IIxp,IIxpxt and IIsym matlab objects

C. Ladroue

October 17, 2007

1 Introduction

IIxp and IIsym `Matlab` objects have been made to easily manipulate 1-dimensional iterated integrals. This document explains how they work and how to use them. IIxpxt objects are similar to IIxp but extended to the multi-dimensional case.

Iterated Integrals are integrals of the form:

$$X_t^{(1221)} = \int_{0 \leq u_1 \cdots \leq u_4 \leq t} dx_1(u_1) dx_2(u_2) dx_2(u_3) dx_1(u_4) \tag{1}$$

We call words the sequence of drivers defining the iterated integrals, for example (1221).

Interstingly, it is possible to write the product of two iterated integrals as a sum of other iterated integrals:

$$X_t^{(w)} X_t^{(v)} = \sum_{z \in w \sqcup\!\sqcup v} X_t^{(z)} \tag{2}$$

where w,v are two words and $\sqcup\!\sqcup$ is the shuffle product - the set of all possible combinations of w and v such that letters from each word are in their original order.

IIxp and IIsym allow you to do simple operations on iterated integrals, in order to build more and more complex objects for, for example, approximating stochastic differential equations.

Requirements

You need to download the following directories: `Classes/@IIxp`, `Classes/@IIxpxt`, `Classes/@IIsym` and `utils_II` and add them to your `Matlab` path. You will also need to compile `utils_II/II.c` if you want to estimate the iterated integrals. The `Matlab` Symbolic Math toolbox is required for `IIsym` objects. IIsym still needs to be uploaded.

You can download these files using a subversion client with:
`svn co https://coropa.svn.sourceforge.net/svnroot/coropa/trunk/MatlabII`
Or download the tarball directly from
`http://coropa.sourceforge.net/coropa_matlabII.tar.gz`

2 IIxp objects

An IIxp object is a linear combination of iterated integrals. Please have a look at `utils_II/test_IIxp.m` for more examples.

Creating an IIxp

To create a new IIxp object, simply use the IIxp constructor as following:

```
>> P=IIxp([],2,[1],1.5,[2 2],-3)
+2
+1.5.X^(1)
-3.X^(2 2)
MaxDepth:5
```

P is an IIxp object, representing $2 + 1.5X_t^{(1)} - 3X_t^{(22)}$. `MaxDepth` is a cut-off point from which no more coefficients will be evaluated. The default is 5, it can be specified as the last variable of the constructor. When IIxp's are multiplied, the number of coefficients grows exponentially and computations can take a very long time. A cut-off point avoids this by truncating the linear combination. In practice, there is no need to go further than 5 or 8.

The rest of the parameters are pairs of words/real numbers.

Arithmetic

A few simple operations are defined for IIxp objects: $+, -, *$ and power. They are used the same way you would use them for regular numbers:

```
>> P=IIxp([],1,[1],4,[2 1],5)
+1
+4.X^(1)
+5.X^(2 1)
MaxDepth:5
>> Q=IIxp([],1,[1],-2,[2],3)
+1
-2.X^(1) +3.X^(2)
MaxDepth:5
>> P+Q
+2
+2.X^(1) +3.X^(2)
+5.X^(2 1)
MaxDepth:5
>> P-Q
+6.X^(1) -3.X^(2)
+5.X^(2 1)
MaxDepth:5
>> P*Q
```

```
+1
+2.X^(1) +3.X^(2)
-16.X^(1 1) +12.X^(1 2) +17.X^(2 1)
-10.X^(1 2 1) -20.X^(2 1 1) +15.X^(2 1 2) +30.X^(2 2 1)
MaxDepth:5
>>  P^3
+1
+12.X^(1)
+96.X^(1 1) +15.X^(2 1)
+384.X^(1 1 1) +120.X^(1 2 1) +240.X^(2 1 1)
+480.X^(1 1 2 1) +960.X^(1 2 1 1) +1440.X^(2 1 1 1) +150.X^(2 1 2 1) +300.X^(2 2 1 1)
+600.X^(1 2 1 2 1) +1200.X^(1 2 2 1 1) +1200.X^(2 1 1 2 1) +2400.X^(2 1 2 1 1) +3600.X^(
MaxDepth:5
```

Note that P^3 should contain words of length 6 but those are not evaluated since `MaxDepth` is set to 5.

Integrating

It's also possible to integrate with respect to one or more drivers:

```
>> P=IIxp([],2,[1],1.5,[2 2],-3)
+2
+1.5.X^(1)
-3.X^(2 2)
MaxDepth:5
>> X(P,[2 1])
+2.X^(2 1)
+1.5.X^(1 2 1)
-3.X^(2 2 2 1)
MaxDepth:5
```

It has the effect of adding the number of the driver on the right of each integral.

Estimating

When the actual value of the drivers are known, it is possible to evaluate the IIxp with `estimate.m`. This function calls `II.c`, written by Terry Lyons.

```
>> P=IIxp([],2,[1],1.5,[2 2],-3)
+2
+1.5.X^(1)
-3.X^(2 2)
MaxDepth:5
>> dt=1E-3;resolution=1E3;
>> Drivers=[dt*(0:resolution-1);[0 cumsum(randn(1,resolution-1)*sqrt(dt))]];
>> plot(Drivers(1,:),estimate(P,Drivers));
```

Example: stochastic differential equation

Let the SDE $dy_t = (-4t^2+5t+2)dt+(-t+2)dW_t$. A solution can be computed recursively by the Picard iteration principle:

$$\begin{aligned} y_t^0 &= 0 \\ y_t^{n+1} &= y_0 + \int_0^1 (-4(y_t^n)^2 + 5y_t^n + 2)dt + \int_0^1 (-y_t^n + 2)dW_t \end{aligned}$$

This can be easily implemented with an IIxp object up to a finite number of iterations[1]:

```
>> P=IIxp;y0=1;
>> for k=1:5
P=y0+X(-4*P^2+5*P+2,'1')+X(-P+2,'2');
end;
>> P
+1
+3.X^1 +1.X^2
-9.X^11 -3.X^12 -3.X^21 -1.X^22
-45.X^111 +9.X^112 -15.X^121 +3.X^122 -15.X^211 +(...)
+783.X^1111 +45.X^1112 +189.X^1121 -9.X^1122 +(...)
-54.X^11111 -810.X^11112 -1098.X^11121 -54.X^11122 +(...)
```

And given the drivers' value, we can evaluate the solution:

```
>> D=[linspace(0,1,500);brown(500,[0 1])];
>> y=estimate(P,D);
>> plot(y)
```

The Picard iteration algorithm is implemented in `utils_II/picard_poly.m`.

3 IIxpxt objects

IIxpxt are a natural extension of IIxp's to the multi-dimensional case, where each dimension contains an IIxp object. `utils_II/test_IIxpxt.m` demonstrates some of the functions associated with these objects. It is possible to do simple arithmetic, directly reference and assign coefficients, evaluate polynomial of more than one variables and compute the approximate solution of a polynomial SDE via the Picard Iteration algorithm.

For example, given the 2D SDE

$$dX = \begin{pmatrix} 0 & -0.1 \\ -0.2 & 0 \end{pmatrix} X dt + \begin{pmatrix} 0.003 & 0 \\ 0 & 0.04 \end{pmatrix} dW_t$$

where dW_t is a two-dimensional Brownian motion, we calculate an expansion of the solution up to depth 5 with:

[1] Again, note that coefficients for integrals whose word is of length more than `MaxDepth` (here 5) are not evaluated.

```
>> M=cell(2,3);
>> M{1,1}=[0 1 -0.1];
>> M{1,2}=[0 0 0.03];
>> M{1,3}=0;
>> M{2,1}=[1 0 -0.2];
>> M{2,2}=0;
>> M{2,3}=[0 0 0.04];
>> X0=[1;1];
>> R=Picard_IIxpxt(X0,M,5)
+1
-0.1.X^(1) +0.03.X^(2)
+0.02.X^(1 1) -0.004.X^(3 1)
-0.002.X^(1 1 1) +0.0006.X^(2 1 1)
+0.0004.X^(1 1 1 1) -8e-05.X^(3 1 1 1)
+1.2e-05.X^(2 1 1 1 1)
MaxDepth:5
+1
-0.2.X^(1) +0.04.X^(3)
+0.02.X^(1 1) -0.006.X^(2 1)
-0.004.X^(1 1 1) +0.0008.X^(3 1 1)
+0.0004.X^(1 1 1 1) -0.00012.X^(2 1 1 1)
+1.6e-05.X^(3 1 1 1 1)
MaxDepth:5
```

R is then a two-dimensional IIxpxt object containing the expansion of the solution of the SDE, when $X_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

4 IIsym objects

Description

IIsym objects are similar to IIxp objects but work as symbolic quantities, that is, you don't have to specify values for the coefficients of the IIxp. For example, it is possible to define objects such as $1+aX_t^{(1)}+(b-c*a)X_t^{(21)}$, and manipulate them without having to give the program values for a, b and c:

```
>> P=IIsym([1],'a',[1 2],'a+b',[2 1],'c')
+a.X^(1)
+a+b.X^(1 2) +c.X^(2 1)
MaxDepth:5
```

Please have a look at `utils_II/test_IIsym.m`. @IIsym requires the symbolic toolbox.

Arithmetic

Simple operations are defined ($+, -, *$ and power), with the added bonus that you can now use symbolic variables:

```
>> P*'a-b*c'
+(a-b*c)*a.X^(1)
+(a-b*c)*(a+b).X^(1 2) +(a-b*c)*c.X^(2 1)
>> P*P
+2*a^2.X^(1 1)
+4*a*(a+b).X^(1 1 2) +2*a*(a+b)+2*a*c.X^(1 2 1) +4*a*c.X^(2 1 1)
+4*(a+b)^2.X^(1 1 2 2) +2*(a+b)^2+2*(a+b)*c.X^(1 2 1 2) +4*(a+b)*c.X^(1 2 2 1) +4*(a+b)*
```

The Picard iteration can be implemented as easily as before. For example, given the SDE $dy_t = (at^2+bt+c)dt+(et+f)dW_t$, we simply write the following loop:

```
>> P=IIsym;
>> for k=1:5,P='y0'+X('a'*P^2+'b'*P+'c',[1])+X('e'*P+'f',[2]);end;
>> P
+y0
+a*y0^2+b*y0+c.X^(1) +e*y0+f.X^(2)
+2*a*(a*y0^2+b*y0+c)*y0+b*(a*y0^2+b*y0+c).X^(1 1) +e*(a*y0^2+b*y0+c).X^(1 2)+(...)
+a*(2*y0*(2*a*(a*y0^2+b*y0+c)*y0+b*(a*y0^2+b*y0+c))+2*(a*y0^2+b*y0+c)^2)+(...)
+(...)
```

This is notably slower[2] than the IIxp version but we usually need to compute the object only once: we can use it afterwards by giving actual values to the variables to approximate the solution of all SDEs of that particular form.

Instantiation and estimation

Instantiation refers to the operation of giving values to some or all the abstract variables present in an IIsym object. Estimation is the operation of giving values to the iterated integrals, providing we know the actual values of the drivers.
instantiation is done with the `instantiate.m` function. The result is a new IIsym object. If there is no abstract variables left in the IIsym object, it can be turned into a IIxp object. `findsym.m` displays the abstract variables used in an IIsym object.

```
>> P=IIsym([1],'a',[1 2],'b+c',[],1);
>> disp(findsym(P))
a, b, c
>> instantiate(P,{'a' 'b' 'c'},[-1 0.2 3])
+1
-1.X^(1)
+16/5.X^(1 2)
```

[2] A bit more than 11 seconds on a personal laptop

```
>> toIIxp(ans)
+1
-1.X^(1)
+3.2.X^(1 2)
```

There are two ways of estimating an IIsym object:`estimate.m` and `fastestimate.m`. The first one takes an IIsym object P and a list of drivers and replaces the iterated integrals by their actual values. The result is a symbolic object, a vector of length d, where d the number of datapoints. One could then evaluate the object by setting particular values to the abstract variables. This, however, is so slow as being completely impractical. If your goal is to estimate the actual value of an IIsym object given the value of all its variables and the drivers, use `fastestimate.m`. This function requires values for all variables and the drivers but is 1000 times faster[3] than using `estimate.m` first and then `subs.m` to instantiate the symbolic object.

As a example, let's approximate the SDE $dy_t = (at + b)dt + (ct + d)dW_t$ in the general case, and use it for particular values.

```
>> P=IIsym;
>> for k=1:5,P='y0'+X('a'*P+'b',[1])+X('c'*P+'d',[2]);end;
>> dt=1E-3;resolution=1E3;
>> Drivers=[dt*(0:resolution-1);[0 cumsum(randn(1,resolution-1)*sqrt(dt))]];
>> y=fastestimate(P,{'a' 'b' 'c' 'd' 'y0'},{1 2 3 4 1},Drivers);plot(y);hold on
>> y=fastestimate(P,{'a' 'b' 'c' 'd' 'y0'},{-1 2 -3 4 1},Drivers);plot(y);
```

[3]Basically it boils down to the difference between manipulating a big array of symbolic objects and a big array of doubles.