C++ Interval & Constraint Solving Package

Gregory D. Hager
John H. Lu

Research Report YALEU/DCS/RR-953
February 1993

C++ Interval & Constraint Solving Package

Gregory D. Hager, John M. Lu
hager@cs.yale.edu

February 15, 1993

1 Introduction

This package is intended to be a reasonably portable and efficient implementation of interval arithmetic and interval-based constraint solving [4, 1, 2]. Thus far it has been developed under AT&T C++ and is not guaranteed to work with any other C compiler without some modification.

There are three main modules to the package:

1. `Interval.[h,cc]` contains the basic code for working with intervals. It includes all of the algebraic operations and a number of trigonometric and transcendental functions.
2. `IntervalVec.[h,cc]` provides the extension to vectors of intervals and operations on vectors.
3. `PElement.List.h,Fit.[h,cc],compare.[h,cc]` provide support for expressing systems of constraints and computing solutions to those constraints.

The program `gearwcenter.cc` is a sample program that can be compiled and run. This program is described in Section 7 of this document.

2 Installation

This section assumes you have retrieved and unpacked a tarred version of the interval directories. This means that you have established three directories: `Interval`, `Bisection` and `Apps`. In addition, you should have a makefile. Simply typing `make` should set up the rest of the directory structure and generate some libraries. There are three variables to set in any makefile: `USE_DIRECTED_ROUNDING` and `NOWARNINGS` and `USE_INLINES`. The first enables the use of directed rounding in all interval computations, ensuring complete correctness of all results. Since directed rounding incurs a significant (about 400%) performance hit at the moment, this option should only be enabled if there is reason to doubt the fidelity of the interval computations. For most practical applications, the precision supplied by double precision arithmetic has proved to be sufficient. The

second flag controls the checking and printing of warnings. A marginal (about 20%) performance improvement results by turning off checking for division by zero and similar numerical error checks. The third flag allows the use of inline functions for primitive operations such as Interval multiply. These flags can be set at the top level, or set individual for each makfile.

3 Basic Intervals

Declarations and Assignment

```
Interval x;
Interval y = 5;
Interval z = Interval (5);
Interval q = Interval(4,5);
```

All declare a single interval variable. The second two initialize the interval to $[5, 5]$, and the last initializes the value to $[4, 5]$.

Logical Operators The logical operations `==`, `!=`, `<`, `<=`, `>`, `>=`, and `&&` are defined. All operators are overloaded to take two interval arguments, or combinations of intervals and doubles. The `==` operator returns true when the lower bound and the upper bound of the two intervals are equal. (e.g. `(1,5)==(1,5)` is true); `!=` returns true if either of the coordinates are different. All comparison operations return true if and only if the same operation would return true for every pair of real numbers (x,y) where x comes from the first interval and y comes from the second interval. The last operation is true if the intersection of the two intervals is not empty.

Arithmetic Operators The arithmetic operators `+`, `-`, `*`, `/` and their assignment extensions `+=`, `-=`, `*=`, `/=` are all defined for intervals. A double can be substituted for all intervals except as an Lvalue for the assignment extensions.

Set Operators The set operators `&`, `|` and their assignment extensions `&=`, `|=` are defined for intervals. `|` returns the smallest interval containing both intervals. `&` returns the intersection of two intervals. If the two intervals do not intersect, `&` returns (NaN,NaN). A double can be substituted for all intervals except as a Lvalue for the assignment extensions.

Transcendental Functions The following transcendental functions are defined for intervals: `pow(x,y)`, `exp(x)`, `log(x)`, `sqr(x)`, `sqrt(x)`, `cos(x)`, `sin(x)`, `atan(x)`, `atan2(x,y)`, `abs(x)`.

Miscellaneous Member function `width()` returns the difference between the upper and lower bounds of the interval as a double. Member function `center()` returns the average of the upper and lower bounds of the interval as a floating point. The lower bound of an interval can be accessed

with the member function `lower_value()`. The upper bound of an interval can be accessed with the member function `upper_value()`. These functions will return a value and not a reference, so there is no way to directly reassign a upper or lower bound to an interval.

Intervals can be displayed by the << operator.

```
ostream << x;
```

will display x on `ostream`. Input is handled with the >> operator.

```
instream >> x;
```

4 Interval Vectors

The Interval vector class is based on a GNU Interval vector class. A vector of intervals obeys roughly the same laws as scalar intervals. There are additional vector operations such as `Map` or `Apply` which are not described here but can be found in the `IntervalVec.h`.

