A simple SLAM example with IBEX

Swim 2013

Gilles Chabert

June 6th

Outline

Introduction

The goal of this talk is to show how to implement

- a simple contractor strategy
- for a SLAM problem
- with the IBEX library.

`http://www.emn.fr/z-info/ibex/`

Background

- principles of contractor programming
- basic knowledge of C++

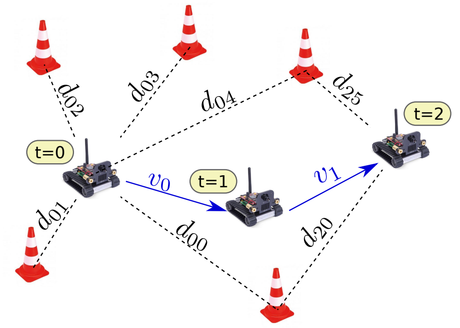
Note. For the sake of simplicity, we shall always use dynamic allocation :

```
MyClass* x = new MyClass(...)
```

just to avoid potential memory fault when pointing to temporary objects.

Outline

Description of the problem



The goal is to charaterize the trajectory of an autonomous robot by enclosing in a box its position x_t for each time step $t = 0 \dots T$.

Problem Description

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)
- and its "speed" vector $v(t_i) = x(t_{i+1}) - x(t_i)$.

Each measurement is subject to uncertainty : *position of the beacons, distances and speed vector*.

Furthermore, we shall consider outliers.

Problem Description

First of all, let us assume that the measurements are all simulated in a seperate 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
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 Nx2 matrix) v[t] is the delta
                                     // vector between x[t+1] and x[t].
/*=========================================================================*/
```

Outline

First strategy (no outlier)

First, we consider no outlier. A simple strategy consists in :

1. creating a contractor for each measurement,
2. calling all these contractors in sequence (composition)
3. performing a fixpoint loop

Let us start by creating contractors for measurements, that is, those related to equations.

Outline

Entering equations and functions

A measurement is an equation.

To enter an equation in `Ibex`, we use the `NumConstraint` class. 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.

Entering equations and functions

Example. For creating the equations :

$$\forall t < T, \quad x_{t+1} - x_t = v_t$$

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]);
        ...

    }
}
```

Entering equations and functions

Sometimes, different constraints are based on the same pattern. It is then often convenient to declare first a `Function` object.

Example. For distances 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]);
    ...
    }
}
```

Outline

Creating basic contractors

We can create now contractors.

To create a contractor with respect to an equation we use the `CtcFwdBwd` class (stands for *forward-backward*).

Example with the constraint $x = 1$:

```
Variable x;
NumConstraint* c=new NumConstraint(x,x=1);
Ctc* ctc=new CtcFwdBwd(*c);
```

Node : 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.

Outline

Combining contractors

We are now ready to build our first strategy. We create all the contractors and push them in a vector `ctc` :

```
vector<Ctc*> ctc;
for (int t=0; t<T; t++) {

    for (int b=0; b<N; b++) {
        NumConstraint* c=new NumConstraint(
                        x,dist(x[t],beacons[b])=d[t][b]);
        ctc.push_back(new CtcFwdBwd(*c));
    }


    if (t<T-1) {
        NumConstraint* c=new NumConstraint(x,x[t+1]-x[t]=v[t]);
        ctc.push_back(new CtcFwdBwd(*c));
    }
}
```

detection of beacons

speed measurement

This vector will be necessary for the composition.

Combining contractors

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 << "initial box =" << box << endl;
fix.contract(box);
cout << "final box =" << box << endl;
```

Outline

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, 0.89360367
05218675] ; [8.02606474898637, 8.443231343191313] ; [1.260805141843543, 1.717998691821
917] ; [8.339079785600994, 8.756246379805939] ; [0.3110569716146419, 0.768250521593016
])
```

The real positions are :

x[0]=8.806965820867086 y[0]=0.6934996231894474
x[1]=8.240950936914649 y[1]=1.517894644489497
x[2]=8.553965973529273 y[2]=0.5681464742605957
. . .

Outline

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` among `N` :

`Ibex` provides the `CtcQInter` operator.
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.

Outline

Second strategy

Let us replace :

```
for (int b=0; b<N; b++) {
    NumConstraint* c=new NumConstraint(
                         x,dist(x[t],beacons[b])=d[t][b]);
    ctc.push_back(new CtcFwdBwd(*c));
}
```

by :

```
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    NumConstraint* c=new NumConstraint(
                         x,dist(x[t],beacons[b])=d[t][b]);
    cdist.push_back(new CtcFwdBwd(*c));
}
ctc.push_back(new CtcQInter(cdist,N-NB_OUTLIERS));
```

Problem : the program runs almost endlessly ! . . . Why ?

Second strategy

. . . 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.

Outline

Inverse contractor

Definition (Inverse contractor)

Given

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

the inverse of C by f is a contractor from $\mathbb{IR}^n \to \mathbb{IR}^n$ that maps a box $[x]$ as follows :

$$[x] \mapsto \{x \in [x], \exists y \in C(f([x]))\}$$

Back to SLAM. Applying the q-intersection on the subset of variables x_t amounts to apply the inverse of this contractor by the projection function

$$x \mapsto x_t$$

Outline

Third strategy

We replace :

```
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    NumConstraint* c=new NumConstraint(
                    x,dist(x[t],beacons[b])=d[t][b]);
    cdist.push_back(new CtcFwdBwd(*c));
}
ctc.push_back(new CtcQInter(cdist,N-NB_OUTLIERS));
```

By :

```
vector<Ctc*> cdist;
for (int b=0; b<N; b++) {
    Variable xt(2);
    NumConstraint* c=new NumConstraint(
                    xt,dist(xt,beacons[b])=d[t][b]);
    cdist.push_back(new CtcFwdBwd(*c));
}

CtcQInter* q=new CtcQInter(cdist,N-NB_OUTLIERS);
ctc.push_back(new CtcInverse(*q,*new Function(x,x[t])));
```

projection function

Third strategy

And now, the program terminates instantaneously. The dispaly shows a (larger) box :

```
initial box =([0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10] ; [0, 10])

final box =([6.231652427835122, 10] ; [0.1253531489288515, 6.09647827181987
2] ; [5.876991438125185, 9.433985116047563] ; [0.9497481702289008, 6.908285
568945011] ; [6.19000647473981, 9.747000152662189] ; [0, 5.958537398716111]
)
```

Outline

Conclusion

- Contractor programmming wih `Ibex` basically amounts to :
  1. enter your mathematical model using `Function` and `NumConstraint`
  2. build basic contractors (`CtcFwdBwd` in our case) with respect to the equations
  3. apply operators to these contractors to yield new (more sophisticated) contractors
- We have seen a simple SLAM example that eventually involves 5 different contrators :
  - `CtcFwdBwd`
  - `CtcCompo`
  - `CtcFixPoint`,
  - `CtcQInter`
  - `CtcInverse`.