A short introduction to the CAPD library

Maciej Capiński

January 30, 2019

This is a brief introduction to the `CAPD` (Computer Assisted Proofs in Dynamics) library. The library is written in `C++` and provides an extensive selection of tools for rigorous-interval-enclosure based computations. Here we present only some of its features. Our aim is to keep things as simple as possible. The below presented tools are more than enough though to perform highly nontrivial proofs in dynamical systems.

For more advanced tools we encourage the user to visit the `CAPD` home page: http://capd.ii.uj.edu.pl

Contents

1 Linear algebra - linalg.cpp

1.1 Matrixes and vectors

```
IVector x(3);
IVector y(2);
IMatrix A(2,3);

x[0] = interval(9.0,11.0)/interval(10.0);
x[1] = -x[0];
x[2] = x[0];

A[0][0] = interval(1); A[0][1] = interval(2); A[0][2] = interval(3);
A[1][0] = -interval(3); A[1][1] = -interval(2); A[1][2] = -interval(1);

y = A*x;
```

- `x` is an interval-vector of dimension 3, `y` is an interval-vector of dimension 2 and `A` is a 2×3 interval matrix. After specifying the values of `A` and `x` we compute `y`.
- Instead of using `IVector, IMatrix` we can use `DVector, DMatrix`, which results in having computations performed in doubles. Then we cannot use intervals of course and need to use standard numbers (for example `x(1) = 1.1`)

1.2 Basic operations

```
IMatrix B = transpose(A);
IMatrix C = (A+A)*B;
IMatrix invC = gaussInverseMatrix(C);

IVector v = gauss(C,y);

y = transpose(gaussInverseMatrix(A*B)+C)*y;
```

- Here we show how to compute a transposition of a matrix, how to multiply matrixes, and how to compute an inverse.
- `v = gauss(C,y)` solves the equation `y = C*v` for `v`. In principle we could compute `v = gaussInverseMatrix(C)*y`, but this is much slower than using `v = gauss(C,y)`.
- We can perform a number of computations at the same time.
- All of the above operations work identically for standard matrixes and vectors using `DMatrix` and `DVector`.

1.3 Commands for intervals and interval vectors

For an interval `a` and interval-vectors `x,y` we have the following useful commands:

```
midVector(x);
subsetInterior(x,y);

a.mid();
a.left();
a.right();
a.leftBound();
a.rightBound();
```

- `midVector(x)` returns an interval-vector consisting of a single point which lies in the middle of `x`. `subsetInterior(x,y)` verifies whether `x` is contained in the interior of `y`. If the answer is "yes" then this function returns 1, if "no" then it returns 0.
- `a.mid()`, `a.left()`, `a.right()` return single point intervals which lie in the middle, to the left and to the right of the interval `a` respectively.
- `a.leftBound()` and `a.rightBound()` perform the same tasks as `a.left()` and `a.right()`, but instead of returning an interval they return a number (of type `double`).

1.4 Eigenvectors

CAPD library computes eigenvalues and eigenvectors for matrixes. **This feature is only implemented for non-rigorous computations using `DVector` and `DMatrix` !**

```
DMatrix D(2,2);
D[0][0] = 5;    D[0][1] = 1;
D[1][0] = 3;    D[1][1] = 6;

DVector rE(2), iE(2);
DMatrix rVec(2,2), iVec(2,2);

computeEigenvaluesAndEigenvectors(D,rE,iE,rVec,iVec);
```

- Vectors `rE` and `iE` hold real and imaginary parts of eigenvalues respectively. This means that the `k`-th eigenvalue is equal to $\texttt{rE}(\texttt{k}) + \texttt{iE}(\texttt{k})i$.
- Matrixes `rVec` and `iVec` hold real and imaginary parts of eigenvectors respectively. The eigenvectors are stored in matrixes for convenience. For example, for our matrix `D` both eigenvalues are real, which gives us

  ```
  gaussInverseMatrix(rVec)*D*rVec;
  ```

  as the Jordan form of `D`.

2 Maps - map.cpp

```
IMap f = "par:a,b;var:x,y;fun:1-a*x^2+y,b*x;";
f.setParameter("a", interval(14)/interval(10));
f.setParameter("b", interval(3) /interval(10));

IVector x(2);
IVector y(2);
IMatrix Df(2,2);

x[0] = interval(9,11)/interval(10);
x[1] = interval(-1,1)/interval(10);

y = f(x);
Df = f[x];
```

- Above we have an example of a map $f(x, y) = (1 - ax^2, bx)$ with $a = 1.4$ and $b = 0.3$.
- `f(x)` computes an image of an interval-vector `x`.
- `f[x]` computes the derivative of f at `x`.
- To conduct non-rigorous computations the code needs to be slightly changed:
  - `IMap` needs to be replaced with `DMap`,
  - `IVector` needs to be changed into `DVector`,
  - `IMatrix` needs to be changed into `DMatrix`,
  - all intervals need to be changed to numbers.

