Pseudocode Premia ACDP

Jacopo Corbetta

March 3, 2020

The structure ACDP [2] has 3 parameters λ, D, σ for the meaning we refer to http://www.matapp.unimib.it/~fcaraven/download/papers/acdp-final.pdf. We focus on different methods, and for the variance reduction we use a stratification method based on the number of crises between today and the maturity time.

The Model

This model relates only on 4 parameters and 3 sources of randomness:

(P1) $\lambda \in (0, +\infty)$ which represents the inverse of the average waiting time between two crises;

(P2) $D \in (0,\ 1/2]$ which determines the change of time $t \mapsto t^{2D}$ which expresses the trading time right after a crisis;

(P3) $\sigma \in (0, \infty)$ proportional to the average volatility;

(P4) ν probability measure on $(0, \infty)$ related to the distribution of the volatility;

(R1) $W = (W_t)_{t\geq 0}$ a standard brownian motion;

(R2) $\mathcal{T} = (\tau_n)_{n\in\mathbb{Z}}$ a Poisson process on $\mathbb{R}$ with intensity λ;

(R3) a sequence $\Sigma = (\sigma_n)_{n\geq 0}$ of positive i.i.d random variables with law ν.

Assume that $W, \mathcal{T}, \Sigma$ are defined on the proability space $(\Omega, \mathscr{F}, \mathbb{P})$, are independent, and number the points of $\mathcal{T}$ in such a way that $\tau_0 < 0 < \tau_1$. For $t \geq 0$ define

$$i(t) := \sup\{n \geq 0 : \tau_n \leq t\} = \#\{\mathcal{T} \cap (0, t]\}$$

so that $\tau_{i(t)}$ is the last point in $\mathcal{T}$ before t. Then they introduce the process

$$I_t = I(t) := \sigma^2_{i(t)}(t - \tau_{i(t)})^{2D} + \sum_{k=1}^{i(t)} \sigma^2_{k-1}(\tau_k - \tau_{k-1})^{2D} - \sigma_0(-\tau_0)^{2D}. \tag{1}$$

and define the model $X = (X_t)_{t\geq 0}$ by

$$X_t = W_{I_t}. \tag{2}$$

The model can be represented also in the following way:

$$dX_t = v_t dB_t \tag{3}$$

where $(B_t)_{t\geq 0}$ is also a standard brownian motion and $(v_t)_{t\geq 0}$ is an indipendent process defined by

$$v_t := \sqrt{I'(t)} = \sqrt{2D}\sigma_{i(t)}(t-\tau_{i(t)})^{D-\frac{1}{2}} \tag{4}$$

where $I'(s) = \frac{d}{ds}I(s)$. We remark that when $D < \frac{1}{2}$ the process v_t has singularities in τ_n.
This model despite its simplicity, displays some interesting features of financial series like the multiscaling of the moments and heavy tails distributions. The main criticism to this model is that the volatility (represented by the process v_t) is too smooth compared with what is observed in real data. Therefore I have tried to improve this aspect by introducing more irregularity with two different multifractal methods: one using multiplication and the other using addiction. Via numerical calibration it has een showed that the random variables $(\sigma_n)_{n\in\mathbb{N}}$ are in reality constant so we will consider them in such a way.

Given the fact that the model is a Black and Scholes one, with stochastic volatility which is independent of the driving brownian motion the idea for the pricingis to conditioning the Black and Scholes formula with respect to the volatility.

The riparametrization

Recently a new choiche of parametres has been made, in particular it has been introduced the new parameter V such that

$$V^2 = \sigma^2\Gamma(2D+1)\lambda^{(1-2D)}$$

In this way we have that $I(T) = V^2TY$ where Y is a random variable with expected value 1, so that we have $\mathbb{E}[I(T)] = V^2T$ displaying a linear behaviour in T. Thus we can interpret the parameter V as the expected istant volatility.

Jump Generator

In this model the most important thing are the jumps which represent the crises so this method takes as entries the model, the time to maturity, the number of jumps and a pointer to a vector of size k+2.We begin by generating a $k+2$-vector of uniform r.v. (called Valori) with which we generate the values of the jumps. In the first entry of the vector it puts τ_0 simulated as an exponential of parameter λ.

```
Tau[0]=log(Valori[0])/modelo.lambda;
```

while the next k entries of Tau are filled in a uniform way over $[0,T]$ in the following way

```
double fattcom;
double prodpar;
fattcom=1.0;
prodpar=1.0;
for(i=1;i<=k+1;i++){
```

```
fattcom=fattcom*Valori[i];
}
for(i=1;i<=k;i++){
prodpar=prodpar*Valori[i];
Tau[i]=T*log(prodpar)/log(fattcom);
}
```

Finally the last element of Tau is the maturity T.

Changed Time Generator

This model is substantially a changed time model, driven by the Poisson process, so we use this method to obtain the final changed time to insert in the BS formula. It takes in entry the model, the final time T, and the numbers of jumps k. Using the method *Jump Generator* then we use main formula of the model

$$I(T) = \sigma^2 * (T - \tau_{i(T)})^{2D} + \sum_{j=1}^{i(T)} \sigma^2 (\tau_j - \tau_{j-1})^{2D} - (-\tau_0)^{2D};$$

with a for cycle to get the final value which will be returned.
We have two different methods which can do this, one that takes as a entry the number of jumps and generates them inside, the other that takes as a entry also the exact time of jumps

```
 //gives the final time with the jumps as a entry
double changedTimeGeneratorVec(double sigma,double D, int k, double* Jumps){
  for cicle in order to get
  finale=sigma*sigma*(finale+pow((Jumps[k+1]-Jumps[k]),2*D));
  return finale;
}


//gives the final time with the number of jumps as entry
double changedTimeGenerator(double lambda, double sigma, double D, double T, int k){
  create a jump vector with jump generator and then use the
  same method as changedTimeGenerator
}
```

Path generator

Since we can give an exact formula for the discretized path once we know the jumps we won't use Euler's scheme but the following formula

$$\begin{cases} X_{t_{i+1}} = X_{t_i} + \sqrt{I(t_{i+1}) - I(t_i)} N_{i+1} + r(t_{i+1} - t_i) - \frac{I(t_{i+1}) - I(t_i)}{2} & X_0 = 0 \\ S_{t_{i+1}} = S_0 e^{X_{t_{i+1}}}, \end{cases} \tag{5}$$

where $\{N_i\}_{i \in \mathbb{N}}$ is a sequence of independent normal random variables. The main issue in this representation is to create a discretization path such that $t_{i+1} - t_i$

and $I(t_{i+1}) - I(t_i)$ are of the same order. particoular attention should be given to the jumps moments because in that moments the graphics of $I(t)$ is almost vertical. The first idea was to discretize in a uniform way but in this case when there is a jump there is a big discrepancy between the term rt e the one in $I(t)$ so we decided to make thicker the path right after the jumps, so using the $100 - k * 10\%$ for the main discretization and put 10% beetween any jump-time and the second istant of the original discretization, but this method works only if k is small (in our first code we have imposed that $k = 0, 1, 2$).

```
 M= number of discretization
jump generator(k,Tau)
int orig_numb=M*(10-k)/10;
delta=10/orig_numb;
for(int i=0; i< orig_numb;i++){
  orig_discr[i]=i*delta;
}
if(k=0){
  discr_fin=orig_discr;
}
if(k=1){
  find j such that orig_discr[j-1]<Tau[1] and orig_discr[j]=>Tau[1]
  num1=0.1*M;
  delta1=(orig_discr[j]-Tau[1])/num1;
  for(int l=0;l<j;l++){
    discr_fin[l]=orig_discr[l];
  }
  discr_fin[j]=Tau[1];
  for(l=1;l<= num1;l++){
    discr_fin[j+l]=Tau[1]+l*delta1;
  }
  for(l=1;l<=orig_num +2-j;l++){
    discr_fin[j+num1+l]=orig_discr[j+l+1];
  }
}
if(k=2){
find j such that orig_discr[j-1]<Tau[1] and orig_discr[j]=>Tau[1] and
     q such that orig_discr[q-1]<Tau[2] and orig_discr[q]=>Tau[2]
  num1=0.1*M;
  delta1=(orig_discr[j+1]-Tau[1])/num1;
  for(int l=0;l<j;l++){
    discr_fin[l]=orig_discr[l];
  }
  discr_fin[j]=Tau[1];
  for(l=1;l<= num1;l++){
    discr_fin[j+l]=Tau[1]+l*delta1;
  }
  for(l=1;l<=q-j-2;l++){
    discr_fin[j+num1+l]=orig_discr[j+l+1];
  }
  int s=j+num1+l;
```

```
  forfor(l=1;l<= num1;l++){
    discr_fin[s+l]=Tau[2]+l*delta1;
  }
  for(l=1;l<=orig_num +2-q;l++){
    discr_fin[s+num1+l]=orig_discr[q+l+1];
  }
}
calculate I(t) on discr_fin
calculate X(t) and S(t) on discr_fin
```

But then in this case we are not so sure that we can controll the general increment term $r(t_{i+1}-t_i)-\frac{I(t_{i+1})-I(t_i)}{2}$ so we thought of choosing a discretization path of variable lenght (but of minimum M steps) such that

$$\frac{I(t_{i+1})-I(t_i)}{2} < r\delta \qquad \text{with } \delta = \frac{T}{M}$$

and such that the jump moments are included in the path. This can give some problems with the next part especially for the integration. Since we are not able to use the class *vector* we have some problems in creating the array of "random" lenght but using the internal class *pnlVector* so that at every iteration we create a longer vector

```
void generaIstantiTraiettoria(double lambda, double sigma, double D,
    int Minimal_lenght, double T,double r, pnlVec* Tau, pnlVec* value,
    pnlVec* trj_temp){
  //fix delta(T/M) and delta1, if the next jump is before present+delta
  the next discretization is the jump, else we findthe istant temp1 in which
  we have the realtion before and we choose the smallest. we repeat
}
```

Now that we have the discretization times creating the trajectory is not that difficult, again the main problem is to return not only one (pnl) vector but two vectors (values of the underlying and times).

```
void genTraiettoria(double lambda, double sigma, double D, int Minimal_lenght,
double S0, double r,double T,pnlVec* traiettoria,
pnlVec* tempi, pnlVec* Tau){
  //I generate the discretization
  generaIstantiTraiettoria(lambda,sigma, D, Minimal_lenght,T,r,Tau, valori_temp,tempi);
  //creation of vector of times of exact lenght
  pnlVec esponente;
  lungh=tempi->size;
  esponente= pnl_vect_create(lungh);
  //computation of the values of the exponent and the underlying
}
```

This function gives back the values of the underlying, the discretized path and their (common) lenght. The only problem in using it is that we have to create at the beginning two very big vectors vectors even if we care only for a small group of events, so that after we exit the function we should create two new vectors of lenght $size+1$ where we copy the results obtained.

European Call

This method take in entrance the model, the initial value, the time to maturity, the interest rate (thought constant),the total number of tries, and the strike, in order to give the value of a call.
Using the fact that in this model the time change and the brownian motion are independent we can use a Black and Scholes formula conditioning with respect to $I(T)$, that is using as volatility $\sqrt{\frac{I(T)}{T}}$. We also use a stratification method (with respect to the number of jumps) for another reduction of variance. The first idea is to use just the probability associated with the number of jumps (with the parameters found in ACDP with a 30 days maturity we have approximately 97.1% for no jumps, 2.8% with one jump, 0.05% with 2 jumps. The idea now is to improve this first stratification using the conditional volatility: use a 10% of the samples equally divided in the three cases just to find those values and the dividing the other 90% divided with the new stratification (QUESTION: is it worthy? seen how much they are unequally divided we risk a grater loss than the gain: we risk to have the 0 jumps case which becames "underestimated").
In the case of 0 jumps we are looking for a semiclose formula in the sense

$$\mathbb{E}\left(BS\left(\frac{I(T)}{T}\right)|k=0\right) = \int_{\mathbb{R}^+} BS((\sigma^2(T+t)^{2D} - (t)^{2D})/T)\lambda e^{-\lambda t}dt \qquad (6)$$

which can be solved in a numerical way using the alrady implemented function *pnl_integration*.In this case I decided to integrate beetween 0 and $\frac{10}{\lambda}$, using as number of discretization the theoretical number of drawings I should have used if I use a Montecarlo method.
Since $D < \frac{1}{2}$, and the crises are relatively rare events in this model (so that λ is very small)we have that the value of $(T+t)^{2D} - (t)^{2D}$ is close to 0 when the time t is far away in the past; this implies that when we compute the Black & Scholes formula with $t \geq \frac{1}{\lambda}$ we have that $|d_1| \simeq |d_2| \lll 0$ with the sign depending only from

$$\log\left(\frac{S_0}{K}\right) + rT$$

so we can approximate the tail simply adding

$$\left(S_0 - e^{-rT}K\right)^+ \int_{\frac{10}{\lambda}}^{\infty} \lambda e^{-\lambda x}dx = \left(S_0 - e^{-rT}K\right)^+ e^{-10}$$

For the other two cases we use just a simple Montecarlo and then we average all the results together.

```
10% get conditional volatility;->vol0, vol1,vol2;
p0=exp(-lambda*T)
p1=exp(-lambda*T)*lambda*T;
p2=exp(-lambda*T)*(lambda*T)^2/2;
tenta[0]=N*p0*vol0/(p0*vol0+p1*vol1+p2*vol2)
tenta[1]=N*p1*vol1/(p0*vol0+p1*vol1+p2*vol2);
tenta[2]=N*p2*vol2/(p0*vol0+p1*vol1+p2*vol2)

price[0]=closed formula;(using pnl_integration, Tenta[0] discretization)+tail
price[1]=montecarlo(BS, tenta[1], 1 jump)
```

```
price[2]=montecarlo(BS, tenta[2], 2 jump)

priceCall=price[0]*tenta[0]/N+price[1]*tenta[1]/N+price[2]*tenta[2]/N
```

We have been pointed out that such a method is not particoularly robust if the user wants to input strange values of λ, that is to say he thinks that there is a crisis everyday so we decide to implement the exact method but with the maximal number of jumps that depends on $T\lambda$ that is the average number of jumps: instead of 2 we have taken $kMax = T\lambda + 2$ so that for ususal values we will still have a 2 jumps at maximum, while in the other cases we can have more.
There is also a second option, that is to choose $kMax$ such that the probability of having more than $kMax - 2$ jumps is less than 0.5%, what actually happens in the first trials is that this probability is much smaller

```
  //I find kMax (number of layers)
  for cicle in order to find j such that
  P(jumps in [0,T]>k)<0.005)
  kMax=j +2;

  //stratification method using N/10 of the total (kMax layers)
  I use both probability and variance in order to obtain the
  number of essay (Tenta[k]) in each level. The prices are
  calculated via integration (0-level) or black and scholes

  //I find the price with no crises by integrating
  prezzi[0]=pnl_integration(&func, 0.0, 10./lambda, Tenta[0],
  "trap")+exp(-10)*max(iniziale-(exp(-r*T)*strike),0);
  //I find the prices fo the other layers (k=1,2,..,kMax) with
  a for cicle using the close formula, each time Tenta[k] times

  //I find the final price
  prezzo=prezzo/tentatot;

  //I divided by tentatot which is the actual number of guesses
  //(usually it differs from N by 1 or 2)
  return prezzo;
}
```

The problem of this method is that it entroduces a natural bias in the simulation, even if a little one. so we decided to use a 4 layers stratification: the first three with 0, 1 and 2 jumps and the last with more then 2 jumps. So as before we utilize 10% of the draws to compute the ripartition for the stratification of the remainig 90%. The first three groups pose no problems, for the last one (the "more than 2") the situation becomes a little more complex, infact in this case we have to simulate the number (and time) of the jumps conditioning that there are more than 3 (and for this we will create a new function *JumpGenerator3*): we find the third jumping time (conditioning it is less then T) and we get the first and the second as uniforms on $[0, \tau_3]$ than we add exponential of parameter λ till they surpass T. In order to find τ_3 knowing it is less than T we have to compute

$$\mathbb{P}(\tau_3 < T) = \mathbb{P}(Ga < T)$$

where Ga follows a $\Gamma(3, \frac{1}{\lambda})$, simulate a uniform ($u \sim U([0,1])$) and solve

$$\frac{\mathbb{P}(Ga < t)}{\mathbb{P}(Ga < T)} = u$$

which can be done using the functions already present in the library (*pnl_root.h*). So we have the jumpGenerator3 which generates the jumps vector of random length when there are more than 3 jumps

```
 //creates a vector with more than 3 jumps(last is T)
void jumpGenerator3(double lambda, double sigma, double D, double T, pnlVec* Tau){
  //I solve in t ( p(3jumps<t)/p(3jumps<T)=u)
  PnlFunc func;
  func.function = terzoSalto;
  double tmp[] = {lambda,T,u};
  func.params = (void*) tmp;
  t = pnl_root_brent(&func, 0.0, T ,&tol);
  //where u is a uniform random variable and terzoSalto
  is the function
  f(t,T,lambda,u)= p(3jumps<t)/p(3jumps<T)-u
  with each jump is an exponential of parameter lambda

  //do-while code in order to add the jumps
}
```

This procedure has to be obviousely implemented also in the computation of the prices in the fourth layer, and so we get this code for the call

```
double pnl_ACDP_call2(double lambda, double v, double D , double T, double strike, double
  //four layer stratification
  0-> integration
  1,2->simple black and scholes
  3+ -> jumpGenerator3 +black and scholes
}
```

We decide to implement also a second method for the pricing, a more direct one even if slower. In fact we decide to simulate the entire trajectory and then give the price as $Max(S_T - k, 0)$. This model takes as data the parameters of the model, the number of simulations in the Montecarlo method and the minimal number of discretizations for each path.

```
double pnl_walk_ACDP_call(double lambda, double v, double D , double T,
          double strike, double r, int N, double s0, int Minimal_lenght){

I use 10% of the essais in order to determine the layers
using Black and Scholes

  0,1,2-> jump generator +create path
  3+ -> jumpGenerator3 + create path

}
```

1 The code

Here is the real code in all of its parts.
•The indroduction with all the necessary classes:

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "pnl/pnl_vector.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_finance.h"
#include "pnl/pnl_integration.h"
#include "pnl/pnl_specfun.h"
#include "pnl/pnl_root.h"
#include "pnl/pnl_cdf.h"
```

•The functions which give the values of a uniform and a gaussian random variable:

```
 double unif(){
  return (double)rand()/RAND_MAX;
}

double gauss(){
  return sqrt(-2*log(unif()))*cos(2*M_PI*unif());
}
```

There is also a **pnl** version but it obliges me to include other classes and it is not as userfriendly as mine.
•The definition of the functions I use in the integration for the calculus of the price of the call when there are no jumps, and the one i use in order to find the third jump before T knowing that there are at least 3 jumps before the expiring date.

```
 static double integranda (double x, void *v){
  double y;
  double * p = (double*)v;
  y=SQR(p[0])*(pow((p[1]+x),2*p[2])-pow(x,2*p[2]));
  return pnl_bs_call(p[4],p[5],p[1],p[6],0,sqrt(y/p[1]))*p[3]*exp(-p[3]*x);
}

static double terzoSalto(double x, void *v){
//v={lambda, T,u}
  double *vi = (double*)v;
  double p,q;
  p=1-(exp(-vi[0]*x)*(1+vi[0]*x+0.5*x*x*vi[0]*vi[0]));
  q=1-(exp(-vi[0]*vi[1])*(1+vi[0]*vi[1]+0.5*vi[1]*vi[1]*vi[0]*vi[0]));
  return ((p/q)-vi[2]);
}
```

•The function that creates a vector of crises-time of length $k+2$, where k is the number of crises between today and the expiring date. the first value is the last past crise, while the final is the expiring date.

```
void jumpGenerator(double lambda, double sigma, double D, double T, int k,
PnlVect* Tau){
int i=0;
PnlVect* Valori;
Valori= pnl_vect_create(k+2);
double fattcom;
double prodpar;

for(i=0;i<=k+1;i++){
  pnl_vect_set(Valori,i,unif());
}
pnl_vect_set(Tau,0,log(pnl_vect_get(Valori,0))/lambda);
if(k>0){

fattcom=1.0;
prodpar=1.0;
for(i=1;i<=k+1;i++){
fattcom=fattcom*pnl_vect_get(Valori,i);
}
for(i=1;i<=k;i++){
prodpar=prodpar*pnl_vect_get(Valori,i);
pnl_vect_set(Tau,i,T*log(prodpar)/log(fattcom));
}
}
pnl_vect_set(Tau,k+1,T);
pnl_vect_free(&Valori);
}
```

•The function which creates the crises-time vector, knowing that there will be at least 3 crises between today and the expiryng date

```
void jumpGenerator3(double lambda, double sigma, double D,
    double T, PnlVect* Tau){
   int i=0;
int crazy;
double x;
double tmp[3];
double u;
double t;
double tol;
double fattcom;
double prodpar;
PnlVect* Valori;

Valori= pnl_vect_create(3);
tol=T/1000;
crazy=pnl_vect_resize (Tau, 1);
```

```
x=unif();
pnl_vect_set(Tau, 0, log(x)/lambda);
u=unif();
tmp[0]= lambda;
tmp[1]=T;
tmp[2]=u;
PnlFunc func;
   func.function = terzoSalto;
   func.params = (void*) tmp;
t = pnl_root_brent(&func, 0.0, T ,&tol);
crazy=pnl_vect_resize(Tau,4);
pnl_vect_set(Tau, 3, t);

for(i=0;i<=2;i++){
  pnl_vect_set(Valori,i,unif());
}
fattcom=1.0;
prodpar=1.0;
for(i=0;i<=2;i++){
fattcom=fattcom*pnl_vect_get(Valori,i);
}
for(i=0;i<2;i++){
prodpar=prodpar*pnl_vect_get(Valori,i);
pnl_vect_set(Tau,i+1, t*(log(prodpar)/log(fattcom)));
}
i=4;
do{
  x=unif();
t=t-log(x)/lambda;
if(t<T){
crazy=pnl_vect_resize(Tau,i+1);
pnl_vect_set(Tau, i, t);
i=i+1;
}
}while(t<T);
crazy=pnl_vect_resize(Tau,i+1);
pnl_vect_set(Tau, i, T);
pnl_vect_free(&Valori);
}
```

•The functions which return $I(T)$ (one generates the crises time vector inside, the other takes it as a entry)

```
double changedTimeGenerator(double lambda, double sigma, double D,
      double T, int k){
  int i=0;
  double finale;
  PnlVect* Jumps;
  Jumps=pnl_vect_create(k+2);
  jumpGenerator(lambda,sigma,D,T,k,Jumps);
```

```
  finale=-pow(-pnl_vect_get(Jumps,0),2*D);
  if(k>0){
    for(i=1;i<=k;i++){
      finale=finale+pow((pnl_vect_get(Jumps,i)-
                  pnl_vect_get(Jumps,i-1)),2*D);
    }
  }
  finale=SQR(sigma)*(finale+pow((pnl_vect_get(Jumps,k+1)
                   -pnl_vect_get(Jumps,k)),2*D));
  return finale;
}

double changedTimeGeneratorVec(double sigma,double D,
                               int k, PnlVect* Jumps){
  int i=0;
  double finale;
  finale=-pow(- pnl_vect_get(Jumps,0),2*D);
  if(k>0){
    for(i=1;i<=k;i++){
      finale=finale+pow((pnl_vect_get(Jumps,i)
                       -pnl_vect_get(Jumps,i-1)),2*D);
    }
  }
  finale=SQR(sigma)*(finale+pow((pnl_vect_get(Jumps,k+1)
                        -pnl_vect_get(Jumps,k)),2*D));
  return finale;
}
```

•The fuction which gives the price of a call using the first stratification method (find k such that $\mathbb{P}$(more than k jumps$< 0.5\%$) and then choose kMax=k+2) and which was not implemented.

```
double pnl_ACDP_call(double lambda, double sigma, double D,
                     double T, double strike, double r,
                     int N, double iniziale){
  if(D==0.5){
    return pnl_bs_call(iniziale,strike,T,r,0,sigma);
  }
  int kMax;
  int j;
  int x;
  j=0;
  double somma;
  double pro;
  somma=exp(-lambda*T);
  while(somma < 0.995)
  {
    j=j+1;
    x= pnl_sf_fact(j);
    pro=(exp(-lambda*T)*pow(lambda*T,j)/x);
    somma=somma+pro;
```

```
}
kMax=j +2;
printf("%d\n",kMax);
int Tenta[kMax+1];
int k;
double ciccio= N;
int primo=ciccio /(10*(kMax+1));
double p[kMax+1];
double time;
double temp[primo];
double meancond[kMax+1];
double volcond[kMax+1];
double prezzi[kMax+1];
for(k=0;k<=kMax;k++){
  prezzi[k]=0.0;
}
double prezzoPar;
double prezzo=0.0;
double sigma1;
int tentatot=0;
PnlFunc func;
func.function = integranda;
double tmp[] = {sigma,T,D,lambda,iniziale,strike,r};
func.params = (void*) tmp;
for(k=0;k<=kMax;k++){
  x=pnl_sf_fact(k);
  p[k]=exp(-lambda*T)*(pow(lambda*T,k))/x;
  meancond[k]=0.0;
  for(j=0;j<primo;j++){
    time=changedTimeGenerator(lambda,sigma,D,T,k);
    sigma1=sqrt(time/T);
    temp[j]= pnl_bs_call(iniziale,strike,T,r,0,sigma1);
    meancond[k]=meancond[k]+temp[j];
  }
  meancond[k]=meancond[k]/primo;
  for(j=0;j<primo;j++){
    volcond[k]=volcond[k]+pow(temp[j]-meancond[k],2);
  }
  volcond[k]=volcond[k]/primo;
  tentatot=tentatot+primo;
}
double fatcom=0.0;
for(k=0;k<=kMax;k++){
  fatcom=fatcom+(volcond[k]*p[k]);
}
for(k=0;k<=kMax;k++){
  Tenta[k]=MAX(N*0.9*volcond[k]*p[k]/fatcom,1);
  tentatot=tentatot+Tenta[k];
}
prezzi[0]=pnl_integration(&func, 0.0, 10.0/lambda, Tenta[0],"trap")
```

```
                +exp(-10)*MAX(iniziale-(exp(-r*T)*strike),0);
  for(k=1;k<=kMax;k++){
    if(Tenta[k]!=0){
      for(j=0;j<Tenta[k];j++){
        time=changedTimeGenerator(lambda, sigma, D,T,k);
        prezzoPar=pnl_bs_call(iniziale,strike,T,r,0,sqrt(time/T));
        prezzi[k]=prezzi[k]+prezzoPar;
      }
    }
  }
  prezzo=meancond[0]*primo+prezzi[0]*Tenta[0];
  for(k=1;k<=kMax;k++){
    prezzo=meancond[k]*primo+prezzi[k]+prezzo;
  }
  prezzo=prezzo/(tentatot);
  return prezzo;
}
```

•The function which gives the price of a call and the delta using the second method (4 stratifications, the last is a generic "more than 2 jumps")

```
void pnl_cf_ACDP_call(double lambda, double v, double D ,
double T, double strike, double r, int N,
double s0, double *ptprice, double *ptdelta){

if(D==0.5){
pnl_cf_call_bs(s0,strike,T,r,0,sigma,ptprice,ptdelta);
}
else{
int kMax=3;
PnlVect* Taun;
int j;
int x;
int k;
double tmp[7];
double ciccio;
int primo;
PnlVect* Tenta;
PnlVect* p;
double time;
PnlVect* temp;
PnlVect* temp2;
double temp3;
double temp4;
PnlVect* meancond;
PnlVect* meancond2;
PnlVect* volcond;
PnlVect* volcond2;
PnlVect* prezzi;
PnlVect* delte;
double prezzoPar;
```

```
double deltaPar;
double prezzo;
double delta;
double sigma1;
double fatcom;
int tentatot;
PnlFunc func;
PnlFunc func1;
func.function = integranda;
func1.function = integranda1;
double sigma;

sigma=v*sqrt(pow(lambda,2*D-1)/pnl_tgamma(2*D+1));
   tmp[0] = sigma;
   tmp[1] = T;
   tmp[2] = D;
   tmp[3] =lambda;
   tmp[4] = s0;
   tmp[5] = strike;
   tmp[6] = r;
   func.params = (void*) tmp;
   func1.params = (void*) tmp;

ciccio=N;
primo=ciccio /(40);
fatcom=0.0;
prezzo=0.0;
prezzoPar=0.0;
deltaPar=0.0;
delta=0.0;
tentatot=0;

Tenta= pnl_vect_create(4);
p= pnl_vect_create(4);
temp= pnl_vect_create(primo);
temp2= pnl_vect_create(primo);
meancond= pnl_vect_create(4);
meancond2= pnl_vect_create(4);
volcond= pnl_vect_create(4);
volcond2= pnl_vect_create(4);
prezzi= pnl_vect_create(4);
delte= pnl_vect_create(4);


//beginning of the stratification

for(k=0;k<kMax;k++){
x=pnl_sf_fact(k);
pnl_vect_set(p,k,exp(-lambda*T)*(pow(lambda*T,k))/x);
pnl_vect_set(meancond,k,0.0);
```

```
for(j=0;j<primo;j++){
time=changedTimeGenerator(lambda,sigma,D,T,k);
sigma1=sqrt(time/T);
pnl_cf_call_bs(s0,strike,T,r,0,sqrt(time/T),
&prezzoPar,&deltaPar);
pnl_vect_set(temp,j,prezzoPar);
pnl_vect_set(temp2,j,deltaPar);
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)+
 prezzoPar);
pnl_vect_set(meancond2,k,pnl_vect_get(meancond2,k)
 + deltaPar);
}
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)/primo);
pnl_vect_set(meancond2,k,pnl_vect_get(meancond2,k)/primo);
for(j=0;j<primo;j++){
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)
+pow(pnl_vect_get(temp,j)-pnl_vect_get(meancond,k),2));
pnl_vect_set(volcond2,k,pnl_vect_get(volcond2,k)
+pow(pnl_vect_get(temp2,j)-pnl_vect_get(meancond2,k),2));
}
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)/primo);
pnl_vect_set(volcond2,k,pnl_vect_get(volcond2,k)/primo);
tentatot=tentatot+primo;
}
k=3;
x=pnl_sf_fact(k);
pnl_vect_set(p,k,1-pnl_vect_get(p,0)-pnl_vect_get(p,1)
-pnl_vect_get(p,2));
pnl_vect_set(meancond,k,0.0);
for(j=0;j<primo;j++){
Taun=pnl_vect_new();
jumpGenerator3(lambda, sigma, D,T, Taun);
time=changedTimeGeneratorVec(sigma,D,Taun->size-2, Taun);
sigma1=sqrt(time/T);
pnl_cf_call_bs(s0,strike,T,r,0,sigma1,&temp3,&temp4);
pnl_vect_set(temp,j, temp3);
pnl_vect_set(temp2,j, temp4);
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)
+pnl_vect_get(temp,j));
pnl_vect_set(meancond2,k,pnl_vect_get(meancond2,k)
+pnl_vect_get(temp2,j));
pnl_vect_free(&Taun);

}
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)/primo);
pnl_vect_set(meancond2,k,pnl_vect_get(meancond2,k)/primo);
for(j=0;j<primo;j++){
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)
  +pow(pnl_vect_get(temp,j)
  -pnl_vect_get(meancond,k),2));
```

```
pnl_vect_set(volcond2,k,pnl_vect_get(volcond2,k)
+pow(pnl_vect_get(temp2,j)
-pnl_vect_get(meancond2,k),2));
}
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)/primo);
pnl_vect_set(volcond2,k,pnl_vect_get(volcond2,k)/primo);
tentatot=tentatot+primo;

for(k=0;k<=kMax;k++){
fatcom=fatcom+((pnl_vect_get(volcond,k)+
pnl_vect_get(volcond2,k))*pnl_vect_get(p,k));
}
for(k=0;k<=kMax;k++){
pnl_vect_set(Tenta,k(int)MAX(N*0.9*(pnl_vect_get(volcond,k)
+pnl_vect_get(volcond2,k))*pnl_vect_get(p,k)/fatcom,1));
tentatot=tentatot+pnl_vect_get(Tenta,k);
}
//end of stratification

//first layer
k=0;
pnl_vect_set(prezzi,0,pnl_integration(&func, 0.0, 10.0/lambda,
 (int)pnl_vect_get(Tenta,0), "trap")
 +exp(-10)*MAX(s0- (exp(-r*T)*strike),0));
pnl_vect_set(delte,0,pnl_integration(&func1, 0.0, 10.0/lambda,
 (int)pnl_vect_get(Tenta,0), "trap"));
//second and third layer
for(k=1; k<kMax ;k++){
if(pnl_vect_get(Tenta,k)!=0){
for(j=0;j<pnl_vect_get(Tenta,k);j++){
time=changedTimeGenerator(lambda, sigma, D,T,k);
pnl_cf_call_bs(s0,strike,T,r,0,sqrt(time/T),
&prezzoPar,&deltaPar);
pnl_vect_set(prezzi,k,pnl_vect_get(prezzi,k)+prezzoPar);
pnl_vect_set(delte,k,pnl_vect_get(delte,k)+deltaPar);
}
}
}
//fourth layer
k=3;
if(pnl_vect_get(Tenta,k)!=0){
for(j=0;j<pnl_vect_get(Tenta,k);j++){
Taun=pnl_vect_new ();
jumpGenerator3(lambda, sigma, D,T, Taun);
time=changedTimeGeneratorVec(sigma,D,(Taun->size-2), Taun);
pnl_cf_call_bs(s0,strike,T,r,0,sqrt(time/T),
                      &prezzoPar,&deltaPar);
pnl_vect_set(prezzi,k,pnl_vect_get(prezzi,k)
                                 +prezzoPar);
pnl_vect_set(delte,k,pnl_vect_get(delte,k)+
```

```
                                    deltaPar);
pnl_vect_free(&Taun);
}
}
prezzo=pnl_vect_get(meancond,0)*primo+ pnl_vect_get(prezzi,0)*
                                        pnl_vect_get(Tenta,0);
delta=pnl_vect_get(meancond2,0)+ pnl_vect_get(delte,0)*
                                pnl_vect_get(Tenta,0);
for(k=1;k<=kMax;k++){
    prezzo=pnl_vect_get(meancond,k)*primo +
                              pnl_vect_get(prezzi,k)+prezzo;
    delta= pnl_vect_get(meancond2,k) +
                               pnl_vect_get(delte,k)+ delta;
}
prezzo=prezzo/(tentatot);
delta=delta/(tentatot);
*ptprice=prezzo;
*ptdelta=delta;
pnl_vect_free(&Tenta);
pnl_vect_free(&p);
pnl_vect_free(&temp);
pnl_vect_free(&temp2);
pnl_vect_free(&meancond);
pnl_vect_free(&meancond2);
pnl_vect_free(&volcond);
pnl_vect_free(&volcond2);
pnl_vect_free(&prezzi);
pnl_vect_free(&delte);

}
}
```

•The function which gives the discretization of the trajectory.

```
void generaIstantiTraiettoria(double lambda, double sigma, double D,
                int Minimal_lenght, double T,double r, PnlVect* Tau,
                 PnlVect* value, PnlVect* trj_temp){
    int crazy;
   double temp=0.0;
   double delta=T/Minimal_lenght;
   double delta1=r*delta;
   int size=1;
   double temp1;
   double temp2;
   double temp_value;
   int j=1;
   int salti, minori;
   double expinv;
   expinv=1/(2*D);
   salti=0;
```

```
   minori=0;
   crazy=pnl_vect_resize(trj_temp,1);
   pnl_vect_set(trj_temp,0,temp);
   crazy=pnl_vect_resize(value,1);
   pnl_vect_set(value,0,0.0);
   while(temp<T){
     //at this moment temp is the last inserted istant
                     in the discretization
     temp2=temp;
     if(temp+delta>pnl_vect_get(Tau,j)){
     //in this case we choose as discr. point the jump
         temp=pnl_vect_get(Tau,j);
       temp_value=(pow((temp-pnl_vect_get(Tau,j-1)),2*D)-
                  pow((temp2-pnl_vect_get(Tau,j-1)),2*D))*sigma*sigma;
       j=j+1;
       salti=salti+1;
     }
     else{
       temp1=pnl_vect_get(Tau,j-1)+pow(((2*delta1/(sigma*sigma))+
       pow(temp-pnl_vect_get(Tau,j-1),2*D)),expinv);
       //temp1 is the istant such that I(t)-I(temp)=r*delta;
      //I choose the minimum between temp1 and temp+delta
       if(temp1<temp+delta){
       minori=minori+1;
temp=temp1;
       }
       else{
temp=temp+delta;
      }
  temp_value=sigma*sigma*(pow(temp-pnl_vect_get(Tau,j-1),2*D)
  -pow(temp2-pnl_vect_get(Tau,j-1),2*D));
    }
    crazy=pnl_vect_resize(trj_temp,size+1);
    crazy=pnl_vect_resize(value,size+1);
     pnl_vect_set(trj_temp,size,temp);
     pnl_vect_set(value,size,temp_value);
size=size+1;
}

}
```

•The function that gives the values of the underlying in the discretization time

```
void genTraiettoria(double lambda, double sigma, double D,
               int Minimal_lenght,double S0, double r,double T,
               PnlVect* traiettoria, PnlVect* tempi, PnlVect* Tau){
  int lungh;
  int crazy;
  double finalTime;
```

```
  PnlVect* valori_temp;
  int i;
  double x;
  PnlVect* esponente;
  finalTime=changedTimeGeneratorVec(sigma,D, Tau->size-2, Tau);
  valori_temp=pnl_vect_new();
  generaIstantiTraiettoria(lambda,sigma, D,
                  Minimal_lenght,T,r,Tau, valori_temp,tempi);
  int i;
  double x;
  PnlVect* esponente;
  lungh=tempi->size;
  esponente= pnl_vect_create(lungh);
  pnl_vect_set(esponente, 0, 0);
  for(i=1; i<lungh;i++){
    x= sqrt(pnl_vect_get(valori_temp,i))*gauss()-
                (pnl_vect_get(valori_temp,i)/2)+r*(pnl_vect_get(tempi,i)
                -pnl_vect_get(tempi,i-1))+pnl_vect_get(esponente,i-1);
    pnl_vect_set(esponente, i, x);
  }
  crazy=pnl_vect_resize(traiettoria,lungh);
  for(i=0;i<lungh;i++){
    pnl_vect_set(traiettoria,i,S0*exp(pnl_vect_get(esponente,i)));
  }
  pnl_vect_free(&esponente);
  pnl_vect_free(&valori_temp);

}
```

•The function which computes the price of a call using the path.

```
double pnl_walk_ACDP_call(double lambda, double v, double D , double T,
 double strike, double r, int N, double s0, int Minimal_lenght){

if(D==0.5){
return pnl_bs_call(s0,strike,T,r,0,v);
}
else{
int kMax=3;
PnlVect* Taun;
PnlVect* Traiettorian;
PnlVect* prezzin;
int j;
int x;
int k;
double tmp[7];
double ciccio;
int primo;
PnlVect* Tenta;
```

```
PnlVect* p;
double time;
PnlVect* temp;
PnlVect* meancond;
PnlVect* volcond;
PnlVect* prezzi;
double prezzoPar;
double prezzo;
double sigma1;
double fatcom;
int tentatot;
PnlFunc func;
func.function = integranda;
double sigma;

sigma=v*sqrt(pow(lambda,2*D-1)/pnl_tgamma(2*D+1));
   tmp[0] = sigma;
   tmp[1] = T;
   tmp[2] = D;
   tmp[3] =lambda;
   tmp[4] = s0;
   tmp[5] = strike;
   tmp[6] = r;
   func.params = (void*) tmp;

ciccio=N;
primo=ciccio /(40);
fatcom=0.0;
prezzo=0.0;
prezzoPar=0.0;
tentatot=0;

Tenta= pnl_vect_create(4);
p= pnl_vect_create(4);
temp= pnl_vect_create(primo);
meancond= pnl_vect_create(4);
volcond= pnl_vect_create(4);
prezzi= pnl_vect_create(4);


//beginning of the stratification

for(k=0;k<kMax;k++){
x=pnl_sf_fact(k);
pnl_vect_set(p,k,exp(-lambda*T)*(pow(lambda*T,k))/x);
pnl_vect_set(meancond,k,0.0);
for(j=0;j<primo;j++){
time=changedTimeGenerator(lambda,sigma,D,T,k);
sigma1=sqrt(time/T);
pnl_vect_set(temp,j,pnl_bs_call(s0,strike,T,r,0,sigma1));
```

```
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)+ prezzoPar);
}
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)/primo);
for(j=0;j<primo;j++){
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)+
pow(pnl_vect_get(temp,j)-pnl_vect_get(meancond,k),2));
}
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)/primo);
tentatot=tentatot+primo;
}
k=3;
x=pnl_sf_fact(k);
pnl_vect_set(p,k,1-pnl_vect_get(p,0)-pnl_vect_get(p,1)-pnl_vect_get(p,2));
pnl_vect_set(meancond,k,0.0);
for(j=0;j<primo;j++){
Taun=pnl_vect_new();
jumpGenerator3(lambda, sigma, D,T, Taun);
time=changedTimeGeneratorVec(sigma,D,Taun->size-2, Taun);
sigma1=sqrt(time/T);
pnl_vect_set(temp,j, pnl_bs_call(s0,strike,T,r,0,sigma1));
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)+pnl_vect_get(temp,j));
pnl_vect_free(&Taun);
}
pnl_vect_set(meancond,k,pnl_vect_get(meancond,k)/primo);
for(j=0;j<primo;j++){
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)+
pow(pnl_vect_get(temp,j)-pnl_vect_get(meancond,k),2));
}
pnl_vect_set(volcond,k,pnl_vect_get(volcond,k)/primo);

tentatot=tentatot+primo;


for(k=0;k<=kMax;k++){
fatcom=fatcom+((pnl_vect_get(volcond,k))*pnl_vect_get(p,k));
}
for(k=0;k<=kMax;k++){
pnl_vect_set(Tenta,k,(int)MAX(N*0.9*(pnl_vect_get(volcond,k))*
pnl_vect_get(p,k)/fatcom,1));
tentatot=tentatot+pnl_vect_get(Tenta,k);
}
//end of stratification

//first second and third layer

for(k=0; k<kMax ;k++){
if(pnl_vect_get(Tenta,k)!=0){
for(j=0;j<pnl_vect_get(Tenta,k);j++){
Taun= pnl_vect_create(k+2);
Traiettorian=pnl_vect_new();
```

```
prezzin=pnl_vect_new();
jumpGenerator(lambda, sigma, D,T,k, Taun);
genTraiettoria(lambda,sigma, D, Minimal_lenght, s0,r,T,
prezzin,Traiettorian , Taun);
prezzoPar= MAX(pnl_vect_get(prezzin,prezzin->size-1)-strike,0);
pnl_vect_set(prezzi,k,pnl_vect_get(prezzi,k)+prezzoPar);
pnl_vect_free(&Taun);
pnl_vect_free(&Traiettorian);
pnl_vect_free(&prezzin);
}
}
}
//fourth layer
k=3;
if(pnl_vect_get(Tenta,k)!=0){
for(j=0;j<pnl_vect_get(Tenta,k);j++){
Taun=pnl_vect_new();
Traiettorian=pnl_vect_new();
prezzin=pnl_vect_new();
jumpGenerator3(lambda, sigma, D,T, Taun);
genTraiettoria(lambda,sigma, D, Minimal_lenght, s0,r,T,
prezzin,Traiettorian , Taun);
prezzoPar= MAX(pnl_vect_get(prezzin,prezzin->size-1) - strike,0);
pnl_vect_set(prezzi,k,pnl_vect_get(prezzi,k)+prezzoPar);
pnl_vect_free(&Taun);
pnl_vect_free(&Traiettorian);
pnl_vect_free(&prezzin);
}
}
prezzo=pnl_vect_get(meancond,0)*primo+ pnl_vect_get(prezzi,0);
for(k=1;k<=kMax;k++){
prezzo=pnl_vect_get(meancond,k)*primo + pnl_vect_get(prezzi,k)+prezzo;
}
prezzo=prezzo/(tentatot);
return prezzo;
pnl_vect_free(&Tenta);
pnl_vect_free(&p);
pnl_vect_free(&temp);
pnl_vect_free(&meancond);
pnl_vect_free(&volcond);
pnl_vect_free(&prezzi);

}
}
```

References

[1] A. Andreoli *Scaling and multiscaling in financial series: a simple model*, Ph. D. thesis 2011.

[2] A. Andreoli, F. Caravenna, P. Dai Pra and G.Posta, *Scaling and multiscaling in financial series: a simple model*, Adv. in Appl. Probab. 44 (2012), 1018-1051. 1