Initialization

1. `IntervalVec x` will create an uninitialized interval vector to which any length interval vector can be assigned.
2. `IntervalVec x(<vector_length>)` will create an uninitialized IntervalVec with length `<vector_length>`
3. `IntervalVec x = IntervalVec(<vector_length>,<fill_value>)` will create an IntervalVec of length `<vector_length>` with `<fill_value>` in all dimensions, where `<fill_value>` is an interval.
4. `IntervalVec x = IntervalVec(<old_intervalvec>)` will create a copy of `<old_intervalvec>`.
5. `x[<dimension>] = <Interval>` will set `<dimension>`-th dimension of the previously declared IntervalVec, x, to `<Interval>`.

Arithmetic Operators The arithmetic operators +, -, *, / are all defined. The appropriate operation is carried out componentwise between the two arguments, which must be the same length. A double or an interval value can be substituted in which case the value is cast to an interval vector of the appropriate length containing that value.

Set Operators The set operators &, | are defined for interval vectors. | returns the smallest interval vector containing both intervals. & returns the intersection of two intervals. If the two intervals do not intersect, & returns a vector of (NaN,NaN). A double can be substituted for all interval vectors except as an Lvalue for the assignment extensions.

Miscellaneous The member function `center()` returns an interval vector of point values. The member function `volume()` returns the volume of an interval vector. Interval vectors can be displayed by the `<<` operator.

```
ostream << x;
```

IntervalVect There is a restricted vector class, IntervalVect, that is only used for temporary results. Its purpose is to allow the compiler to generate efficient code that does not have all the security checks normally provided to an `IntervalVec`.

5 Constraint Solving

Constraint solving is a process of dividing an initial space into regions that contain only feasible points, regions that contain only infeasible points, and regions that are mixed. Mixed regions are further subdivided and classified. This procedure continues until an iteration bound is reached, or a feasible region is found. A complete description of this process is described in [3]. This report is included as `report.ps` in this `doc` directory.

We will first present a detailed explanation of `circle.cc` and then give a general description of how to do constraint solving. The problem that the program addresses is as follows. Given data from a circle observed with bounded error, determine whether the radius of the circle exceeds a specific value. Mathematically, the circle is described by the constraint

$$g(p, x) = (p_1 - x_1)^2 + (p_2 - x_2)^2 - p_3^2 = 0$$

The question to be answered is $p_3 > B$ for some value B.

5.1 Annotated Example

A complete listing of `circle.cc` is in Section 7 of this document. In the program, data is generated artificially by the call:

```
gen_ellipse_data(13.0,13.0)
```

which creates data for a circle of radius 13.0 centered at the origin. Each data element is a `PElement`, which is an IntervalVec with some extensions necessary for constraint solving. The each data element is constructed by generating an ideal observation, and then constructing an interval about this observation large enough to account for sensing error. All the data is returned in one data structure called a `PElementList`. In real applications, the data list is generated by reading a sensor and adding an interval representing bounds on sensing error to the value returned.

This data is used to narrow down a search space that is initially one large interval vector. In the program, this is initial search space is in the `PElement ii`.

```
PElement ii(SIZE);

ii[0] = Interval(-10.5,10.5);
ii[1] = Interval(-10.5,10.5);
ii[2] = Interval(3.0,30.0);
```

The SIZE argument to ii serves to define the length of the IntervalVec in ii. The first dimension holds the search space for the x-dimension, the second holds the space for the y-dimension, and the third holds the space for the radius.

The initial search space is the space that will be divided and subdivided into feasible, unfeasible and mixed regions using a *constraint procedure* that is exact analog of g above:

```
void
constraint(const PElement& soln, const PElement& sensor_data)
{

  Interval xdiff,ydiff;

  xdiff = soln[0]- sensor_data[0];
  ydiff = soln[1]- sensor_data[1];

  compare(sqr(xdiff) + sqr(ydiff) - sqr(soln[2]),EQ,0);

}
```

soln is a portion of the search space. sensor_data is one of the data elements generated by gen_ellipse_data(). The compare() call is where the classification actually takes place.