3 Integration of ODEs, time shift map - tmap.cpp

In this example we consider the flow $\Phi_t(x)$ generated by an ODE

$$x' = f(x),$$

and show how to compute the map

$$x \to \Phi_T(x)$$

for a given fixed $T \in \mathbb{R}$.

3.1 Non-rigorous computation

```
int order = 20;
DMap f="var:x,y;fun:x*(1-(x^2+y^2)^(-0.5))-y,x+y*(1-(x^2+y^2)^(-0.5));";
DOdeSolver solver(f,order);
DTimeMap Phi(solver);
double T =3.14159265358979;

DVector x(2);
x[0] = 1.;
x[1] = 0.;
DVector y(2);

y = Phi(T,x);

DMatrix der(2,2);

y = Phi(T,x,der);
```

- The integration is performed using a Taylor method. We need to specify the order of this method.
- A map is declared as in Section 2. We then specify that we use a Taylor method for the map of a given order. `Phi` is the flow of the vector field `f`.
- `T` will be the time for our time map Φ_T, `y` will hold the result Φ_T(`x`), the matrix `der` will hold the derivative $D\Phi_T$.
- `y = Phi(T,x)` computes the time shift map.
- When we execute `y = Phi(T,x,der)` then at the same time the derivative of the map is computed (`der` $= D\Phi_T$(`x`) is computed behind scenes).
- If we are not interested in the derivative then we should use `Phi(T,x)` since it is faster than `Phi(T,x,der)`.

3.2 Interval computation

```
int order = 20;
IMap f = "var:x,y;fun:x*(1-(x^2+y^2)^(-0.5))-y,x+y*(1-(x^2+y^2)^(-0.5));";
IOdeSolver solver(f,order);
ITimeMap Phi(solver);
interval T = 2*4*atan(interval(1));

IVector x(2);
x[0] = interval(1);
x[1] = interval(0);
C0Rect2Set R(x);
IVector y(2);

y = Phi(T,R);

IMatrix der(2,2);
C1Rect2Set S(x);

y = Phi(T,S,der);
```

- The code is very similar to the program from Section 3.1. The difference is that we use intervals instead of doubles.
- The `2*4*atan(interval(1))` returns an interval which contains 2π.
- To compute $\Phi_T(\mathtt{x})$ the map `Phi` cannot work on interval-vectors, but needs to work on sets of type "`C0Rect2Set`". The underlying reason is that objects of type `C0Rect2Set` carry more information. We can create a set `R` of type `C0Rect2Set` that is equal to `x` by calling

  ```
  C0Rect2Set R(x);
  ```

  and then turn it back to an `IVector` if we wish by calling

  ```
  x = R;
  ```

- The `R`, which is a `C0Rect2Set` can also be declared in the following way:

  ```
  C0Rect2Set R(x,A,B);
  ```

  where `A` is an `IMatrix` and `b` is an `IVector`. This is equivalent to `R = x+A*b`. If `A` is well aligned with the dynamics of the flow, then **such representation can give much better results**.
- To compute $\Phi_T(\mathtt{x})$ together with $D\Phi_T(\mathtt{x})$ we need to work on sets of type "`C1Rect2Set`". The way we use them is identical to the way that we handle `C0Rect2Set`. In particular, we could also declare:

  ```
  C1Rect2Set S(x,A,b);
  ```

4 Poincaré map - pmap.cpp

Here we consider an ODE $x' = f(x)$ with a Poincaré section $V = \{s = 0\}$ where s is some function $s : \mathbb{R}^n \to \mathbb{R}$. We shall show how to compute the Poincaré map

$$P : V \to V. \tag{1}$$

4.1 Non-rigorous computation

```
int order = 20;
DMap f = "var:x,y,z;fun:x*(1-(x^2+y^2)^(-0.5))-y,x+y*(1-(x^2+y^2)^(-0.5)),-z;";
DCoordinateSection section(3,0);
DOdeSolver solver(f,order);
DPoincareMap P(solver,section);

DVector x(3);
x[0] = 0;
x[1] = 1;
x[2] = 0.5;
DVector y(3);

y = P(x);

DMatrix DPhi(3,3);
DMatrix DP(3,3);

y = P(x,DPhi);

DP = P.computeDP(y,DPhi);
```

- We initiate out map `f`.
- `DCoordinateSection` initiates a section. This reads as follows: we choose the first coordinate (with index 0) out of a three dimensional space. This means that our section is $\{x = 0\}$. We could also choose sections $\{y = 0\}$ and $\{z = 0\}$ by calling, respectively:

  ```
  DCoordinateSection section(3,1);
  ```

  ```
  DCoordinateSection section(3,2);
  ```

- To initiate our Poincaré map `P` we need to specify the Taylor method for integration and the `section`.
- It is essential to highlight one feature. The image of the Poincaré map `y = P(x)` is computed in the **full phase space**. Our section was defined as $\{x = 0\}$. It is therefore natural to view the map only in coordinates (y, z). In such case the image is $(\mathtt{y}[1], \mathtt{y}[2])$, since these are the (y, z) coordinate of vector `y`.

