```cpp
#include "ibex.h"

using namespace std;
using namespace ibex;

int main() {
{
    // Example #1
    // ------------------------------------------------
    // Basic interval arithmetic
    // > create the interval x=[1,2] and y=[3,4]
    // > calculate the interval sum x+y
    // > calculate interval image of sin(x+1)
    // ------------------------------------------------
    Interval x(1,2);
    Interval y(3,4);
    cout << "x+y=" << x+y << endl;
    cout << "sin(x+1)=" << sin(x+1) << endl;
    // ------------------------------------------------
}

{
    // Example #2
    // ------------------------------------------------
    // 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[][2]={{0,1},{0,1},{0,1}};
    IntervalVector x(3,_x);

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

    cout << "M*x=" << M*x << endl;
    cout << "M'*x=" << M.transpose()*x << endl;
    // ------------------------------------------------
}
```

```
{
    // Example #3
    // ------------------------------------------------
    // Mixing real/interval vector/matrices
    // > calculate the magnitude of an interval matrix (a real matrix)
    // > calculate the mid-vector 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[][2]={{0,1},{0,1},{0,1},
                    {0,2},{0,2},{0,2}};
    IntervalMatrix M(3,3,_M);
    Matrix m2=M.mag();
    Vector x2=x.mid();
    cout << m2*x2 << endl;
    // ------------------------------------------------
}

{
    // Example #4
    // ------------------------------------------------
    // Basic "projection"/"backward arithmetic"
    //
    // (Contraction of x w.r.t. to f(x)=y)
    //
    // > create three intervals x,y and z with z=x+y
    // > project sin(z)=-1 onto z (contracts z)
    // > project x+y onto x and y (contracts x and y)
    // ------------------------------------------------
    Interval x(1,2);
    Interval y(3,4);
    Interval z=x+y;
    cout << "x before =" << x << endl;
    cout << "y before =" << y << endl;
    cout << "z before =" << z << endl << endl;
    proj_sin(-1.0,z);
    cout << "z after =" << z << endl;
    proj_add(z,x,y);
    cout << "x after =" << x << endl;
    cout << "y after =" << y << endl;
```

```
    // ------------------------------------------------
}

{
    // Example #5
    // ------------------------------------------------
    // Example of projection leading to an empty set
    //
    // Same example as Example #3 except that the
    // projection of sin(z)=1 onto z leads to an empty set.
    // ------------------------------------------------
    Interval x(1,2);
    Interval y(3,4);
    Interval z=x+y;
    cout << "x before =" << x << endl;
    cout << "y before =" << y << endl;
    cout << "z before =" << z << endl << endl;
    proj_sin(1.0,z);
    cout << "z after =" << z << endl;
    proj_add(z,x,y);
    cout << "x after =" << x << endl;
    cout << "y after =" << y << endl;
    // ------------------------------------------------
}

{
    // Example #6
    // ------------------------------------------------
    // Vector/Matrix projection
    // > create a matrix M centered on the identity
    // > create two 1-column vectors x and y
    // > set artificially one entry of M to a large interval
    // > contract M with respect to M*x=y and observe
    //   that the uncertainty on this entry has been reduced
    // ------------------------------------------------
    double _x[][2]={{1,1},{1,1},{1,1}};
    IntervalVector x(3,_x);
    IntervalMatrix M=Matrix::eye(3) +
                     Interval(-0.1,0.1)*Matrix::ones(3);
    cout << "x before=" << x << endl;
    cout << "M before=" << M << endl << endl;
    cout << "M*x=" << M*x << endl << endl;
```

```
    M[2][2]=Interval(0.5,1.1);
    IntervalVector y=Vector::ones(3);
    cout << "M modified=" << M << endl << endl;

    proj_mul(y,M,x,1e-04);
    cout << "x after=" << x << endl;
    cout << "M after=" << M << endl << endl;
    // ------------------------------------------------
}

{
    // Example #7
    // ------------------------------------------------
    // Function definition
    //
    // > Create the function (x,y)->sin(x+y)
    // > Display it
    // ------------------------------------------------
    Variable x("x");
    Variable y("y");
    Function f(x,y,sin(x+y));
    cout << f << endl;
    // ------------------------------------------------
}

{
    // Example #8
    // ------------------------------------------------
    // Function evaluation/projection
    //
    // > Create the function f:(x,y)->sin(x+y)
    // > Evaluate the interval image of f on a box
    // > Contract (x,y) w.r.t f(x)=-1.
    // ------------------------------------------------
    Variable x;
    Variable y;
    Function f(x,y,sin(x+y));

    double _box[][2]= {{1,2},{3,4}};
    IntervalVector box(2,_box);
    cout << "initial box=" << box << endl;
```

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

    f.proj(-1.0,box);
    cout << "box after proj=" << box << endl;
    // ------------------------------------------------
}

{
    // Example #9
    // ------------------------------------------------
    // Gradient computation
    //
    // > Create the Euclidian distance function "dist"
    //   between two points (xa,ya) and (xb,yb)
    // > Contract xa and ya w.r.t. dist(xa,ya,1,2)=5.0
    // > Display the result (the box enclosing the circle
    //   centered on xb=1,yb=2).
    // > Calculate the gradient of the function
    // ------------------------------------------------
    Variable xa,xb,ya,yb;
    Function dist(xa,xb,ya,yb, sqrt(sqr(xa-xb)+sqr(ya-yb)));

    double init_xy[][2] = { {-10,10}, {1,1},
                            {-10,10}, {2,2} };
    IntervalVector box(4,init_xy);
    cout << "initial box=" << box << endl;

    dist.proj(5.0,box);

    cout << "box after proj=" << box << endl;

    IntervalVector g(4);
    dist.gradient(box,g);
    cout << "gradient=" << g << endl;
    // ------------------------------------------------
}

{
    // Example #10
    // ------------------------------------------------
    // Function with vector variables
```

```
    //
    // Same example as Example #9 except that the function
    // dist has 2 vector arguments a and b instead of 4
    // real arguments xa, ya, xb and yb.
    // ------------------------------------------------
    Variable a(2);
    Variable b(2);
    Function dist(a,b,sqrt(sqr(a[0]-b[0])+sqr(a[1]-b[1])));

    double init_xy[][2] = { {-10,10}, {-10,10}, {1,1}, {2,2} };
    IntervalVector box(4,init_xy);
    dist.proj(5.0,box);
    cout << "box after proj=" << box << endl;
    // ------------------------------------------------
}

{
    // Example #11
    // ------------------------------------------------
    // Function composition
    //
    // > Create the function dist:(a,b)->||a-b||
    // > Create the function f: x->(dist(x,(1,2)),
    // > Perform the same contraction as in the
    //   previous examples
    // ------------------------------------------------
    Variable a(2);
    Variable b(2);
    Function dist(a,b,sqrt(sqr(a[0]-b[0])+sqr(a[1]-b[1])),"dist");

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

    Variable x(2);
    Function f(x,dist(x,pt));

    double init_xy[][2] = { {-10,10}, {-10,10} };
    IntervalVector box(2,init_xy);
    f.proj(5.0,box);
    cout << "box after proj=" << box << endl;
    // ------------------------------------------------
```

```
}

{
    // Example #12
    // ------------------------------------------------
    // Vector-valued functions, Jacobian matrix
    //
    // > create the function dist:(x,pt)->||x-pt||
    // > create the function f:x->(dist(x,pt1),dist(x,pt2)
    // > perform a contraction w.r.t. f(x)=(sqrt(2)/2,sqrt(2)/2)
    // > 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)));

    cout << f << endl;

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

    IntervalVector d=0.5*sqrt(2)*Vector::ones(2);

    f.proj(d,box);

    cout << "box after proj=" << box << endl;

    box[0]=3;
    box[1]=2;
    IntervalMatrix J(2,2);
    f.jacobian(box,J);
    cout << "J=" << J << endl;
    // ------------------------------------------------
}


{
```

```
    // Example #13
    // ------------------------------------------------
    // Contractor
    //
    // > Create the function (x,y)->( ||(x,y)||-d,  ||(x,y)-(1,1)||-d )
    // > Create the contractor "projection of f=0"
    //   (i.e., contraction of the input box w.r.t. f=0)
    // > Embed this contractor in a generic fix-point loop
    // ------------------------------------------------

    Variable x("x"),y("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));
    cout << f << endl;

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

    CtcProj c(f);
    c.contract(box);
    cout << "box after proj=" << box << endl;

    CtcFixPoint fp(c,1e-03);

    fp.contract(box);
    cout << "box after fixpoint=" << box << endl;
    // ------------------------------------------------
}

{
    // Example #14
    // ------------------------------------------------
    // Combining Projection and Newton Contractor
    //
    // > Create the projection contractor on the same function
    //   as in the last example
    // > Contract the initial box x to x'
    // > Create the Newton iteration contractor
    // > Contract the box x'
    // ------------------------------------------------
    Variable x("x"),y("y");
```

```
    double d=1.0;
    Function f(x,y,Return(sqrt(sqr(x)+sqr(y))-d,
                          sqrt(sqr(x-1.0)+sqr(y-1.0))-d));
    cout << f << endl;

    double init_box[][2] = { {0.9,1.1},{-0.1,0.1} };
    IntervalVector box(2,init_box);