Likewise, termination criteria are expressed in a constraint procedure:

```
void
bigenough(const PElement& val)
{
   compare(val[2].GE.THRESH);
}
```

These four elements—sensor data, search space, and constraint, and termination criteria—are used by the Fit class to perform the constraint solving. In order to perform constraint solving, an instance of the Fit class must be created and initialized. In this case, the instance state is declared with:

```
Fit state(SIZE);
```

SIZE is the number of dimensions in the solution space. The sensor data, search space and constraint are added into state with the lines:

```
state.Initialize(ii); state.add_constraint(constraint,data,"Circle Constraint");
state.add_termination(bigenough)
```

When these elements are in place, the call:

```
state.bisect(atoi(argv[1]));
```

will start the constraining process. `atoi(argv[1])` is the greatest number of iterations that should be run before stopping. This is a complete program to decide whether the radii of observed circles exceeds a value. Other programs performing decision-making contain exactly the same elements. They differ only in the form, number, and complexity of constraint functions. The listing in the appendix contains some other options that will be explained later.

5.2 Data Elements

As noted above, there are two specialized structures, `PElement` and `PElementList`, that are used by the constraint solver. For all practical purposes, a `PElement` can be though of as an `IntervalVec` (from which it is derived). `PElementList` is a list of `PElement` values. This is based on the GNU list class. This class provides a number of functions for list manipulation. We refer to the GNU documentation (a postscript version is included in the `doc` directory) for a complete listing of the functions provided. Most commonly used member functions are:

`PElementList::push(<val>)` adds a PElement to the front of a list.

`PElementList::pop()` removes the first PElement from the front of a list.

`PElementList::append(<list>)` appends `<list>` to the current list.

In principle, interval vectors and lists of interval vectors would suffice for interfacing with the constraint solver. Subsequent releases of the software may institute this change.

5.2.1 Describing Constraints

There are two comparison operations used by the constraint solver:

`require(<boolean>)` takes any boolean expression an insists that the value of that expression be true.

`compare(<ival>,<comp>,<ival>)` stipulates that an equality or inequality condition must hold among two interval expressions. The operator `<comp>` must be one of `GE,LE,LT,GT,EQ`.

These procedures are incorporated into procedures describing data constraints or termination constraints. These procedures take one of the following two forms:

```
void
cfn(PElement& parms)
{
    code
}
```

or

```
void
cfn(PElement& parms, PElement& data)
{
    code
}
```

The first of these is a single constraint on model parameters. The second form specifies a relationship between model parameters an data. It will be called once for each data item bound to the constraint (this is further explained below). Each form of the constraint procedure tests the `parms PElement`, which is portion of the potential solution space, to see if it satisfies the constraints expressed using `require` or `compare`. Each procedure can have many occurrences of `require` or `compare`. In this case, the failure of any condition is enough to reject the `PElement`. `require`'s will not affect the way data selection and bisection proceed, but `compare`'s will. `compare`'s should be used for constraints that are well behaved and can be expressed as interval inequalities; `require`'s should be used for all other constraints.

5.3 Fit

As discussed above, there is a predefined class, `Fit` that is used to define an instantiation of a constraint solving problem. Any number of `Fit` class instances can be created and operated on within a single program. The routines described below are members of and operate on this class.

`Initialize(<PElementList>)`

`Initialize(<PElement>)` These functions supply the initial search space for a problem instantiation. The search space will be the union of the members of the supplied list.

`add_constraint(<cfn>)`

`add_constraint(<cfn>,<data>,<name>)` These functions permit model constraints to be specified. Any number of constraints can be supplied. `<cfn>` is a constraint function as described above, `<data>` is a `PElementList`, and `<name>` is a character string.

`add_termination(<cfn>,<name>)` Specifies termination conditions. If all active portions of the solution space are found to satisfy this termination condition then further bisection is not needed and the program reports that the termination condition for `<name>` can be satisfied

and stops. Any number of termination conditions can be specified. The algorithm will execute until *one* of the termination procedures are satisfied. `<cfn>` is a constraint function as described above and `<name>` is a character string.

`add_tolerance_level(<ivec>)` permits specification of a tolerance level to which parameters should be computed. Each element of the interval vector `<ivec>` should be an interval centered on zero, whose width corresponds to the tolerance desired. **Tolerances and termination criteria should not be specified for the same instantiation. The behavior of the algorithms under these circumstances is not specified.**

`undecided_points()` returns the list of intervals on the active queue.

`decision()` returns the number corresponding to the decision reached, and returns -1 if no decision has been reached.

`bisect(<n>)` perform up to n iterations of bisection.

`trace()` toggles trace mode. When tracing, reams of information is printed out to help debug programs.

`declare_integer(<dim>)` indicates that the value of `<dim>` takes on only integral values.

`inhibit_bisection(<dim>)` indicates that no bisection takes place on dimension `<dim>`.

5.4 Selection for Efficiency

In [Hager,1992], It was shown that a n-dimensional space can be represented by 2n constraints. The data that generates these 2n constraints, for our search space can be guessed at with a Jacobian approximation of the constraint function. These data points that are guessed to produce the best constraints can be chosen supplying the appropriate information when a constraint is declared. The full form of a constraint declaration is:

```
addconstraint(<cfn>,<data>,<constraint_name>,<rejection_level>, <selection method>,(JacobianProcedur
<JacobianProcedure>)
```

`<rejection_level>` is discussed in the next section; its default value is 0. The default setting for `selection method` is `NO_SELECTION`.

The functions described in this section are still somewhat experimental and not guaranteed to always act you one may expect. They will probably stabilize in future releases.

5.4.1 Jacobian

For jacobian selection, `<selection method>` must be specified as `JACOBIAN` and `<JacobianProcedure>` must be supplied. `<JacobianProcedure>` computes the constraints that can be put on each data field by the Jacobian approximation of the `constraint_name` function. The data that produces the strongest constraints on the n dimensions of the solution space will be used for further computation. A `<JacobianProcedure>` must be of the form

```
IntervalVec jacobian(const PElement& soln, const PElement& sensor_data)
{
   code
}
```

The IntervalVec that is returned must have the same length as `soln` and have the range generated as the bounds of the n-th dimension of `soln` in the n-th interval of the returned IntervalVec. If no bound is needed for the n-th dimension, the n-th returned interval should be `(-HUGE,HUGE)`.

There is a jacobian compiler, `jacobian.m` that is built on top of Mathematica that will quickly create a constraint and jacobian function given a symbolic description of the constraint in `/math/jacobian.m`. Fuller documentation can be found in the `/math/jacobian.doc` file.

5.4.2 Sensitivity Selection

Data selection can also be done by determining for each dimension, which piece of data causes the constraint function to be most sensitive to shrinking of the search space along that dimension. This form of selection can be used by setting the selection field of `addconstraint` to `SENSITIVITY`.

6 Looking at What You've Done

6.1 Interpreting Output

Here is the output generated from one iteration of `gearwcenter`.

```
Iteration 1
Big enough-->   und: 2  in: 1   out: 1

Current Undecided: 4
[  (-1.5, 1.5)  (-1.5, 1.5)  (10, 30)  (1, 10)  (1, 10)  (20, 73.3333)  (0, 0.628319)  (-0.523599, C
 task Big enough with work 3
Working on task Big enough with work 3
Looking for intervals that are undecided
Top rating: 89298.1
[  (-1.5, 1.5)  (-1.5, 1.5)  (10, 30)  (1, 10)  (4, 7)  (46.6667, 73.3333)  (0, 0.628319)  (-0.5235

Task Big enough undecided
Rating sensor = 1.45815 dim 5
```

After the iteration number, a status report is printed for each termination condition.

```
Big enough-->   und: 2  in: 1   out: 1
```

There is only one task constraint for **gearwcenter**, which is called **Big enough.**

After this the number of undecided regions still to be considered is displayed along with a bounding vector of all the undecided regions.

```
Current Undecided: 4
[  (-1.5, 1.5)  (-1.5, 1.5)  (10, 30)  (1, 10)  (1, 10)  (20, 73.3333)  (0, 0.628319)  (-0.523599, C
```

Then a list of all task constraints is printed along with a measure of how much work needs to be done before the final status of that task constraint can be resolved. This measure is the number of undecided regions plus the minimum of the number of regions inside or outside.

```
task Big enough with work 3
```

After this, the task constraint that will be worked on and what is to be done with it are printed.

```
Working on task Big enough with work 3
Looking for intervals that are undecided
```

Then the rating or volume of the smallest undecided element, and the element itself are printed.

```
Top rating: 89298.1
[  (-1.5, 1.5)  (-1.5, 1.5)  (10, 30)  (1, 10)  (4, 7)  (46.6667, 73.3333)  (0, 0.628319)  (-0.5235
```

Then the status of that element with respect to each task constraint is printed.

```
Task Big enough undecided
```

Then the best dimension to bisect upon and a rough measure of how good bisecting on that dimension is, is printed.

```
Rating sensor = 1.45815 dim 5
```

The terminal status report for **gearwcenter** running thirty-one iterations is:

```
Selection method was NO_SELECTION
Total selection time 0.002
Bisection took 500.501

Final undecided:
[  (-1.5, 1.5)  (-1.5, 1.5)  (10, 30)  (1, 10)  (4, 6)  (37.7778, 73.3333)  (0, 0.628319)  (-0.5235

The answer is YES at iteration 31
```

The first three lines give the selection method, time to select data, and time to bisect. Then the bounding vector of the remaining undecided points is printed and the final answer.