- We can compute the map together with the derivative $D\Phi_\tau$, where τ is the return time to the section, by calling `P(x,DPhi)`. Then, we convert `DPhi` to the derivative of the Poincaré map P by calling

  ```
  DP = P.computeDP(y,DPhi);
  ```

- The derivative is computed in **full dimension of the system**. In our example, since our section was defined as $\{x = 0\}$, the derivative restricted to the (y, z) coordinate on the section is

$$\begin{pmatrix} \texttt{DP[1][1]} & \texttt{DP[1][2]} \\ \texttt{DP[2][1]} & \texttt{DP[2][2]} \end{pmatrix}$$

4.2 Interval computation

```
int order = 20;
IMap f = "var:x,y,z;fun:x*(1-(x^2+y^2)^(-0.5))-y,x+y*(1-(x^2+y^2)^(-0.5)),-z;";
ICoordinateSection section(3,0);
IOdeSolver T(f,order);
IPoincareMap P(T,section);

IVector x(3);
x[0] = interval(0);
x[1] = interval(1);
x[2] = interval(1)/interval(2);
IVector y(3);

C0Rect2Set R(x);

y = P(R);

IMatrix DPhi(3,3);
IMatrix DP(3,3);
C1Rect2Set S(x);

y = P(S,DPhi);
DP = P.computeDP(y,DPhi);
```

- The code is a mirror of the program from Section 4.1, rewritten for intervals.
- Similar to the computation of the time map from Section 3.1, to compute the image of the Poincaré map we need to work on a `C0Rect2Set` type. To compute the image of the Poincaré map together with its derivative we need to work with `C1Rect2Set` type.
- Here also, as in Section 4.1, the image of the Poincaré map and its derivative is computed **in the full phase space**.
- In some cases one might wish to compute a second, third or higher order iterate of the Poincaré map. The following code gives `y` as the third iterate of the map

  ```
  P(R);
  P(R);
  y = P(R);
  ```

  Each time that `P(R)` is called, the set `R` is transformed to its image by `P`.

5 2D Plots and sequences of IVectors - plot-seq.cpp

```
int N = 64;
vector<IVector> p(N);

IMap r = "par:phi;var:x,y;fun:x*cos(phi)-y*sin(phi),x*sin(phi)+y*cos(phi);";
r.setParameter("phi",interval(1)/interval(10));

IVector x(2);
x[0] = interval(-1,1)/interval(1000);
x[1] = x[0]+interval(1);
p[0] = x;

for(int i=1;i<N;i++) p[i] = r(p[i-1]);
```

- The first two lines create a sequence of `N` IVectors called `p`.
- We declare a map `r`, which is a rotation by an angle `phi`.
- We assign the first element `p[0]` from the sequence `p` to be the IVector `x`. Next we let successive elements in `p` to be the successive iterates of `x` by the map `r`.

Now we shall show how we can write our sequence into a file:

```
ofstream outdata;
outdata.open("NameOfFile.txt");
outdata.precision(10);

double xl,xr,yl,yr;
for(int i=0;i<N;i++)
{
    xl = p[i][0].leftBound(); xr=p[i][0].rightBound();
    yl = p[i][1].leftBound(); yr=p[i][1].rightBound();
    outdata << (xr+xl)/2. <<" ";
    outdata << (yr+yl)/2. <<" ";
    outdata << (xr-xl)/2. <<" ";
    outdata << (yr-yl)/2. <<" "<< endl;
}
outdata.close();
```

- We write our results into a file named `NameOfFile.txt` (any name can be given, of course).
- We fill our file with lines of the format

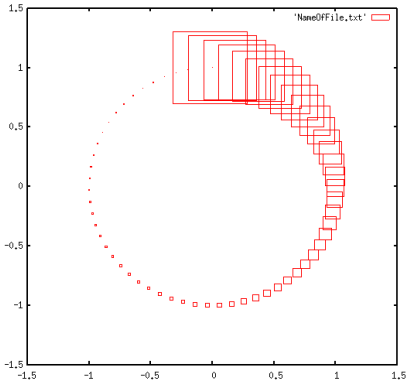
$$x_m \quad y_m \quad r_x \quad r_y$$

each line for a single IVector `p[i]` from our sequence. The numbers are interpreted as follows: (x_m, y_m) is a point which lies in the middle of the IVector `p[i]`. r_x and r_y are the radii of the intervals on the x and y coordinates respectively.

- By running Gnuplot in the directory containing the file `NameOfFile.dat` and typing:

  ```
  plot "NameOfFile.txt" with boxxyerrorbars
  ```

  we obtain a graph of our sequence:



  Typing simply

  ```
  plot "NameOfFile.txt"
  ```

  gives a plot of mid-points for our sequence: