
IBEX Documentation

Release 2.6.0

Gilles Chabert

Sep 19, 2018

Contents

1	The Core Library	1
1.1	Introduction	1
1.2	Installation (Release 2.6)	3
1.3	Tutorial	8
1.4	Interval Computations	22
1.5	Functions	39
1.6	Constraints	49
1.7	Systems	50
1.8	The Minibex Language	55
1.9	Contractors	63
1.10	Separators	85
1.11	Sets	90
1.12	Strategies	101
1.13	References	107
1.14	A complete Example: SLAM with outliers	109
1.15	Do it Yourself!	118
2	IbexSolve	129
2.1	IbexSolve	129
3	IbexOpt	137
3.1	IbexOpt	137
4	Java Plugin (for Choco)	143
4.1	Java Plugin (for Choco)	143
5	Indices and tables	147

1.1 Introduction

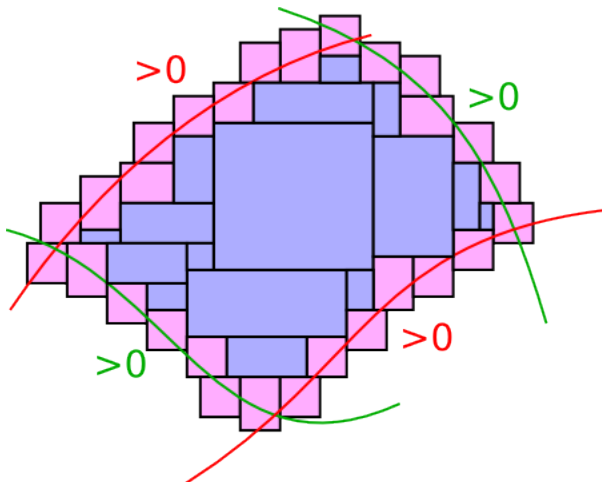
Ibex is a C++ numerical library based on **interval arithmetic** and **constraint programming**.

It can be used to solve a variety of problems that can roughly be formulated as:

Find a reliable characterization with boxes (Cartesian product of intervals) of sets implicitly defined by constraints

Reliable means that all sources of uncertainty should be taken into account, including:

- approximation of real numbers by floating-point numbers
- round-off errors
- linearization truncatures
- model parameter uncertainty
- measurement noise



Example of problem: Given a set of nonlinear inequalities $f_1 \leq 0, \dots, f_n \leq 0$, find two sets of boxes S_1 and S_2 such that

$$S_1 \subseteq \{x, f_1(x) \leq 0 \wedge \dots \wedge f_n(x) \leq 0\} \subseteq S_2$$

1.1.1 The API

The API of Ibex can be broken into three layers:

- An extended (symbolic-numeric) interval calculator
- A contractor programming library
- A system solver / global optimizer (supplied as plugins since Release 2.2)

Each usage corresponds to a different layer and each layer is built on top of the previous one.

Ibex does not include low-level interval arithmetic but uses a third library (Ibex is currently automatically compiled either with [Gaol](#) or [Filib](#), depending on your platform).

1.1.2 An extended interval calculator

Ibex allows you to declare symbolically a mathematical function and to perform interval computations with it. For example:

```
Variable x;  
Function f(x, sin(x)+1);
```

defines the “mathematical” object $x \mapsto \sin(x) + 1$.

Note: *Functions* (as well as equalities or inequalities) can either be entered programmatically (using C++ operator overloading) or using a parser of an AMPL-like language called Minibex. Functions accept vector or matrix variables or values; similarities with Matlab are shared on purpose. See the modeling guide.

Now that functions are built, you can use them to perform interval or symbolic operations. Example:

```
Interval x(0,1);  
Interval y=f.eval(x);           // calculate the image of x by f  
Function df(f,Function::DIFF); // derivate f  
Interval z=df.eval_affine(x);   // calculate the image of x by df using affine forms
```

All the classical operations with intervals can be performed with the previously defined functions, including relational (backward) operators, inner arithmetics, automatic differentiation, affine arithmetic, etc.

1.1.3 Contractor programming

Ibex gives you the ability to build high-level interval-based algorithms declaratively through the *contractor programming* paradigm [[Chabert & Jaulin, 2009](#)].

A contractor is basically an operator that transforms a box to a smaller (included) box, by safely removing points with respect to some mathematical property.

The first property one usually wants to enforce is a numerical constraint, like an equation or inequality:

```
Variable x,y,z;
Function f(x,y,z,...);
NumConstraint c(x,y,z,f(x,y,z)=0);
CtcFwdBwd ctc(c); // build the contractor w.r.t f(x,y,z)=0
```

Contraction is performed with a call to the function `contract(...)`:

```
IntervalVector box(3); // build a box for x, y and z
box[0]=...;
box[1]=...;
box[2]=...;
ctc.contract(box); // contract the box
```

Note: A *contractor* is the equivalent of a propagator in finite domain solvers except that it is a pure numerical function (no state).

More complex properties are obtained by combining contractors. For instance:

```
Ctc& c1=... ;
Ctc& c2=... ;
Ctc& c3=... ;
CtcUnion u(CtcInter(c1,c2),c3);
```

will define the contractor $((C_1 \cap C_2) \cup C_3)$.

Ibex contains a variety of built-in operators (HC4, Shaving, ACID, X-newton, q-intersection, etc.).

1.1.4 System solving and global optimization

Finally, Ibex proposes various plugins. In particular, the `IbexSolve` and `IbexOpt` plugins are dedicated to system solving and optimization, and come both with a default black-box solver and global optimizer for immediate usage. See the [IbexSolve](#) and [IbexOpt](#) documentations.

1.2 Installation (Release 2.6)

1.2.1 Standard install

Note: Ibex can either be compiled with [Gaol](#), [Filib](#) or [Profil/Bias](#). The library installed by default is Gaol, since our experiments have shown that this library prevails over the others. So the standard installation will automatically extract and build the Gaol library (and its dependencies) from the bundle. However, because of some installation problems with Gaol under Windows, the library installed by default under this platform as of today is Filib (this will be changed in the future).

Linux and MacOS

The installation assumes your system meets some [requirements](#).

Save the archive `ibex-2.6.0.tar.gz` in some Ibex folder and:

```
~/Ibex/$ tar xvfz ibex-2.6.0.tar.gz
~/Ibex/$ cd ibex-2.6.0
~/Ibex/ibex-2.6.0/$ ./waf configure
~/Ibex/ibex-2.6.0/$ sudo ./waf install
```

Windows

- **Install MinGW+Msys.** Select the following components to be installed:
 - A MSYS Installation for MinGW Developers
 - A Basic MinGW Installation
 - The GNU C++ Compiler
- With recent releases of MinGW, you have to run the executable `pi.bat` in `C:\MinGW\msys\1.0\postinstall`
- Install **Python2** (**warning:** the script are currently not compatible with python3)
- Create a new directory `Ibex` in the shell of MinGW (to open the shell of MinGW, click on `Start -> MinGW -> MinGWShell`):

```
mkdir Ibex
```

Note: if you don't have MinGW in your Start menu, run the executable file `msys.bat` in `C:\MinGW\msys\1.0`.

- Assuming you have installed MinGW in `C:\`, the `Ibex` directory you have created should be located on your system here:

```
C:\MinGW\msys\1.0\home\[user]\Ibex
```

We will assume now that this folder is the root folder of `ibex`.

- Save the archive `ibex-2.6.0.tar.gz` in `C:\MinGW\msys\1.0\home\[user]\Ibex`
- Configure `Ibex` (still in the shell of MinGW):

```
~/Ibex/$ export PATH="$PATH:/c/Python27"  
~/Ibex/$ tar xvfz ibex-2.6.0.tar.gz  
~/Ibex/$ cd ibex-2.6.0  
~/Ibex/ibex-2.6.0/$ ./waf configure --prefix=/c/MinGW/msys/1.0/home/[user]/Ibex/  
→ibex-2.6.0
```

Note: the paths must be entered in Linux-style (don't use backslash ("`\"`") as separator).

- Install `Ibex`:

```
~/Ibex/ibex-2.6.0/$ ./waf install
```

Note: For mysterious reasons, the command sometimes gets frozen (this was observed while compiling `Filib`). Use `Control-C` to interrupt the command and run it again. Do this several times until compilation is over.

Note: if `g++` is not found, it probably means that you have not run the "postinstall" script of MinGW (see above).

Requirements

The following applications must be installed.

- `g++`
- `gcc`
- `flex`

- bison
- python2.x (**warning:** the script are currently not compatible with python3)
- make
- pkg-config (*optionnal*)

On Ubuntu, you can install all you need with:

```
~$ sudo apt-get install -y python2.7 flex bison gcc g++ make pkg-config
```

1.2.2 Configuration options

The full list of options supported by `waf configure` can be obtained with:

```
~/Ibex/ibex-2.6.0/$ ./waf --help
```

This will display the full list of installed interval/LP libraries and plugins with their specific options, as well as benchmarking features.

In particular, `waf configure` supports the following options:

--enable-shared Compile Ibex as a dynamic library.

If Ibex is compiled as a dynamic library in a local folder, you must set the library path accordingly to execute a program. Under Linux:

```
$ export LD_LIBRARY_PATH=[prefix]/lib/
```

Under MacOS:

```
$ export DYLD_LIBRARY_PATH=[prefix]/lib
```

Under MinGW:

```
$ export PATH=$PATH:/c/MinGW/msys/1.0/home/[user]/Ibex/
↪ibex-2.6.0/lib
```

Under a Windows command window:

```
> set PATH=%PATH%;C:\MinGW\msys\1.0\home\[user]\Ibex\ibex-
↪2.6.0\lib;C:\MinGW\bin
```

--prefix=PREFIX Set the folder where Ibex must be installed (by default, `/usr/local`).

You can use this option to install Ibex in a local folder.

--with-debug Compile Ibex in debug mode

Compiler optimizations are all discarded (`-O0`), low-level assertions in the code are activated and debug information is stored (`-g -pg`)

Once Ibex has been compiled with this option, you should also compile your executable in debug mode. If you use the makefile of `examples/`, simply write:

```
make DEBUG=yes ...
```

--interval-lib=gaol Use Gaol as interval library (recommended)

- interval-lib=filib** Use Filib as interval library
- interval-lib=bias** Use Profil/Bias as interval library (legacy: support not guaranteed)
- interval-lib=direct** Use non-rigorous interval arithmetic (essentially for embedded systems with specific processor architectures that do not support rounding modes) (experimental: support not guaranteed)
- with-solver** Enable IbexSolver (the plugin is installed by default)
- with-optim** Enable IbexOpt (the plugin is installed by default)
- lp-lib=soplex** Install Ibex with the LP solver Soplex. The plugin archive contains a version of soplex so it is not necessary to have Soplex already installed on your system. Soplex is under [ZIB](#) academic licence. If you intend to use Ibex with Soplex commercially, you may consider contacting Soplex for a commercial licence.

If you install your own version of Soplex, use the following argument:

```
make ZLIB=false
```

and if Ibex is installed as a dynamic library (`--enable-shared`), Soplex must also be installed as a dynamic library. For this, add the option `SHARED=true` to the previous command:

```
make ZLIB=false SHARED=true
```

Under Windows, add also `SHAREDLIBEXT=dll` to the previous command.

Warning: The current release of Ibex is not compatible with Soplex 2.0.

- soplex-path=PATH** Set the (absolute) path of Soplex to `PATH` (to be used with `--lp-lib=soplex`). The plugin archive contains a version of Soplex so this option is not required. `PATH` is the absolute path where Soplex is installed (don't use relative path like `--soplex-path=../soplex-xx`).

If Ibex is compiled as a shared library, you must also add the libpath of Soplex in `LD_LIBRARY_PATH`:

```
~/Ibex/ibex-2.6.0/$ export LD_LIBRARY_PATH=[prefix]/lib/  
↪:[soplex-path]/lib/
```

Under Windows, if you run a program from a command window, the `PATH` variable must also be updated:

```
> set IBEX_PATH=C:\MinGW\msys\1.0\home\[user]\Ibex\ibex-2.  
↪6.0  
> set SOPLEX_PATH=...  
> set PATH=%PATH%;%IBEX_PATH%\lib;%SOPLEX_PATH%\lib;  
↪C:\MinGW\bin
```

- lb-lib=clp** Install Ibex with the LP solver CLP (from the COIN-OR project). The plugin archive contains a version of CLP so it is not necessary to have CLP already installed on your system. This option is **experimental**, i.e., support for installation issues may not be guaranteed.
- clp-path=PATH** Set the (absolute) path of CLP to `PATH` (to be used with `--lp-lib=clp`). The plugin archive contains a version of CLP so this option is not required.

PATH is the absolute path where CLP is installed (don't use relative path like `--clp-path=./clp-xx`). If Ibex is compiled as a shared library, you must also add the libpath of CLP in `LD_LIBRARY_PATH`.

- lp-lib=cplex** Install Ibex with the LP Solver CPLEX. The path of CPLEX must be provided with the `--cplex-path` option. This option is **experimental**, i.e., support for installation issues may not be guaranteed.
- cplex-path=PATH** Set the path of CPLEX (to be used with `--lp-lib=cplex`). PATH is the absolute path where CPLEX is installed (don't use relative path). If Ibex is compiled as a shared library, you must also add the libpath of CPLEX in `LD_LIBRARY_PATH`.

1.2.3 Compiling a Test Program

Copy-paste the following example code in a file named `foo.cpp`

```
#include "ibex.h"
#include <iostream>

using namespace std;
using namespace ibex;

int main(int argc, char** argv) {
    Interval x(0,1);
    cout << "My first interval:" << x << endl;
}
```

There is a simple “makefile” in the `examples` folder that you can use to compile your own programs (note: this makefile uses the extended syntax of GNU make).

This makefile however assumes `pkg-config` is installed on your system, which is done by default on many Linux distribution). To install `pkg-config` under MinGW, follow the steps given [here](#).

So, place the file `foo.cpp` in the `examples/` folder and:

```
~/Ibex/ibex-2.6.0/$ cd examples
~/Ibex/ibex-2.6.0/examples$ make foo
~/Ibex/ibex-2.6.0/examples$ ./foo
```

Note:

1. It may be necessary to set the `PKG_CONFIG_PATH` to `[prefix]/share/pkgconfig` where `[prefix]` is `/usr/local` by default or whatever path specified via `--prefix`:

```
~/Ibex/ibex-2.6.0/$ export PKG_CONFIG_PATH=/usr/local/share/pkgconfig/
```

Under Windows, if you have compiled Ibex with `--enable-shared` you can run the program from a command window. Just update the path to dynamically link against Ibex:

```
> set IBEX_PATH=C:\MinGW\msys\1.0\home\[user]\Ibex\ibex-2.6.0
> set PATH=%PATH%;%IBEX_PATH%\lib;C:\MinGW\bin
> cd %IBEX_PATH%\examples
> foo.exe
```

1.2.4 Running unit tests

You can also run the whole unit tests suite with the **installed** version of Ibex.

To this end, you must install first the [cppunit library](#). Then run:

```
~/Ibex/ibex-2.6.0/$ ./waf utest
```

Note also the following command:

```
~/Ibex/ibex-2.6.0/$ ./waf check
```

as a handy shortcut for:

```
~/Ibex/ibex-2.6.0/$ ./waf build install clean utest
```

1.2.5 Uninstall

Simply type in the path of IBEX (under the shell of MinGW for Windows):

```
~/Ibex/ibex-2.6.0$ sudo ./waf uninstall
~/Ibex/ibex-2.6.0$ ./waf distclean
```

Note: sudo is useless under MinGW or if Ibex is installed in a local folder.

It is highly recommended to uninstall Ibex like this before upgrading to a new release or installing a plugin.

1.2.6 Troubleshooting

Headers of Gaol not found

When running `waf configure`, I get messages like this:

```
Checking for header ['gaol/gaol.h', 'gaol/gaol_interval.h'] : not found
...
```

Does it mean that Ibex is not properly installed?

Answer: No, this message simply indicates that gaol was not found on your system and that it will be automatically extracted from the bundle. It is not an error message.

Linking problem with CoinOR

If the linker fails with undefined reference to `dgetrf` and `dgetrs`, it is probably because you have installed Lapack. You can either:

- try to adapt the makefile to link with Lapack. Remove Lapack, reinstall Ibex and reinstall Lapack (in this order).

1.3 Tutorial

Note: This tutorial is for the Ibex core library. If you just want to solve **equations** or an **optimization problem**, jump to the documentation of the *IbexSolve* or *IbexOpt* plugins.

1.3.1 Basic Interval computations

Start a program

To write a program with Ibex, use the following canvas:

```
#include "ibex.h"

using namespace std;
using namespace ibex;

int main(int argc, char** argv) {

    // write your own code here

}
```

You can execute by yourself all the code snippets of this tutorial, using this canvas.

To compile a program, the easiest way is to copy-paste the makefile of the `examples/` subfolder of Ibex. See also *Compiling a Test Program*.

Creating intervals

Here are examples of intervals

```
Interval x(1,2);           // create the interval [1,2]
Interval y;                // create the interval (-oo,oo)
Interval z=Interval::ALL_REALS; // create the interval (-oo,oo)
Interval w=Interval::EMPTY_SET; // create the empty interval
```

Operation between intervals

C++ operator overloading allows you to calculate the sum of two intervals by using directly the “+” symbol:

```
// - create the interval x=[1,2] and y=[3,4]
// - calculate the interval sum x+y
Interval x(1,2);
Interval y(3,4);
cout << "x+y=" << x+y << endl; // display [4,6]
```

You can use the other operators similarly ($-$, $*$, $/$).

Applying a function to an interval

All the elementary functions can be applied to intervals, and composed in an arbitrarily way:

```
Interval x(0,1);
Interval y=exp(x+1); //y is [1,7.389...]
```

Interval vectors

You can create an interval vector by using an intermediate array of $n \times 2$ double, representing the lower and upper bounds of each components. The first argument of the constructor of `IntervalVector` in this case is the dimension (here, 3), the second the array of `double`.

```
double _x[3][2]={ {0,1}, {2,3}, {4,5} };
IntervalVector x(3,_x); //create ([0,1],[2,3],[4,5])
```

You can also create an interval vector by duplicating a given interval or simply create the empty interval vector.

```
IntervalVector x(3,Interval(1,2)); //create ([1,2],[1,2],[1,2])
IntervalVector y=IntervalVector::empty(3); //create a vector of 3 empty
↪ intervals
```

Interval matrices

Interval matrices can be created in a similar way. However, since we cannot build 3-dimensional arrays in C++, all the bounds must be set in a single $n \times 2$ array representing the matrix row by row (and n is the total number of entries of the matrix). The two first arguments of the constructor are the number of rows and columns respectively. The last one is the array of double. Here is an example of a 3×3 matrix:

```
double _M[9][2]={ {0,1}, {0,1}, {0,1},
                  {0,2}, {0,2}, {0,2},
                  {0,3}, {0,3}, {0,3} };
IntervalMatrix M(3,3,_M);
//create ([0,1] [0,1] [0,1]) ; ([0,2] [0,2] [0,2]) ; ([0,3] [0,3] [0,3]))
```

Operations between matrices and vectors

You can use the usual operations of linear algebra between matrices and vectors (*sum of vectors, transpose of vectors, sum of matrices, left multiplication of a matrix by a scalar, etc.*).

```
// -----
// Vector/matrix interval arithmetic
// - create an interval vector x
// - create an interval matrix M
// - calculate M*x
// - calculate M'*x, where M' is the transpose of M
// -----

double _x[3][2]={ {0,1}, {2,3}, {4,5} };
IntervalVector x(3,_x);

double _M[9][2]={ {0,1}, {0,1}, {0,1}, // 3*3 matrix of intervals
                  {0,2}, {0,2}, {0,2},
```

(continues on next page)

(continued from previous page)

```

        {0,1},{0,1},{0,1}};

IntervalMatrix M(3,3,_M);
IntervalVector y=M*x;           // matrix-vector multiplication
IntervalMatrix N=M.transpose(); // N is M^T

```

Midpoint, radius, magnitude, etc.

These usual properties can be obtained for intervals. They are also all extended to interval vectors or matrices componentwise. For instance, the radius of an interval matrix is the (real) matrix of the radii.

As a consequence, Ibex also has classes to handle real (versus interval) vectors and matrices. Mathematical Operations (like the sum) can also be applied to these objects but, of course, using this times floating-point arithmetic (not interval).

```

// -----
// Mixing real/interval vector/matrices
// - calculate the magnitude of an interval matrix (a real matrix)
// - calculate the midvector of an interval vector (a real vector)
// - multiply the latters (floating point arithmetic)
// -----

double _x[][2]={0,1},{0,1},{0,1}};
IntervalVector x(3,_x);

double _M[9][2]={0,1},{0,1},{0,1},
                {0,2},{0,2},{0,2},
                {0,1},{0,1},{0,1}};
IntervalMatrix M(3,3,_M);

Matrix M2=M.mag(); // the matrix of magnitudes
Vector x2=x.mid(); // the vector of midpoints
Vector y=M2*x2;    // a matrix-vector product (subject to roundoff errors)

```

1.3.2 Functions

Creating functions

The easiest way to create a function is with a string directly:

```
Function f("x","y","sin(x+y)"); // create the function (x,y)->sin(x+y)
```

However, this has some limitations (see *Advanced examples*). Another (more flexible) way to create function is using C++ operator overloading. The only difference is that you must have to first build *variables*:

```

Variable x("x");
Variable y("y");
Function f(x,y,sin(x+y)); // create the function (x,y)->sin(x+y)

```

Constants inside functions

You can insert interval constants in the expression of a function, even in C++-style. For instance, if you want to create the function $x \mapsto \sin(2x)$, just write:

```
Variable x;  
Function f(x, sin(2*x)); // create the function (x,y)->sin(2*x)
```

Assume now that the function to be created is $x \mapsto \sin(\pi x)$. It is still possible to use a double representing approximately π ; but to keep numerical reliability, it is required in this case to use an interval constant enclosing π . Next function must be seen as a “thick” function that rigorously encloses $\sin(\pi x)$:

```
Interval pi(3.1415, 3.1416);  
Variable x;  
Function f(x, sin(pi*x)); // create the function (x,y)->sin(pi*x)
```

Or with strings directly:

```
Function f("x", "sin([3.1415, 3.1416]*x)"); // create the function (x,y)->  
↪ sin(π*x)
```

Functions with vector arguments

Arguments of a function are not necessarily scalar variables. They can also be vectors or matrices. In the following example, we build the distance function: $dist : (a, b) \mapsto \|a - b\|$ where a and b are 2-dimensional vectors.

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

We can also create the same function with string directly (note that the syntax quite *differs*).

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

Note: *Evaluation* of a thick function will necessarily result in an interval with non-null diameter, even if the argument is reduced to a point.

Composing functions

You can compose functions to build new functions. We build here the function that maps a vector x to its distance with a constant point $(1, 2)$. To this end, we first define a generic distance function $dist(a, b)$ as above.

```
/* create the distance function with 2 arguments */  
Variable a(2);  
Variable b(2);  
Function dist(a, b, sqrt(sqr(a[0]-b[0])+sqr(a[1]-b[1])));  
  
/* create the constant vector pt=(1,2) */  
Vector pt(2);  
pt[0]=1;  
pt[1]=2;  
  
/* create the function x->dist(x,pt). */  
Variable x(2);  
Function f(x, dist(x, pt));
```

The display is as follows. 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>)))
```

Vector-valued functions

Let us start with a basic example: the function $x \mapsto (x - 1, x + 1)$.

With strings:

```
Function f("x", "(x-1,x+1)");
```

With operator overloading:

```
Variable x;
Function f(x,Return(x-1,x+1));
```

Note: The Return keyword is only necessary when the output of a function is a vector (or a matrix).

Now, in line with the previous sections, let us define a more complicated example: the function that associates to a vector x its distance with two fixed points `pt1` and `pt2` initialized in our program to (0,0) and (1,1):

$$f : x \mapsto (\|x - (1, 1)\|, \|x - (0, 0)\|).$$

```
// -----
// Vector-valued functions
// -----
/* create the distance function with 2 arguments */
Variable x(2,"x");
Variable pt(2,"p");
Function dist(x,pt,sqrt(sqr(x[0]-pt[0])+sqr(x[1]-pt[1])), "dist");

/* create the two constant vectors */
Vector pt1=Vector::zeros(2);
Vector pt2=Vector::ones(2);

/* create the function x->(dist(x,pt1),dist(x,pt2)). */
Function f(x,Return(dist(x,pt1),dist(x,pt2)), "f");

cout << f << endl;
```

The last construction is much more cumbersome with strings.

Matrix-valued functions

You can also create functions that return matrices. Here is an example of a function from R to $R^{2 \times 2}$ where:

$$f : x \mapsto ((2x, -x); (-x, 3x)).$$

With strings:

```
Function f("x", "((2*x,x);(-x,3*x))");
```

With C++ operator overloading:

```
Variable x("x");
Function f(x, Return(Return(2*x, x, ExprVector::ROW), Return(-x, 3*x,
↪ExprVector::ROW)));
```

The boolean value `true` given here to the two embedded `Return` means that, each time, the two components must be put in rows, and not in column as it is by default. In contrast, the enclosing `Return` keeps the default behaviour since the two rows are put in column in order to form a 2x2 matrix.

Using the Minibex syntax

To create sophisticated functions we advice you to use an intermediate “minibex” input file as follows instead of embedding the function directly in your C++ program. The previous example can be written in a plain text file:

```
function f(x)
  return ((2*x, -x); (-x, 3*x));
end
```

Save this file under the name “myfunction.txt”. Now, you can load this function in your C++ program:

```
Function f("myfunction.txt");
```

Minibex syntax with intermediate variables

When several occurrences of the same subexpression occur in a function, it is a good idea for readability (and, actually, efficiency) to put this subexpression into intermediate variables.

The following example is the function that returns the rotation matrix from the three Euler angles. In this function an expression like `cos(phi)` occurs several times.:

```
/* Computes the rotation matrix from the Euler angles:
   roll(phi), the pitch (theta) and the yaw (psi) */

function 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
```

Evaluation over floating-point numbers

Given input double values `x`, you can obtain a rigorous inclusion of `f(x)` either using `eval`, `eval_vector` or `eval_matrix`. These functions return interval enclosures of the true result.

These functions are presented below in a more general setting where the inputs are intervals as well.

So, to get the image by `f` of fixed floating-point values, simply create degenerated intervals in the next examples.

Interval evaluation

The interval evaluation of f is the image of the given input interval vector $[x]$ by f , this range being noted by $f([x])$:

$$f([x]) := \{f(x), x \in [x]\}.$$

Let us start with a real-valued function f with scalar arguments:

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

double _x[2][2]={ {1,2},{3,4}};
IntervalVector xy(2,_x); // build xy=([1,2],[3,4])
Interval z=f.eval(xy); // z=f(xy)=sin([4,6])=[-1, -0.27941]
```

The sine function is not monotonic on $[4,6]$ and actually reaches its minimum at $3\pi/2$.

Note that the `eval` takes an `IntervalVector` as argument, even if there is only one variable. So, in the latter case, you have to build a vector reduced to a single component.

We consider now a vector-valued function. Since the return type of an evaluation is not anymore an `Interval` but an `IntervalVector`, we have to use a method with a different signature, namely, `eval_vector`:

```
Variable a;
Function f(a,Return(sqr(a),-a));

IntervalVector x(1,Interval(1,2)); // build x=([1,2])

/* calculate y=f(x)=([1, 4] ; [-2, -1]) */
IntervalVector y=f.eval_vector(x);
```

Finally, for a matrix-valued function, the evaluation is obtained via `eval_matrix`. We assume again that the following matrix-valued function

$$f : x \mapsto ((2x, -x); (-x, 3x))$$

has been written in a “minibex” input file (see above).

```
Function f("myfunction.txt");

IntervalVector x(1,Interval(0,1));

// calculate M=f(x)=([0, 2] , [-1, -0]) ; ([-1, -0] , [0, 3])
IntervalMatrix M=f.eval_matrix(x);
```

Interval gradient

For a scalar-valued function, you can get an interval enclosure of the gradient:

```
Variable x,y,z;
Function f(x,y,z,x*y+z*y);

double _xyz[3][2]={ {0,1},{0,2},{0,3}};
IntervalVector xyz(3,_xyz);

/* calculate g=grad_f(x)=(y,x+z,y)=[0, 2] ; [0, 4] ; [0, 2]) */
IntervalVector g=f.gradient(xyz);
```

Interval Jacobian matrix

For a vector-valued function, you can get an interval enclosure of the Jacobian matrix:

```
// -----
// Vector-valued functions, Jacobian matrix
//
// > create the function dist:(x,pt)->||x-pt||
// > create the function f:x->(dist(x,pt1),dist(x,pt2))
// > calculate the Jacobian matrix of f over the box
// -----

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)));

double init_box[][2] = { {-10,10},{-10,10} };
IntervalVector box(2,init_box);

/* calculate J as a m*n interval enclosure of the Jacobian matrix */
IntervalMatrix J=f.jacobian(box);
// -----
```

Backward (or contraction)

One of the main feature of Ibex is the ability to *contract* a box representing the domain of a variable x with respect to the constraint that $f(x)$ belongs to a restricted input range $[y]$. Rigorously, given two intervals $[x]$ and $[y]$, the contraction gives a new interval $[z]$ such that

$$\forall x \in [x], \quad f(x) \in [y] \implies x \in [z] \subseteq [x]$$

One way to do this is by using the famous *backward* algorithm. This algorithm does not return a new interval $[z]$ but contract the input interval $[x]$ which is therefore an input-output argument.

In the following snippet we require the function $\sin(x+y)$ to take the value -1 (a degenerated interval). With an initial box $(x,y)=([1,2],[3,4])$, we obtain the result that (x,y) must lie in the subdomain $([1, 1.7123] ; [3, 3.7124])$.

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

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

/* the backward sets box to ([1, 1.7123] ; [3, 3.7124]) */
f.backward(-1.0,box);
```

1.3.3 Constraints

To create a constraint, you can also either use strings or C++ objects:

With strings:

```
NumConstraint c("x", "y", "z", "x+y<=z");
```

With C++ objects:

```
Variable x,y,z;
NumConstraint c(x,y,z,x+y<=z);
```

You can also refer to a previously defined function *f* to create, e.g., $f(x) \leq 0$:

```
Variable x,y,z;
Function f(x,y,z,x+y-z);
NumConstraint c(f, LEQ);
```

1.3.4 Contractors

What is a contractor programming?

The key idea behind *contractor programming* [Chabert & Jaulin, 2009] is to abstract the algorithm from the underlying constraint and to view it a function “*C*”:

$$C : \mathbb{IR}^n \rightarrow \mathbb{IR}^n \text{ such that } C([x]) \subseteq [x],$$

where \mathbb{IR} denotes the set of real intervals.

In other word, we take as primary concept the *operational* definition of a constraint.

In this way, operators (like the intersection and the others below) can be extended to contractors.

Since contractors implicitly represent sets, the fundamental advantage of extending operations to contractors is that we actually extend these operations to sets.

All contractors in Ibex are algorithms represented by different classes. See the [strategy pattern](#) for more information on this design choice. Classes representing contractors are prefixed by `Ctc`.

Forward-Backward

The standard way to contract with respect to a constraint is by using the *forward-backward* algorithm. The corresponding class is `CtcFwdBwd`.

A constraint has to be built first using the `NumConstraint` class. In the following piece of code, we build a forward-backward contractor with respect to $x+y=z$.

```
Variable x,y,z;
NumConstraint c(x,y,z,x+y=z);
CtcFwdBwd ctc(c);
```

Of course, the expression of a constraint can involve a previously defined function. Furthermore, if the constraint is simply “ $f=0$ ”, where *f* is a `Function` object, it is not necessary in this case to build an intermediate `NumConstraint` object. One can directly give the function *f* that has to be nullify to `CtcFwdBwd`. In the next example, we consider the problem of finding the point which distance from both (0,0) and (1,1) is $\sqrt{2}/2$. The solution is (0.5,0.5).

```
Variable x,y;
double d=0.5*sqrt(2);
Function f(x,y,Return(sqrt(sqr(x)+sqr(y))-d, sqrt(sqr(x-1.0)+sqr(y-1.0))-d));

IntervalVector box(2,Interval(-10,10));

/* we give f directly (means that the constraint is f=0) */
CtcFwdBwd c(f);
c.contract(box);

/* display ([0.2929, 0.7072] ; [0.2929, 0.7072]) */
cout << box << endl;
```

Of course, the result is rather crude. Remember that the purpose of `CtcFwdBwd` is to contract *quickly* with respect to *any* numerical constraint: it is widely applicable and takes a time that is only proportional to the expression size. In the other hand, it is not accurate in general.

See more

Fixpoint

The fixpoint operator applies a contractor `C` iteratively:

$$\text{fixpoint}(C) : [x] \mapsto C(\dots C([x]) \dots),$$

while the “gain” is more than the given `ratio`. More precisely, the “gain” is the relative Hausdorff distance between the input box $[x]$ and the output box $C([x])$ but, often, you can ignore the precise meaning of this gain and just consider that the procedure will loop until the contracted box will roughly differ “by ratio” from the input one.

Let us now follow the previous example. As said, the solution is (0.5,0.5). We can see that simply embedding the `CtcFwdBwd` contractor in a fixpoint loop (with a `ratio` set to 0.1) gives a box with sharp bounds.

```
Variable x,y;
double d=0.5*sqrt(2);
Function f(x,y,Return(sqrt(sqr(x)+sqr(y))-d, sqrt(sqr(x-1.0)+sqr(y-1.0))-d));

IntervalVector box(2,Interval(-10,10));

CtcFwdBwd c(f);
CtcFixPoint fp(c,1e-03);

fp.contract(box);
/* display ([0.4990, 0.5001] ; [0.4990, 0.5001]) */
cout << "box after fixpoint=" << box << endl;
```

Intersection, union & composition

Given two or more contractors, we can apply the two logical operators *union* and *intersection*:

$$\begin{aligned} \text{union}(C_1, \dots, C_n) : [x] &\mapsto C_1([x]) \cup \dots \cup C_n([x]). \\ \text{inter}(C_1, \dots, C_n) : [x] &\mapsto C_1([x]) \cap \dots \cap C_n([x]). \end{aligned}$$

However, the latter operation is barely used and usually replaced by the *composition*:

$$\text{compo}(C_1, \dots, C_n) : [x] \mapsto C_n(\dots (C_1([x]) \dots).$$

Indeed, one can see that the composition amounts to the same logical operation (the intersection of each contractor's set), but in a more efficient way since we take advantage of the contraction performed by C_1, \dots, C_{i-1} when contracting with C_i . In contrast, the intersection operator calls each contractor independently on the same initial box.

The corresponding classes are `CtcUnion` and `CtcCompo`.

As a rule of thumb, use `CtcUnion` for the union of two contractors and `CtcComp` for the intersection. Here is an example with the union:

```
Variable x;
NumConstraint c1(x,x<=-1);
NumConstraint c2(x,x>=1);
CtcFwdBwd ctc1(c1);
CtcFwdBwd ctc2(c2);
IntervalVector box(1,Interval::POS_REALS); // the box [0,oo)

CtcUnion ctc3(ctc1,ctc2); // a contractor w.r.t. (x<=-1 or x>=1)
ctc3.contract(box); // box will be contracted to [1,oo)
cout << box << endl;
```

Here is an example with the intersection (composition):

```
Variable x;
NumConstraint c1(x,x>=-1);
NumConstraint c2(x,x<=1);
CtcFwdBwd ctc1(c1);
CtcFwdBwd ctc2(c2);
IntervalVector box(1,Interval::ALL_REALS); // the box (-oo,oo)
CtcCompo ctc3(ctc1,ctc2); // a contractor w.r.t. (x>=-1 and x<=1)
ctc3.contract(box); // box will be contracted to [-1,1]
cout << box << endl;
```

Interval Newton

When a function is “square” (the dimension is the same as the codimension, i.e., $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$), you can contract a box with respect to the constraint $f(x)=0$ using the interval Newton iteration.

You just have to build a `CtcNewton` object with the function and call `contract`.

This operator can give extremely accurate bounds proving that the input box is already “sufficiently” small (that is, “inside the convergence basin” of Newton’s iteration). In the following example, we give a box that encloses the solution (1,0) with a radius of 10^{-3} . Newton’s iteration contracts this box down to the maximal precision:

```
Variable x,y;
Function f(x,y,Return(sqrt(sqr(x)+sqr(y))-1, sqrt(sqr(x-1.0)+sqr(y-1.0))-1));
double init_box[][2]={0.999,1.001},{-0.001,0.001}};
IntervalVector box(2,init_box);

// Build an interval Newton iteration
// for solving f(x)=0 where f is
// a vector-valued function representing
// the system.
CtcNewton newton(f);

/* Contract the box with Newton */
newton.contract(box);
```

(continues on next page)

(continued from previous page)

```
/* display a very small box enclosing (1,0) */
cout << box << endl;
```

Propagation

The *propagation* operator calculates the fixpoint of (the composition of) n contractors by using a more sophisticated (“incremental”) strategy than a simple loop. So, semantically, the propagation operator can be defined as follows:

$$\text{propagation}(C_1, \dots, C_n) := \text{fixpoint}(\text{compo}(C_1, \dots, C_n)).$$

(see above for the definition of the fixpoint and composition operators).

The key idea behind this operator is to avoid calling contractors that will certainly leave the box intact. Contractors that can potentially enforce a contraction are determined typically from the syntax of their underlying constraint. Consider for instance two contractors, C_1 w.r.t. $f(x,z)=0$ and C_2 w.r.t. $g(x,y)=0$. Assume that the fixpoint for C_1 is reached with the current box $([x],[y],[z])$. If a call to C_2 only contracts the second interval (the one corresponding to y), it is then useless to call C_1 again.

So, by using such principle, the propagation calculates the fixpoint by “awaking” contractors only when necessary. Of course, the more sparse the constraint system, the more valuable the propagation, when compared to a simple fixpoint.

See more

Q-Intersection (robustness w.r.t. outliers)

The Q-intersection is typically used in a context where we have a set of contractors that result from measurements (each measurement enforces a constraint), some of which can be incorrect.

If we are sure that at least q measurements are correct (which amounts to say that the number of outliers is bounded by $N-q$) then we can contract the box in a robust way, by calculating the union of the boxes resulting from the contraction with all combinaisons of q contractors among N .

Mathematically, with (i_1, \dots, i_q) ranging over the set of all q distinct indices between 0 and $N-1$:

$$q - \text{inter}(C_1, \dots, C_n, q) := \text{union}(\dots, \text{inter}(C_{i_1}, \dots, C_{i_q}), \dots)$$

Here is a simple example inspired from parameter estimation.

We assume a point (x,y) has to be localized. We measure 4 distances “bD” from 6 (approximately known) points (bX,bY) . Each position bX , bY and each distance bD has an uncertainty $[-0.1,0.1]$. We also know there may be at most one outlier.

The solution point is: $x=6.32193$ $y=5.49908$

First of all, let us enter the coordinates of the points (bX,bY) and the distances. This data will simulate our measurements.

```
const int N=6;
/* The measurements (coordinates of the points and distances) */
double bx[N]={5.09392,4.51835,0.76443,7.6879,0.823486,1.70958};
double by[N]={0.640775,7.25862,0.417032,8.74453,3.48106,4.42533};
double bd[N]={5.0111,2.5197,7.5308,3.52119,5.85707,4.73568};
```

define the measurement intervals (with uncertainty taken into account)


```

Interval bX[N];
Interval bY[N];
Interval bD[N];

/* add uncertainty on measurements */
for (int i=0; i<N; i++) {
    bX[i]=bx[i]+Interval(-0.1,0.1);
    bY[i]=by[i]+Interval(-0.1,0.1);
    bD[i]=bd[i]+Interval(-0.1,0.1);
}

```

Now, we artificially introduce an outlier by shifting the interval for one measurement (here, x position n°5) by a large value:

```
bX[5]+=10;
```

Now, all our simulated data is set up. We just have to define the contractors. We first declare the distance function and then 6 contractors corresponding to the distance with each (bX,bY):

```

Variable x(2);
Variable px,py;
Function dist(x,px,py,sqrt(sqr(x[0]-px)+sqr(x[1]-py)));

Function f0(x,dist(x,bX[0],bY[0])-bD[0]);
Function f1(x,dist(x,bX[1],bY[1])-bD[1]);
Function f2(x,dist(x,bX[2],bY[2])-bD[2]);
Function f3(x,dist(x,bX[3],bY[3])-bD[3]);
Function f4(x,dist(x,bX[4],bY[4])-bD[4]);
Function f5(x,dist(x,bX[5],bY[5])-bD[5]);

CtcFwdBwd c0(f0);
CtcFwdBwd c1(f1);
CtcFwdBwd c2(f2);
CtcFwdBwd c3(f3);
CtcFwdBwd c4(f4);
CtcFwdBwd c5(f5);

```

We can contract now a box with the q-intersection of these contractors:

```

/* The initial box: [0,10]x[0,10] */
IntervalVector initbox(2,Interval(0,10));

/* Create the array of all the contractors */
Array<Ctc> array(c0,c1,c2,c3,c4,c5);
/* Create the q-intersection of the N contractors */
CtcQInter q(array,5); // 2 is the number of variables, 5 the number of
→correct measurement
/* Perform a first contraction */
IntervalVector box=initbox;
q.contract(box);
cout << "after q-inter = " << box << endl;

```

The displayed result is ([3.9667, 7.2381] ; [4.5389, 8.1479]). Of course, we can do better by calculating a fixpoint of the q-intersection:

```

/* Build a Fix-point of the q-intersection */
CtcFixPoint fix(q);

```

(continues on next page)

(continued from previous page)

```

/* Perform a stronger contraction with the fixpoint */
fix.contract(box);
cout << "after fix+q-inter =" << box << endl;

```

The displayed result is ([5.9277, 6.8836] ; [5.0914, 5.7996]) which, indeed, better encloses the solution point $x=6.32193$ $y=5.49908$.

Build your own contractor

To create a contractor, you just have to - declare a class that extends `Ctc` - create inside a function `contract` that takes a reference to a box (`IntervalVector&`) and contracts it. The function returns `void`.

In the following example, we create a contractor that simply divides by two the radius of each component.

```

class MyContractor : public Ctc {

public:
    MyContractor(int nb_var) : Ctc(nb_var) {}

    void contract(IntervalVector& box) {
        box=box.mid()+0.5*Interval(-1,1)*box.rad();
    }

};

```

Then, if we create this contractor and applies it several time to the same box, we can observe the expected result:

```

/* build the contractor for 3-dimensional boxes. */
MyContractor c(3);

/* create the box [0,1]x[0,1]x[0,1] */
IntervalVector x(3,Interval(0,1));

c.contract(x);
cout << x << endl; // ([0.25, 0.75] ; [0.25, 0.75] ; [0.25, 0.75])

c.contract(x);
cout << x << endl; // ([0.375, 0.625] ; [0.375, 0.625] ; [0.375, 0.625])

c.contract(x);
cout << x << endl; // ([0.4375, 0.5625] ; [0.4375, 0.5625] ; [0.4375, 0.5625])

```

This contractor can now be combined with the ones built-in. For instance, we can decide to calculate the fixpoint. Then, the result is a small box enclosing (0.5,0.5,0.5):

```

CtcFixPoint fp(c,0.001);
fp.contract(x);
cout << x << endl; // ([0.4999999999999999, 0.5000000000000001], ...)

```

1.4 Interval Computations

This chapter present the basic structures used to represent sets and the operations directly related to them.

1.4.1 Intervals, vectors and matrices

Ibex allows you to create interval, interval vectors, interval matrices and array of interval matrices. All these objects represent sets. These objects are respectively of the following classes:

- `Interval`
- `IntervalVector`
- `IntervalMatrix`
- `IntervalMatrixArray`.

It should be emphasized that, mainly for efficiency reasons, the previous list is not an inheritance hierarchy. So, an `Interval` is **not** a particular `IntervalVector` and so on, although one might legitimately expect this (this would also be a perfectly valid design). This is in contrast with Matlab, where, basically, everything is an array of interval matrices.

Some functions return points, not intervals; this is typically the case for the function `mid()` that, given an interval, returns the midpoint. The returned type of `Interval::mid()` is therefore `double`. In the case of an interval vector, the function `mid()` returns a vector of points. In order to keep our system homogeneous, a type has been introduced to represent a vector of reals, namely, `Vector` and similarly for matrices or matrices arrays.

So the following types allow to represent real (floating-point) values:

- `double` (primitive type)
- `Vector`
- `Matrix`
- `MatrixArray`.

The empty set

One unpleasant consequence of our inheritance-free design is that the empty set does not have a unique representation in Ibex. The empty set is “typed” so that an empty interval is not the same object as an empty interval vector or an empty interval matrix. But, of course, they can all be interpreted as the same mathematical object, *the empty set*.

Worse, an empty interval vector has a size just as any other vectors (and similarly, an empty interval matrix has a certain number of rows/columns, etc.). So the empty vector of 3 components is not the same object as the empty vector of 4 components. Imposing a size for the empty vector may seem clumsy at first glance, but it actually simplifies programming in several situations. For instance, if one asks for the complementary of the empty set represented by a n -sized vector, the result will be the n -dimensional box $(-\infty, \infty) \times \dots \times (-\infty, \infty)$. Anyway, this multiple representation is harmless in practice because we always manipulate sets of a priori known dimensions.

It holds the same with matrices and arrays of matrices.

Intervals

Intervals are represented by the class `Interval` (this class wraps the actual class of the underlying sub-library that implements interval arithmetic).

Here are examples of interval definitions. In particular, it is possible to give a single value to define a degenerated interval (reduced to a point). Note the usage of the constants `NEG_INFINITY` and `POS_INFINITY` for infinite bounds.

Note: In Ibex, an interval should never contains a “NaN” (*not a number*), whatever operation you perform with it. If this occurs, something is wrong.

```

Interval x1; // (-oo, +oo)
Interval x2(2); // [2, 2]
Interval x3(x2); // [2, 2]
Interval x4=x2; // [2, 2]

Interval x5(1,2); // [1, 2]

Interval x6(1, POS_INFINITY); // [1, +oo)
Interval x7(NEG_INFINITY, -1); // (-oo, -1]
Interval x8=Interval::EMPTY_SET; // empty set

```

Interval Constants

Some commonly used intervals are already defined as static variables:

Interval::PI	a thin enclosure of π
Interval::TWO_PI	a thin enclosure of π
Interval::HALF_PI	a thin enclosure of $\pi/2$
Interval::EMPTY_SET	\emptyset
Interval::ALL_REALS	$(-\infty, +\infty)$
Interval::ZERO	$[0, 0]$
Interval::ONE	$[1, 1]$
Interval::POS_REALS	$[0, +\infty)$
Interval::NEG_REALS	$(-\infty, 0]$

Let us print all these constants:

```

output << " EMPTY_SET =\t " << Interval::EMPTY_SET << endl;
output << " PI =\t\t " << Interval::PI << endl;
output << " 2 PI =\t\t " << Interval::TWO_PI << endl;
output << " 1/2 PI =\t " << Interval::HALF_PI << endl;
output << " ONE =\t\t " << Interval::ONE << endl;
output << " ZERO =\t\t " << Interval::ZERO << endl;
output << " ALL_REALS =\t " << Interval::ALL_REALS << endl;
output << " POS_REALS =\t " << Interval::POS_REALS << endl;
output << " NEG_REALS =\t " << Interval::NEG_REALS << endl;

```

The output is:

Vectors or Boxes

Vectors of reals are represented by the class `Vector` and interval vectors (or “boxes”) are represented by the class `IntervalVector`.

Creation and initialization of interval vectors can either be done separately (you first create a vector of the desired size and then fill it with the intervals) or simultaneously, by giving an array of `double` in argument of the constructor. The following example creates twice $([0,1],[2,3],[4,5])$:

```

// creates ([0,1],[2,3],[4,5]) in several steps.
IntervalVector x1(3); // so far x1 is ((-oo,oo), (-oo,oo), (-oo,oo))
x1[0]=Interval(0,1); // we initialize each component
x1[1]=Interval(2,3);
x1[2]=Interval(4,5);

```

(continues on next page)

(continued from previous page)

```
// creates the same vector in two steps only
double _x2[3][2]={0,1},{2,3},{4,5}};
IntervalVector x2(3,_x2);
```

This is the same for vectors:

```
// creates (0.1,0.2,0.3) in several steps.
Vector x1(3); // so far x1 is (0,0,0)
x1[0]=0.1;    // we initialize each component
x1[1]=0.2;
x1[2]=0.3;

// creates the same vector in two steps only
double _x2[3]={0.1,0.2,0.3};
Vector x2(3,_x2);
```

You can also create an (interval) vector by duplicating n times the same double/interval and copy another (interval) vector:

```
IntervalVector x2(3,Interval(1,2)); // create ([1,2],[1,2],[1,2])
IntervalVector x3(3,0.1);          // create ([0.1,0.1],[0.1,0.1],[0.
↪1,0.1])
Vector          x4(3,0.1);          // create (0.1,0.1,0.1)

IntervalVector x5(x2);              // create a copy of x2
Vector x6(x4);                     // create a copy of x4
```

Finally, you can create an empty interval vector of a given size (see *The empty set*) or create a degenerated box from a vector of reals:

```
IntervalVector x1=IntervalVector::empty(3); // create a vector of 3 empty_
↪intervals

Vector x(3,0.1);
IntervalVector x2(x);                // create ([0.1,0.1],[0.1,0.1],[0.
↪1,0.1])
```

Matrices and Array of matrices

They work exactly the same way as vectors. Here are first listed different ways to build a Matrix:

```
// create [(1,2,3);(4,5,6)] in several steps.
Matrix m1(2,3); // a 2x3 matrix filled with zeros
m1[0][0]=1;     // we initialize each component
m1[0][1]=2;
m1[0][2]=3;
m1[1][0]=4;
m1[1][1]=5;
m1[1][2]=6;
output << "m1=" << m1 << endl << endl;

// create the same matrix in two steps only
double _m2[2*3]={1,2,3,
                 4,5,6};
```

(continues on next page)

(continued from previous page)

```

Matrix m2(2,3,_m2);
output << "m2=" << m2 << endl << endl;

// create a 2x3 matrix filled with ones
Matrix m3(2,3,1.0);
output << "m3=" << m3 << endl << endl;

// the same matrix using a built-in static function
Matrix m4=Matrix::ones(2,3);

// create a copy of m3
Matrix m5(m3);
output << "m5=" << m5 << endl;

```

The output is:

Now, for interval matrices, we shall take as example a matrix that frequently appears, the “perturbed” identity, that is:

$$\begin{pmatrix} [1-\varepsilon, 1+\varepsilon] & \dots & [-\varepsilon, \varepsilon] \\ \vdots & \ddots & \vdots \\ [-\varepsilon, \varepsilon] & \dots & [1-\varepsilon, 1+\varepsilon] \end{pmatrix}$$

```

double eps=1e-02;

// create in several steps.
IntervalMatrix m1(3,3); // a 3x3 matrix filled with (-oo,oo)
m1=Matrix::eye(3);      // set m1 to the identity matrix
m1+=Interval(-eps,eps)*Matrix::ones(3); // add [-eps,eps] to each component
output << "m1=" << m1 << endl << endl;

// create the same matrix with a matrix of double
// warning: the matrix of double has a number of rows
// equal to the total number of components (9) and
// 2 columns (left bound, right bound).
double _m2[3*3][2]={ {1-eps,1+eps}, {-eps,eps}, {-eps,eps},
                     {-eps,eps}, {1-eps,1+eps}, {-eps,eps},
                     {-eps,eps}, {-eps,eps}, {1-eps,1+eps}};

IntervalMatrix m2(3,3,_m2);
output << "m2=" << m2 << endl << endl;

// create a 3x3 matrix filled with [-eps,eps]
IntervalMatrix m3(3,3,Interval(-eps,eps));
m3+=Matrix::eye(3);
output << "m3=" << m3 << endl << endl;

// create a copy of m3
IntervalMatrix m4(m3);
output << "m4=" << m4 << endl;

```

The output is:

1.4.2 Container operations

This section presents operations on interval vectors and interval matrices viewed as containers.

Most of the operations are also possible with vectors (`Vector`) and matrices (`Matrix`) by replacing `Interval` with `double` and so on. They are not detailed again here, please refer to the API.

Table 1: Container operations for an interval vector `x`

Return type	C++ code	Meaning
int	<code>x.size()</code>	The size (number of components)
Interval&	<code>x[i]</code> (<i>int i</i>)	The <i>i</i> th component. By reference.
IntervalVec-tor	<code>x.subvector(i,j)</code> (<i>int i, int j</i>)	Return the subvector of length (<i>j-i+1</i>) starting at index <i>i</i> and ending at index <i>j</i> (both included). By copy. [<i>x</i>] must be non-empty .
void	<code>x.resize(n)</code> (<i>int n</i>)	Resize the vector. If the vector has been enlarged the extra components are set to $(-\infty, +\infty)$ even if the vector is empty.
void	<code>x.put(i, y)</code> (<i>int i, IntervalVec-tor& y</i>)	Write the vector [<i>y</i>] in [<i>x</i>] at index <i>i</i> .
void	<code>x.clear()</code>	Set all the elements to [0,0] (even if [<i>x</i>] is empty). Emptiness is “overridden”.
void	<code>x.init(y)</code> (<i>Interval& y</i>)	Set all the elements to [<i>y</i>] (even if [<i>x</i>] is empty). Emptiness is “overridden”.

Table 2: Container operations for an interval matrix *m*

Return type	C++ code	Meaning
int	<code>m.nb_rows()</code>	Number of rows
int	<code>m.nb_cols()</code>	Number of column
Interval&	<code>m[i][j]</code> (<i>int i, int j</i>)	The (i,j)th entry. By reference.
IntervalVector&	<code>m.row(i)</code> (<i>int i</i>)	The ith row. By reference (fast).
IntervalVector	<code>m.col(j)</code> (<i>int j</i>)	The jth column. By copy (slow, in $O(n)$)
IntervalMatrix	<code>m.rows(i1,i2)</code> (<i>int i1, int i2</i>)	The submatrix obtained by selecting rows between index i1 and i2. By copy.
IntervalMatrix	<code>m.cols(j1,j2)</code> (<i>int j1, int j2</i>)	The submatrix obtained by selecting columns between index j1 and j2. By copy.
IntervalMatrix	<code>m.submatrix(i1,i2,j1,j2)</code> (<i>int i1, int i2, int j1, int j2</i>)	The submatrix by selecting rows from index i1 to i2 (both included) and columns from index j1 to j2 (both included).
void	<code>m.resize(n1,n2)</code> (<i>int n1, int n2</i>)	Resize the matrix to a $n1 \times n2$ matrix. Extra components are set to $(-\infty, +\infty)$ even if the matrix is empty.
void	<code>m.set_row(i, y)</code> (<i>int i, IntervalVector& y</i>)	Write the vector [y] in the ith row of [m].
void	<code>m.set_col(i, y)</code> (<i>int i, IntervalVector& y</i>)	Write the vector [y] in the ith column of [m].
void	<code>m.put(i,j,m2)</code> (<i>int i, int j, IntervalMatrix& m2</i>)	Write the matrix [m2] in [m] at (i,j) and subsequent indices
void	<code>m.put(i,j,v,b)</code> (<i>int i, int j, IntervalVector& v, bool b</i>)	Write the vector at (i,j) either in row (if <code>b==true</code>) or in column (if <code>b==false</code>).
void	<code>m.clear()</code>	Set all the elements to [0,0] (even if [m] is empty). Emptiness is “overridden”.
void	<code>m.init(y)</code> (<i>Interval& y</i>)	Set all the elements to [y] (even if [m] is empty). Emptiness is “overridden”.

1.4.3 Set-membership operations

The operations described here are valid for any “sets” [x] and [y] and any “point” p, where a “set” designates here on object of either of the following classes:

- `Interval`
- `IntervalVector`
- `IntervalMatrix`
- `IntervalMatrixArray`.

Similarly, a “point” either refers to a `double` or an instance of:

- `Vector`
- `Matrix`
- `MatrixArray`.

Given equidimensional sets $[x]$, $[y]$ and a point p , the following table gives the set-membership predicates currently available in Ibex (from release 2.6.0) and their mathematical interpretation:

<i>C++ code</i>	<i>Meaning</i>	available from
<code>x==y</code>	$[x] = [y]$	
<code>x!=y</code>	$[x] \neq [y]$	
<code>x.is_empty()</code>	$[x] = \emptyset$	
<code>x.is_subset(y)</code>	$[x] \subseteq [y]$	
<code>x.is_strict_subset(y)</code>	$[x] \subseteq [y] \wedge x \neq [y]$	
<code>x.is_interior_subset(y)</code>	$[x] \subseteq [y]$	release 2.6.0
<code>x.is_strict_interior_subset(y)</code>	$[x] \subseteq [y] \wedge x \neq [y]$	release 2.6.0
<code>x.is_superset(y)</code>	$[x] \supseteq [y]$	
<code>x.is_strict_superset(y)</code>	$[x] \supseteq [y] \wedge [x] \neq [y]$	
<code>x.contains(p)</code>	$d \in [x]$	
<code>x.interior_contains(p)</code>	$d \in [x]$	release 2.6.0
<code>x.intersects(y)</code>	$[x] \cap [y] \neq \emptyset$	release 2.6.0
<code>x.overlaps(y)</code>	$[x] \cap [y] \neq \emptyset$	release 2.6.0
<code>x.is_disjoint(y)</code>	$[x] \cap [y] = \emptyset$	release 2.6.0

Possible set-membership operations between sets are:

<i>C++ code</i>	<i>Meaning</i>
<code>x&y</code>	$[x] \cap [y]$
<code>x y</code>	$\square([x] \cup [y])$
<code>x.set_empty()</code>	$[x] \leftarrow \emptyset$
<code>x=y</code>	$[x] \leftarrow [y]$
<code>x&=y</code>	$[x] \leftarrow ([x] \cap [y])$
<code>x =y</code>	$[x] \leftarrow \square([x] \cup [y])$

Finally, given two interval vectors $[x]$ and $[y]$ one can call:

```
cart_prod(x,y)
```

to create the Cartesian product.

Complementary and set difference

These set operations are only available for intervals (from release 2.6.0) and interval vectors.

In the case of intervals, the function `complementary` stores the complementary of `*this` in two intervals `c1` and `c2`. More exactly, the result is the closure of the complementary since open bounds are not representable (except for infinity and minus infinity). E.g, the “complementary” of $(-\infty, 0]$ is $[0, +\infty)$.

```
Interval x(0,1);
Interval c1,c2; // to store the result

int n=x.complementary(c1,c2);
output << "complementary of " << x << " = " << c1;
if (n>1) output << " and " << c2;
output << endl;
```

The output is:

In the case of interval vectors, the function `complementary` calculates the complementary under the form of a union of non-overlapping interval vectors, and stores this union into an array that is allocated by the function itself (the variable `result` in the code below). The function returns the size of the union/array.

To illustrate this, let us first build a function that prints the complementary of an interval vector:

```
void print_compl(const IntervalVector& x) {
    IntervalVector* result;
    int n=x.complementary(result);
    output << "complementary of " << x << " = " << endl;
    for (int i=0; i<n; i++) {
        output << "\t" << result[i] << endl;
    }

    delete[] result; // don't forget to free memory!
}
```

We can call it now with different vectors. Note that when the vector is the empty set with n components, the complementary is a n -dimensional box: $(-\infty, \infty) \times \dots \times (-\infty, \infty)$. Note also that if the difference is empty, `result` is an array of one element set to the empty box. It is *not* a zero-sized array containing no element (this is illegal in ISO C++). However the returned number is 0 (not 1). The interesting point is that you can call `delete[]` safely, in all cases.

```
print_compl(IntervalVector::empty(3));

print_compl(IntervalVector(3));

print_compl(IntervalVector(3, Interval(0,1)));
```

The output is:

The set difference works exactly the same way except that the function takes as first argument another set `[y]`.

Example with intervals:

```
// set difference between two intervals
Interval x(0,3);
Interval y(1,2);
Interval c1,c2; // to store the result

int n=x.diff(y,c1,c2);
output << x << " \ " << y << " = " << c1;
if (n>1) output << " and " << c2;
output << endl;
```

Example with interval vectors:

```
// set difference between two boxes
IntervalVector x(2, Interval(0,3));
IntervalVector y(2, Interval(1,2));
IntervalVector* result; // to store the result

int n=x.diff(y,result);
output << x << " \ " << y << " = " << endl;
for (int i=0; i<n; i++) {
    output << "\t" << result[i] << endl;
}
delete[] result; // don't forget to free memory!
```

1.4.4 Geometric operations

Here are the functions that can be applied to an **interval** `[x]`, seen as a segment of the line.

<i>Re- turn type</i>	<i>C++ code</i>	<i>Meaning</i>
double	<code>x.lb()</code>	\underline{x} , the lower (left) bound of <code>[x]</code>
double	<code>x.ub()</code>	\overline{x} , the upper (right) bound of <code>[x]</code>
double	<code>x.diam()</code>	diameter, $ \overline{x} - \underline{x} $
double	<code>x.rad()</code>	radius, half of the diameter
double	<code>x.mid()</code>	the midpoint, $((\underline{x} + \overline{x})/2)$
Interval	<code>x.inflate(eps)</code> (double eps)	an interval with the same midpoint and radius increased by <code>eps%</code>
double	<code>distance(x,y)</code> (Interval& y)	the (Hausdorff) distance between <code>[x]</code> and <code>[y]</code> . The distance is 0 iff <code>[x]==[y]</code> . Otherwise, it returns the minimal value by which one of the intervals (<code>[x]</code> or <code>[y]</code>) has to be inflated so that it entirely overlaps the other interval.
double	<code>x.rel_distance(y)</code> (Interval& y)	the “relative” distance, that is, <code>distance(x,y)/x.diam()</code>
bool	<code>x.is_unbounded()</code>	true iff <code>[x]</code> has one of its bounds infinite.
bool	<code>x.is_bisectable()</code>	true iff <code>[x]</code> , as an interval of floating point numbers, can be bisected in two non-degenerated intervals of floating point numbers. The empty interval or an interval of two consecutive floating points are not bisectable.

Here are the functions that can be applied to an **interval vector** `[x]`, seen as a box.

<i>Return type</i>	<i>C++ code</i>	<i>Meaning</i>
Vector	<code>x.lb()</code>	lower-left corner (vector of lower bounds of [x])
Vector	<code>x.ub()</code>	upper-right corner (vector of upper bounds of [x])
Vector	<code>x.diam()</code>	vector of diameters, $ \overline{x_i} - x_i $
double	<code>x.min_diam()</code>	minimal diameter, among all components of [x]
double	<code>x.max_diam()</code>	maximal diameter, among all components of [x]
int	<code>x.extr_diam_index(b)</code> (<i>bool b</i>)	the index of a component with minimal (if <code>b==true</code>) (if <code>b==false</code>) diameter.
void	<code>x.sort_indices(b, tab)</code> (<i>bool b, int tab[]</i>)	Write into <code>tab</code> the indices of all the components, sorted by increasing (if <code>b==true</code>) /decreasing (if <code>b==false</code>) diameters. The array <code>tab</code> must have been allocated before calling this function.
Vector	<code>x.rad()</code>	vector of radii (halves of diameters)
Vector	<code>x.mid()</code>	the midpoint, $((\underline{x} + \overline{x})/2)$
double	<code>x.volume()</code>	the volume of the box
double	<code>x.perimeter()</code>	the perimeter of the box
bool	<code>x.is_flat()</code>	true if the volume is null (one dimension is degenerated)
IntervalVector	<code>x.inflate(eps)</code> (<i>double eps</i>)	a box with the same midpoint and each radius increased by <code>eps%</code>
double	<code>distance(x,y)</code> (<i>IntervalVector& y</i>)	the (Hausdorff) distance between [x] and [y]. The distance is 0 iff <code>[x]==[y]</code> . Otherwise, it returns the minimal value by which one of the boxes ([x] or [y]) has to be inflated so that it entirely overlaps the other box.
double	<code>x.rel_distance(y)</code> (<i>IntervalVector& y</i>)	the “relative” distance, that is, <code>distance(x,y)/x.max_diam()</code>
bool	<code>x.is_unbounded()</code>	true iff [x] has one of its bounds infinite.
bool	<code>x.is_bisectable()</code>	true iff at least one component of [x] is bisectable (see above).

Bisection

Bisecting a box is a fundamental operation in Ibex. The choice of the component to be bisected is often critical and defining a strategy for choosing this component is the purpose of *Bisectors*.

To bisect the `i`th component, the `bisect` function can be used. The first argument is the index of the dimension `i` along which the box has to be splitted. The second argument is optional and gives at which point the `i`th interval is bisected. This argument is entered as a ratio of the interval diameter, 0.5 means “midpoint” (and the default value is precisely 0.5). Example:

```
IntervalVector x(3, Interval(0,1)); // [0,1]x[0,1]x[0,1]

std::pair<IntervalVector, IntervalVector> p = x.bisect(1,0.4); // bisect the
↪second component with ratio 0.4

output << "first box=" << p.first << endl;
output << "second box=" << p.second << endl;
```

The output is:

1.4.5 Miscellaneous

Given an interval $[x]$:

C++ code	Meaning
<code>x.mig()</code>	Mignitude, $\min_{x \in [x]} x $ (a double). Also exists for interval vectors and matrices (gives a vector or matrix of mignitudes).
<code>x.mag()</code>	Magnitude, $\max_{x \in [x]} x $ (a double). Also exists for interval vectors and matrices (gives a vector or matrix of magnitudes).
<code>integer(x)</code>	The largest interval $[a,b]$ of integers included in $[x]$.

Finally, for an interval vector $[v]$:

<code>v.random()</code>	A random point inside x
-------------------------	---------------------------

1.4.6 Interval arithmetic

Interval arithmetic is the main device upon which Ibex is built.

For a complete introduction of interval arithmetic and interval analysis, see *[Moore 1966]*, *[Neumaier 1990]* or *[Jaulin et al. 2001]*.

The interval arithmetic defines for each elementary function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ an interval function $[f] : \mathbb{IR}^n \rightarrow \mathbb{IR}$ according to the formula:

$$[f]([x]) \supseteq \square\{f(\alpha), \alpha \in [x]\}$$

where \square (the “hull” symbol) in front of a set means “the smallest box enclosing this set”. The hull symbol is here to highlight the fact that an enclosure of the possibly discontinuous set is returned. Note that, among all the functions below, only `tan` and `sign` fail to be continuous. So, in the other cases, the “hull” operator can be omitted. The possible roundoff error accounts for the \supseteq symbol.

The function $[f]$ is called an “interval extension” of the original function f (applied on real numbers).

We start by detailing the binary operators (+, -, *, /) and then proceed with the nonlinear ones.

Addition, subtraction, multiplication

The following tables summarize the basic operations of linear algebra in Ibex. It is important to notice that (so far?), these operations are **not** optimized, as compared to specialized softwares like *Intlab*. They are all based on naive implementations.

- Symbols x and y denote scalars (double or instances of `Interval`);
- Symbols v and w denote vectors (instances of `Vector` or `IntervalVector`)
- Symbols A and B denotes matrices (instances of `Matrix` or `IntervalMatrix`).

Notice that it is not possible (at least in the current release) to apply such operations with objects of type `IntervalMatrixArray`, the latter being considered as a pure “set” or, more pragmatically, as a matrix container. In other words, arrays of interval matrices are not a 3rd order tensor.

Possible additions and subtractions (with optionally assignments) are:

$-x$	$x+y$	$x-y$	$x+=y$	$x-=y$
$-v$	$v+w$	$v-w$	$v+=w$	$v-=w$
$-A$	$A+B$	$A-B$	$A+=B$	$A-=B$

Possible multiplications are:

<i>C++ code</i>	<i>Meaning</i>
$x*y$	multiplication of two “scalars” (double/intervals)
$x*=y$	multiplication and assignment
$x*v$	scalar multiplication of a vector
$x*A$	scalar multiplication of a matrix
$v*w$	dot product ($v^T w$)
<code>hadamard_product(v,w)</code>	$(v_1 w_1, \dots, v_n w_n)$
<code>outer_product(v,w)</code>	outer product ($v w^T$)
$A*x$	matrix-vector product
$A*B$	matrix product

Division

First, it is possible to use division between two interval/doubles using C++ operator overloading, as for the previous operators. You can either write x/y (division) or $x/=y$ (division and assignment).

However, as a set-membership operator, and because the division of reals is non-continuous, dividing two intervals may result in a union of two intervals. E.g.,

$$[2, 3]/[-1, 2] = \{x/y, x \in [2, 3], y \in [-1, 2]\} = (-\infty, -2] \cup [1, +\infty)$$

This variant of the interval division is called “generalized division” in the literature. The previous division operator / simply gives (in Ibex) the hull of this union. So, in C++, $[2,3] / [-1,2]$ gives $(-\infty, \infty)$. If you want to handle the possible outcome of two intervals, you have to call the “div2” function.

Given four intervals $[x], [y], [out1]$ and $[out2]$:

```
div2(x, y, out1, out2)
```

will store the union representing $[x]/[y]$ in $[out1]$ and $[out2]$. If the result is a single interval, $[out2]$ is set to the empty interval. If the result is the empty set, both $[out1]$ and $[out2]$ are set to the empty interval.

Note: Contrary to the “cset” theory, the result is empty if $y=[0,0]$ (whatever $[x]$ is).

For convenience, there is also a function in the `Interval` class that simultaneously performs division and intersection:

```
bool div2_inter(const Interval& x, const Interval& y, Interval& out2)
```

This function sets `*this` to the intersection of itself with the division of two others, `[x]` and `[y]`.

In return, `*this` and `[out2]` contains the lower and upper part respectively of the division. If the result of the generalized division and intersection is a single interval, `[out2]` is set to the empty interval.

The function returns true if the intersection is non empty.

Example:

```
Interval intv(-10,10);
Interval out2;
bool result=intv.div2_inter(Interval(2,3), Interval(-1,2), out2);
output << "the intersection is " << (result? "not":"" ) << " empty" << endl;
output << "left part=" << intv << " right part=" << out2 << endl;
```

Non-linear elementary functions

The following operations are allowed for an interval `[x]`.

Power and roots

C++ code	Meaning
<code>sqr(x)</code>	$[x]^2$
<code>sqrt(x)</code>	$\sqrt{[x]}$
<code>pow(x,n)</code>	$[x]^n$
<code>pow(x,y)</code>	$[x]^{[y]} = e^{[y] \log([x])}$
<code>root(x,n)</code>	$\sqrt[n]{[x]}$

Exponential, logarithm

<code>exp(x)</code>	<code>log(x)</code>
---------------------	---------------------

Trigonometric and hyperbolic functions

<code>cos(x)</code>	<code>sin(x)</code>	<code>tan(x)</code>
<code>acos(x)</code>	<code>asin(x)</code>	<code>atan(x)</code>
<code>cosh(x)</code>	<code>sinh(x)</code>	<code>tanh(x)</code>
<code>acosh(x)</code>	<code>asinh(x)</code>	<code>atanh(x)</code>
<code>atan2(y,x)</code>		

Max, min and miscellaneous

The sign function below is the interval extension of

$$\text{sign}(\alpha) := \begin{cases} -1 & \text{if } \alpha < 0 \\ 0 & \text{if } \alpha = 0 \\ 1 & \text{if } \alpha > 0 \end{cases}$$

<i>C++ code</i>	<i>Meaning</i>
<code>abs(x)</code>	interval extension of the absolute value. Not to be confused with magnitude; e.g. with $[x]=[-2,1]$, $\text{abs}([x])=[0,2]$ and the magnitude is 2.
<code>max(x,y)</code>	interval extension of the maximum. Not to be confused with $\max([x] \cup [y])$; e.g., $\max([0,3],[1,2])=[1,3]$.
<code>min(x,y)</code>	interval extension of the minimum. Not to be confused with $\min([x] \cup [y])$; e.g., $\min([0,3],[1,2])=[0,2]$.
<code>sign(x)</code>	(see above)

1.4.7 Backward arithmetic

The “backward” or “relationnal” arithmetic consists in contraction operators for elementary constraints like

$$y = x_1 + x_2.$$

More precisely, given a function $f : \mathbb{R} \rightarrow \mathbb{R}$ ($\sin, \cos, \exp \dots$) the backward operator of f is an interval function $[f]_{bwd} : \mathbb{IR}^2 \rightarrow \mathbb{IR}$ that satisfies:

$$[f]_{bwd}([x], [y]) \supseteq \{x \in [x] \mid \exists y \in [y], y = f(x)\}.$$

Similarly, given a binary function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ ($+, -, \times, \max, \dots$) the backward operator is a function $[f]_{bwd} : \mathbb{IR}^2 \rightarrow \mathbb{IR}$ that satisfies:

$$[f]_{bwd}([x]_1, [x]_2, [y]) \supseteq \{(x_1, x_2) \in [x]_1 \times [x]_2 \mid \exists y \in [y], y = f(x_1, x_2)\}.$$

In practice, backward operators in Ibex do not return an interval, they directly contracts the input interval $[x]$ (in case of unary functions), that is:

$$[x] \leftarrow [f]_{bwd}([x], [y])$$

or the two intervals $[x]_1, [x]_2$ (in case of binary functions):

$$([x]_1, [x]_2) \leftarrow [f]_{bwd}([x]_1, [x]_2, [y]).$$

So the `bwd_*` functions below all return `void`.

Note: One important feature in Ibex is the ability to contract with respect to any constraints and, in particular, with constraints under the form

$$y = f(x_1, \dots, x_n)$$

where f is an arbitrary *function*. So, as a user, there is probably little interest for you to call the low-level routines presented here. Moreover, these routines only contracts the “pre-image argument” $[x]$; they **don’t contract the “image argument”** $[y]$. This is in contrast with high-level contractors like *Forward-Backward* that do both contractions (on $[x]$ and $[y]$).

<i>C++ code</i>	<i>Relation</i>
<code>bwd_add(y,x1,x2)</code>	$y = x_1 + x_2$
<code>bwd_sub(y,x1,x2)</code>	$y = x_1 - x_2$
<code>bwd_mul(y,x1,x2)</code>	$y = x_1 \times x_2$

Continued on next page

Table 3 – continued from previous page

<i>C++ code</i>	<i>Relation</i>
<code>bwd_div(y,x1,x2)</code>	$y = x_1/x_2$
<code>bwd_sqr(y,x)</code>	$y = x^2$
<code>bwd_sqrt(y,x)</code>	$y = \sqrt{x}$
<code>bwd_pow(y,n,x)</code>	$y = x^n$
<code>bwd_pow(y,x1,x2)</code>	$y = x_1^{x_2}$
<code>bwd_root(y,n,x)</code>	$y = \sqrt[n]{x}$
<code>bwd_exp(y,x)</code>	$y = \exp x$
<code>bwd_log(y,x)</code>	$y = \ln x$
<code>bwd_cos(y,x)</code>	$y = \cos x$
<code>bwd_sin(y,x)</code>	$y = \sin x$
<code>bwd_tan(y,x)</code>	$y = \tan x$
<code>bwd_acos(y,x)</code>	$y = \arccos x$
<code>bwd_asin(y,x)</code>	$y = \arcsin x$
<code>bwd_atan(y,x)</code>	$y = \arctan x$
<code>bwd_cosh(y,x)</code>	$y = \cosh x$
<code>bwd_sinh(y,x)</code>	$y = \sinh x$
<code>bwd_tanh(y,x)</code>	$y = \tanh x$
<code>bwd_acosh(y,x)</code>	$y = \operatorname{arcosh}(x)$
<code>bwd_asinh(y,x)</code>	$y = \operatorname{arsinh}(x)$
<code>bwd_atanh(y,x)</code>	$y = \operatorname{artanh}(x)$
<code>bwd_atan2(y,x1,x2)</code>	$y = \operatorname{atan2}(x_1, x_2)$
<code>bwd_abs(y,x)</code>	$y = x $
<code>bwd_sign(y,x)</code>	$y = \operatorname{sign}(x)$ (see def. above)
<code>bwd_integer(y,x)</code>	$y = x \wedge x \in \mathbb{N}$
<code>bwd_min(y,x1,x2)</code>	$y = \min(x_1, x_2)$

Continued on next page

Table 3 – continued from previous page

<i>C++ code</i>	<i>Relation</i>
<code>bwd_max(y,x1,x2)</code>	$y = \max(x_1, x_2)$

1.4.8 Inner arithmetic

The inner arithmetic [Chabert & Beldiceanu 2010] [Araya et al 2014] consists in two types of operators. Given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$:

- the **forward inner operator** is an interval function $]f[: \mathbb{IR}^n \rightarrow \mathbb{IR}$ that satisfies the formula:

$$]f[([x]) \subseteq \{f(\alpha), \alpha \in [x]\}$$

- the **backward inner operator** is an interval function $]f[_{bwd} : \mathbb{IR}^{n+1} \rightarrow \mathbb{IR}$ that satisfies the formula:

$$]f[_{bwd}([x], [y]) \subseteq \{x \in [x] \mid \exists y \in [y], y = f(x)\}.$$

Notice that the inclusion symbol has just been reversed, as compared to the classical interval (forward) and backward arithmetic.

Forward operators

Note: The inner operators are not implemented for all elementary functions yet.

<i>C++ code</i>	<i>Function</i>
<code>iadd(x1,x2)</code>	$x1+x2$
<code>isub(x1,x2)</code>	$x1-x2$
<code>imul(x1,x2)</code>	$x1*x2$
<code>idiv(x1,x2)</code>	$x1/x2$
<code>isqr(x)</code>	x^2
<code>iminus(x)</code>	$-x$
<code>ilog(x)</code>	$\ln(x)$
<code>iexp(x)</code>	$\exp(x)$
<code>iacos(x)</code>	$\cos(x)$
<code>iasin(x)</code>	$\sin(x)$
<code>iatan(x)</code>	$\tan(x)$

Backward operators

Let us denote

$$\mathcal{S} := \{x \in [x] \mid \exists y \in [y], y = f(x)\}.$$

The backward operator also takes an optional argument `xin` that represents an “inner” box, that is a box which already satisfies

$$[x]_{in} \subseteq \mathcal{S}.$$

If this argument is set, the operator guarantees that the contracted box $[x]$ will enclose $[xin]$. The operator works in this case as an *inflator*, the inner box $[xin]$ is indeed “inflated” to a larger inner box $[x]$. More precisely, we have:

$$[x]_{in} \subseteq]f[bwd([x], [y], [x]_{in}) \subseteq ([x] \cap \mathcal{S}).$$

Note: The inner operators are not implemented for all elementary functions yet.

C++ code	Relation
ibwd_add(y,x1,x2,xin1,xin2)	$y=x1+x2$
ibwd_sub(y,x1,x2,xin1,xin2)	$y=x1-x2$
ibwd_mul(y,x1,x2,xin1,xin2)	$y=x1*x2$
ibwd_div(y,x1,x2,xin1,xin2)	$y=x1/x2$
ibwd_sqr(y,x,xin)	$y=x^2$
ibwd_sqrt(y,x,xin)	$y = \sqrt{x}$
ibwd_minus(y,x,xin)	$y=-x$
ibwd_abs(y,x,xin)	$y = x $
ibwd_pow(y,x,p,xin)	$y=x^p$
ibwd_log(y,x,xin)	$y=\ln(x)$
ibwd_exp(y,x,xin)	$y=\exp(x)$
ibwd_cos(y,x,xin)	$y=\cos(x)$
ibwd_sin(y,x,xin)	$y=\sin(x)$
ibwd_tan(y,x,xin)	$y=\tan(x)$

1.5 Functions

1.5.1 Introduction

The purpose of this chapter is to show how to create and manipulate objects corresponding to the mathematical concept of *function*.

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.

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

Arguments versus variables

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

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

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.

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 $x \mapsto x + 1$. Functions play, in contrast, a big role.

In other words, x is nothing but a syntactic 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.

Dimensions and ordering

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

Let us consider again this function:

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

From the user standpoint, the function f (once built) is “flattened” (or “serialized”) 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 $[b] = ([b]_1, \dots, [b]_5)$ 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 move now to [function operations](#).

Class name and fields

As we have just said, arguments are just symbols in expression. For this reason, they are represented by a class 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 programmer’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 “argument”. Anyway, the programmer 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++*.

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`.

1.5.2 Interval Computations

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

Evaluation (forward computation)

Take a look first at the [tutorial](#) for introductory examples.

Since function overloading does not work for return types in C++, you have to either call `eval`, `eval_vector` or `eval_matrix` depending if your function respectively returns a scalar, a vector or a matrix.

All `eval_XXX` functions expects a single box in argument that represents all the arguments (scalars, vectors, matrices) stored in a single flat array (see [Dimensions and ordering](#)).

To build this vector, the best is to use *backward projection functions*.

Here is an example with $f(A,B,C)=A+B-C$ where A , B and C are matrices.

```
const int nb_rows=2;
const int nb_cols=2;

Variable a(nb_rows,nb_cols),b(nb_rows,nb_cols),c(nb_rows,nb_cols);
```

(continues on next page)

(continued from previous page)

```

Function pA(a,b,c,a);
Function pB(a,b,c,b);
Function pC(a,b,c,c);

Function f(a,b,c, (a+b-c));

double _A[nb_rows*nb_cols][2]={ {2,2},{-1,-1},{-1,-1},{2,2}};
IntervalMatrix MA(2,2,_A);

double _B[nb_rows*nb_cols][2]={ {1,1},{-1,-1},{1,1},{1,1}};
IntervalMatrix MB(2,2,_B);

double _C[nb_rows*nb_cols][2]={ {1,1},{0,0},{0,0},{1,1}};
IntervalMatrix MC(2,2,_C);

IntervalVector box(3*nb_rows*nb_cols);

// the backward call on pA will force the sub-vector of
// "box" that represents the domain of "a" to contain the
// interval matrix "MA".
pA.backward(MA,box);

// idem
pB.backward(MB,box);
pC.backward(MC,box);

IntervalMatrix M = f.eval_matrix(box);

output << "A+B-C=" << M << endl;

```

Backward

One of the main feature of Ibex is the ability to *contract* a box representing the domain of a variable x with respect to the constraint that $f(x)$ belongs to a restricted input range $[y]$. The range $[y]$ can be any constant (real value, interval, interval vector, etc.). Rigorously, given two intervals $[x]$ and $[y]$, the contraction gives a new interval $[z]$ such that

$$\forall x \in [x], \quad f(x) \in [y] \implies x \in [z] \subseteq [x].$$

One way to do this is by using the famous *forward-backward* (alias HC4Revise). 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. This algorithm does not return a new interval $[z]$ but contract the input interval $[x]$ which is therefore an input-output argument.

In the following snippet we require the function $\sin(x+y)$ to take the value -1 (a degenerated interval). With an initial box $(x,y)=[1,2],[3,4]$, we obtain the result that (x,y) must lie in the subdomain $([1, 1.7123] ; [3, 3.7124])$.

```

Variable x;
Variable y;
Function f(x,y,sin(x+y));

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

/* the backward sets box to ([1, 1.7123] ; [3, 3.7124]) */
f.backward(-1.0,box);

```

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

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;
```

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, thinner 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, \dots, x_n) denotes the midvector of $[x]$.

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

Here is an example:

```
Variable x,y;  
Function f(x,y,Return(sqr(x)*y,sqr(y)*x));  
IntervalMatrix H(2,2);  
IntervalVector box(2,Interval(1,2));  
f.hansen_matrix(box,H);  
output << "Hansen matrix:\n" << H << endl;
```

The display is:

1.5.3 Creating functions (in C++)

Function objects are very easy to build.

This section explains how to build them using C++ operator overloading but using the *Minibex* syntax is even simpler.

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.

Creating vector and matrix arguments (in C++)

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
```

Examples

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)); // create the function (x,y)->sin(x+y)
output << f << endl;
```

The display is:

You can directly give up to 20 variables in argument of the Function constructor:

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

If more than 20 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

$$x \mapsto x_1 + \dots + x_7$$

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

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

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]);
output << f << endl;
```

The display is:

Note: Because of a potential conflict with `std::min` (or `std::max`), you might be forced to prefix the min (max) function with `ibex::`:

```
Variable x,y;
Function f(x,y,ibex::min(x,y));
output << f << endl;
```

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");
```

Vector-valued functions

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

See the example in the [tutorial](#).

Advanced examples

Building DAGs (directed acyclic graphs)

C++ operator overloading allows you to create a DAG instead of an expression tree. This will result in a gain in performance. For that, you need to handle references of shared subexpressions with variables types `const ExprNode&`.

In the following example we create the function :

$$f : x \mapsto ((\cos(x) + 1)^2, (\cos(x) + 1)^3)$$

and we want the subexpression `cos(x)+1` to be shared:

```
Variable x;
const ExprNode& e=cos(x)+1;
Function f(x,Return(pow(e,2),pow(e,3)));
```

Iterated sum

Let us build a function that returns the sum of the square of `N` variables, where `N` is some constant.

The only difficulty is that we cannot assign references in C++, so we need to use pointers to (`const ExprNode&`) instead:

```
int N=10;
Variable x(N,"x");

const ExprNode* e=&(sqr(x[0]));
for (int i=1; i<N; i++)
    e = &(*e + sqr(x[i]));

Function f(x,*e,"f");

output << f << endl;
```

The display is:

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 (x_a, y_a) and (x_b, y_b) :

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

Serialization

(to do)

1.5.4 Operations on Functions

Composition

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

See the example in the [tutorial](#).

Applying a function with numerous arguments

We have explained how to create a function with *an arbitrary number of arguments*. We explain now how to call (perform composition) with such function.

It is as simple as storing all the actual arguments in an array structure, namely, a structure of expression nodes (typed `Array<const ExprNode>`).

However, when an actual argument is not a formal expression but a numerical constant (data), it is necessary to explicitly encapsulate this constant in a expression node. This is what the `ExprConstant` class stands for.

Here is an example. We create the function $f : (x,y) \mapsto x + y$ and apply it to the hybrid couple $(z,1)$ where z is another variable. We do it in the generic way, using arrays:

```
Variable x,y;

// formal arguments
Array<const ExprSymbol> vars(2);

vars.set_ref(0,x);
vars.set_ref(1,y);
```

(continues on next page)

(continued from previous page)

```

Function f(vars, x+y, "f");

// actual arguments
const ExprSymbol& z=ExprSymbol::new_("z"); // <=> "Variable z" (but more
→ "safe")
const ExprConstant& c=ExprConstant::new_scalar(1.0);

// =====
// before release 2.1.6:
//   const ExprNode* args[2];
//   args[0]=&z;
//   args[1]=&c;
// before release 2.1.6:

// from release 2.1.6 and subsequents:
Array<const ExprNode> args(2);
args.set_ref(0,z);
args.set_ref(1,c);
// =====

Function g(z,f(args), "g");

output << g << endl;

```

The display is:

You can also use this construct with vector/matrix variables and mix functions declared with different style:

```

// Here is a function that performs a matrix-vector multiplication,
// declared in the usual way (with a list of arguments):
Variable R(3,3);
Variable x(3);
Function f(R,x,R*x);

// Here is a function declared in the generic way (with an array of symbols)
// that applies the function 'f' to the array:
Array<const ExprSymbol> vars(2);
vars.set_ref(0,ExprSymbol::new_("R",Dim::matrix(3,3))); // note: giving "R"
→ is optional
vars.set_ref(1,ExprSymbol::new_("x",Dim::col_vec(3)));

Function g(vars,f(vars[0],vars[1]), "g");

```

Symbolic differentiation

Differentiation of a function is another function. So symbolic differentiation is obtained via a copy constructor where the copy “mode” is set to the special value `Function::DIFF`:

```

Function f("x","y","z","x*y*z");
Function df(f,Function::DIFF);
output << "df=" << df << endl;

```

The output is

1.6 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 contradictory 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 programmer, you may actually face two different situations.

Either you indeed want to use a constraint as a contractor in which case you build a `Ctc` object with this constraint (the actual class depending on the algorithm you chose, as explained in [the tutorial](#) –by default, it is *Forward-Backward*). 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.

1.6.1 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:

Op	def
LT	<
LEQ	≤
EQ	=
GEQ	≥
GT	>

1.6.2 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, like [here](#).

Note: There is currently the important restriction that inequalities can only be formed with real-valued (called “scalar”) functions. It could be possible, in theory, to write $f(x) \leq 0$ with f vector-valued by interpreting the operator componentwise but this is not supporter by Ibex.

1.7 Systems

A *system* in IBEX is a set of *constraints* (equalities or inequalities) with, optionnaly, a goal function to minimize and an initial domain for variables. It corresponds to the usual concept of system in mathematical programming. Here is an example of system:

$$\begin{aligned} &\text{Minimize } x + y, \\ &x \in [-1, 1], y \in [-1, 1] \\ &\text{such that} \\ &\quad x^2 + y^2 \leq 1 \\ &\quad y \geq x^2. \end{aligned}$$

One is usually interested in solving the system while minimizing the criterion, if any.

1.7.1 Class and Fields

The class for representing a system is `System`.

Systems fields

A system is not as simple as a collection of *any* constraints because 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).
- `IntervalVector box`: when a system is *loaded from a file*, 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.

Auxiliary functions

(to be completed)

1.7.2 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:

- declare a new system factory (an object of `SystemFactory`)
- add arguments in the factory using `add_var`.
- (optional) add the expression of the goal function using `add_goal`
- add the constraints using `add_ctr`
- 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 [here](#), 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.

1.7.3 System Transformation

We present in this section the different transformations that can be applied to a system.

Copy

The first transformation you can apply on a system is a simple copy. Of course, this is done via the copy constructor of the `System` class.

When calling the copy constructor, you can decide to copy everything, only the equations or only the inequalities. For this, set the second parameter of the constructor to either:

value	def
COPY	duplicate all the constraints
INEQ_ONLY	duplicate only inequalities
EQ_ONLY	duplicate only equalities

The first argument of the constructor is the system to copy of course.

```
output << "original system:" << endl;
output << "-----" << endl;
output << sys;
output << "-----" << endl << endl;

System sys2(sys, System::INEQ_ONLY);

output << "system with only inequalities" << endl;
output << "-----" << endl;
output << sys2;
output << "-----" << endl << endl;
```

The display is:

Normalization

It is comfortable in some situations to assume that a system is made of inequalities only, and that each inequality is under the forme $g(x) \leq 0$, that is, it is a “less or equal” inequality. This is called a “normalized” system.

This need arises, e.g., in optimization methods where the calculation of Lagrange multipliers is simplified when the normalization assumption holds.

It is possible to automatically transform a system into a normalized one. The process is immediate. If a constraint is:

- $g(x) \leq 0$ it is already normalized so it is left unchanged.
- $g(x) < 0$ it is replaced by $g(x) \leq 0$ (yes, there is a little loss of precision here)
- $g(x) > 0$ or $g(x) \geq 0$ it is replaced by $-g(x) \leq 0$
- $g(x) = 0$ it is replaced by two constraints: $g(x) \leq 0$ and $-g(x) \leq 0$. It is also possible to introduce an inflation value or “thickness”, that is, to replace the equality by $g(x) \leq \varepsilon$ and $-g(x) \leq \varepsilon$ where ε can be fixed to any value.

Note: There is a special treatment for “thick equalities”, that is, equations of the form $g(x) = [l, u]$. This kind of equations appear often in, e.g., robust parameter estimation problems. In this case, the equality is replaced by two inequalities, $g(x) \leq u$ and $-g(x) \leq -l$, and the ε -inflation is not applied unless $|u - l| < \varepsilon$.

Normalization is done by calling the constructor of `NormalizedSystem`, a sub-class of `System`. Here is an example where `sys` is a system built previously:

```
output << "original system:" << endl;
output << "-----" << endl;
output << sys;
output << "-----" << endl << endl;

// normalize the system with a "thickness"
// set to 0.1
NormalizedSystem norm_sys(sys, 0.1);

output << "normalized system:" << endl;
output << "-----" << endl;
output << norm_sys;
output << "-----" << endl;
```

We get the following display:

Extended System

An extended system is a system where the goal function is transformed into a constraint.

For instance, the extension of the system given above:

Minimize $x + y$,
 $x \in [-1, 1], y \in [-1, 1]$
 such that
 $x^2 + y^2 \leq 1$
 $y \geq x^2$.

is the following unconstrained system of constraints:

$x \in [-1, 1], y \in [-1, 1],_{goal \in (-\infty, \infty)}$ such that $x + y =_{goal} x^2 + y^2 \leq 1 y \geq x^2$

Once built, an extended system is a system like any other one, but it has also some extra information:

- the name of the goal variable which is automatically generated (it is “__goal__” in our previous example).
- the index of the goal variable (the last (2) in our previous example)
- the index of the “goal constraint” (the first (0) in our previous example)

For this reason, an extended system is represented by a subclass of `System` named `ExtendedSystem`.

To create an extended system just use the constructor of `ExtendedSystem`. We assume in the following example that the variable `sys` is a `System` previously built.

```
output << "original system:" << endl;
output << "-----" << endl;
output << sys;
output << "-----" << endl;
output << "  number of variables:" << sys.nb_var << endl;
output << "  number of constraints:" << sys.nb_ctr << endl << endl;
```

(continues on next page)

(continued from previous page)

```

ExtendedSystem ext_sys(sys);

output << "extended system:" << endl;
output << "-----" << endl;
output << ext_sys;
output << "-----" << endl;
output << "  number of variables:" << ext_sys.nb_var << endl;
output << "  number of constraints:" << ext_sys.nb_ctr << endl;
output << "  goal name:" << ext_sys.goal_name() << endl;
output << "  goal variable:" << ext_sys.goal_var() << endl;
output << "  goal constraint:" << ext_sys.goal_ctr() << endl;

```

We get the following display:

Fritz-John (Khun-Tucker) conditions

The generalized Khun-Tucker (aka Fritz-John) conditions can be obtained from a system. This produces a new system of $n+M+R+K+1$ variables where

- n is the number of basic variables (the ones of the original system)
- M is the number of Lagrange multipliers for inequalities (i.e., the number of inequalities in the original system)
- R is the number of Lagrange multipliers for equalities (i.e., the number of equalities in the original system)
- K is the number of Lagrange multipliers for bounding constraints. These bounding constraints correspond to the `box` field of the original system which is taken into account as $2n$ additional inequalities.
- The last variable is the “special coefficient” of the goal function that is equal to 0 in the case where constraint qualification (linear independency of constraints gradients) does not hold.

Generation of the Fritz-John conditions is based on *Applying a function with numerous arguments*.

Example:

```

output << "original system:" << endl;
output << "-----" << endl;
output << sys;
output << "-----" << endl;

FritzJohnCond fj(sys);

output << "Fritz-John system:" << endl;
output << "-----" << endl;
output << fj << endl;
output << "-----" << endl;
output << "  number of variables:" << fj.nb_var << endl;

```

We get the following display. The variable `_u` is the coefficient of the goal function. The variable `_l` is the multiplier of the constraint.

1.8 The Minibex Language

1.8.1 Introduction

There are three possible alternatives for modeling.

- First, you can write C++ code. Variables, functions, constraints and systems are C++ objects that you declare yourself and build by calling the constructors of the corresponding classes
- Entering mathematic formulas 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 *function*, a *constraint* or a *system* from a (plain text) input file, following the (very intuitive) Minibex syntax.
- However, files I/O operations are not always welcome. The third possibility is a kind of compromise. You can initialize a `Function` or `NumConstraint` objects with a string (`char*`) that contains the Minibex code. The syntax is exactly the same (see examples in the tutorial).

In all 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, whether it be created in your C++ program or loaded from a Minibex file.

Here are simple examples where the syntax talks for itself.

1.8.2 Examples

Function

Copy-paste the text below in a file named, say, `function.txt`:

```
function f(x)
  return x+y;
end
```

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

```
Function f("function.txt");
```

and the function you get is $(x,y) \rightarrow x+y$.

Constraint

Copy-paste the text below in a file named, say, `constraint.txt`:

```
Variables
  x, y;

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

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

```
NumConstraint ctr("constraint.txt");
```

and the constraint you get is $x^2+y^2\leq 1$. Notice that the keyword `constraints` has an “s” at the end because the Minibex syntax allows several constraints declaration. If you load a constraint from a Minibex file that contains several constraints, only the first one is considered.

System

Copy-paste the text below in a file named, say, `system.txt`:

```
Variables
  x in [-1,1];
  y in [-1,1];

Minimize
  x+y;

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

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

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

and the system you get is:

Minimize $x + y$,
 $x \in [-1, 1], y \in [-1, 1]$
such that
$$x^2 + y^2 \leq 1$$
$$y \geq x^2.$$

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

1.8.3 Overall structure

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

- constants
- variables
- auxiliary functions
- goal function
- constraints

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

1.8.4 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 ∞ (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]
```

There is a predefined interval constant `pi` representing π .

1.8.5 Constants

Constants are all defined in the same declaration block, started with the `Constants` keyword. This block is always optional.

A constant value can depends on other (previously defined) constants value. Example:

```
Constants
e=0.5772156649;
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 `e` a valid enclosure as below. We just have to replace “=” by “in”:

```
Constants
e in [0.577215664, 0.577215665];
y=e+1;
```

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 2x1 matrix. A row vector is a 1x2 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 not 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 can mix vector with intervals. For instance: $([-\infty, 0]; [0, +\infty])$ 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 10x10 matrix where all components are $[0, 0]$:

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

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

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

$x \text{ in } [-\infty, 0]$	declares a constant $x \in (-\infty, 0]$
$x \text{ in } [0, 1]$	declares an constant $x \in [0, 1]$
$x \text{ in } [0, 0]$	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 $100 * \sin(0.1)$
$x[10] \text{ in } [-\infty, 0]$	declares a 10-sized constant vector x , with each component $x_i \in (-\infty, 0]$
$x[2] \text{ in } ([-\infty, 0]; [0, +\infty])$	declares a 2-sized constant vector x with $x_1 \in (-\infty, 0]$ and $x_2 \in [0, +\infty)$
$\mathbf{x}[3][3] \text{ in } (([0, 1], 0, 0); (0, [0, 1], 0); (0, 0, [0, 1]))$	declares a constant matrix $x \in \begin{pmatrix} [0, 1] & 0 & 0 \\ 0 & [0, 1] & 0 \\ 0 & 0 & [0, 1] \end{pmatrix}.$
$x[10][5] \text{ in } [0, 1]$	declares a matrix x with each entry $x_{ij} \in [0, 1]$.
$x[2][10][5] \text{ in } [0, 1]$	declares an array of two 10x5 matrices with each entry $x_{ijk} \in [0, 1]$.

1.8.6 Variables

Variables need to be declared in two situations:

- when you create a function. In this case, the variables are the arguments of the function and they are only visible inside the body of the function. Let us call them *local variables*. Here is an example:

```
function f(x) // x is a local variable
    ...
end
```

- when you create a constraint or a system of constraints. In this case, the variables are declared globally in a specific block and shared by all the constraints. Let us call them *global variables*:

```
variables
    x;           // x is a global variable
    ...
```

Note that global variables are not visible inside the (auxiliary) functions and conversely. So there is no possible confusion between the global and the local variables.

Local and global variables can be vectors and matrices. Declaring vector and matrix variables follow exactly the same rules as for vector and matrix *constants*. Example:

```
function f1(x[3]) // x is a vector of 3 components
    ...
end
```

It is possible to define up to three dimensional vectors.

Global variables can also be given a domain to initialize each component with. The following examples are valid:

```
variables

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

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

1.8.7 Expressions

The expressions are built by applying operators on constants and variables.

In the following, we assume that:

- $e, e1, e2, \dots$ are expressions
- `real-cst` is a constant expression (not involving variables)
- `int-cst` is a constant integer expression
- `func` is the name of an auxiliary function (see below)

You can use parenthesis and any space characters inside the expression, including new line.

Operators for real-valued expressions are:

-e	opposite
e1+e2	sum
e1-e2	subtraction
e1*e2	multiplication
e1/e2	division
e1^e2	power
e^int-cst	power (note: faster than previous op.)
max(e1,e2,...)	max
min(e1,e2,...)	min
atan2(e1,e2)	atan2
sign(e)	sign of e
abs(e)	absolute value
exp(e)	exponential
ln(e)	neperian logarithm
sqrt(e)	square root
cos(e)	cosine
sin(e)	sine
tan(e)	tangent
acos(e)	inverse cosine
asin(e)	inverse sine
atan(e)	inverse tangent
cosh(e)	hyperbolic cosine
sinh(e)	hyperbolic sine
tanh(e)	hyperbolic tangent
acosh(e)	inverse hyperbolic cosine
asinh(e)	inverse hyperbolic sine
atanh(e)	inverse hyperbolic tangent
func(e1,e2,...)	apply the function “func” to the arguments
(e1,e2,...)	create a row vector of expressions
(e1;e2;...)	create a column vector of expressions

Operators for vector/matrix-valued expressions are:

e'	transposition (like in Matlab)
-e	opposite
e1+e2	sum
e1-e2	subtraction
e1*e2	matrix-vector multiplication or dot/Hadamard product
e(int-cst)	get the ith component of a vector or the ith row of a matrix
e(int-cst,int-cst)	get the (i,j)th entry of a matrix expression
(e1,e2,...)	create a matrix from column vectors
(e1;e2;...)	create a matrix from row vectors

So, indexing vector or matrix variables follow Matlab convention and, remember, indices start from 1.

Ex:

```
Variables
  x[10][10] in [0,∞];
Constraints
  x(1,1)=0;
end
```


Some differences with C++

- Vectors indices are surrounded by parenthesis (not brackets),
- Indices start by 1 instead of 0,
- You have to use the “^” symbol (instead of `sqr` or `pow`).

1.8.8 Functions

A function declared in a Minibex file may have two different usage.

- You need to handle this function in your C++ program. In this case, your Minibex file should only contain that function. The file can then be loaded with the appropriate *constructor* of the Function class.
- You have several constraints that involve the same expression repeatedly. Then, it may be convenient for you to put this expression once for all in a separate function and to invoke this function inside the constraints expressions. We shall talk in this case about *auxiliary functions*.

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, \dots, y_b are various sub-expressions, like in:

```
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 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 that you can access from your C++ program via the *System* class. See *auxiliary functions*.

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 readability 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 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` amounts 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++.

1.8.9 Constraints

Constraints are simply written in sequence. The sequence starts with the keyword `constraints` and terminates with the keyword `end`. They are separated by semi-colon. Here is an example:

```
Variables
  x in [0,∞];
Constraints
  //you can use C++ comments
  x+y>=-1;
  x-y<=2;
end
```

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
```

1.9 Contractors

1.9.1 Introduction

The concept of *contractor* is directly inspired from the ubiquitous concept of *filtering algorithm* in constraint programming.

Given a constraint c relating a set of variables x , an algorithm C is called a filtering algorithm if, given a box (and more generally a “domain” for x , i.e., a set of potential tuples variable can be assigned to):

$$C([x]) \subseteq [x] \wedge \forall x \in [x], c(x) \Rightarrow x \in C([x]).$$

This relation means that:

1. Filtering gives a sub-domain of the input domain $[x]$;
2. The resulting subdomain $C([x])$ contains all the feasible points with respect to the constraint c . No solution is “lost”.

This illustrated in the next picture. The constraint c (i.e., the set of points satisfying c) is represented by a green shape.



Constraint programming is the context where this concept has been formalized and intensively used. But constraint programming is by no means the only paradigm where algorithms complying with this definition have been developed. The most significant example is interval analysis, where a Newton-like iteration has been developed since the 1960's [Moore 1966] that corresponds exactly to a filtering algorithm in the case of a constraint c of the following form:

$$c(x) \iff f(x) = 0$$

where f is any (non-linear) differentiable function from $\mathbb{R}^n \rightarrow \mathbb{R}^n$. The most famous variant of this Newton iteration is probably the Hansen-Sengupta algorithm [Hansen & Sengupta 1980].

Another important example is in picture processing, where there exist algorithms able to reduce the size of a picture to some *region of interest*. This kind of algorithm is today implemented in almost every digital cameras, for an automatic focus adjustment.

In this example, the constraint c is:

images/ladybug.png

$c(x) \iff x \text{ belongs to a region of interest}$

Here, it is clear that the constraint is more a consequence of the algorithm than in the other way around. This last example suggests the next definition.

An algorithm C is called a *contractor* if:

$$\begin{aligned} \forall [x] \in \mathbb{IR}^n, \mathcal{C}([x]) &\subseteq [x] && \text{(contraction)} \\ (x \in [x], \mathcal{C}(\{x\}) = \{x\}) &\Rightarrow x \in \mathcal{C}([x]) && \text{(consistency)} \\ \mathcal{C}(\{x\}) = \emptyset &\Leftrightarrow (\exists \varepsilon > 0, \forall [x] \subseteq B(x, \varepsilon), \mathcal{C}([x]) = \emptyset) && \text{(continuity)} \end{aligned}$$

- The first condition is the same as before.
- The second condition, the one related to the underlying constraint, has been dropped. In fact, the constraint c has been replaced by the set of *insensitive* points, those satisfying $\mathcal{C}(\{x\}) = \{x\}$. So the constraint still exists, but implicitly. With this in mind, it is clear now the second condition here states again that “no solution is lost”.
- The last condition is important for some convergence issues only.

Withdrawing the link to the constraint from the definition forces one to view the contractor as a pure function:

$$C : \mathbb{IR}^n \rightarrow \mathbb{IR}^n,$$

where \mathbb{IR} denotes the set of real intervals. “Pure” means that the execution of the contractor does not depend on a context and does not produce side effects. In the former definition of *filtering algorithm*, the operator was depending on constraints and, in practice, constraints are external objects sharing some structures representing domains of variables. This means that the execution was depending on the context and producing side effects. This centralized architecture is often met in discrete constraint programming. It allows the implementation of many code optimization techniques, but at a price of a huge programming complexity.

In contrast, *contractor programming* is so simple that anyone can build a solver in a few lines. Here is the interface for contractors. As one can see, it could not be more minimalist:

```

class Ctc {
public:

    // Build a contractor on nb_var variables.
    Ctc(int nb_var);

    // Performs contraction.
    // This is the only function that must be implemented in a subclass of Ctc.
    // The box in argument is contracted in-place (in-out argument).
    virtual void contract(IntervalVector& box)=0;
};

```

That’s all. Another advantage of removing the constraint from the definition is that it makes natural the cooperation of heterogenous contractors (would they be linked internally to a numerical constraint, a picture processing algorithm, a combinatorial problem, etc.).

The good news is that some important constraint programming techniques like *propagation*, *shaving* or *constructive disjunction* can actually be generalized to contractors. They don’t intrinsically need the concept of constraint.

These operators all take a set of contractors as input and produce a new (more sophisticated) contractor. The design of a solver simply amounts to the composition of such operators. All these operators form a little functional language, where contractors are first-class citizens. This is what is called contractor programming [Chabert & Jaulin 2009].

We present in this chapter the basic or “numerical” contractors (built from a constraint, etc.) and the operators.

1.9.2 Forward-Backward

Forward- backward (also known as HC4Revise) is a classical algorithm in constraint programming for contracting quickly with respect to an equality or inequality. See, e.g., [Benhamou & Granvilliers 2006], [Benhamou et al. 1999], [Collavizza 1998]. However, the more occurrences of variables in the expression of the (in)equality, the less accurate the contraction. Hence, this contractor is often used as an “atomic” contractor embedded in an higher-level operator like *Propagation* or *Shaving*.

The algorithm works in two steps. The **forward step** apply *Interval arithmetic* to each operator of the *function* expression, from the leaves of the expression (variable domains) upto the root node.

This is illustrated in the next picture with the constraint $(x - y)^2 - z = 0$ with $x \in [0, 10]$, $y \in [0, 4]$ and $z \in [9, 16]$:

The **backward step** sets the interval associated to the root node to $[0,0]$ (imposes constraint satisfaction) and, then, apply *Backward arithmetic* from the root downto the leaves:

This contractor can either be built with a *NumConstraint* object or directly with a function f . In the latter case, the constraint $f=0$ is implicitly considered.

See **examples in the tutorial**.

1.9.3 Intersection, Union, etc.

Basic operators on contractors are :

Class name	arity	Definition
CtcIdentity	0	$[x] \mapsto [x]$
CtcCompo	n	$[x] \mapsto (C_1 \circ \dots \circ C_n)([x])$
CtcUnion	n	$[x] \mapsto \Box C_1([x]) \cup \dots \cup C_n([x])$
CtcFixpoint	1	$[x] \mapsto C^\infty([x])$



Fig. 1: *Forward step*

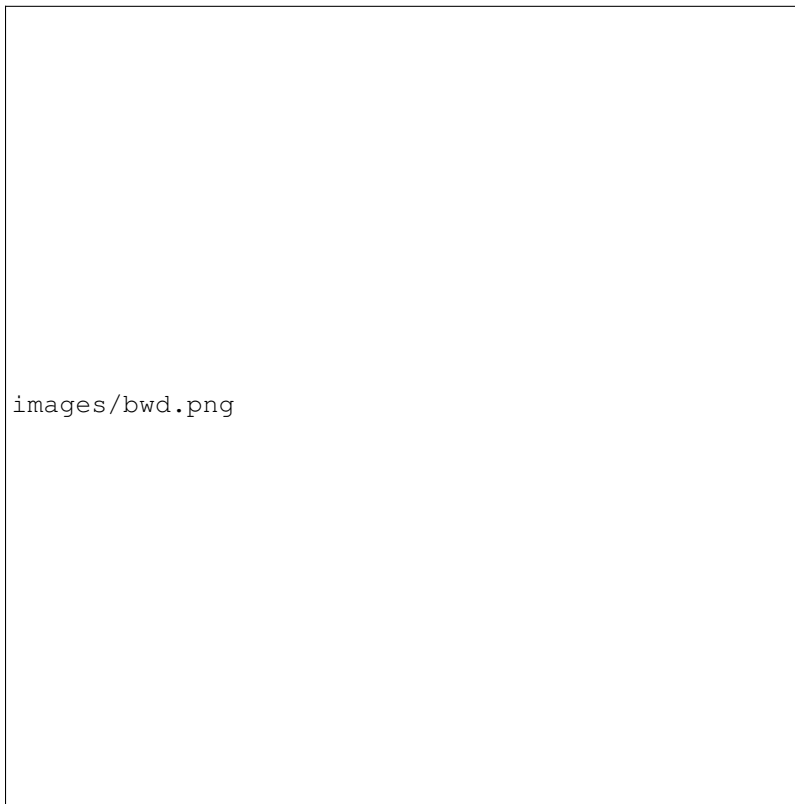


Fig. 2: *Backward step*

Basic examples are given in the *tutorial*.

To create a union or a composition of an arbitrary number of contractors, you need first to store all the contractors references into an Array. This is illustrated in the next example where we create a contractor for the intersection of n half-spaces (delimiting a polygon) and its complementary (see the picture below where the two contractors are used to build a *set*).

```
// Create the function corresponding to an
// hyperplane of angle alpha
Variable x,y,alpha;
Function f(x,y,alpha,cos(alpha)*x+sin(alpha)*y);

// Size of the polygon
int n=7;

// Array to store constraints (for cleanup)
Array<NumConstraint> ctrs(2*n);

// Arrays to store contractors
Array<Ctc> ctc_out(n), ctc_in(n);

for (int i=0; i<n; i++) {
    // create the constraints of the two half-spaces
    // delimited by f(x,y,i*2pi/n)=0
    // and store them in the array
    ctrs.set_ref(2*i, *new NumConstraint(x,y,f(x,y,i*2*Interval::PI/n)
    <=&1));
    ctrs.set_ref(2*i+1,*new NumConstraint(x,y,f(x,y,i*2*Interval::PI/n)>
    <1));

    // create the contractors for these constraints
    // and place them in the arrays
    ctc_out.set_ref(i,*new CtcFwdBwd(ctrs[2*i]));
    ctc_in.set_ref(i, *new CtcFwdBwd(ctrs[2*i+1]));
}

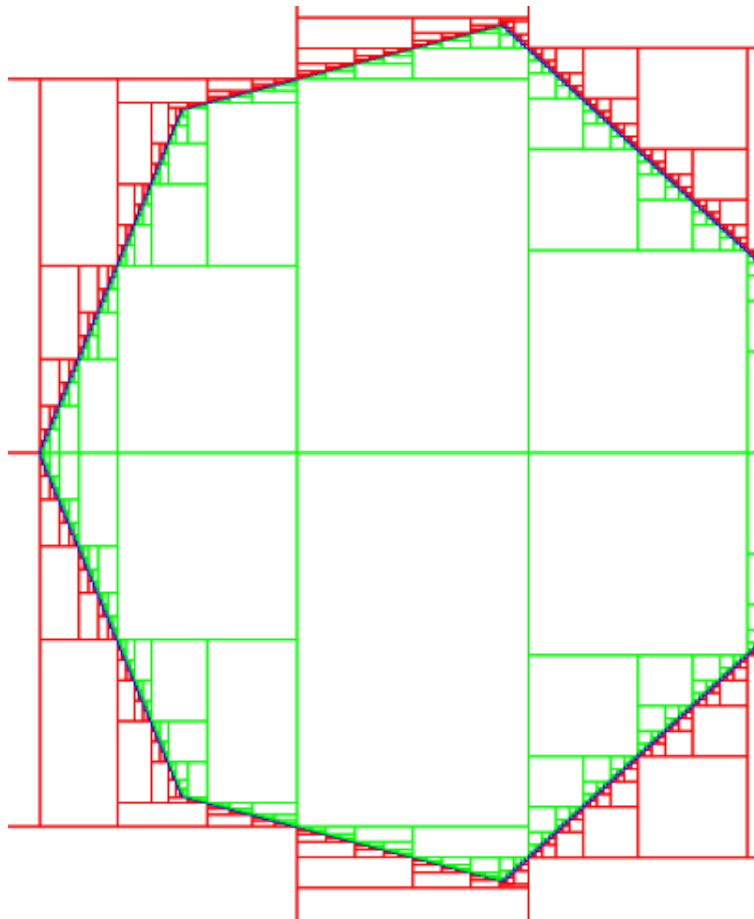
// Composition of the "outer" contractors
CtcCompo ctc_polygon_out(ctc_out);
// Union of the "inner" contractors
CtcUnion ctc_polygon_in(ctc_in);
```

Note: in this example, we have created constraints and contractors dynamically. We have to delete all these pointers after usage:

```
// ***** cleanup *****
for (int i=0; i<n; i++) {
    delete &ctc_in[i];
    delete &ctc_out[i];
}
for (int i=0; i<2*n; i++) {
    delete &ctrs[i];
}
```

1.9.4 Propagation

Propagation [Bessiere 2006] is another classical algorithm of constraint programming.



The basic idea is to calculate the fixpoint of a set of n contractors $C_1 \dots, C_n$, that is:

$$(C_1 \circ \dots \circ C_n)^\infty$$

without calling a contractor when it is unnecessary (as it is explained in the [tutorial](#)).

Let us first introduce for a contractor C two sets of indices: the *input* and *output* dimensions of C :

If:

$$\exists [x] \in \mathbb{IR}^n, \exists [y] \in \mathbb{IR}^n, \quad \forall j \neq i \quad [x]_j = [y]_j \wedge \begin{cases} C([x])_i \neq [x]_i \vee \\ C([y])_i \neq [y]_i \vee \\ \exists j \neq i \quad C([x])_j \neq C([y])_j \end{cases}$$

Then i is in the **input** of C .

If:

$$\exists [x] \in \mathbb{IR}^n, \quad C([x])_i \neq [x]_i$$

Then i is in the **output** of C .

Basically, $input(C)$ contains the variables that **potentially impacts** the result of the contractor while $output(C)$ contains the variables that are **potentially impacted by** the contractor.

We will explain further how this information is set in Ibex.

The propagation works as follows. It creates and maintain a set of *active* contractors \mathcal{A} (called “agenda”). The agenda is initialized to the full set:

$$\mathcal{A} := \{C_1, \dots, C_n\};$$

And the algorithm is:

1. Pop a contractor C from \mathcal{A}
2. Perform contraction: $[x] := C([x])$.
3. If the contraction was effective, push into \mathcal{A} all the contractors C' such that $input(C') \cap output(C) \neq \emptyset$
4. Return to step 1 until $\mathcal{A} = \emptyset$.

Note: The algorithm could be improved by not pushing again in the agenda a contractor C that is idempotent (*under development*).

To illustrate how the propagation can be used and the benefit it provides, we compare in the next example the number of times contractors (that we chose to be forward-backward) are called in a simple fixpoint loop with the number obtained in a propagation loop.

To this aim, we need to increment a counter each time a forward-backward is called. The easiest way to do this is simply to create a subclass of `CtcFwdBwd` that just call the parent contraction function and increments a global counter (static variable named `count`).

Here is the class:

```
class Count : public CtcFwdBwd {
public:
    static int count;

    Count(const NumConstraint& ctr) : CtcFwdBwd(ctr) { }

    void contract(IntervalVector& box) {
```

(continues on next page)

(continued from previous page)

```

        CtcFwdBwd::contract(box);
        count++;
    }
};

int Count::count=0;

```

Now, we load a set of constraints from a file. We chose on purpose a large but very sparse problem (makes propagation advantageous) and create our “counting” contractor for each constraint:

```

// Load a system of constraints
System sys (IBEX_BENCHS_DIR "/polynom/DiscreteBoundary-0100.bch");

// The array of contractors we will use
Array<Ctc> ctc(sys.nb_ctr);

for (int i=0; i<sys.nb_ctr; i++)
    // Create contractors from constraints and store them in "ctc"
    ctc.set_ref(i, *new Count(sys.ctr[i]));

```

Fixpoint ratio

The two contractors (CtFixPoint and CtcPropag) take also as argument a “fixpoint ratio”. The principle is that if a contraction does not remove more that

(ratio \times the diameter of a variable domain),

then this reduction is not propagated. The default value is 0.01 in the case of propagation, 0.1 in the case of fixpoint.

Warning: In theory (and sometimes in practice), the fixpoint ratio gives no information on the “distance” between the real fixpoint and the one we calculate.

Here we fix the ratio to 1e-03 for both:

```

double prec=1e-03;           // Precision upto which we calculate the
↪fixpoint

```

We can finally build the two strategies, call them on the same box and observe the number of calls. We also check that the final boxes are equal, up to the precision required.

With a fix point:

```

// ===== with simple fixpoint
↪=====
Count::count=0;           // initialize the counter

CtcCompo compo(ctc);      // make the composition of all contractors
CtcFixPoint fix(compo,prec); // make the fixpoint

IntervalVector box=sys.box; // tested box (load domains written in the file)

fix.contract(box);

output << " Number of contractions with simple fixpoint=" << Count::count <<
↪endl;

```

With a propagation:

```

// ===== with propagation
Count::count=0; // initialize the counter

CtcPropag propag(ctc, prec); // Propagation of all contractors

IntervalVector box2=sys.box; // tested box (load domains
written in the file)

propag.contract(box2);

output << " Number of contractions with propagation=" << Count::count << endl;

output << " Are the results the same? " << (box.rel_distance(box2)<prec? "YES
" : "NO") << endl;

```

The display is:

Input and Output variables

The input variables (the ones that potentially impacts the contractor) and the output variables (the ones that are potentially impacted) are two lists of variables stored under the form of “bitsets”.

A biset is nothing but an (efficiently structured) list of integers.

These bitsets are the two following fields of the “Ctc” class:

```

/**
 * The input variables (NULL pointer means "unspecified")
 */
BitSet* input;

/**
 * The output variables NULL pointer means "unspecified")
 */
BitSet* output;

```

These fields are not built by default. One reason is for allowing the distinction between an empty bitset and *no bitset* (information not provided). The other is that, in some applications, the number of variables is too large so that one prefers not to build these data structures even if they are very compacted.

To show the usage of these bitsets and their impact on propagation, we consider the same example as before. Let us now force the input/output bitsets of each contractors to contain every variable:

```

class Count2 : public CtcFwdBwd {
public:
    static int count;

    Count2(const NumConstraint& ctr) : CtcFwdBwd(ctr) {

        // The input bitset should have been created
        // by the constructor CtcFwdBwd
        assert(input!=NULL);

        // overwrite the input and output lists calculated
        // by CtcFwdBwd by adding all the variables
        for (int i=0; i<nb_var; i++) {

```

(continues on next page)

(continued from previous page)

```

        input->add(i);
        output->add(i);
    }
}

void contract(IntervalVector& box) {
    CtcFwdBwd::contract(box);
    count++;
}
};

int Count2::count=0;

```

We observe now that the fixpoint with `CtcPropag` is reached with as many iterations as without `CtcPropag`:

If you build a contractor from scratch (not inheriting a built-in contractor like we have just done), don't forget to create the bitsets before using them with `add/remove`.

Here is a final example. Imagine that we have implemented a contraction algorithm for the following constraint (over 100 variables):

$$\forall i, 0 \leq i \leq 49, \quad x[2 \times i] > 0 \implies x[2 \times i + 1] = 0.$$

The the input (resp. output) variables is the set of even (resp. odd) numbers. Here is how the initialization could be done:

```

class MyCtc : public Ctc {

    MyCtc() : Ctc(100) { // my contractor works on 100 variables

        // create the input list with all the variables set by default
        input = new BitSet(BitSet::all(100));
        // remove all the odd variables
        for (int i=0; i<100; i++)
            if (i%2==1) input->remove(i);

        // create the output list with all the variables unset by default
        output = new BitSet(BitSet::empty(100));
        // add all the odd variables
        for (int i=0; i<100; i++)
            if (i%2==1) output->add(i);
    }
}

```

The accumulate flag

The accumulate flag is a subtle tuning parameter you may ignore on first reading.

As you know now, one annoyance with continuous variables is that we have to stop the fixpoint before it is actually reached, which means that insignificant contractions are not propagated.

Now, to measure the significance of a contractor, we look at the intervals after contraction and compare them to the intervals just before contraction.

One little risk with this strategy is when a lot of insignificant contractions gradually contracts domains to a point where the overall contraction has become significant. The propagation algorithm may stop despite of this significant contraction.

The `accumulate` flag avoids this by comparing not with the intervals just before the current contraction, but the intervals obtained after the last significant one. The drawback, however, is that all the insignificant contractions are cumulated and attributed to the current contraction, whence a little loss of efficiency.

To set the `accumulate` flag, just write:

```
CtcPropag propag(...);
propag.accumulate = true;
```

Often, in practice, setting the `accumulate` flag results in a slightly better contraction with a little more time.

1.9.5 HC4

HC4 is the classical “constraint propagation” loop that we found in the literature [Benhamou & al., 1999]. It allows to contract with respect to a *system* of constraints.

In Ibex, the `CtcHC4` contractor is simply a direct specialization of `CtcPropag` (the *propagation contractor*).

The contractors that are propagated are nothing but the default (*Forward-Backward*) contractors associated to every constraint of the system.

Here is an example:

```
// Load a system of equations
System sys(IBEX_BENCHS_DIR "/others/hayes1.bch");
// Create the HC4 propagation loop with this system
CtcHC4 hc4(sys);

// Test the contraction
IntervalVector box(sys.box);
output << " Box before HC4:" << box << endl;
hc4.contract(box);
output << " Box after HC4:" << box << endl;
```

And the result:

1.9.6 Inverse contractor

Given

- a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- a contractor $C : \mathbb{R}^m \rightarrow \mathbb{R}^m$

The **inverse** of C by f is a contractor from $\mathbb{R}^n \rightarrow \mathbb{R}^n$ denoted by $f^{-1}(C)$ that satisfies:

$$\forall [x] \in \mathbb{R}^n, \quad \left(f^{-1}(C) \right) [x] \supseteq \{x \in [x], \exists y \in C(f([x]))\}.$$

To illustrate this we shall consider the function

$$t \mapsto (\cos(t), \sin(t))$$

and a contractor with respect to the constraint

$$x \geq 0, y \geq 0$$

```

// Build a contractor on  $R^2$  wrt  $(x \geq 0$  and  $y \geq 0)$ .

Function gx("x","y","x"); // build  $(x,y) \rightarrow x$ 
Function gy("x","y","y"); // build  $(x,y) \rightarrow y$ 

NumConstraint geqx(gx,GEQ); // build  $x \geq 0$ 
NumConstraint geqy(gy,GEQ); // build  $y \geq 0$ 

CtcFwdBwd cx(geqx);
CtcFwdBwd cy(geqy);

CtcCompo compo(cx,cy); // final contractor wrt  $(x \geq 0, y \geq 0)$ 

// Build a mapping from  $R$  to  $R^2$ 
Function f("t","(cos(t),sin(t))");

// Build the inverse contractor
CtcInverse inv(compo,f);

double pi=3.14;
IntervalVector box(1,Interval(0,2*pi));

inv.contract(box);
output << "contracted box (first time):" << box << endl;

inv.contract(box);
output << "contracted box (second time):" << box << endl;

```

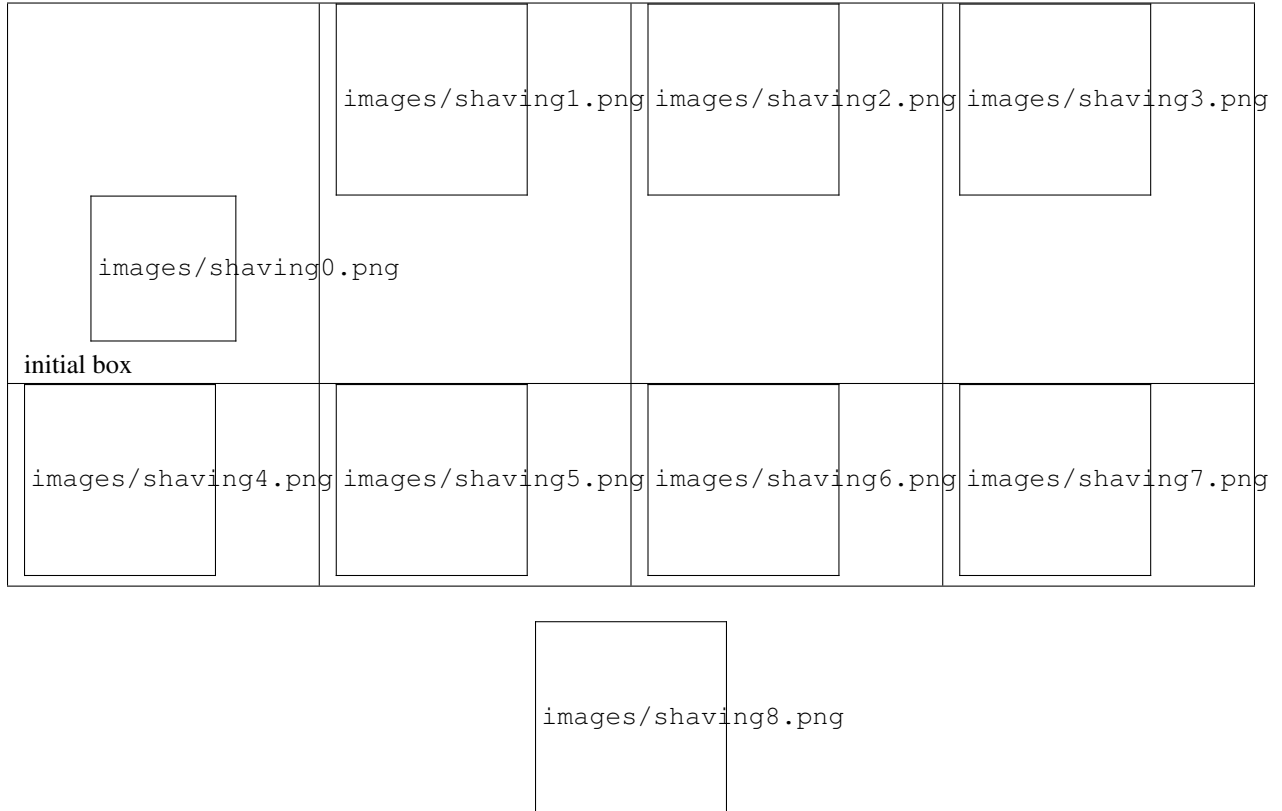
This gives:

1.9.7 Shaving

The shaving operator consists in calling a contractor C onto sub-parts (“slices”) of the input box. If a slice is entirely eliminated by C , the input box can be contracted by removing the slice from the box.

This operator can be viewed as a generalization of the SAC algorithm in discrete domains [Bessiere and Debruyne 2004]. The concept with continuous constraint was first introduced in [Lhomme 1993] with the “3B” algorithm. In this paper, the sub-contractor C was *HC4*.

(to be completed)



1.9.8 Acid & 3BCid

(to be completed)

1.9.9 Polytope Hull

Consider first a system of linear inequalities. Ibex gives you the possibility to contract a box to the *hull* of the polytope (the set of feasible points). This is what the contractor `CtcPolytopeHull` stands for.

This contractor calls the linear solver Ibex has been configured with (Soplex, Cplex, CLP) to calculate for each variable x_i , the following bounds:

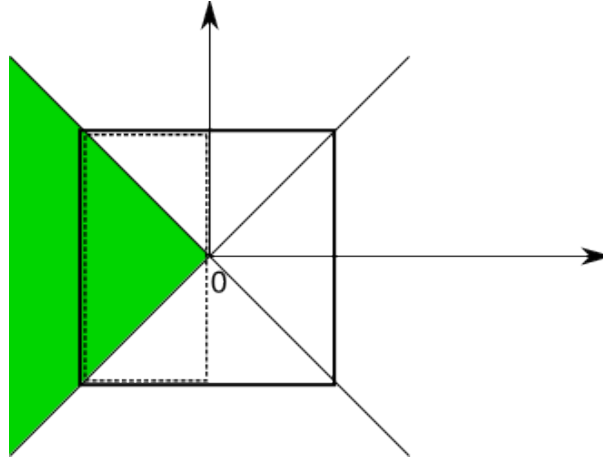
$$\min_{Ax \leq b \wedge x \in [x]} \{x_i\} \quad \text{and} \quad \max_{Ax \leq b \wedge x \in [x]} \{x_i\}.$$

where $[x]$ is the box to be contracted. Consider for instance

$$\begin{cases} x_1 + x_2 \leq 0 \\ x_1 - x_2 \leq 0 \end{cases}$$

Let $[x]$ be $[-1,1] \times [-1,1]$. The following picture depicts the polytope (which is rather a polyhedron in this case) in green, the initial box and the result of the contraction (dashed box).

The corresponding program is:



```
// build the matrix
double _A[4]= {1,1,1,-1};
Matrix A(2,2,_A);

// build the vector
Vector b=Vector::zeros(2);

// create the linear system (with fixed matrix/vector)
LinearizerFixed lin(A,b);

// create the contractor w.r.t the linear system
CtcPolytopeHull ctc(lin);

// create a box
IntervalVector box(2,Interval(-1,1));

// contract it
output << "box before contraction=" << box << endl;
ctc.contract(box);
output << "box after contraction=" << box << endl;
```

The output is:

In case of a non-linear system, it is also possible to call the `CtcPolytopeHull` contractor, providing that you give a way to *linearize* the non-linear system. Next section describes linearization techniques and gives an example of `CtcPolytopeHull` with a non-linear system.

Linearizations

Linearization procedures in Ibex are embedded in a class inheriting the `Linearizer` interface.

There exists some built-in linearization techniques, namely:

- `LinearizerXTaylor`: a corner-based Taylor relaxation [Araya & al., 2012].
- `LinearizerAffine2`: a relaxation based on affine arithmetic [Ninin & Messine, 2009].
- `LinearizerCombo`: a combination of the two previous techniques (the polytope is basically the intersection of the polytopes calculated by each technique)
- `LinearizerFixed`: a fixed linear system (as shown in the example above)

This Linearizer interface only imposes to implement a virtual function called `linearize`:

```
class Linearizer {
public:

    /**
     * Build a linearization procedure on nb_var variables.
     */
    Linearizer(int nb_var);

    /**
     * Add constraints in a LP solver.
     *
     * The constraints correspond to a linearization of the
     * underlying system, on the given box.
     *
     * This function must be implemented in the subclasses.
     *
     * \return the number of constraints (possibly 0) or -1 if the linear system is
     *         infeasible.
     */
    virtual int linearize(const IntervalVector& box, LinearSolver& lp_solver)=0;
};
```

The linearizer takes as argument a box because, most of the time, the way you linearize a nonlinear system depends on the domain of the variables. In other words, adding the $2*n$ bound constraints that represent the box to the system allow to build a far smaller polytope.

The second argument is the linear solver. This is only for efficiency reason: instead of building a matrix A and a vector b , the function directly enters the constraints in the linear solver. Let us now give an example.

Giving an algorithm to linearize a non-linear system is beyond the scope of this documentation so we shall here artificially *linearize a linear system*. The algorithm becomes trivial but this is enough to show you how to implement this interface. So let us take the same linear system as above and replace the `LinearizerFixed` instance by an instance of a home-made class:

```
/**
 * My own linear relaxation of a system
 */
class MyLinearRelax : public Linearizer {
public:
    /**
     * We actually only accept linear systems  $Ax \leq b$  :)
     */
    MyLinearRelax(const Matrix& A, const Vector& b) : Linearizer(2), A(A), b(b) {}

    virtual int linearize(const IntervalVector & box, LPSolver& lp_solver) {
        for (int i=0; i<A.nb_rows(); i++)
            // add the constraint in the LP solver
            lp_solver.add_constraint(A[i], LEQ, b[i]);

        // we return the number of constraints
        return A.nb_rows();
    }

    Matrix A;
    Vector b;
};
```

Replacing the `LinearizerFixed` instance by an instance of `MyLinearizer` gives exactly the same result.

1.9.10 Exists and ForAll

The `CtcExist` and `CtcForAll` contractors allow to deal with quantifiers in constraints, in a generic way.

Assume we have built a contractor `C` w.r.t. a constraint $c(x,y)$ (note that we have deliberately split here the variables of c into two sub-sets: $x = (x_1, \dots)$ and $y = (y_1, \dots)$).

The `CtcExist` operator produces from `C` a contractor with respect to the constraint c where the variable y has been *existentially quantified*. More precisely, the result is a contractor w.r.t. to the constraint c' defined as follows:

$$c'(x) \iff \exists y \in [y], c(x, y).$$

where $[y]$ is a user-defined box.

Similarly, `CtcForAll` produces a contractor w.r.t.:

$$\forall y \in [y], c(x, y).$$

It is important to notice that the contractor `CtcExist/CtcForAll` expects as input two boxes, each playing a different role:

- a box that represents **the domain of y** . This box is read or written in a specific field of `CtcExist/CtcForAll` called `y_init`. The domain of y is a parameter that must be initialized priori to contractions. It can also be dynamically updated between two contractions but it is considered as fixed during one contraction.
- a box that represents **the domain of x** : this is the box that `CtcExist/CtcForAll` waits as a contractor. This box is the one in argument of the `contract (. . .)` function.

The contractors work as depicted on the following pictures (which illustrate the principles of the algorithms, but the actual implementation is a little more clever).

images/ctcquantif1.png

CtcExist. (a) The domain of y is split until a precision of ε is reached.

images/ctcquantif3.png

CtcExist. (b) each box is contracted with the sub-contractor and projected onto x . The result is the union (hull) of all these projections (the interval in green).

<p>images/ctcquantif4.png</p>		
<p>CtcForAll. (a) The domain of y is split until a precision of ε is reached. A box is then formed by taking the midpoint for y.</p>		<p>images/ctcquantif5.png</p> <p>CtcForAll. (b) each box is contracted with the sub-contractor and projected onto x. The result is the intersection of all these projections (the interval in green).</p>

Here is a first example. We consider the constraint $c(x, y) \iff x^2 + y^2 \leq 1$. When a constraint is given directly to a `CtcExist/CtcForAll` operator, this constraint is automatically transformed to a `CtcFwdBwd` contractor.

If the domain for y is set to $[-1, 1]$, the set of all x such that it exists y in $[-1, 1]$ with (x, y) satisfying the constraint is also $[-1, 1]$. This can be observed in the following program:

```

// create a constraint on (x,y)
Variable x,y;
NumConstraint c(x,y,sqr(x)+sqr(y)<=1);

// create domains for x and y
IntervalVector box_x(1, Interval(-10,10));
IntervalVector box_y(1, Interval(-1,1));

// set the precision that controls how much y will be bisected
double epsilon=1.0;

// create a contractor on x by transforming y into an
// existentially-quantified parameter.
CtcExist exist_y(c,y,box_y,epsilon);

// contract the domain of x
output << "box before contraction=" << box_x << endl;
exist_y.contract(box_x);
output << "box after contraction=" << box_x << endl;

```

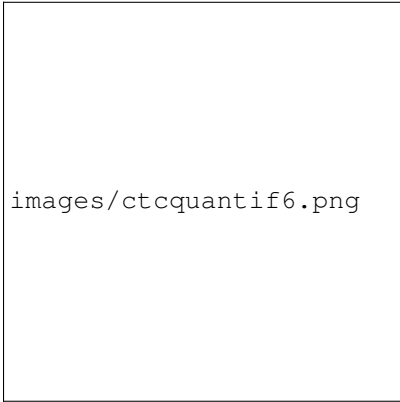
The output is:

In this example, we have set the value of ε to 1.0 which is a very crude precision. In fact, the result would have been the same with an arbitrarily large precision, simply because the contraction of the full box $[-10,10] \times [-1,1]$ is optimal in this case. In other words, splitting y (i.e., using the `CtcExist` operator) is so far totally superfluous.

To create a more interesting example (where optimality of a single contraction is lost) we consider two constraints:

$$x^2 + y^2 \leq 1 \wedge x = y.$$

The set of solutions is the segment depicted in the figure below.



images/ctcquantif6.png

Now the value of ε has an impact on the accuracy of the result, as the next program illustrates:

Note: to create a conjunction of two constraints, we introduce a vector-valued function because the current constructor of `CtcExist` does not accept a `System` (this shall be fixed in a future release). You may also consider in this situation building `CtcExist` directly *from a contractor*.

```

// create a conjunction of two constraint on (x,y)
Variable x,y;
Function f(x,y,Return(sqr(x)+sqr(2*y-y),y-x));
IntervalVector z(2);
z[0]=Interval(0,1);

```

(continues on next page)

(continued from previous page)

```

z[1]=Interval::ZERO;
NumConstraint c(x,y,f(x,y)=z);

// create domains for y
IntervalVector box_y(1, Interval(-1,1));

// observe the result of the contraction for
// different precision epsilon=10^{-_log}
for (int _log=0; _log<=8; _log++) {
    // create the domain for x
    IntervalVector box_x(1, Interval(-10,10));

    // create the exist-contractor with the new precision
    CtcExist exist_y(c,y,box_y,::pow(10,-_log));

    // contract the box
    exist_y.contract(box_x);

    output << "epsilon=1e-" << _log << " box after contraction=" << box_x
    << endl;
}

```

The output is:

Warning: As we have explained, both `CtcExist` and `CtcForAll` deploy internally a search tree on the variables y until some precision ε is reached (the precision is uniform so far). The complexity of `CtcExist` and `CtcForAll` are therefore exponential in the number n_y of variables y and with potentially a “big constant”. Roughly, the complexity is in $O\left(\left[\frac{\max_i \text{rad}(y_i)}{\varepsilon}\right]^{n_y}\right)$. A good way to alleviate this high complexity is to use an *adaptive precision*. but it is anyway strongly recommended to limit the number of quantified parameters.

The generic constructor

If you want to use a specific contractor (either because forward-backward is not appropriate or because your constraint is something more exotic), you can resort to the generic constructor. This constructor simply takes a contractor `C` in argument instead of a constraint. There is however another slight difference. Since there is no constraint anymore, you cannot directly specify the parameters as symbols directly: `CtcExist` cannot map itself these symbols to components of boxes contracted by `C` (remember that contractors are pure numerical functions). So you need to explicitly state the indices of variables that are transformed into parameters. This indices are given in a bitset structure. Here is a “generic variant” of the last example. Note that we take benefit of this generic constructor to give a *HC4* contractor instead of a simple forward-backward, whence a slightly better contraction.

```

// create a system
Variable x,y;
SystemFactory fac;
fac.add_var(x);
fac.add_var(y);
fac.add_ctr(sqr(x)+sqr(2*y-y)<=1);
fac.add_ctr(x=y);

System sys(fac);

CtcHC4 hc4(sys);

// create domains for y

```

(continues on next page)

(continued from previous page)

```

IntervalVector box_y(1, Interval(-1,1));

// Creates the bitset structure that indicates which
// component are "quantified". The indices vary
// from 0 to 1 (2 variables only). The bitset is
// initially empty which means that, by default,
// all the variables are parameters.
BitSet vars(2);

// Add "x" as variable.
vars.add(0);

// observe the result of the contraction for
// different precision epsilon=10^{-_log}
for (int _log=0; _log<=8; _log++) {
    // create the domain for x
    IntervalVector box_x(1, Interval(-10,10));

    // create the exist-contractor with the new precision
    CtcExist exist_y(hc4,vars,box_y,::pow(10,-_log));

    // contract the box
    exist_y.contract(box_x);

    output << "epsilon=1e-" << _log << " box after contraction=" << box_x
    << endl;
}

```

The output is:

Adaptive precision

When a `CtcExist` contractor is embedded in a search, one naturally asks for an adaptive precision. Indeed, when the domain of $[x]$ is large, it is counter-productive to split $[y]$ down to maximal precision. More generally, the idea of adaptive precision is to dynamically set the precision ε accordingly to the size of the contracted box $[x]$.

You may find surprising that such adaptive precision is not part of the `CtcExist/CtcForAll` interface. This is because the adaptive behavior can be easily produced thanks to the contractor paradigm. The idea is to create a contractor that wraps `CtcExist` and simply update the precision. Next program illustrates the concept.

```

class MyCtcExist : public Ctc {
public:

    /*
     * Create MyCtcExist. The number of variables
     * is the number of bits set to "true" in the
     * "vars" structure; this number is obtained
     * via vars.size().
     */
    MyCtcExist(Ctc& c, const BitSet& vars, const IntervalVector& box_y) :
        Ctc(vars.size()), c(c), vars(vars), box_y(box_y) { }

    void contract(IntervalVector& box_x) {
        // Create a CtcExist contractor on-the-fly
        // and set the splitting precision on y
    }
}

```

(continues on next page)

(continued from previous page)

```

        // to one tenth of the maximal diameter of x.
        // The box_x is then contracted
        CtcExist(c, vars, box_y, box_x.max_diam() / 10).contract(box_x);
    }

    // The sub-contractor
    Ctc& c;
    // The variables indices
    const BitSet& vars;
    // The parameters domain
    const IntervalVector& box_y;
};

```

It suffices now to replace a `CtcExist` object by a `MyCtcExist` one.

1.10 Separators

1.10.1 Introduction

A separator is an operator that performs two independent and complementary contractions. The separator is associated with a set (noted S) and the first contraction (called “inner”) removes points inside S . The second contraction (called “outer”) removes points outside S . See [\[Jaulin & Desrochers 2014\]](#).

In concrete terms, given a box $[x]$, the separator produces two sub-boxes $[x_{in}]$ and $[x_{out}]$ that verify:

$$\begin{aligned} ([x] \setminus [x_{in}]) &\subset S \\ ([x] \setminus [x_{out}]) \cap S &= \emptyset \end{aligned}$$

For efficiency reasons, the `separate(...)` function takes two input-output arguments, `<x_in>` and `<x_out>`, each containing initially a copy of the box $[x]$.

The first and more natural way to build a separator is to do it from a set implicitly defined by a constraint. See [Separator from a constraint](#).

Since a separator can be viewed as a pair of contractors, another natural way to build a separator is from two complementary contractors. See [Separator from complementary contractors](#).

A separator is however not necessarily built this way. A separator can also be built from a contractor and a predicate. In this case, the contractor is assumed to work with respect to the boundary of a set (that is, it removes both inner and outer points) and the predicate is assumed to state if a given box is either inside, outside or crossing the boundary of the same set. See [Boundary-Based Separator](#).

Separator from a constraint

When the set S corresponds to an inequality $f(x) < 0$, a separator with respect to the set S can therefore be automatically derived using forward / backward techniques for both contractions. This is what the `SepFwdBwd` class stands for. The outer (resp. inner) contractor is simply a forward-backward with respect to $f < 0$ (resp. $f \geq 0$).

```

// Define the function
Function f("x", "y", "x - y");

// Build the separator for the set X = {x, y | x - y < 0}

```

(continues on next page)

(continued from previous page)

```

SepFwdBwd s(f, LT);

// Note: in a similar way, it is possible to define the separator
// for the complementary set : X = {x, y | x - y >= 0}
//SepFwdBwd s_compl(f, GEQ);

double _box[2][2] = {{0,3},{1,2}};
IntervalVector box(2,_box);
IntervalVector x_in=box;
IntervalVector x_out=box;

s.separate(x_in,x_out);

output << "result of inner contraction=" << x_in << endl;
output << "result of outer contraction=" << x_out << endl;

```

The result is:

Separator from complementary contractors

If two complementary contractors \mathcal{C}_{in} and \mathcal{C}_{out} are available, the separator can build using the SepCtcPair class.

(to be completed)

```

Ctc c_in = ...
Ctc c_out = ...
// Build a separator from two complementary contractors
SepCtcPair s(c_in, c_out);

```

Boundary-Based Separator

Sometimes, we only have contractors with respect to the boundary of a set \mathcal{S} . Isolating the inner and the outer contractor is not possible, or not easy. It is still possible to build a separator in this case, providing that we can also test whether a point belongs to the set or not.

Let us consider a contractor \mathcal{C} w.r.t. the boundary of a set \mathcal{S} . The main idea behind the separator is, first, to contract the input box using \mathcal{C} and, second, to test for each box in $\neg\mathcal{C}$ if it belongs to \mathcal{S} or not.

The test is performed for a given box by picking randomly one point and calling a *predicate*. A predicate is an object of a class extending Pdc. It must implements a method `test(IntervalVector&)` that returns a boolean interval, that is, either YES, NO or MAYBE (see BoolInterval). In the case of a separator, the predicate must be an operator T such that:

$$\begin{aligned}
 T : \mathbb{R}^n &\longrightarrow \{yes, no, maybe\} \\
 [\mathbf{x}] &\longmapsto \begin{cases} yes & \text{if } [\mathbf{x}] \subseteq \mathcal{S} \\ no & \text{if } [\mathbf{x}] \cap \mathcal{S} = \emptyset \\ maybe & \text{otherwise} \end{cases}
 \end{aligned}$$

A separator is built from a contractor and a predicate using the SepBoundaryCtc class.

Note: the predicate is called by SepBoundaryCtc only with points (degenerated boxes) but the interface for Pdc has been made to deal with a more general situation.

The *separator for the constraint “points in polygon”* is an illustration of this type of separator.

As an illustration of the concept, we build here a separator for an inequality using SepBoundaryCtc:

```

// Define the function
Function f("x", "y", "x - y");

// Build a contractor for the boundary
CtcFwdBwd c(f,EQ);

// Build a predicate for the inside
PdcFwdBwd p(f,LEQ);

// Build the separator for the set X = {x, y | x - y < 0}
SepBoundaryCtc s(c,p);

double _box[2][2] = {{0,3},{1,2}};
IntervalVector box(2,_box);
IntervalVector x_in=box;
IntervalVector x_out=box;

s.separate(x_in,x_out);

output << "result of inner contraction=" << x_in << endl;
output << "result of outer contraction=" << x_out << endl;

```

The result is:

1.10.2 Separator Algebra

The Separator algebra is a direct extension of the set algebra. E.g., the intersection of two separators w.r.t \mathcal{S}_1 and \mathcal{S}_2 is a separator w.r.t. $\mathcal{S}_1 \cap \mathcal{S}_2$.

Here are the available operations and the way they are performed. Separators are viewed as pair of contractors denoted $\mathcal{S}_i = (\mathcal{S}_i^{in}, \mathcal{S}_i^{out})$.

$$\begin{aligned}
\overline{\mathcal{S}} &= (\mathcal{S}^{out}, \mathcal{S}^{in}) && \text{(Negation)} \\
\mathcal{S}_1 \cap \mathcal{S}_2 &= (\mathcal{S}_1^{in} \cup \mathcal{S}_2^{in}, \mathcal{S}_1^{out} \cap \mathcal{S}_2^{out}) && \text{(intersection)} \\
\mathcal{S}_1 \cup \mathcal{S}_2 &= (\mathcal{S}_1^{in} \cap \mathcal{S}_2^{in}, \mathcal{S}_1^{out} \cup \mathcal{S}_2^{out}) && \text{(union)} \\
\bigcap_{\{q\}} \mathcal{S}_i &= \left(\bigcap_{\{m-q-1\}} \mathcal{S}_i^{in}, \bigcap_{\{q\}} \mathcal{S}_i^{out} \right) && \text{(relaxed intersection)} \\
\mathcal{S}_1 \setminus \mathcal{S}_2 &= \mathcal{S}_1 \cap \overline{\mathcal{S}_2} && \text{(difference)}
\end{aligned}$$

The following example shows how to combine separator for finding the union and intersection of 3 rings.

(to be completed)

```

// define the center of circle
double ax[] = {3,7,-3};
double ay[] = {4,3,7};
double dist[] = {3,6,6};

Variable x,y;

// Rings definitions
Function f1(x,y,sqrt(sqr(x-ax[0]) + sqr(y-ay[0])));
SepFwdBwd S1(f1,dist[0]);

Function f2(x,y,sqrt(sqr(x-ax[1]) + sqr(y-ay[1])));
SepFwdBwd S2(f2,dist[1]);

```

(continues on next page)

(continued from previous page)

```

Function f3(x,y,sqrt(sqr(x-ax[2]) + sqr(y-ay[2])));
SepFwdBwd S3(f3,dist[2]);

// Negation of separator
SepNot S4(S3);

// union of separators
Array<Sep> AS(S1,S2,S3);
SepUnion SUL = SepUnion(AS); // Union from an array of
→separators
SepUnion SU2 = SepUnion(S1,S2); // Union from two separators
SepUnion SU3 = SepUnion(S1,S2,S3);

// intersection of separators
SepInter SIL = SepInter(AS);
SepInter SI2 = SepInter(S1,S2);
SepInter SI3 = SepInter(S1,S2,S4);

```

1.10.3 Separator for a Polygon

Note: This separator is available in the ENSTA Robotics plugin (–with-ensta-robotics).

(under construction)

Contractor for a Segment

The `CtcSegment` class allows to contract a box w.r.t. a segment (in the plane), that is, w.r.t. to the constraint

$$\mathbf{x} \in [\mathbf{a}, \mathbf{b}]$$

where $\mathbf{a} \in \mathbb{R}^2$, $\mathbf{b} \in \mathbb{R}^2$.

The contractor works by consider the following equivalent constraints :

$$\begin{cases} \det(\mathbf{b} - \mathbf{a}, \mathbf{a} - \mathbf{x}) = 0 \\ \min(\mathbf{a}, \mathbf{b}) \leq \mathbf{x} \leq \max(\mathbf{a}, \mathbf{b}). \end{cases}$$

the *min* and the *max* being interpreted componentwise.

Remark: The first constraint is an equality $f(x) = 0$ and the associated contractor will contract only on the boundary. We only have an approximation of \mathbb{X}^+ because there is no inner part.

Point Inside a Polygon

We consider an oriented polygon \mathcal{P} , convex or not, without self interaction, composed of N segments. The boundary $\partial\mathcal{P}$ of the polygon satisfies the following constraint:

$$\partial\mathcal{P} = \{\mathbf{m} \in \mathbb{R}^2, \exists i \in \llbracket 1, N \rrbracket, \mathbf{m} \in [\mathbf{a}_i, \mathbf{b}_i]\}$$

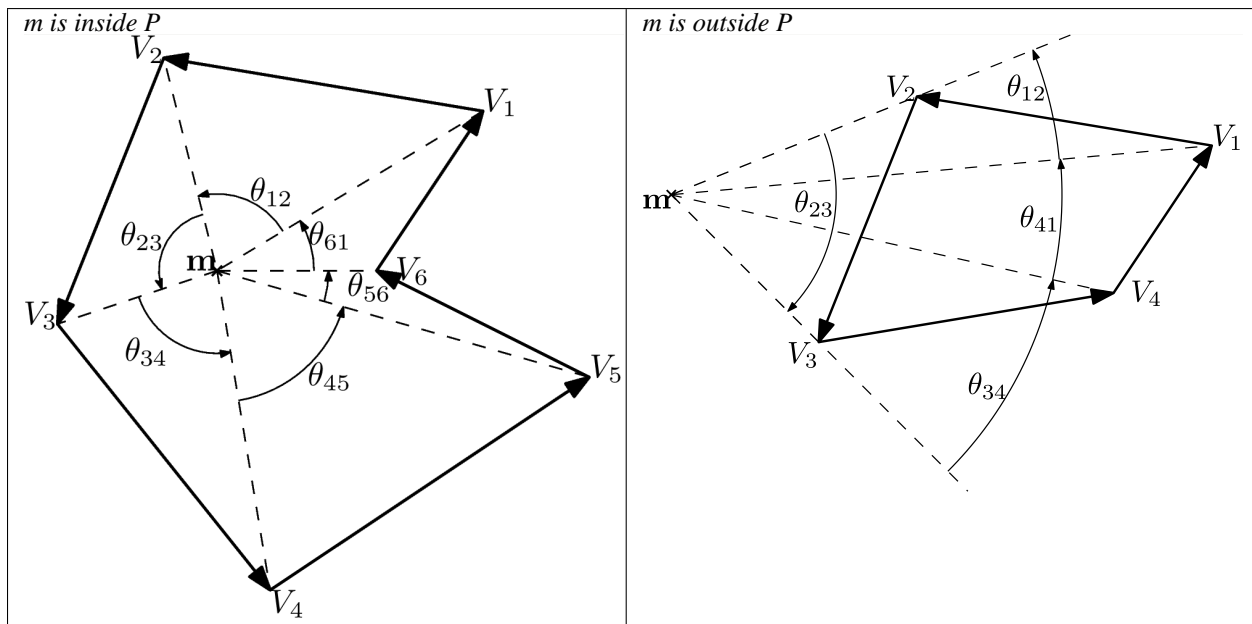
Let's us take \mathcal{C}_{a_i, b_i} as a contractor for the segment $[a_i, b_i]$, the contractor for $\partial\mathcal{P}$ is:

$$\mathcal{C}_{\partial\mathcal{P}} = \bigcup_{i=1}^N \mathcal{C}_{a_i, b_i}$$

Remark: Because the union of minimal contractor is minimal, $\mathcal{C}_{\partial\mathcal{P}}$ is a minimal contractor for the border of the polygon \mathcal{P} .

To identify which part is inside and outside we use a test based on the *Winding Number* which represents the total number of times that curve travels counterclockwise around the point. The winding number depends on the orientation of the curve, and is negative if the curve travels around the point clockwise. Let us take a polygon \mathcal{P} with vertices's $V_1, V_2, \dots, V_n = V_1$ and \mathbf{m} a point not on the border of \mathcal{P} . The Winding Number is defined by:

$$\mathbf{wn}(\mathbf{m}, P) = \frac{1}{2\pi} \sum_{i=1}^n \theta_i = \frac{1}{2\pi} \sum_{i=1}^n \arccos \left(\frac{(V_i - \mathbf{m}) \cdot (V_{i+1} - \mathbf{m})}{\|V_i - \mathbf{m}\| \|V_{i+1} - \mathbf{m}\|} \right)$$



So, if \mathbf{m} is outside P we will have $\mathbf{wn}(\mathbf{m}, P) = 0$, otherwise if \mathbf{m} is inside, $\mathbf{wn}(\mathbf{m}, P) = 1$.

The class implementing a separator for a polygon is `SepPolygon`.

(to be completed)

Example

Let \mathcal{S}_P be the polygon described in figure ...

The following snippet shows how to build the associated separator and make operations with:

(to be completed)

```
// Polygone convex
vector<double> walls_xa,walls_xb,walls_ya,walls_yb;

walls_xa.push_back(6); walls_ya.push_back(-6); walls_xb.push_back(7); walls_yb.
↪push_back(9);
```

(continues on next page)

(continued from previous page)

```

walls_xa.push_back(7); walls_ya.push_back(9); walls_xb.push_back(0); walls_yb.
↳push_back(5);
walls_xa.push_back(0); walls_ya.push_back(5); walls_xb.push_back(-9); walls_yb.
↳push_back(8);
walls_xa.push_back(-9); walls_ya.push_back(8); walls_xb.push_back(-8); walls_yb.
↳push_back(-9);
walls_xa.push_back(-8); walls_ya.push_back(-9); walls_xb.push_back(6); walls_yb.
↳push_back(-6);

SepPolygon S1(walls_xa, walls_ya, walls_xb, walls_yb);

// Make a hole inside the first one
vector<double> walls_xa2,walls_xb2,walls_ya2,walls_yb2;
walls_xa2.push_back(-2); walls_ya2.push_back(3); walls_xb2.push_back(3.5); walls_
↳yb2.push_back(2);
walls_xa2.push_back(3.5); walls_ya2.push_back(2); walls_xb2.push_back(3); walls_
↳yb2.push_back(-4);
walls_xa2.push_back(3); walls_ya2.push_back(-4); walls_xb2.push_back(-3); walls_
↳yb2.push_back(-3);
walls_xa2.push_back(-3); walls_ya2.push_back(-3); walls_xb2.push_back(-2); walls_
↳yb2.push_back(3);

SepPolygon S2(walls_xa2, walls_ya2, walls_xb2, walls_yb2);
SepNot S3(S2);

// Separator for the polygon with a hole in it
SepInter S(S1, S3);

```

Using a paver, the previous separator will produce the following figure :

1.11 Sets

Note: This part of the library is recent and under active development. It will be enriched with new functionalities in future releases.

1.11.1 Introduction

Ibex provides a structure for representing sets of \mathbb{R}^n with boxes and performing operations on sets directly. Another possible name for such a set would be a *paving*.

The structure is organized as a binary tree. Each leaf represents a box with a *status* that indicates whether the box is inside the set, outside the set or potentially crossing the boundary of the set. The status is a boolean interval (BoolInterval) which possible values are either YES (*inside*), NO (*outside*) or MAYBE (*boundary*). This is depicted in the figure below:

1.11.2 A graphical tool: Vibes

In all the examples proposed below, we will display boxes (rectangles) to have a visual rendering of the computations.

In the solutions, we propose to use [Vibes](#) but you can easily adapt the code to use your favorite graphical tool.

If you want to use Vibes allos, here is a few tips (valid for Linux and MacOS).

First, install Vibes:

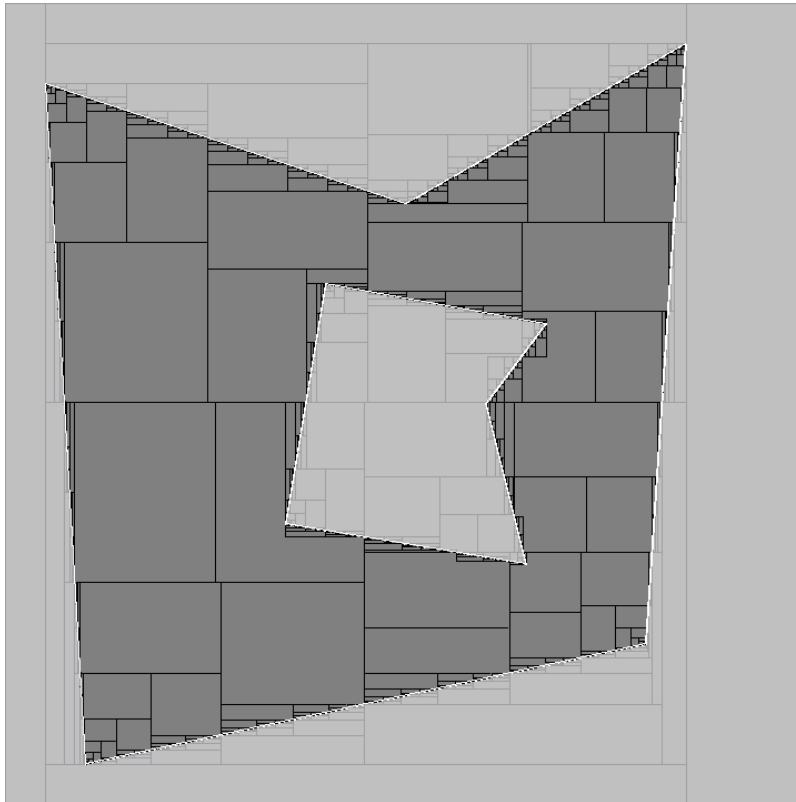


Fig. 3: *Point inside the polygon are in dark gray.*



Fig. 4: **Example of “set”**, obtained from the constraint $\|(x, y)\| \leq 5$.


```
gunzip VIBES-XXX.tar.gz
tar xvf VIBES-XXX.tar
cd VIBES-XXX/viewer
cmake .
make
cd ../../
```

Vibes is based on a client-server approach, the boxes are sent by the client program through a pipe and plot by another program (the viewer).

To run the viewer:

```
VIBES-XXX/viewer/VIBes-viewer&
```

Now, copy the client API in the folder of your C++ program:

```
cp VIBES-XXX/client-api/C++/src/* [my-folder]
```

In the client program, include the Vibes API:

```
#include "ibex.h"
#include "vibes.cpp"

using namespace std;
using namespace ibex;
```

Then, connect to the server with the following instructions.

```
int main() {
    vibes::beginDrawing ();
    vibes::newFigure("...");
    ...
}
```

And disconnect before your program terminantes:

```
vibes::endDrawing();
```

Finally, to plot a box ([a,b],[c,d]) just call:

```
vibes::drawBox(a, b, c, d, "...");
```

What is between the double quotes is the color code of the box. For instance, “b[r]” will paint the box in red and the contour in blue.

1.11.3 Set Creation

A set is an instance of `Set` and can either be initially defined as \mathbb{R}^n itself or a specific box:

```
// create the two-dimensional set (-oo,+oo)x(-oo,+oo)
Set set1(2);

// create the two-dimensional set [0,1]x[0,1]
Set set2(IntervalVector(2,Interval(0,1)));
```

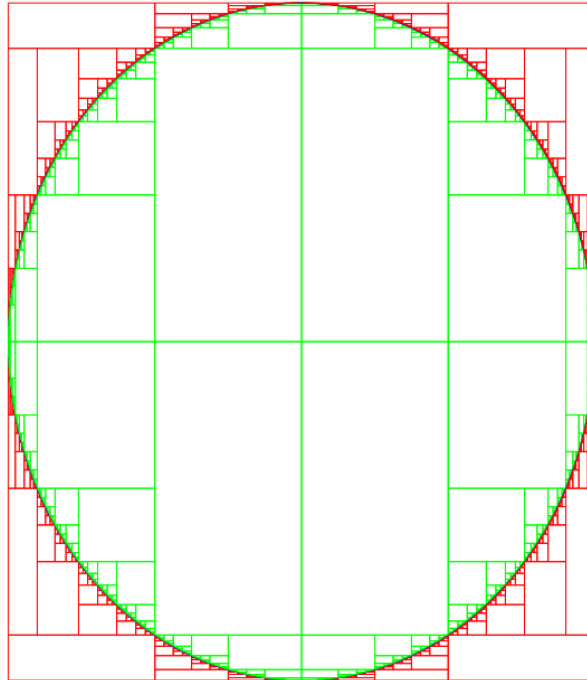
It can also be initialized from a constraint or a system of constraints. In this case, the set is built by performing some computations recursively, until some precision is reached. The second parameter given to the constructor controls this precision.

Note: the computation performed recursively is an application of the *forward-backward separator* associated to the constraint.

```
// Create the constraint  $\|(x,y)\|^2 \leq 25$ 
NumConstraint c("x","y","x^2+y^2<=25");

// Create the set with a precision of 0.01
Set set(c,0.01);
```

The result is the following picture, obtained with Vibes. We explain how we have generated this picture in the next section.



1.11.4 Set Exploration

The internal structure of sets is not intended to be handled directly by users (and may actually change with time). If you want to explore a set, you can use the `SetVisitor` class (following the [visitor design pattern](#)). This class must implement a `visit_leaf` function that will be automatically called for every leaf of the set. It can also optionally implement a `visit_node` function that is called for every intermediate node in the tree, and tell whether or not to visit children of the current node by returning true (default) or false.

A typical usage of a set visitor is for listing or plotting the set and we shall illustrate the mechanism for such usage.

The `visit_leaf` function must take in argument the information that characterizes a leaf, namely, a box (`IntervalVector`) and a status (`BoolInterval`). Here is an example that simply displays the box and the status of a leaf in the standard output:

```
class ToConsole : public SetVisitor {
/**
 * The function that will be called automatically on every boxes (leaves) of the_
↪ set.
 */
void visit_leaf(const IntervalVector& box, BoolInterval status) {
```

(continues on next page)

(continued from previous page)

```
output << box << " : ";

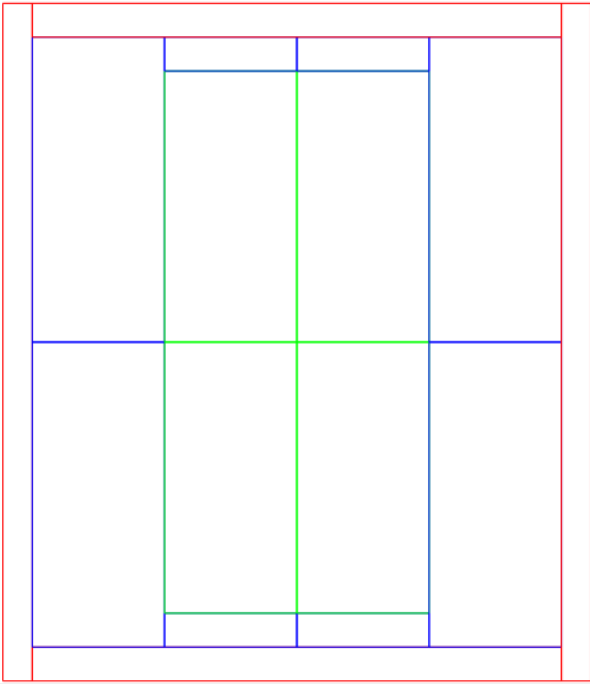
switch (status) {
  case YES:    output << "in"; break;
  case NO:     output << "out"; break;
  case MAYBE : output << "?"; break;
}
output << endl;
}
};
```

Now the following code *load a set* from a file and list all the boxes inside:

```
Set set("set-example");

ToConsole to_console;
set.visit(to_console);
```

The result is:

	
Visiting the set with ToPlot	Visiting the set with ToVibes

We will now show how to plot a set calculated by Ibex with *Vibes*.

We create a new class that implements the `visit_leaf` function as follows:

```
class ToVibes : public SetVisitor {
public:
```

(continues on next page)

(continued from previous page)

```

/**
 * Plot a box within the frame [-max,max]x[-max,max]
 *
 * The frame avoids, in particular, to plot unbounded OUT boxes.
 */
ToVibes(double max) : frame(2,max*Interval(-1,1)) { }

/**
 * Function that will be called automatically on every boxes (leaves) of the set.
 */
void visit_leaf(const IntervalVector& box, BoolInterval status) {

    // Intersect the box with the frame
    IntervalVector framebox=box & frame;

    // Associate a color to the box.
    // - YES (means "inside") is in green
    // - NO (means "outside") is in red
    // - MAYBE (means "boundary") is in blue.
    const char* color;

    switch (status) {
    case YES: color="g"; break;
    case NO: color="r"; break;
    case MAYBE : color="b"; break;
    }

    // Plot the box with Vibes
    vibes::drawBox(framebox[0].lb(), framebox[0].ub(), framebox[1].lb(), framebox[1].
↪ub(), color);
    }

    IntervalVector frame;
};

```

The main code is similar:

```

vibes::beginDrawing ();
vibes::newFigure("visit");

Set set("set-example");

ToVibes to_vibes(10);
set.visit(to_vibes);
vibes::endDrawing();

```

And the result is the picture above.

1.11.5 File operations

You can save a set into a file and load a set from a file.

To save into a file named “set-example”:

```

set.save("set-example");

```

To load a set from a file, use the constructor with string argument:

```
Set set("set-example");
```

1.11.6 Set Intersection

A set can be intersected with another set that can either be explicit (of type `Set`) or implicit. A contractor and a separator are examples of implicit sets. In this case, we talk about *set contraction* rather than intersection (but, conceptually, this is two equivalent terms when dealing with sets).

We consider here intersection of two explicit sets.

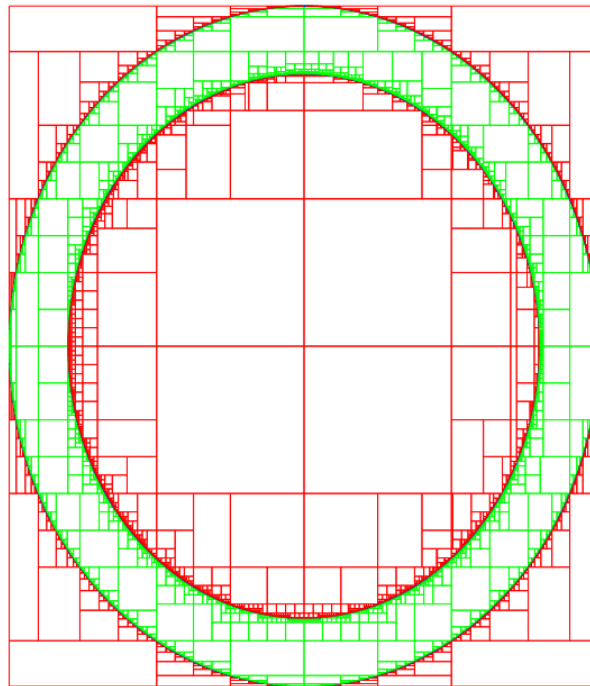
The intersection between two sets is obtained with the `&=` operator. In the next example, we create two sets from separators, one being a circle of radius 5, the other being the complementary of a circle of radius 4. The result of the intersection gives a ring.

```
// Create a first set ||(x,y)||<=5
NumConstraint c("x","y","x^2+y^2<=25");
Set set(c,0.01);

// Create a second set ||(x,y)||>=4
NumConstraint c2("x","y","x^2+y^2>=16");
Set set2(c2,0.01);

// Intersect the first set with the second one
set &= set2;
```

The result is the following picture.



Note: It should be emphasized that it is always better to handle sets explicitly on last resort. This means that, in this example, it would have been more efficient to create one set from the conjunction of the two constraints.

1.11.7 Set Union

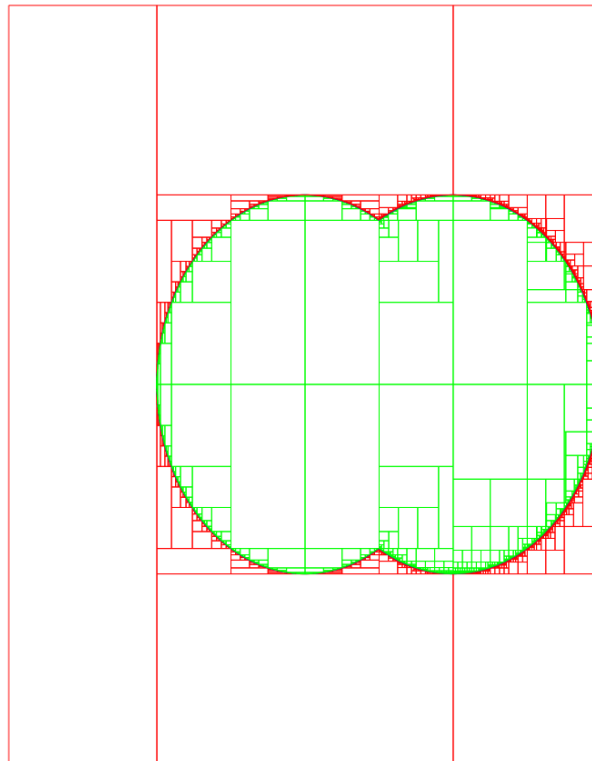
The union works similarly. In the next example, we create two sets, one being a circle centered on the origin and the other the circle centered on the point (5,0). Then we perform the union of the two sets.

```
// Create a first set  $\|(x,y)\| \leq 5$ 
NumConstraint c("x","y","x^2+y^2<=25");
Set set(c,0.01);

// Create a second set  $\|(x-5,y)\| \leq 5$ 
NumConstraint c2("x","y","(x-5)^2+y^2<=25");
Set set2(c2,0.01);

// Make the union
set |= set2;
```

The result is the following picture.



1.11.8 Set Contraction

A *separator* can be used to contract a set.

The operator is recursively applied on the set until some precision is reached (size of boundary boxes).

We illustrate this by calculating again the “ring” but in a different way. We have already shown how to calculate a ring:

- by giving a conjunction of the constraints to the *constructor of Set*
- by computing two sets separately, one for each constraint, and then by performing *an intersection*

We now create a set for one of the constraint and then contracts this set with the other constraint.

```
double eps=0.01;

// create the set with one of the constraints
NumConstraint c("x", "y", "x^2+y^2<=25");
Set set(c,eps);

// create the second constraint
NumConstraint c2("x", "y", "x^2+y^2>=16");
// create a separator for this constraint
SepFwdBwd sep(c2);
// contract the set with the separator
sep.contract(set,eps);
```

1.11.9 Set Intervals

A set interval [S] (or i-set) [Jaulin 2012] is the given of two sets (S_1, S_2) that represent a lower and upper bound (with respect to the inclusion order) of an unknown set S:

$$S_1 \subseteq S \subseteq S_2.$$

A possible notation (that we use in the code below) for the set interval [S] is: $[S_1, S_2]$.

A set interval can be explicitly represented by an instance of the `SetInterval` class. It can also be implicitly represented by a separator. Let us explain how. A separator S have been used so far to represent a “simple” set (not a set interval) by two complementary contractions C_1 and C_1 , being respectively for the inner and outer part. This means that the set associated to the separator can be seen as the following degenerated set interval:

$$\text{set}(S) = [\text{set}(C_1), {}^c\text{set}(C_2)].$$

where $\text{set}(C)$ designates the set associated to C (the insensitive points).

Now, it is possible to change the status of either the inner or outer contraction to the special value `MAYBE`. This means that the contracted part is not inside or outside the set but potentially inside either one. If we change this way the status of C_1 , the separator is now associated to the set interval:

$$\text{set}(S) = [\emptyset, {}^c\text{set}(C_2)].$$

If we change the status of C_2 , we obtain:

$$\text{set}(S) = [\text{set}(C_1), \mathbb{R}^n].$$

The next example illustrates the use of separators to contract a set interval with the following information:

- the set is enclosed in the circle centered on the origin and of radius 2
- the set encloses n little circles centered on different points and of radius 1

A set interval can be visited exactly like a set and we have used the same `ToVibes` class as above for producing the picture below.

```
double eps=0.001;

// Create the distance function between (x,y)
// and the point (cos(alpha), sin(alpha))
Variable x,y,alpha;
```

(continues on next page)

(continued from previous page)

```

Function f(x,y,alpha,sqr(x-cos(alpha))+sqr(y-sin(alpha)));

// Build the initial box
IntervalVector box(2);
box[0]=Interval(-2,2);
box[1]=Interval(-2,2);

// Create the initial i-set [emptyset,[box]]
SetInterval set(box);

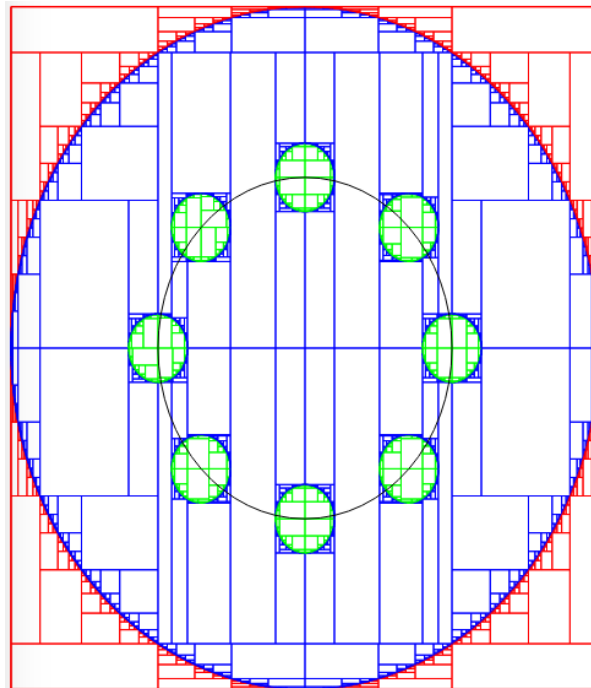
NumConstraint ctr(x,y,sqr(x)+sqr(y)<=4);
// Create a separator for the i-set [emptyset,ctr]
// Initially, the i-set is [ctr,ctr]
SepFwdBwd sep(ctr);

// We set the status of the first contraction to
// "maybe" so that the i-set associated to the separator becomes [emptyset,ctr]
// We contract the set with the separator, i.e.,
// we compute [emptyset,[box]] & [emptyset,ctr]
sep.contract(set,eps,MAYBE,NO);

// Number of points
int n=8;

for (int i=0; i<n; i++) {
  NumConstraint ctr(x,y,f(x,y,i*2*Interval::PI/n)<=0.04);
  SepFwdBwd sep(ctr);
  // We set the status of the second contraction to
  // "maybe" so that the i-set associated to the separator becomes [ctr,R^n]
  sep.contract(set,eps,YES,MAYBE);
}

```



1.12 Strategies

1.12.1 Box Properties

(In release 2.7)

The box (`IntervalVector` class) is the central concept in Ibex and often this structure is too simple and required to be extended.

Consider for example a search tree, such as the one performed by `ibexsolve`. Assume you have several contractors involved in this search that need to calculate at some point the image of the current box by a function f . Imagine also that this function has quite a long expression so that calculating an image takes significant time:

```
// A contractor
class MyCtc : public Ctc {
public:
    void contract(IntervalVector& box) {
        Interval y = f.eval(box); // this line is assumed to be expensive
        // ...
    }
};
```

Note: For simplicity, we assume from now on that f is a fixed object that has been declared somewhere in the context (it could also be given in argument of all objects).

Recalculating this image each time a contractor is called represents a waste of time if the box hasn't changed meanwhile. One would like to store this information in the box. This is the kind of things "box properties" allow to do.

All is based on the `Bxp` interface. This `Bxp` interface allows to extend the simple `IntervalVector` data structure and to make this extension being propagated through a strategy (search tree). The extended box is then visible by all operators involved in the strategy (contractors, bisectors, cell buffers, etc.).

Note that the name of this interface is a trigram (like `Ctc` or `Bsc`) just to encourage programmers to prefix subclasses by `Bxp` (this is a recommended usage). A box property, such as the image of the box by a given function, has to be a subclass of `Bxp` so let us name it `BxpImage`:

```
class BxpImage : public Bxp {
public:
    // to be completed...

    Interval image;
};
```

Of course, this class contains a field named `image` that will store the information. We could also add whatever data needed.

Constructor

It is natural to ask the constructor of `BxpImage` to take a box in argument and to set the `image` field appropriately.

The constructor of the mother class `Bxp` also requires an identifying number. Here is why. A box property is actually a set of *instances* of the `Bxp` interface: if the solver handles 1000 boxes at a given time, every box has its own image, hence its specific instance of `BxpImage`. These 1000 instances represent the same "property" and since there may be other properties attached to the boxes at the same time, we can retrieve a given property thanks to its `id` field. (**Note:** using the class name directly as identifier would be too restrictive as there may be different `BxpImage`

properties attached to different functions f). You can simply fix this identifier to any random `long` number or use the `next_id()` function of Ibex as follows:

```
class BxpImage : public Bxp {
public:
    BxpImage(const IntervalVector& box) :
        Bxp(id), image(f.eval(box)) { }

    // to be completed...

    Interval image;
    static const long id;
};

const long BxpImage::id = next_id();
```

Note: In the case of several `BxpImage` properties (one for each function f) you can store identifying numbers in a map structure (see examples in the Ibex code).

To avoid confusion, we call for now “property value” an instance of the same property. So, `BxpImage` is a property (or a set of properties, one for each f) and the instances of `BxpImage` are property values.

Property update

The next step is to specify how property values are updated when the box is modified. This amounts to implement an `update(...)` function as follows. This function will be called at different points of the strategy, through the *trust chain* principle to be explained further.

```
void update(const BoxEvent& event, const BoxProperties& prop) {
    image = f.eval(event.box);
}
```

Note that a *smarter implementation* is often desired. This is however not enough. You also have to state how the property is transformed when the box is copied (copies occur in a search each time a box is bisected or, e.g., to perform some temporary computations). This is done by implementing the `copy()` function:

```
Bxp* copy(const IntervalVector& box, const BoxProperties& prop) const {
    return new BxpImage(*this); // implicit copy constructor is fine
}
```

Using properties

Now, let us modify the implementation of our contractor. To take benefit of properties, two steps are required. First, we have to override the `add_property` function of the `Ctc` interface. This function is called by all strategies at initialization. This function takes as argument the initial box (of the search) and a container for property values: an instance of `BoxProperties`. This object basically just stores pointers to `Bxp*`, except that it can handle *inter-dependencies*.

Second, we have to override a variant of the `contract` function, which takes in argument not only the box, but also a `ContractContext` object which contains, among other things, the current property values container (again, an instance of `BoxProperties`). The `BoxProperties` class works like a simple map: by using the bracket operator `[...]` with the property id inside the brackets, you get the corresponding property value associated to the box:

```

class MyCtc : public Ctc {
public:

    /* Add the required property inside the map.
     * (This function is automatically called by the search). */
    void add_property(IntervalVector& box, BoxProperties& prop) {
        // if the property is not already in the list
        // (which is possible since another operator requiring it may
        // have already added it)
        if (!prop[BxpImage::id])
            // create an initial property value, and add it
            prop.add(new BxpImage(box));
    }

    /* Contract a box with associated properties. */
    void contract(IntervalVector& box, ContractContext& context) {
        // Get the desired property from the map, by its id
        // (a cast is necessary because all properties are typed Bxp*)
        BxpImage* bxp = (BxpImage*) context.prop[BxpImage::id];

        if (bxp == NULL) {
            // This happens if the property is not present.
            // It is much more safe to handle this case
            // ...
        } else {
            // Obtain directly the image (without recalculating it)
            Interval y = bxp->image;
            // .....
        }
    }
};

```

Lazy update

So far, we have centralized in a unique place the result of the image computation which is already good but not optimal at all. Worse, the running time of our program will likely be longer than without introducing this property! Indeed, the update function will be called basically whenever an operator change the box, which means that the function *f* will be evaluated again and again!

This event-oriented design of a property can be sometimes interesting but, clearly, it does not fit well here.

What we actually want is the function to postpone the evaluation at the latest time, that is, when someone requires it. This is called laziness. This principle can be simply applied here as follows:

```

class BxpImage : public Bxp {
public:
    // The class contains 2 new fields:
    // box: a reference to the box needs to be stored
    // to perform the evaluation at any time.
    // up2date: memorize whether the image is up to date.
    BxpImage(const IntervalVector& box) : Bxp(id), box(box), up2date(false) { }

    void update(const BoxEvent& event, const BoxProperties& prop) {
        // do nothing for the moment!
        up2date = false;
    }
}

```

(continues on next page)

(continued from previous page)

```

Bxp* copy(const IntervalVector& box, const BoxProperties& prop) const {
    return new BxpImage(box, image, up2date);
}

// Return the image of f.
const Interval& get() {
    if (!up2date) {
        image = f.eval(box);
        up2date=true;
    }
    return image;
}

static const long id;

protected:
    BxpImage(const IntervalVector& box, const Interval& image, bool up2date) :
        Bxp(id), box(box), image(image), up2date(up2date) { }

    const IntervalVector& box;
    Interval image;
    bool up2date;
};

```

It is easy to adapt this code so that an update is performed only when the box modification is significant (e.g., when a contraction removes more than 1% of a component width).

Dependencies

It may happen that a property is based on another one. Imagine that you want to create a property that stores the width of the function image (of course, this example is caricatural as the width is not something you really need to store). You can extend the BxpImage class but you can also create a separate property, say BxpImageWidth.

The BxpImageWidth need to “see” the BxpImage property in the update_xxx(...) function. This is why there is also a BoxProperties map in the argument of these functions. Furthermore, we must be sure that the BxpImage is updated before BxpImageWidth. To this end, we simply have to add the identifier of BxpImage in the dependencies of BxpImageWidth. This must be done in the constructor of BxpImageWidth as shown in the following code.

```

class BxpImageWidth : public Bxp {
public:
    BxpImageWidth(const IntervalVector& box) : Bxp(id), w(box.max_diam()) {
        // dependencies is a field inherited from Bxp
        dependencies.push_back(BxpImage::id);
    }

    void update(const BoxEvent& event, const BoxProperties& prop) {
        BxpImage* bxp=(BxpImage*) prop[BxpImage::id];

        if (bxp==NULL) {
            // This happens if the property is not present.
            // It is much more safe to handle this case
            // ...
        } else {

```

(continues on next page)

(continued from previous page)

```

    // note: we lose laziness here
    w=bxp->get().diam();
}
}

double w; // the width
static const long id;
};

```

For the sake of concision, we haven't used laziness in this code. A lazy variant is necessary here.

Trust chain principle

The trust chain principle is the following:

- Property values are always up-to-date when given to argument of a function.

Consider a function that handles properties (e.g., an implementation of the `contract` variant with `BoxProperties`, as above). If the box is modified at several points in the code, it is not necessary to perform updates as long as properties are not used elsewhere. The update can be postponed to the point where property values are transmitted to another function or, on last resort, before returning.

Note that updating all property values can be simply done via the `update` function of `BoxProperties` (this also allows to respect dependencies).

As a consequence, if the function does not modify itself the box (would it calls other functions that potentially modify it), it does not have to perform at all any update of property values.

1.12.2 Bisectors

A bisector is an operator that takes a box $[x]$ as input and returns two sub-boxes $([x]^{(1)}, [x]^{(2)})$ that form a partition of $[x]$, that is, $[x] = [x]^{(1)} \cup [x]^{(2)}$. This partition is obtained by selecting one component $[x_i]$ and splitting this interval at some point.

Each bisector implements a specific strategy for choosing the component. The bisection point in the interval is defined as a *ratio* of the interval width, e.g., a ratio of 0.5 corresponds to the midpoint.

Bisecting each component in turn

(to be completed)

Bisecting the largest component

(to be completed)

Setting different precision for variables

In real-world applications, variables often correspond to physical quantities with different units. The order of magnitude greatly varies with the unit. For example, consider Coulomb's law:

$$F = k_e \frac{q_1 q_2}{r^2}$$

applied to two charges q_1 and q_2 or $\sim 1e-6$ coulomb, with a distance r of $\sim 1e-2$ meter. With Coulomb's constant $\sim 1e10$, the resulting force will be in the order of $1e2$ Newton.

If one introduces Coulomb's equation in a solver, using a bisection that handles variables uniformly, i.e., that uses the same precision value for all of them, is certainly not adequate.

Each bisection can be given a vector of different precisions (one for each variable) instead of a unique value. We just have to give a `Vector` in argument in place of a `double`. For instance, with the round-robin bisection:

```
double _prec[]={1e-8,1e-8,1e-4,1};

Vector prec(4,_prec);

RoundRobin rr(prec);
```

Respecting proportions of a box

If you want to use a relative precision that respects the proportion between the interval widths of an “initial” box, you can simply initialize the vector of precision like this:

```
Vector prec=1e-07*box.diam(); // box is the initial domain
```

The Smear Function

(to be completed)

Smear function with maximum impact

(to be completed)

Smear function with maximal sum of impacts

(to be completed)

Smear function with maximal normalized impact

(to be completed)

Smear function with maximal sum of normalized impacts

(to be completed)

maximal sum of impacts

1.12.3 Cell buffers

(to be completed)

Cell Stack

(to be completed)

Cell Heap

(to be completed)

1.13 References

NOTE: This page is *under construction* and a lot of references are still missing.

1.13.1 Articles in Journal and Conferences

This page is under construction

	Author(s)	Title	Journal/Conference	Year	Link
[Jaulin & Desrochers 2014]	L. Jaulin, B. Desrochers	<i>Introduction to the Algebra of Separators with Application to Path Planning</i>	Engineering Applications of Artificial Intelligence	2014	PDF
[Araya et al. 2014]	I. Araya, G. Trombettoni, B. Neveu and G. Chabert	<i>Upper Bounding in Inner Regions for Global Optimization under Inequality Constraints</i>	Journal of Global Optimization	2014	PDF
[Jaulin 2012]	L. Jaulin	<i>Solving set-valued constraint satisfaction problem</i>	Computing	2012	PDF
[Araya et al. 2012]	I. Araya, G. Trombettoni, and B. Neveu	<i>A Contractor Based on Convex Interval Taylor</i>	CPAIOR	2012	PDF
[Trombettoni et al. 2011]	G. Trombettoni, I. Araya, B. Neveu and G. Chabert	<i>Inner Regions and Interval Linearizations for Global Optimization</i>	AAAI	2011	PDF
[Chabert & Beldiceanu 2010]	G. Chabert and N. Beldiceanu	<i>Sweeping with Continuous Domains</i>	CP	2010	PDF
[Ninin & Messine, 2009]	J. Ninin and F. Messine	<i>An Automatic Linear Reformulation Technique Based on Afne Arithmetic</i>	ISMP	2009	Slides
[Chabert & Jaulin, 2009]	G. Chabert and L. Jaulin	<i>Contractor Programming</i>	Artificial Intelligence	2009	PDF
[Chabert & Jaulin, 2009bis]	G. Chabert and L. Jaulin	<i>Hull Consistency under Monotonicity</i>	CP	2009	PDF
[Trombettoni & Chabert 2007]	G. Trombettoni and G. Chabert	<i>Constructive Interval Disjunction</i>	CP	2007	PDF
[Bessiere & Debruyne 2004]	C. Bessiere and R. Debruyne	<i>Theoretical Analysis of Singleton Arc Consistency</i>	ECAI	2004	PDF
[Benhamou & Granvilliers, 2006]	F. Benhamou and L. Granvilliers	<i>Continuous and Interval Constraints</i>	Handbook of Constraint Programming	2006	
[Bessiere 2006]	C. Bessiere	<i>Constraint Propagation</i>	Handbook of Constraint Programming	2006	
108			Programming	Chapter 1. The Core Library	
[Benhamou et al. 1999]	F. Benhamou, F. Goualard, L. Granvilliers	<i>Revising Hull and Box Consistency</i>	ICLP	1999	

1.13.2 Books

	Author(s)	Title	Publisher	Year
[Jaulin et al. 2001]	L. Jaulin, M. Kiefer, O. Didrit and E. Walter	<i>Applied Interval Analysis</i>	Springer	2001
[Kearfott 1996]	R.B. Kearfott	<i>Rigorous Global Search: Continuous Problems</i>	Springer	1996
[Hansen 1992]	E.R. Hansen	<i>Global Optimization using Interval Analysis</i>	Marcel Dekker	1992
[Neumaier 1990]	A. Neumaier	<i>Interval Methods for Systems of Equations</i>	Cambridge University Press	1990
[Moore 1966]	R. Moore	<i>Interval Analysis</i>	Prentice-Hall	1966

1.14 A complete Example: SLAM with outliers

1.14.1 Introduction

The goal of this example is to implement a simple contractor strategy for a SLAM problem with the IBEX library. SLAM means *simultaneous localization and map building* and is a classical problem in mobile robotics.

We will see that contractor programming with Ibex basically amounts to:

- enter your mathematical model using *Functions* and *Constraints*;
- build basic contractors (CtcFwdBwd in general) with respect to the equations;
- apply operators to these contractors to yield new (more sophisticated) contractors.

The code we build here will eventually involve 5 different contractors:

- CtcFwdBwd
- CtcCompo
- CtcFixPoint
- CtcQInter
- CtcInverse.

We shall implement a strategy that is similar to a predictor-corrector approach (like the Kalman filter for instance) in the sense that we also use odometry and observation to reduce the uncertainty on the robot's position. However, both information are considered on the same footing and there is no distinction such as *prediction* versus *correction*. They are just contractors that can be used in any order and we will even calculate a fixpoint of them (so the strategy is not a *recursive* filter).

note For the sake of simplicity, we shall always use dynamic *allocation*:

```
MyClass* x = new MyClass(...)
```

just to avoid potential memory fault when pointing to temporary objects. Of course, all these objects should be deallocated afterwards.

Download

The full code can be found under `.../examples/slam` (this subfolder is included in the Ibex package).

Problem description

The goal is to characterize the trajectory of an autonomous robot by enclosing in a box its position $x[t]$ for each time step $t=0 \dots T$.



We have no direct information on its position (including the initial one) but the robot measures at each time step:

- its distance from a set of N fixed beacons ($\rightarrow N$ measurements) as if it was equipped with a telemeter;
- its “speed” (delta) vector $v[t]=x[t+1]-x[t]$.

Each measurement is subject to uncertainty: *distances and speed vector* but also the position of the beacons, that is supposed to be measured a priori.

Furthermore, we shall consider outliers.

First of all, let us assume that the measurements are all simulated in a separate unit. The header file of this unit contains:

```
/*===== data =====*/
extern const int N;           // number of beacons
extern const int T;           // number of time steps
extern const double L;        // limit of the environment (the
                               // robot is in the area [0,L]x[0,L])
extern const int NB_OUTLIERS; // maximal number of outliers per
                               // time units
```

(continues on next page)

(continued from previous page)

```

extern IntervalMatrix beacons;      // (a Nx2 matrix) beacons[i] is the
                                   // position (x,y) of the ith beacon

extern IntervalMatrix d;            // (a TxN matrix) d[t][i]=distance
                                   // between x[t] and the ith beacon

extern IntervalMatrix v;            // (a Tx2 matrix) v[t] is the delta
                                   // vector between x[t+1] and x[t].
/*=====*/

```

1.14.2 First strategy (no outlier)

First, we consider no outlier. A simple strategy consists in :

- creating a contractor for each measurement,
- calling all these contractors in sequence (composition)
- performing a fixpoint loop

Let us start by creating contractors for measurements, that is, those related to equations.

Entering equations and functions

A measurement is an equation.

To enter an equation in Ibex, we use the `NumConstraint` class (see [Constraints](#)). A `NumConstraint` object contains a mathematical condition, or “constraint”.

To define a constraint mathematically, we must specify how many variables it relates and in which order these variables must be taken.

That is why we need to create first some `Variable` objects. But keep in mind that these objects are just a C++ trick for the only purpose of declaring a constraint.

Once declared, a constraint is self-contained and depends on nothing else.

Example: For creating the equations:

$$\forall t < T, x[t+1] - x[t] = v[t]$$

The corresponding code in Ibex is:

```

Variable x(T,2); // create a Tx2 variable

for (int t=0; t<T; t++) {
    if (t<T-1) {
        NumConstraint* c=new NumConstraint(x,x[t+1]-x[t]=v[t]);
        ...
    }
}

```

note Here, `v` is not a variable but a constant (see `data.h`).

Sometimes, different constraints are based on the same pattern. It is then often convenient to declare first a `Function` object.

Example: For distance constraints, we may first declare the distance function:

```
// create the distance function beforehand
Variable a(2);           // "local" variable
Variable b(2);
Function dist(a,b,sqrt(sqr(a[0]-b[0])+sqr(a[1]-b[1])));
```

and then the equation for each time step and each beacon:

```
for (int t=0; t<T; t++) {
  for (int i=0; i<N; i++) {
    NumConstraint* c=new NumConstraint(
      x,dist(x[t],beacons[i])=d[t][i]);
    ...
  }
}
```

Creating basic contractors

We can create now contractors.

To create a contractor with respect to an equation we use the `CtcFwdBwd` class (see *Forward-Backward*).

Example: With the constraint $x=1$:

```
Variable x;
NumConstraint* c=new NumConstraint(x,x=1);
Ctc* ctc=new CtcFwdBwd(*c);
```

Note: The `Ctc` prefix indicates that this class is a contractor (i.e., it can be composed with other contractors). `Ctc` is also the name of the generic contractor class.

Combining contractors

We are now ready to build our first strategy. We create all the contractors and push them in a vector `ctc` (this vector will be necessary for the composition):

```
vector<Ctc*> ctc;
for (int t=0; t<T; t++) {
  vector<Ctc*> cdist;
  for (int b=0; b<N; b++) {
    // Push the contractor corresponding to
    // the detection of beacon n°b at time t
    NumConstraint* c=new NumConstraint(
      x,dist(transpose(x[t]),beacons[b])=d[t][b]);
    ctc.push_back(new CtcFwdBwd(*c));
  }

  if (t<T-1) {
    // Push the contractor corresponding to
    // the speed measurement at time t
    NumConstraint* c=new NumConstraint(x,x[t+1]-x[t]=transpose(v[t]));
    ctc.push_back(new CtcFwdBwd(*c));
  }
}
```

Now, we can create the composition of all these contractors using `CtcCompo` (the vector `ctc` being given in argument) and a fixpoint of the latter using `CtcFixPoint`. This gives:

```
// Composition
CtcCompo compo(ctc);

// FixPoint
CtcFixPoint fix(compo);
```

We are done. We just have to call the top-level contractor on the initial box.

```
// the initial box [0,L]x[0,L]x[0,L]x[0,L]
IntervalVector box(T*2, Interval(0,L));

cout << endl << "  initial box =" << box << endl;
fix.contract(box);
cout << endl << "  final box =" << box << endl << endl << endl;
```

Result

The execution shows that the final box contains the real trajectory:

```
initial box =([0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10])

final box =([8.592079632938807, 9.009246227143752] ; [0.4364101205434934, 0.
↪8936036705218675] ; ... )
```

The real positions are:

$$\begin{array}{ll} x_0 = 8.806965820867086 & y_0 = 0.6934996231894474 \\ x_1 = 8.240950936914649 & y_1 = 1.517894644489497 \\ x_2 = 8.553965973529273 & y_2 = 0.5681464742605957 \\ \vdots & \vdots \end{array}$$

1.14.3 Second strategy (with outliers)

We consider now that at most NB_OUTLIERS outliers may occur for each time step.

To contract rigorously despite of outliers, we must use the “q-intersection” operator that basically consider all possible combinations of (N-NB_OUTLIERS) contractors among N:

Ibex provides the CtcQInter contractor.

Q-intersection

We must only place all the contractors related to the same time step in another temporary vector (called `cdist`) and give this vector in argument of `CtcQInter`:

Let us see what happens if we do this.

Let us replace:

```
if (t<T-1) {
  // Push the contractor corresponding to
  // the speed measurement at time t
  NumConstraint* c=new NumConstraint(x,x[t+1]-x[t]=v[t]);
  ctc.push_back(new CtcFwdBwd(*c));
}
```

by:

```
// create a temporary subvector
// for collecting all the contractors corresponding
// to the detections at time t
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    NumConstraint* c=new NumConstraint(
        x,dist(x[t],beacons[b])=d[t][b]);
    // push the detection of beacon n°b
    cdist.push_back(new CtcFwdBwd(*c));
}
// Push the q-intersection of all
// the contractors in "cdist" in the main
// vector "ctc"
ctc.push_back(new CtcQInter(cdist,N-NB_OUTLIERS));
```

Problem: the program runs almost endlessly! ... Why?

... because the q-intersection runs exponentially in the dimension of the input box, which is 2T.

Of course, the implementation should take advantage of the fact that only 2 variables are actually impacted. But the current code is not optimized in this way.

Anyway, it is often necessary to apply a contractor strategy to only a subset of variables (here, to the two components of $x[t]$).

For this end, we will make use of the *Inverse contractor*.

Projection using the inverse contractor

Applying the q-intersection on the subset of variables x_t amounts to apply the inverse of this contractor by the projection function $x \mapsto x[t]$.

We replace:

```
// create a temporary subvector
// for collecting all the contractors corresponding
// to the detections at time t
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    NumConstraint* c=new NumConstraint(
        x,dist(x[t],beacons[b])=d[t][b]);
    // push the detection of beacon n°b
    cdist.push_back(new CtcFwdBwd(*c));
}
// Push the q-intersection of all
// the contractors in "cdist" in the main
// vector "ctc"
ctc.push_back(new CtcQInter(cdist,N-NB_OUTLIERS));
```

By:

```
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    // Create the distance constraint with 2
    // (instead of 2*T) variables
    Variable xt(2);
```

(continues on next page)

(continued from previous page)

```

NumConstraint* c=new NumConstraint (
    xt,dist(xt,beacons[b])=d[t][b]);
cdist.push_back(new CtcFwdBwd(*c));
}

// q-intersection with 2 variables only
CtcQInter* q=new CtcQInter(cdist,N-NB_OUTLIERS);
// Push in the main vector "ctc" the application
// of the latter contractor to x[t]
ctc.push_back(new CtcInverse(*q,*new Function(x,x[t])));

```

And now, the program terminates instantaneously. With NB_OUTLIERS set to 1, the display shows a slightly larger box:

```

initial box =([0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10])

final box =([8.542599451371126, 9.032225305125761] ; [0.3807126686643456, 1.
↪002241041162326] ; ...)

```

1.14.4 Third strategy (how can this scale?)

The program we have proposed so far does not really scale. For example, setting `T=200000` in `data.cpp` will make the program run for a long time and crash after a memory overflow. We see now a more efficient variant. This variant, however, will be less concise. We will also partially lose the elegance of contractor programming. In particular, we will do ourselves the loop that compose the contractors as time increases. But, after all, a programming language is always a compromise between efficiency and elegance so if you really look for efficiency, you should accept to sacrifice a little bit of elegance.

Let us first explain why the current program does not scale. In the program, we build a `NumConstraint` object for most of the contractors and each of these `NumConstraint` objects builds (silently) a `Function` object. For instance, by writing:

```
NumConstraint* c=new NumConstraint(xt,dist(xt,beacons[b])=d[t][b]);
```

The following function is created somewhere

$$x \mapsto \text{dist}(x[t], \text{beacons}[b]) - d[t][b]$$

Now, you must be aware that the construction of `Function` objects is both time and memory consuming. The good point however is that, once built, these objects are fast to use (evaluation, gradient, etc.).

The `CtcQInter` objects are also costly because each contains a set of `N` references.

It is clear in our context that the we keep on creating the same contractors again and again, with only one of the parameters changing with time (in our previous example, `d[t][b]`). The key idea is to factorize all these contractors and create a “parametrized” contractor where the time is set dynamically. Let us start with the detection constraints.

Detection contractor

The following class declares a contractor for the detection of a given beacon. This is a handcrafted contractor so we need to create a new class that extends `Ctc` and implements the `contract()` function. Time (contrary to the beacon number) is not set at construction so that one instance of this contractor can be used for any time:

```

/*
 * Contractor for the detection of beacon n°b.
 *
 * This is a contractor parametrized by the time "t".
 * It means that a call to contract() must be
 * preceded by a call to set_time(t).
 */
class Detection : public Ctc {
public:
    /*
     * The contractor is for a specific beacon "b" which
     * is specified in argument of the constructor.
     */
    Detection(int b) : Ctc(2), b(b) {
        Variable x(2);
        // This function will be created once for the T time steps!
        dist = new Function(x, sqrt(sqr(x[0]-beacons[b][0])+sqr(x[1]-
↪beacons[b][1])));
    }

    /*
     * Allow to set the time dynamically
     */
    void set_time(int t) {
        this->t=t;
    }

    void contract(IntervalVector& x) {
        // by simplicity, we call the backward
        // operator on the function directly
        dist->backward(d[t][b],x);
    }

protected:
    int b;           // beacon number
    int t;           // time number
    Function* dist;  // distance function
};

```

Speed contractor

We do the same with the second set of time-dependant constraints, namely the “speed” or “delta” constraints between two consecutive time steps.

```

/*
 * Contractor for the "speed" constraint.
 *
 * This is a contractor parametrized by the time "t".
 * It means that a call to contract() must be
 * preceded by a call to set_time(t).
 */
class Speed : public Ctc {
public:
    Speed() : Ctc(2) {
        Variable a(2);

```

(continues on next page)

(continued from previous page)

```

        Variable b(2);
        delta = new Function(a,b,b-a);
    }

    void contract(IntervalVector& x) {
        delta->backward(v[t],x);
    }

    void set_time(int t) {
        this->t=t;
    }

protected:
    int t;
    Function* delta;
};

```

Scan contractor (q-intersection)

Again, we create a parametrized contractor for the q-intersection of the N detections occurring at a given time. This set of measurements somehow forms a scanning of the environment so we name this contractor `Scan`:

```

/*
 * Scanning contractor that aggregates
 * the N detections occurring at a given time t.
 */
class Scan : public Ctc {
public:
    Scan() : Ctc(2) {

        // The N detections
        detect = new Detection*[N];

        // The q-intersection is created as before,
        // using a temporary vector "cdist"
        vector<Ctc*> cdist;
        for (int b=0; b<N; b++) {
            cdist.push_back(detect[b]=new Detection(b));
        }
        qinter = new CtcQInter(cdist,N-NB_OUTLIERS);
    }

    void contract(IntervalVector& x) {
        qinter->contract(x);
    }

    void set_time(int t) {
        // we set the time of each sub-contractor
        for (int i=0; i<N; i++)
            detect[i]->set_time(t);
    }

protected:
    Detection** detect;
    CtcQInter* qinter;
}

```

(continues on next page)

(continued from previous page)

```

    int t;
};

```

Trajectory

We can create now the final contractor that calls 2T times an instance of the Scan and Speed contractors (there is only one instance of each). A call to `set_time()` precedes every call to `contract()`. Note that a fix-point would not be reasonable here.

```

/*
 * The contractor for the whole trajectory.
 *
 * It will contract every positions of the robot using
 * detections and speed data, in a single pass (no
 * fixpoint).
 */
class Trajectory : public Ctc {
public:
    Trajectory() : Ctc(2*T) { }

    void contract(IntervalVector& x) {
        for (int t=0; t<T; t++) {
            // Get a copy of the domain of x[t]
            IntervalVector xt=x.subvector(2*t,2*t+1);
            // Set the time
            scan.set_time(t);
            // Contract with the scanning
            scan.contract(xt);
            // Update the box "x" with the new domain for x[t]
            x.put(2*t,xt);

            if (t<T-1) {
                // Get a copy of the domain of x[t] and x[t+1]
                IntervalVector xtt1=x.subvector(2*t,2*(t+1)+1);
                // Set the time
                speed.set_time(t);
                // Contract with the speed vector
                speed.contract(xtt1);
                // Update the box
                x.put(2*t,xtt1);
            }
        }

        Scan scan;
        Speed speed;
    };
};

```

1.15 Do it Yourself!

The examples in this page are presented under the form of “labs” so that you can use them for practicing.

The complete source codes are available under the `examples/` folder.

In these labs, we use Vibes to plot boxes but you can easily adapt the code to use your own graphical tool.

Fast instructions for installing and using Vibes are given [here](#).

1.15.1 Set image

The complete code can be found here: `examples/lab/lab1.cpp`.

Introduction

The goal of this first lab is to calculate the image of a box by a function using interval arithmetic:

If we denote by f the function and $[x]$ the initial box, then the set S to calculate is:

$$S := \{f(x), x \in [x]\}.$$

Applying directly an *interval evaluation* of the function f to $[x]$ will give a single box that only represents an enclosure of S .

To fight with the wrapping effect, we will split $[x]$ into smaller boxes and evaluate the function with every little box as argument. This will result in a better description of the set that will eventually converge to S as the size of the boxes tend to zero.

This will consist in three tasks:

- creating the function f
- creating the initial box $[x]$ and splitting it into small boxes
- evaluating the function of every boxes
- plotting the results

Question 1

Create in the `main` the function

$$f : (x, y) \in \mathbb{R}^2 \mapsto \begin{pmatrix} \sin(x + y) \\ \cos(x + 0.9 \times y) \end{pmatrix}.$$

Question 2

Create the box $([x],[y])=([0,6],[0,6])$ and split each dimension into n slices, where n is a constant.

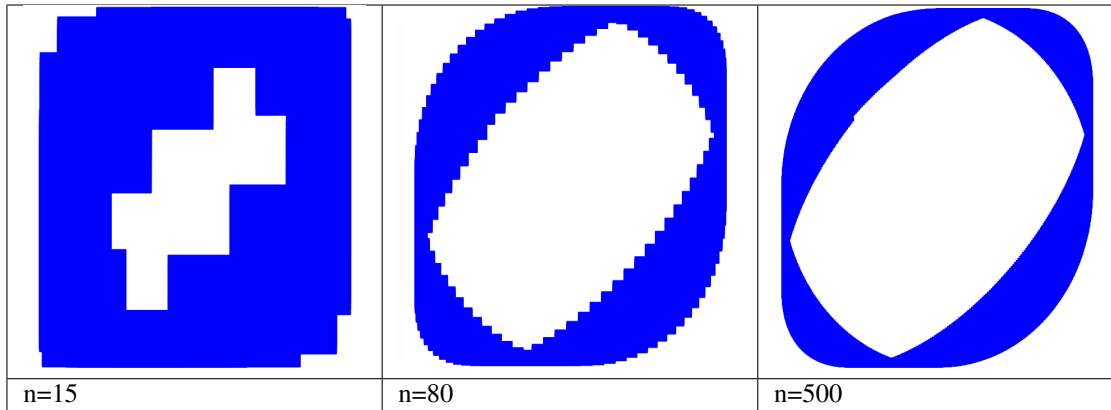
Question 3

Evaluate the function on each box and plot the result with Vibes.

Question 4

Compare the result with $n=15$, $n=80$ and $n=500$.

You should obtain the following pictures:



1.15.2 Set inversion (basic)

The complete code can be found here: `examples/lab/lab2.cpp`.

Introduction

The goal of this lab is to program **Sivia** (*set inversion with interval analysis*) [Jaulin & Walter 1993] [Jaulin 2001], an algorithm that draws a paving representing a set E defined implicitly as the preimage of an interval $[z]$ by a non-linear function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ (here $n=2$).

$$E := \{(x, y) \in \mathbb{R}^2, f(x, y) \in [z]\}.$$

The Sivia algorithm performs a recursive exploration of the initial box by applying the following steps:

- **inner test**: if the image of $([x],[y])$ by f is a subset of $[z]$, the box is painted in green;
- **outer test**: if the image does not intersect $[z]$, the box is painted in red;
- if none of these test succeeds and if $([x],[y])$ has a maximal diameter greater than ε , the box is split and the procedure is recursively called on the two subboxes.

Question 1 (Initialisation)

Create the `Function` object that represents

$$(x, y) \mapsto \sin(x + y) - 0.1 \times x \times y.$$

and the initial bounding box $([-10,10],[-10,10])$.

Question 2 (Initialisation)

We shall use a `stack` for implementing the recursivity. This stack is a container that will be used to store boxes.

Create a `C++ stack` and set the precision of bisection to 0.1.

Push the initial box in the stack. Define the image interval $[z]$ and initialize it to $[0,2]$.

Question 3

Create the loop that pop boxes from the stack until it is empty. Define a local variable `box` to be the current box (the one on top of the stack).

Hint: use the `top()` and `pop()` functions of the `stack` class.

Question 4

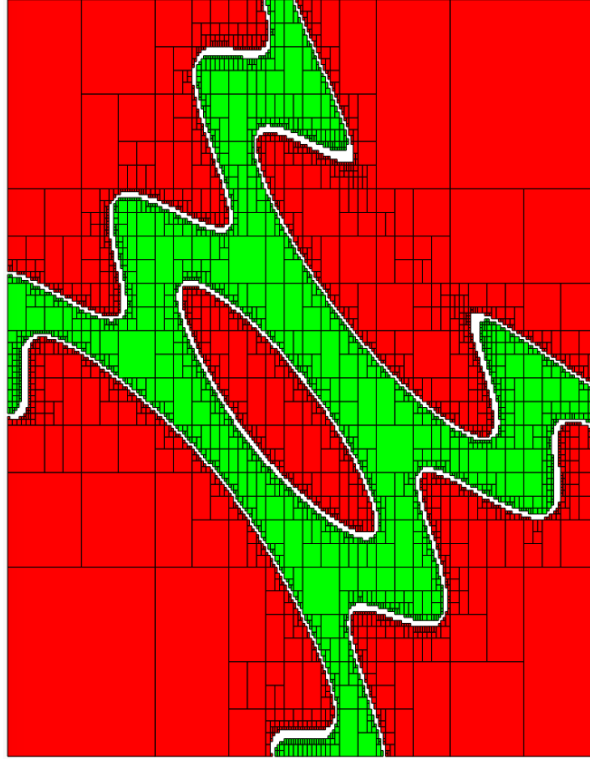


Fig. 5: **Sivia (basic variant)**. Result obtained with $f(x,y)=\sin(x+y)-0.1xy$ and $[z]=[0,2]$, by simply alternating an evaluation and bisection phase. For a precision of $\varepsilon = 0.1$, the number of boxes generated by the algorithm is **11891**.

Implement the inner test (see above).

Hint: use `is_subset`.

Question 5

Implement the outer test (see above).

Hint: use `intersects`.

Question 6

If none of these test succeeds, split the box. We will split the box on the axis of its largest size. Finally, the two subboxes are pushed on the stack.

Hint: use `extr_diam_index` and `bisect`.

1.15.3 Set inversion (with contractors)

The complete code can be found here: `examples/lab/lab3.cpp`.

Introduction

We will improve the **Sivia** algorithm by replacing in the loop the inner and outer tests by contractions. This leads to a more compact paving and a smaller number of boxes (see figure below).

The first part of the code is unchanged:

```
int main() {
    vibes::beginDrawing ();
    vibes::newFigure("lab3");

    // Create the function we want to apply SIVIA on.
    Variable x,y;
    Function f(x,y, sin(x+y)-0.1*x*y);

    // Build the initial box
    IntervalVector box(2);
    box[0]=Interval(-10,10);
    box[1]=Interval(-10,10);

    // Create a stack (for depth-first search)
    stack<IntervalVector> s;

    // Precision (boxes of size less than eps are not processed)
    double eps=0.1;

    // Push the initial box in the stack
    s.push(box);
    ...
}
```

The idea is to contract the current box either with respect to the constraint

$$f(x, y) \in [z],$$

in which case the contracted part will be painted in red, or

$$f(x) \notin [z],$$

in which case the contracted part will be painted in green.

Given a contractor c , the contracted part is also called the *trace* of the contraction and is defined as $[x] \setminus c([x])$.

Question 1

Build forward-backward contractors for the four constraints (see [the tutorial](#)):

$$f(x) < 0, \quad f(x) \geq 0, \quad f(x) \leq 2 \quad \text{and} \quad f(x) > 2.$$

Question 2

Thanks to the [composition](#), build a contractor w.r.t. $f(x) \in [0, 2]$.

Similarly, thanks to the union, build a contractor w.r.t. $f(x) \notin [0, 2]$.

Question 3

Create the function `contract_and_draw` with the following signature:

```
void contract_and_draw(Ctc& c, IntervalVector& box, const char* color);
```

This function must contract the box `box` in argument with the contractor `c` and plot the trace of the contraction (see above) with Vibes, with the specified color `color`.

Hints: use the [diff](#) function of `IntervalVector` to calculate the set difference between two boxes.

Question 4

Replace in the loop the inner/outer tests by contractions.

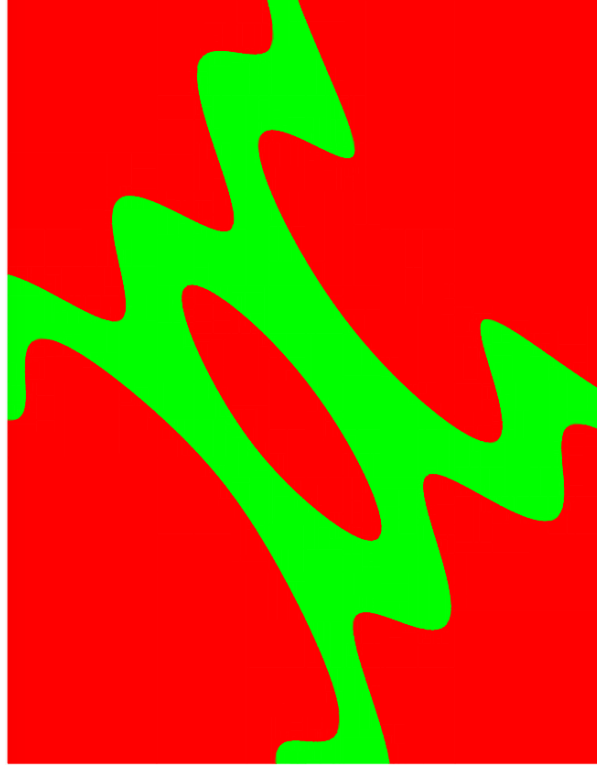


Fig. 6: **Sivia (with contractors)**. Result obtained with $f(x,y)=\sin(x+y)-0.1xy$ and $[z]=[0,2]$. For a precision of $\varepsilon = 0.1$, the number of boxes generated by the algorithm is **5165**.

1.15.4 Set Inversion (using “Sets”)

The complete code can be found here: `examples/lab/lab4.cpp`.

Introduction

The purpose of this exercise is just to get familiar with the structure proposed in Ibex for representing sets (or pavings).

The set inversion is naturally one of the main features proposed in this part of the library. We will solve the same problem as before but this time with the `Set` class directly. This will take only a few lines of code.

Give first a look at the [documentation on sets](#).

Question 1

Create the function $(x, y) \mapsto \sin(x + y) - 0.1 \times x \times y$ and the *forward-backward separator* associated to the constraint

$$0 \leq f(x, y) \leq 2.$$

Question 2

Build the initial set $[-10,10] \times [-10,10]$ and contract it with the separator with a precision of 0.1.

Question 3

Plot the set with Vibes using a `SetVisitor`.

Solution: copy-paste the code given [here](#).

1.15.5 Parameter Estimation

The complete code can be found here: `examples/lab/lab5.cpp`.

Introduction

This exercise is inspired by this [video](#).

The problem is to find the values of two parameters (p_1, p_2) of a physical process that are consistent with some measurements. Measurements are subject to error and we want a guaranteed enclosure of all the feasible parameters values.

The physical process is modeled by a function $f_{p_1, p_2} : t \mapsto y$ and a measurement is a couple of input-output (t_i, y_i) . We assume the input has no error. The error on the output is represented by an interval.

The model is:

$$f_{p_1, p_2} : t \mapsto 20 \exp(-p_1 t) - 8 \exp(-p_2 t).$$

We have the following series of measurements:

t	y
1	[4.5, 7.5]
2	[0.67, 4.6]
3	[-1, 2.8]
4	[-1.7, 1.7]
5	[-1.9, 0.93]
6	[-1.8, 0.5]
7	[-1.6, 0.24]
8	[-1.4, 0.09]
9	[-1.2, 0.0089]
10	[-1, -0.031]

Question 1

Build the function f as a mapping of 3 variables, p_1 , p_2 and t .

Question 2

Build two interval vectors \mathbf{t} and \mathbf{y} of size 10 that contain the measurements data (even if the input has no error, we will enter times as degenerated intervals).

Hint: build interval vectors *from array of double*.

Question 3

Build a system using a *system factory*. The system must contain the 10 constraints that represent each measurements and the additional bound constraints on the parameters:

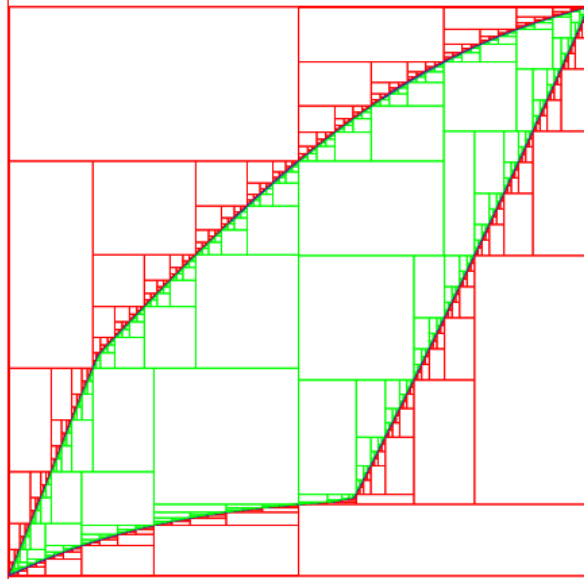
$$0 \leq p_1 \leq 1, \quad 0 \leq p_2 \leq 1.$$

Question 4

Calculate the parameter values using set inversion (see [lab n°4](#)). You should obtain the following picture:

1.15.6 Parameter Estimation (advanced)

The complete code can be found here: `examples/lab/lab6.cpp`.



Introduction

This lab is a follow-up of the previous one.

We now introduce uncertainty on the input variable t . However, we will try somehow to make our parameter estimation *robust* with respect to this uncertainty. This means that the values of our parameters should be consistent with our output whatever is the actual value of the input. Mathematically, we require p_1 and p_2 to respect the following constraints.

$$\forall i, \quad \forall t \in [t_i], \quad f(p_1, p_2, t) \in [y_i].$$

As a contractor-oriented library, Ibex does not provide quantifiers at the modeling stage. This means that you cannot write directly a constraint like this one. You have to build contractors and apply “quantifiers” on contractors. Read the documentation about *contractors and quantifiers*.

Question 1

Like in the previous lab, create the function and the vector of measurements.

Then, define a constant `delta_t` (the uncertainty on time) and *inflate* the vector of representing input times by this constant:

Question 2

We will use the *generic constructor* of `CtcForAll`. Create a bitset that will indicate among the arguments of the function f which ones will be treated as variables and which ones will be treated as quantified parameters.

Question 3

Create the inner and outer contractor for the “robust” parameter estimation problem. Set the splitting precision ε of the parameter to `tdelta/5`.

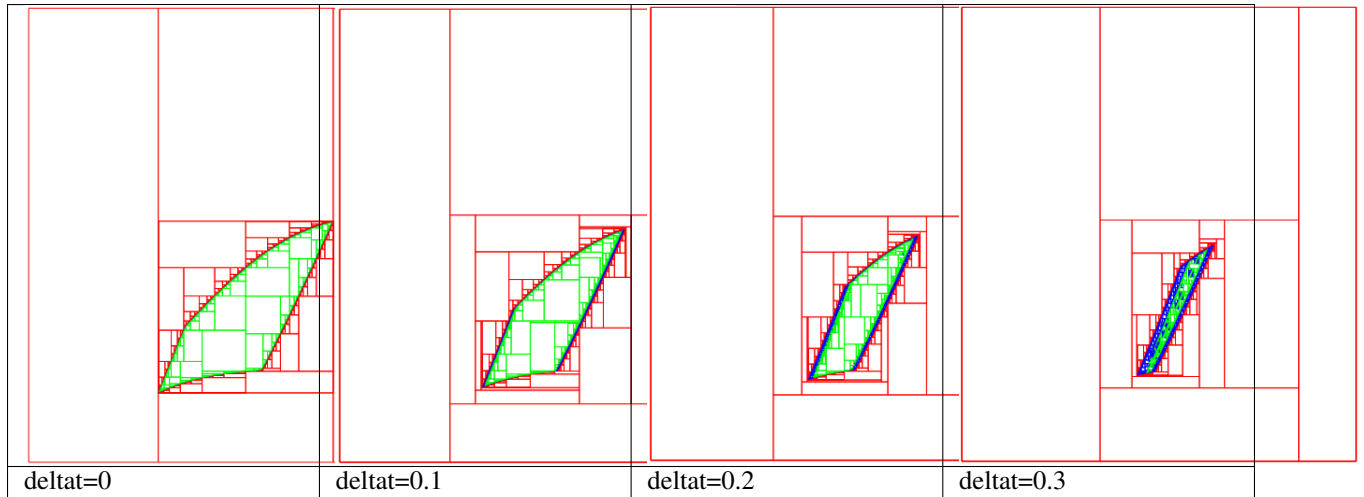
Hints: follow the same idea as in the contractor variant of set inversion (see [lab n°3](#)). Also, we need here to deal with n inner/outer contractors. To perform the composition/union of several contractors, see [here](#).

Question 4

- Create a *separator from the two contractors*;
- Create a *set from the box* $[0,1] \times [0,1]$;

- *Contract the set* with the separator.

The pictures below show the results obtained for increasing values of `deltat` (0,0.1,0.2 and 0.3). As expected, the larger the error on input, the smaller the set of feasible parameters.



1.15.7 Stability

The complete code can be found here: `examples/lab/lab7.cpp`.

Introduction

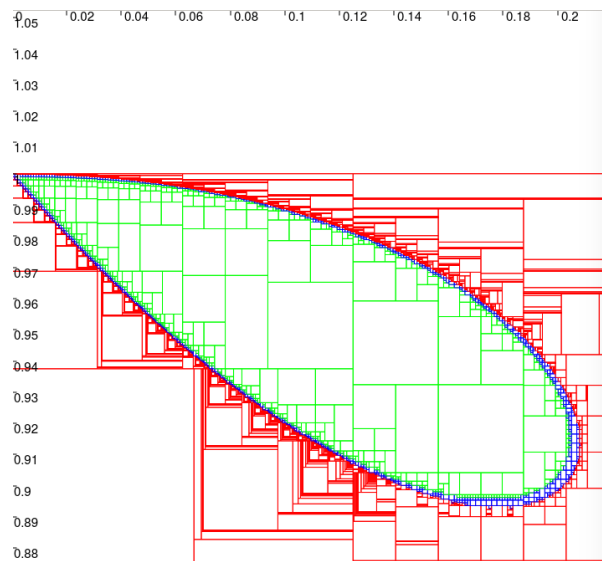
The goal of this lab is to cast a classical problem in control theory into a set inversion problem.

We have a dynamical system $y(t)$ governed by the following linear differential equation:

$$y^{(4)}(t) + ay^{(3)}(t) + by^{(2)}(t) + (1 - b)y'(t) + ay(t) = 0.$$

where a and b are two unknown parameters.

Our goal is to find the set of couples (a,b) that makes the origin $y=0$ stable. It is depicted in the figure:



Hint: apply the Routh-Hurwitz criterion to the characteristic polynomial of the system.

1.15.8 Unstructured Mapping

The complete code can be found here: `examples/lab/lab8.cpp`.

Introduction

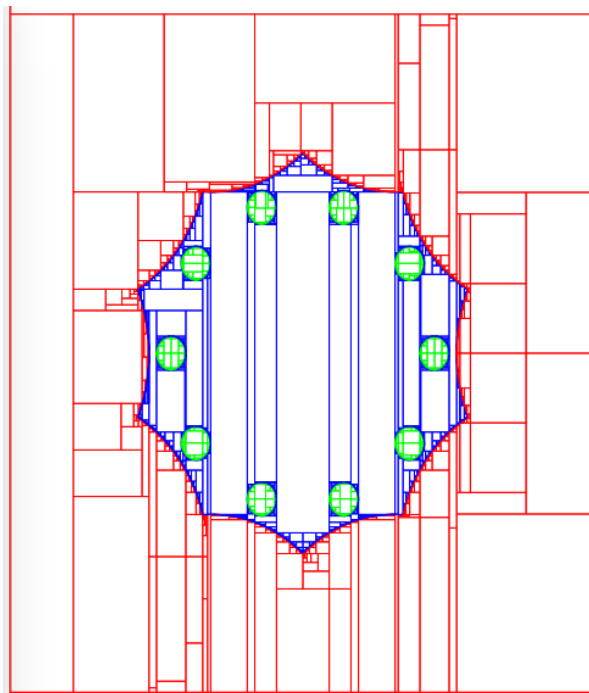
A robot is moving in a rectangular area $[-L,L] \times [-L,L]$ (with $L=2$) and tries to build a map while avoiding obstacles. The map is precisely the shape of obstacles inside the area.

The only information we have is when its euclidian distance to an obstacle get smaller than 0.9. It receives an alert, a vector of measurements which contain its own position (named `x_rob` and `y_rob` in the code) and the position of the detected obstacle point (named `x_obs` and `y_obs` in the code). We also know that all the points that are less distant than 0.1 from the detected point belong to the obstacle.

The robot has a series of $n=10$ measurements. The goal is to build an approximation of the map using *set intervals*.

You can first copy-paste the data:

```
// positions of the robot
double x_rob[n]={2, 1.61803, 0.618034, -0.618034, -1.61803, -2, -1.61803, -0.618034,
↪0.618034, 1.61803};
double y_rob[n]={0, 1.17557, 1.90211, 1.90211, 1.17557, 0, -1.17557, -1.90211, -1.
↪90211, -1.17557};
// positions of the obstacle point
double x_obs[n]={0.9, 0.728115, 0.278115, -0.278115, -0.728115, -0.9, -0.728115, -0.
↪278115, 0.278115, 0.728115};
double y_obs[n]={0, 0.529007, 0.855951, 0.855951, 0.529007, 0, -0.529007, -0.855951, -
↪0.855951, -0.529007};
```



2.1 IbexSolve

This page describes IbexSolve, the plugin installed with the `--with-solver` option.

2.1.1 Getting started

IbexSolve is a end-user program that solves a system of nonlinear equations rigorously (that is, it does not lose any solution and return each solution under the form of a small box enclosing the true value). It resorts to a unique black-box strategy (whatever the input problem is) and with a very limited number of parameters. Needless to say, this strategy is a kind of compromise and not the best one for a given problem.

Note that this program is based on the *generic solver*, a C++ class that allows to build a more customizable solver.

You can directly apply this solver on one of the benchmark problems distributed with Ibex. The benchmarks are all written in the *Minibex syntax* and stored in an arborescence under `plugins/solver/benchs/`.

Open a terminal, move to the `bin` subfolder and run IbexSolve with, for example, the problem named `kolev36` located at the specified path:

```
~/ibex-2.6.0$ cd bin
~/ibex-2.6.0/bin$ ./ibexsolve ../plugins/solver/benchs/others/kolev36.bch
```

The following result should be displayed:

```
***** setup *****
file loaded:      ../plugins/solver/benchs/others/kolev36.bch
output file:      ../plugins/solver/benchs/others/kolev36.mnf
*****

running.....

solving successful!
```

(continues on next page)

(continued from previous page)

```
number of inner boxes:      1
number of boundary boxes:   0
number of unknown boxes:    0
number of pending boxes:    0
cpu time used:              0.0640001s
number of cells:            25

results written in ../plugins/solver/benchs/others/kolev36.mnf
```

The number of “inner boxes” correspond to the number of solutions found (there is just one here). To see the solution, use the option `-s`.

In the report, the “number of cells” correspond to the number of hypothesis (bisections) that was required to solve the problem.

2.1.2 Options

-e<float>, -eps-min=<float>	Minimal width of output boxes. This is a criterion to <i>stop</i> bisection: a non-validated box will not be larger than 'eps-min'. Default value is 1e-3.
-E<float>, -eps-max=<float>	Maximal width of output boxes. This is a criterion to <i>force</i> bisection: a validated box will not be larger than 'eps-max' (unless there is no equality and it is fully inside inequalities). Default value is +oo (none)
-t<float>, -timeout=<float>	Timeout (time in seconds). Default value is +oo (none).
-i<filename>, -input=<filename>	Manifold input file. The file contains a (intermediate) description of the manifold with boxes in the MNF (binary) format.
-o<filename>, -output=<filename>	Manifold output file. The file will contain the description of the manifold with boxes in the MNF (binary) format.
-s, -sols	Display the "solutions" (output boxes) on the standard output.
-bfs	Perform breadth-first search (instead of depth-first search, by default)
-trace	Activate trace. "Solutions" (output boxes) are displayed as and when they are found.
-boundary=...	Boundary test strength. Possible values are: <ul style="list-style-type: none"> • <code>true</code>: always satisfied. Set by default for under constrained problems ($0 < m < n$). • <code>full-rank</code>: the gradients of all constraints (equalities and potentially activated inequalities) must be linearly independent. • <code>half-ball</code>: (not implemented yet) the intersection of the box and the solution set is homeomorphic to a half-ball of \mathbb{R}^n • <code>false</code>: never satisfied. Set by default if $m=0$ or $m=n$ (inequalities only/square systems)
-random-seed=<float>	Random seed (useful for reproducibility). Default value is 1.
-q, -quiet	Print no report on the standard output.

2.1.3 Calling IbexSolve from C++

You can call IbexSolve (the default solver) and get the solutions from C++.

Two objects must be built: the first represents the problem (namely, a *system*), the second the solver itself. Then, we just run the solver. Here is a simple example:

```
/* Build a system of equations from the file */
System system(IBEX_BENCHS_DIR "/others/kolev36.bch");

/* Build a default solver for the system and with a precision set to 1e-07 */
DefaultSolver solver(system, 1e-07);

solver.solve(system.box); // Run the solver
```

(continues on next page)

(continued from previous page)

```
/* Display the solutions. */
output << solver.get_manifold() << endl;
```

The output is:

2.1.4 The generic solver

The generic solver is the main C++ class behind the implementation of `ibexsolve`. It is a classical branch and prune algorithm that interleaves contraction and branching (bisection) until boxes get sufficiently small. However, it performs a more general task than just finding solution points of square systems of equations: it also knows how to deal with under-constrained systems and handle manifolds.

Note: A more detailed documentation about under-constrained systems will be available soon.

Compared to `ibexsolve`, the generic solver allows the following additional operators as inputs:

1. a **contractor**

Operator that contracts boxes by removing non-solution points. The contraction operator must be compatible with the system given (equations/inequalities). The solver performs no check (it is the user responsibility). See [Contractors](#).

2. a **bisector**

Operator that splits a box. Note that some bisectors have a *precision* parameter: the box is bisected providing it is large enough. But this precision is not directly seen by the solver which has its own precision variables (see `-e`` and ``-E`). If however the bisector does not split a box, this will generate an exception caught by the solver, which will not continue the search and backtrack. So fixing the bisector internal precision gives basically the same effect as fixing it with `--e`. See [Bisectors](#) for more details.

3. a **cell buffer**

Operator that manages the list of pending boxes (a *cell* is a box with a little bit of extra information used by the search). See [Cell buffers](#) for more details.

Our next example creates a solver for the intersection of two circles of radius d , one centered on $(0, 0)$ and the other in $(1, 0)$.

To this end we first create a vector-valued function:

$$(x, y) \mapsto \begin{pmatrix} x^2 + y^2 - d \\ (x - 1)^2 + y^2 - d \end{pmatrix}$$

Then, we build two contractors; a *forward-backward* contractor and (because the system is square), an *interval Newton* contractor.

We chose as bisection operator the *round-robin* operator, that splits each component in turn. The precision of the solver is set to $1e-7$.

Finally, the cell buffer is a stack, which leads to a depth-first search.

```
/* Create the function (x,y)->( ||(x,y)||-d, ||(x,y)-(1,0)||-d ) */
Variable x,y;
double d=1.0;
```

(continues on next page)

(continued from previous page)

```

Function f(x,y,Return(sqrt(sqr(x)+sqr(y))-d,
                        sqrt(sqr(x-1.0)+sqr(y))-d));

/* Create the system f(x,y)=0. */
SystemFactory factory;
factory.add_var(x);
factory.add_var(y);
factory.add_ctr(f(x,y)=0);
System system(factory);

/* Create the domain of variables */
double init_box[][2] = { {-10,10},{-10,10} };
IntervalVector box(2,init_box);

/* Create a first contractor w.r.t f(x,y)=0 (forward-backward) */
CtcFwdBwd fwdBwd(f);

/* Create a second contractor (interval Newton) */
CtcNewton newton(f);

/* Compose the two contractors */
CtcCompo compo(fwdBwd,newton);

/* Create a round-robin bisection heuristic and set the
 * precision of boxes to 0. */
RoundRobin bisector(0);

/* Create a "stack of boxes" (CellStack), which has the effect of
 * performing a depth-first search. */
CellStack buff;

/* Vector precisions required on variables */
Vector prec(6, 1e-07);

/* Create a solver with the previous objects */
Solver s(system, compo, bisector, buff, prec, prec);

/* Run the solver */
s.solve(box);

/* Display the solutions */
output << s.get_manifold() << endl;

```

The output is:

2.1.5 Implementing IbexSolve (the default solver)

IbexSolve is an instance of the generic solver with (almost) all parameters set by default.

We already showed how to *Calling IbexSolve from C++*. To give a further insight into the generic solver and its possible settings, we explain now how to re-create the default solver by yourself.

The contractor of the default solver is obtained with the following recipe. This is a *composition* of

1. *HC4*
2. *ACID*

3. *Interval Newton* (only if it is a square system of equations)
4. **A *fixpoint* of the *Polytope Hull* of two linear relaxations combined:**
 - the relaxation called X-Taylor;
 - the relaxation generated by affine arithmetic. See *Linearizations*.

The bisector is based on the *The Smear Function* with maximal relative impact.

So the following program exactly reproduces the default solver.

```

System system(IBEX_BENCHS_DIR "/others/kolev36.bch");

/* ===== building contractors ===== */
CtcHC4 hc4(system, 0.01);

CtcHC4 hc4_2(system, 0.1, true);

CtcAcid acid(system, hc4_2);

CtcNewton newton(system.f_ctrs, 5e+08, 1e-07, 1e-04);

LinearizerCombo linear_relax(system, LinearizerCombo::XNEWTON);

CtcPolytopeHull polytope(linear_relax);

CtcCompo polytope_hc4(polytope, hc4);

CtcFixPoint fixpoint(polytope_hc4);

CtcCompo compo(hc4, acid, newton, fixpoint);
/* ===== */

/* Create a smear-function bisection heuristic. */
SmearSumRelative bisector(system, 1e-07);

/* Create a "stack of boxes" (CellStack) (depth-first search). */
CellStack buff;

/* Vector precisions required on variables */
Vector prec(6, 1e-07);

/* Create a solver with the previous objects */
Solver s(system, compo, bisector, buff, prec, prec);

/* Run the solver */
s.solve(system.box);

/* Display the solutions */
output << s.get_manifold() << endl;

/* Report performances */
output << "cpu time used=" << s.get_time() << "s." << endl;
output << "number of cells=" << s.get_nb_cells() << endl;

```

2.1.6 Parallelizing search

It is possible to parallelize the search by running (in parallel) solvers for different subboxes of the initial box.

Be aware however that Ibex has not been designed (so far) to be parallelized and the following lines only reports our preliminary experiments.

Here are the important observations:

- The sub-library gaol is **not** thread-safe. You must compile Ibex with **filib** which seems to be OK (see [Configuration options](#)).
- The linear solver Soplex (we have not tested yet with Cplex) seems to be thread-safe but sometimes generates error messages on the console like:

```
ISOLVE56 stop: 0, basis status: PRIMAL (2), solver status: RUNNING (-1)
```

So, calling Soplex several times simultaneously seems not to be allowed, but Soplex at least manages the case properly, that is, stops. As far as we have observed, we don't lose solutions even when this kind of message appear.

- Ibex objects are not thread-safe which means that the solvers run in parallel must share no information. In particular, each solver must have its **own copy** of the system.

Here is an example:

```
// Get the system
System sys1(IBEX_BENCHS_DIR "/polynom/ponts-geo.bch");

// Create a copy for the second solver
System sys2(sys1, System::COPY);

// Precision of the solution boxes
double prec=1e-08;

// Create two solvers
DefaultSolver solver1(sys1, prec);
DefaultSolver solver2(sys2, prec);

// Create a partition of the initial box into two subboxes,
// by bisecting any variable (here, n^4)
pair<IntervalVector, IntervalVector> pair=sys1.box.bisect(4);

// =====
// Run the solvers in parallel
// =====
#pragma omp parallel sections
{
    solver1.solve(pair.first);
#pragma omp section
    solver2.solve(pair.second);
}
// =====

output << "solver #1 found " << solver1.get_manifold().size() << endl;
output << "solver #2 found " << solver2.get_manifold().size() << endl;
```

If I remove the `#pragma` the program displays:

```
solver #1 found 64
solver #2 found 64

real 0m5.121s          // <----- total time
user 0m5.088s
```

With the `#pragma`, I obtain:

```
solver #1 found 64
solver #2 found 64

real 0m2.902s          // <----- total time
user 0m5.468s
```

Note: It is pure luck that by bisecting the 4th variable, we obtain exactly half of the solutions on each sub-box. Also, looking for the 64 first solutions takes here around the same time than looking for the 64 subsequent ones, which is particular to this example. So, contrary to what this example seems to prove, splitting the box in two subboxes does not divide the running time by two in general. Of course :)

If you are afraid about the messages of the linear solver, you can replace the `DefaultSolver` by your own dedicated solver that does not resort to the simplex, ex:

```
Vector eps_min(sys1.nb_var,prec);
Vector eps_max(sys1.nb_var,POS_INFINITY);
Solver solver1(sys1,*new CtcCompo(*new CtcHC4(sys1),*new CtcNewton(sys1.f_
↪ ctrs)), *new RoundRobin(prec), *new CellStack(), eps_min, eps_max);
Solver solver2(sys2,*new CtcCompo(*new CtcHC4(sys2),*new CtcNewton(sys2.f_
↪ ctrs)), *new RoundRobin(prec), *new CellStack(), eps_min, eps_max);
```

3.1 IbexOpt

This page describes IbexOpt, the plugin installed with the `--with-optim` option.

3.1.1 Getting started

IbexOpt is a end-user program that solves a NLP problem (non-linear programming). It minimizes a (nonlinear) objective function under (nonlinear) inequality and equality constraints. It resorts to a unique black-box strategy (whatever the input problem is) and with a very limited number of parameters. Needless to say, this strategy is a kind of compromise and not the best one for a given problem.

Note that this program is based on the *generic optimizer*, a C++ class that allows to build a more customizable optimizer.

You can directly apply this optimizer on one of the benchmark problems distributed with Ibex. The benchmarks are all written in the *Minibex syntax* and stored in an arborescence under `plugins/optim/benchs/`. If you compare the Minibex syntax of these files with the ones given to IbexSolve, you will see that a “minimize” keyword has appeared.

Open a terminal, move to the `bin` subfolder and run IbexSolve with, for example, the problem named `ex3_1_3` located at the specified path:

```
~/ibex-2.6.0$ cd bin
~/ibex-2.6.0/bin$ ./ibexopt ../plugins/optim/benchs/easy/ex3_1_3.bch
```

The following result should be displayed:

```
***** setup *****
file loaded: ../plugins/optim/benchs/easy/ex3_1_3.bch
*****
running.....
```

(continues on next page)

(continued from previous page)

```

optimization successful!

best bound in: [-310.3099999984,-309.9999999984]
relative precision obtained on objective function: 0.001 [passed]
absolute precision obtained on objective function: 0.309999999985 [failed]
best feasible point: (4.9999999999 ; 1 ; 5 ; 0 ; 5 ; 10)
cpu time used: 0.004000000000001s.
number of cells: 1

```

The program has proved that the minimum of the objective lies in a small interval enclosing -310. It also gives a point close to (5 ; 1 ; 5 ; 0 ; 5 ; 10) which satisfies the constraints and for which the value taken by the objective function is inside this interval. The process took less than 0.005 seconds.

3.1.2 Options

-r<float>, -rel-eps-f=<float>	Relative precision on the objective. Default value is 1e-3.
-a<float>, -abs-eps-f=<float>	Absolute precision on the objective. Default value is 1e-7.
-eps-h=<float>	Equality relaxation value. Default value is 1e-8.
-t<float>, -timeout=<float>	Timeout (time in seconds). Default value is +oo.
-random-seed=<float>	Random seed (useful for reproducibility). Default value is 1.
-eps-x=<float>	Precision on the variable (Deprecated). Default value is 0.
-initial-loup=<float>	Initial “loup” (a priori known upper bound).
-rigor	Activate rigor mode (certify feasibility of equalities).
-trace	Activate trace. Updates of loup/uplo are printed while minimizing.

3.1.3 Calling IbexOpt from C++

Calling the default optimizer is as simple as for the *default solver*. The loaded system must simply correspond to an optimization problem. The default optimizer is an object of the class `DefaultOptimizer`.

Once the optimizer has been executed(), the main information is stored in three fields, where *f* is the objective:

- `loup` (“lo-up”) is the lowest upper bound known for $\min(f)$.
- `uplo` (“up-lo”) is the uppest lower bound known for $\min(f)$.
- `loup_point` is the vector for which the value taken by *f* is less or equal to the `loup`.

Example:

```

/* Build a constrained optimization problem from the file */
System sys(IBEX_OPTIM_BENCHS_DIR "/easy/ex3_1_3.bch");

/* Build a default optimizer with a precision set to 1e-07 for f(x) */
DefaultOptimizer o(sys,1e-07);

o.optimize(sys.box); // Run the optimizer

/* Display the result. */
output << "interval for the minimum: " << Interval(o.get_uplo(),o.get_loup()) <<
endl;
output << "minimizer: " << o.get_loup_point() << endl;

```

The output is:

3.1.4 Getting an enclosure of all global minima

Given a problem:

$$\begin{aligned} & \text{Minimize } f(x) \\ & \text{s.t. } h(x) = 0 \wedge g(x) \leq 0 \end{aligned}$$

IbexOpt gives a feasible point that is *sufficiently* close to the real minimum f^* of the function, i.e., a point that satisfies

$$\begin{aligned} & \text{uplo} \leq f(x) \leq \text{loup} \\ & \text{s.t. } h(x) = 0 \wedge g(x) \leq 0 \end{aligned}$$

with *loup* and *uplo* are respectively a valid upper and lower bound of f^* , whose accuracy depend on the input precision parameter (note: validated feasibility with equalities requires “rigor” mode).

From this, it is possible, in a second step, to get an enclosure of all global minima thanks to *IbexSolve*. The idea is simply to ask for all the points that satisfy the previous constraints. We give now a code snippet that illustrate this.

```
// ===== 1st step =====
// Build the original system:
Variable x,y;
SystemFactory opt_fac;
opt_fac.add_var(x);
opt_fac.add_var(y);
// minimize f(x)=-x-y
opt_fac.add_goal(-x-y);
// s.t. x^2+y^2<=1
opt_fac.add_ctr(sqr(x)+sqr(y)<=1);
System opt_sys(opt_fac);
// build the default optimizer
DefaultOptimizer optimizer(opt_sys,1e-01);
// run it with (x,y) in R^2
optimizer.optimize(IntervalVector(2));

// ===== 2nd step =====
// Build the auxiliary system
SystemFactory sol_fac;
sol_fac.add_var(x);
sol_fac.add_var(y);
sol_fac.add_ctr(sqr(x)+sqr(y)<=1);
sol_fac.add_ctr(-x-y>=optimizer.get_uplo());
sol_fac.add_ctr(-x-y<=optimizer.get_loup());
System sol_sys(sol_fac);
DefaultSolver solver(sol_sys,0.01);
solver.solve(IntervalVector(2));
// Get an enclosure of all minima, (under
// the form of manifold)
const Manifold& minima=solver.get_manifold();
```

3.1.5 The generic optimizer

Just like the *generic solver*, the generic optimizer is the main C++ class (named `Optimizer`) behind the implementation of `IbexOpt`. It takes as the solver:

- a **contractor**
- a **bisector**

- a cell buffer

but also requires an extra operator:

- a **loup finder**. A loup finder is in charge of the *goal upper bounding* step of the optimizer.

We show below how to re-implement the default optimizer from the generic `Optimizer` class.

3.1.6 Implementing `IbexOpt` (the default optimizer)

The contraction performed by the default optimizer is the same as the default solver (see *Implementing IbexSolve (the default solver)*) except that it is not applied on the system itself but the *Extended System*.

The loup finder (`LoupFinderDefault`) is a mix of different strategies. The basic idea is to create a continuum of feasible points (a box or a polyhedron) where the goal function can be evaluated quickly, that is, without checking for constraints satisfaction. The polyhedron (built by a `LinearizerXTaylor` object) corresponds to a *linerization technique* described in [Araya et al. 2014] and [Trombettoni et al. 2011], based on *inner region extraction*. It also resorts to the linerization offered by affine arithmetic (a `LinearizerAffine` object) if the affine plugin is installed. The box (built by a `LoupFinderInHC4` object) is a technique based on *inner arithmetic* also described in the aforementioned articles.

Finally, by default, the cell buffer (`CellDoubleHeap`) is basically a sorted heap that allows to get in priority boxes minimizing either the lower or upper bound of the objective enclosure (see *Cell Heap*).

```
System system(IBEX_OPTIM_BENCHS_DIR "/easy/ex3_1_3.bch");

double prec=1e-7; // precision

// normalized system (all inequalities are "<=")
NormalizedSystem norm_sys(system);

// extended system (the objective is transformed to a constraint y=f(x))
ExtendedSystem ext_sys(system);

/* ===== building contractors ===== */
CtcHC4 hc4(ext_sys,0.01);

CtcHC4 hc4_2(ext_sys,0.1,true);

CtcAcid acid(ext_sys, hc4_2);

LinearizerCombo linear_relax(ext_sys,LinearizerCombo::XNEWTON);

CtcPolytopeHull polytope(linear_relax);

CtcCompo polytope_hc4(polytope, hc4);

CtcFixPoint fixpoint(polytope_hc4);

CtcCompo compo(hc4,acid,fixpoint);
/* ===== */

/* Create a smear-function bisection heuristic. */
SmearSumRelative bisector(ext_sys, prec);

/** Create cell buffer (fix exploration ordering) */
CellDoubleHeap buffer(ext_sys);
```

(continues on next page)

(continued from previous page)

```
/** Create a "loup" finder (find feasible points) */
LoupFinderDefault loup_finder(norm_sys);

/** Create a solver with the previous objects */
Optimizer o(system.nb_var, compo, bisector, loup_finder, buffer, ext_sys.goal_
↪var());

/** Run the optimizer */
o.optimize(system.box,prec);

/** Display a safe enclosure of the minimum */
output << "f* in " << Interval(o.get_uplo(),o.get_loup()) << endl;

/** Report performances */
output << "cpu time used=" << o.get_time() << "s."<< endl;
output << "number of cells=" << o.get_nb_cells() << endl;
```


4.1 Java Plugin (for Choco)



The Java plugin of Ibex allows to use Ibex with [Choco](#), for solving mixed integer-continuous CSP (constraint satisfaction problems).

4.1.1 Installation (with Ibex 2.6)

Note: Read how to *install Ibex* before installing this plugin.

Warning: The option `--with-java-package` has been changed to `--java-package-name`

The installation of the plugin will generate, in addition to the Ibex library, the `libibex-java` library that contains the glue code between C++ and Java.

Note: Under Windows, Ibex is compiled as a 32-bit library although the platform is 64 bits (this is mainly because the MinGW environment is 32-bits). Hence, Java will fail in loading Ibex unless you have a 32-bits JVM.

The following instructions must be typed in the shell of MinGW.

Uncompress the archive `ibex-2.6.0.tar.gz` in some Ibex folder:

```
~/Ibex/$ tar xvfz ibex-2.6.0.tar.gz
```

Uncompress the archive `ibex-java.tar.gz` in the plugin folder:

```
~/Ibex/$ tar xvfz ibex-java.tar.gz --directory=ibex-2.6.0/plugins
```

Set the environment variable `JAVA_HOME`. Typical paths are `/Library/Java/Home` (MacOS) or `/usr/lib/jvm/java-7-openjdk-i386` (Linux). Example:

```
~/Ibex/$ export JAVA_HOME=/Library/Java/Home
```

Under MinGW, the variable must be set in Liux-style (don't use backslash ("`\`") as separator), e.g.:

```
~/Ibex/$ export JAVA_HOME=/c/Java/jdk1.7.1_17
```

Then configure Ibex as follows:

```
~/Ibex/$ cd ibex-2.6.0
~/Ibex/ibex-2.6.0/$ ./waf configure [...] --enable-shared --with-solver --with-jni --
↳ java-package-name=org.chocosolver.solver.constraints.real
```

Note: the `--enable-shared` option is mandatory. The `--with-solver` option can be omitted as this plugin is automatically installed (with Release 2.6.0).

4.1.2 Configuration options

The IbexOpt plugin supports the following options (to be used with `waf configure`) :

- with-jni** Activate the Java plugin.
- java-package-name=PACKAGE_NAME** The plugin will create the `PACKAGE_NAME.jar` file. This file is put into the `[prefix]/share/java` where `[prefix]` is `/usr/local` by default or whatever path specified via `--prefix`.

4.1.3 Troubleshooting

UnsatisfiedLinkError with Choco

When running the “CycloHexan” example from Choco using Ibex, the following error appears:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: org.chocosolver.solver.
↳ constraints.real.Ibex.add_ctr(ILjava/lang/String;I)V
at org.chocosolver.solver.constraints.real.Ibex.add_ctr(Native Method)
at org.chocosolver.solver.constraint.propagators.real.RealPropagator.<init>()V
↳ (RealPropagator.java:77)
at org.chocosolver.solver.constraints.real.RealConstraint.addFunction(RealConstraint.
↳ java:82)
```

(continues on next page)

(continued from previous page)

```
at samples.real.CycloHexan.buildModel(CycloHexan.java:87)
at samples.AbstractProblem.execute(AbstractProblem.java:130)
at samples.real.CycloHexan.main(CycloHexan.java:134)
```

Solution: You probably did not set the Java package properly. The java package of the Ibex class in Choco is `org.chocosolver.solver.constraints.real`, try:

```
./waf configure [...] --java-package-name=org.chocosolver.solver.constraints.real
```

JAVA_HOME does not seem to be set properly

I get this message when running waf configure.

Solution: The `JAVA_HOME` must be the path of the JDK and contain a subdirectory include which, in turn, contains the `jni.h` header file. On MacOS this path can be `/Library/Java/JavaVirtualMachines/jdkXXXX.jdk/Contents/Home`.

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`