7 Example Programs

The following program determines the position, cut, and number of teeth needed to fit a gear to data.

```
/* gearwcenter.cc */
#include <stream.h>
#include <math.h>

#include "PElement.List.h"
#include "compare.h"
#include "Fit.h"

#define SIZE 7

extern void mathout(PElementList&);

PElementList
gen_gear_wcenter_data(double cut, int nteeth, double hubsize, double size)

{
  float i;
  Interval noise(-0.1,0.1);
  IntervalVec t(2);
  PElementList temp;

  for (i=0.0;i<2*M_PI;i+=M_PI/100.0) {
    t[0] = noise + cos(i)*(size + cut * sin(i * nteeth));
    t[1] = noise + sin(i)*(size + cut * sin(i * nteeth));

    temp.push(t);

    t[0] = noise + cos(i)*hubsize;
    t[1] = noise + sin(i)*hubsize;

    temp.push(t);
  }

  return temp;
}

// 0 and 1 are translation
// 2 is k
// 3 is cut
// 4 is hubsize
// 5 is size
// 6 is orientation

void
gear_constraint(const PElement& x, const PElement& y)
```

```
{

  Interval xdiff,ydiff, atanval, temp;

  // Make sure the rotation is as small as possible.
  // Since this is not a structural thing, I'll just stipulate
  // that it has to be true, but it won't be used in bisection
  // decisions.

  require(nlt(2*M_PI/x[2],x[6]));

  // Now, check the fit.

  xdiff = x[0] - y[0];
  ydiff = x[1] - y[1];

  // This is for the center hub.

  temp = sqr(xdiff) + sqr(ydiff);

  // This is an or constraint that checks the above first, and
  // the latter only if the first one doesn't make it.

  compare((temp - sqr(x[4]),EQ,0.0)*
          (temp - sqr((x[5] + x[3] *
sin(x[6]+x[2]*atan2(ydiff,xdiff))))), EQ,0.0);
}

void
bigenough(const PElement& parms)
{
  compare(parms[4],LE,6.0);
}

main (int argc,char** argv)
{
  PElement ii(SIZE);
  PElementList data;
  Fit state(SIZE);

  which_select_data = SENSITIVITY;

  if (argc < 2) {
    cout << "Usage: " << argv[0] << " <iterations>\n";
    exit(1);
  }

  ii[0] = Interval(-1.5,1.5);
  ii[1] = Interval(-1.5,1.5);
  ii[2] = Interval(10,30);
  ii[3] = Interval(1.0,10.0);
```

```
  ii[4] = Interval(1.0,10.0);
  ii[5] = Interval(20.0,100.0);
  ii[6] = Interval(0.0,2*M_PI/10.0);

  state.Initialize(ii);

  data = gen_gear_wcenter_data(3.0,20,5.0,50.0);
  state.add_constraint(gear_constraint,data);
  state.add_termination(bigenough,"Big enough");
  state.declare_integer(2);

  state.bisect(atoi(argv[1]));
}
```

8 To Appear

- Global optimization support. In future releases, it will be possible specify an objective function that should be optimized in addition to hard constraints.
- Second order methods. We will eventually support Gauss-Newton iterations to refine parameters.
- Display. Eventually, there will be some sort of graphics interface.
- Data segmentation. We plan to implement support for segmentation when multiple models are used to describe data.
- Data organization. We plan to implement some support for organized collections of data.

Acknowledgements: This research was supported by DARPA grant N00014-91-J-1577, by National Science Foundation grants IRI-9109116 and DDM-9112458, and by funds provided by Yale University.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, N.Y., 1983.

[2] G. D. Hager. Deciding not to decide using resource-bounded sensing. In *Sensor Fusion III: 3-D Perception and Recognition*, volume 1383, pages 379–390. SPIE, Bellingham, WA, 1990.

[3] G. D. Hager. Task-directed computation of qualitative decisions from sensor data. DCS RR-921, Yale University, New Haven, CT, August 1992. Submitted for Review to the IEEE Transactions on Robotics and Automation.

[4] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.