    CtcProj c(f);
    c.contract(box);
    cout << "box after proj=" << box << endl;

    CtcNewton newton(f);
    newton.contract(box);
    cout << "box after newton=" << box << endl;
    // ------------------------------------------------
}

{
    // Example #15
    // ------------------------------------------------
    // A basic example of Robust Parameter estimation using
    // contractor programming.
    //
    // A point (x,y) has to be localized. We measure
    // 4 distances "bD" from 6 different points (bX,bY).
    // Each measurement bD has an uncertainty [-0.1,0.1]
    // We know there may be at most one outlier.
    //
    // The solution point is: x=6.32193 y=5.49908
    //
    // ------------------------------------------------
    int N=6;

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

    // The corresponding intervals (uncertainty is 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);
}
// We introduce an outlier
bX[5]+=10;

// declare the distance function
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]);

// Declare a projection contractor for each
// function
CtcProj c0(f0);
CtcProj c1(f1);
CtcProj c2(f2);
CtcProj c3(f3);
CtcProj c4(f4);
CtcProj c5(f5);

// The initial box
double _box[][2]={{0,10},{0,10}};
IntervalVector initbox(2,_box);

// Create the array of all the contractors
Array<Ctc> array(c0,c1,c2,c3,c4,c5);
CtcCompo all(array);

IntervalVector box=initbox;
```

```
    // Create the q-intersection of the N contractors
    CtcQInter q(array,5);

    // Perform a first contraction
    box=initbox;
    q.contract(box);
    cout << "after q-inter =" << box << endl;

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

    // Perform a second contraction with
    // the fixpoint loop
    fix.contract(box);

    cout << "after fix+q-inter =" << box << endl;
    // ------------------------------------------------
}

{
    // Example #16
    // ------------------------------------------------
    // Solver (with an hard-coded function)
    //
    // > Create the function (x,y)->( ||(x,y)||-d,  ||(x,y)-(1,1)||-d )
    // > Create two contractors w.r.t f(x,y)=0, one using backward
    //   arithmetic (CtcProj), the other using interval Newton iteration
    //   (CtcNewton)
    // > Create a round-robin bisection heuristic
    // > Create a "stack of boxes" (CellStack), which has the effect of
    //   performing a depth-first search.
    // > Create a solver with the previous objects
    // > Run the solver and obtain the solutions
    // ------------------------------------------------

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

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

```
    CtcProj c(f);
    CtcNewton newton(f);
    RoundRobin rr;
    CellStack buff;

    Solver s(c,rr,buff,1e-07);

    vector<IntervalVector> sols=s.solve(box);

    for (int i=0; i<sols.size(); i++)
        cout << "solution n°" << i << "=\t" << sols[i] << endl;

    cout << "number of cells=" << s.nb_cells << endl;
    // ------------------------------------------------
}

{
    // Example #17
    // ------------------------------------------------
    // Solver (with a function loaded from a file)
    //
    // Solve a system using interval Newton and constraint
    // propagation.
    // ------------------------------------------------


    // Load a system of equations
    // --------------------------
    System sys("ponts.txt");

    // Build contractor #1:
    // --------------------------
    // A "constraint propagation" loop.
    // Each constraint in sys.ctrs is an
    // equation.
    CtcHC4 hc4(sys.ctrs,0.1);
    hc4.accumulate=true;

    // Build contractor #2:
    // --------------------------
    // An interval Newton iteration
```

```
    // for solving f(x)=0 where f is
    // a vector-valued function representing
    // the system.
    CtcNewton newton(sys.f);

    // Build the main contractor:
    // ---------------------------
    // A composition of the two previous
    // contractors
    CtcCompo c(hc4,newton);

    // Build the bisection heuristic
    // --------------------------
    // Round-robin means that the domain
    // of each variable is bisected in turn
    RoundRobin rr;

    // Chose the way the search tree is explored
    // ------------------------------------
    // A "CellStack" means a depth-first search.
    CellStack buff;

    // Build a solver
    // -------------
    // The last parameter (1e-07) is the precision
    // required for the solutions
    Solver s(c,rr,buff,1e-03);

    //s.trace=true;

    // Get the solutions
    vector<IntervalVector> sols=s.solve(sys.box);

    // Display the number of solutions
    cout << "number of solutions=" << sols.size() << endl;

    // Display the number of boxes (called "cells")
    // generated during the search
    cout << "number of cells=" << s.nb_cells << endl;
    // ------------------------------------------------
}
}
```