Discretization of Random Variables

Niels O. Nygaard
The University of Chicago

- Probability Distributions form a Monad (Lawvere, Erwig-Kollmansberger,Ramsey-Pfeffer)
- Monad Transformer: List Monad and Writer Monad (Kidd)
- Writer Monad:
  ‘a -> ‘a*Prob
- Type Constructor (Functor)

```
type Pr<'a>=Pr of 'a*Prob
```

Monoid

```
type Prob=
    | Prob of float
    member this.p=match this with
                  |Prob x when x>=0.0 || x<=1.0 ->x
                  |Prob x when x<0.0 -> 0.0
                  |Prob x->1.0
    static member (*) (P1:Prob,P2:Prob)
                          =Prob(P1.p*P2.p)
```

Monoid Multiplication

- Distributive Law: `(Pr<'a list> -> Pr<'a> list)`

```
let distribList (ls:Pr<'a list>)=match ls with
                                 |Pr(s,P)->s|>List.map(fun x->Pr(x,P))
```

- Probability Distribution

```
type ProbDist<'a>=
    |PD of Pr<'a> list
```

- Monad Structure

```
let map f (D:ProbDist<'a>)=D.List|>List.map(Pr.map f)|>PD
let unit x = PD [Pr.unit x]
let join (d:ProbDist<ProbDist<'a>>) = d.List|>PSeq.map(fun (Pr(pd,P))->Pr.distribList (Pr(pd.List,P)))
                                            |>PSeq.concat
                                            |>PSeq.map(fun x->Pr.join x)
                                            |>(fun u->PD(u|>PSeq.toList))
```

- [Pr([Pr(a(0,0),P(0,0));..Pr(a(0,n0),P(0,n0))],Q0);…
  Pr([Pr(a(m,0),P(m,0));..;Pr(a(m,nm),P(m,nm))],Qm]
  ->
  [Pr(a(0,0),P(0,0)*Q0);Pr(a(0,1),P(0,1)*Q0);…
  ..;Pr(a(m,nm),P(m,mn)*Qm)]

Example

```
let uniform (ls:'a list)=ls|>List.map(fun s->Pr(s,Prob(1.0/(float ls.Length))))|>PD
```

```
val die : ProbDist<int> =
  PD
    [Pr (1,Prob 0.1666666667); Pr (2,Prob 0.1666666667);
     Pr (3,Prob 0.1666666667); Pr (4,Prob 0.1666666667);
     Pr (5,Prob 0.1666666667); Pr (6,Prob 0.1666666667)]
```

```
let pd=new ProbDistBuilder()

let twoDice=pd{ let! a = die
                let! b = die
                return (a,b) }
```

```
val twoDice : ProbDist<int * int> =
  PD
    [Pr ((1, 4),Prob 0.02777777778); Pr ((1, 2),Prob 0.02777777778);
     Pr ((1, 1),Prob 0.02777777778); Pr ((1, 6),Prob 0.02777777778);
     Pr ((1, 3),Prob 0.02777777778); Pr ((1, 5),Prob 0.02777777778);
     Pr ((2, 1),Prob 0.02777777778); Pr ((2, 2),Prob 0.02777777778);
     Pr ((2, 3),Prob 0.02777777778); Pr ((2, 4),Prob 0.02777777778);
     Pr ((2, 5),Prob 0.02777777778); Pr ((2, 6),Prob 0.02777777778);
     Pr ((3, 1),Prob 0.02777777778); Pr ((3, 2),Prob 0.02777777778);
     Pr ((3, 3),Prob 0.02777777778); Pr ((3, 4),Prob 0.02777777778);
     Pr ((3, 5),Prob 0.02777777778); Pr ((3, 6),Prob 0.02777777778);
     Pr ((4, 1),Prob 0.02777777778); Pr ((4, 2),Prob 0.02777777778);
     Pr ((4, 3),Prob 0.02777777778); Pr ((4, 6),Prob 0.02777777778);
     Pr ((4, 4),Prob 0.02777777778); Pr ((4, 5),Prob 0.02777777778);
     Pr ((5, 1),Prob 0.02777777778); Pr ((5, 2),Prob 0.02777777778);
     Pr ((5, 3),Prob 0.02777777778); Pr ((5, 4),Prob 0.02777777778);
     Pr ((5, 5),Prob 0.02777777778); Pr ((5, 6),Prob 0.02777777778);
     Pr ((6, 2),Prob 0.02777777778); Pr ((6, 1),Prob 0.02777777778);
     Pr ((6, 3),Prob 0.02777777778); Pr ((6, 4),Prob 0.02777777778);
     Pr ((6, 5),Prob 0.02777777778); Pr ((6, 6),Prob 0.02777777778)]

>
```

Random Variables

- In Probability Theory a Random Variable is a function from a Probability Distribution to the Reals
- We can model this again as a Monad Transformer: Writer-List where this time the Writer is defined using the Monoid
  W=Prob*float
  with multiplication
  (P,x)*(Q,y)=(P*Q,x+y)

```
type Rv<'a> =
    |Rv of 'a*Prob*float
```

```
let distribList (ls:Rv<'a list>)= ls.Point
                                  |>PSeq.map(fun a->Rv(a,ls.P,ls.Value))
                                  |>PSeq.toList
```

```
type RandomVar<'a when 'a:equality>=
    |RV of Rv<'a> list
```

- If we use the monadic composition the size of the list grows exponentially which is not always what we want

```
let lift f (X:RandomVar<'a>)= X.Values
                              |>PSeq.groupBy(fun (Pr(a,P),x)-> f a)
                              |>PSeq.map(fun (b,s)
                                          ->(b,Prob(s|>PSeq.sumBy(fun (Pr(a,P),x)->P.p))),s|>PSeq.sumBy(fun (Pr(a,P),x)->x*P.p))
                              |>PSeq.map(fun ((b,P),x)->Rv(b,P,x/P.p))
                              |>PSeq.toList|>RV
```

```
> RV.lift;;
> val it :
  (('a -> 'b) -> RandomVar<'a> -> RandomVar<'b>)
    when 'a : equality and 'b : equality = <fun:clo@3-1>
```

- In mathematical terms: a function from a Probability Distribution to a set

$$f: \Omega \to S$$

defines a partition $\mathcal{F}$ on Ω by the subsets indexed by S: $\Omega(s)=f^{-1}(s)$

- If $X: \Omega \to \mathbb{R}$ is a Random Variable we can define the Conditional Expectation

$$E(X|\ \mathcal{F})(s)=\sum_{\omega\in f^{-1}(s)} X(\omega)Prob(\omega) / Prob(f^{-1}(s))$$

- We define a function

```
let recombine X=lift (fun a->a ) X
```

- The Monad structure on RandomVar is then

```
let unit x= RV [Rv.unit x]
let join (X:RandomVar<RandomVar<'a>>)= X.List
                                      |>PSeq.map(fun (Rv(Z,P,x))->Rv.distribList(Rv(Z.List,P,x)))
                                      |>PSeq.concat
                                      |>PSeq.map Rv.join
                                      |>PSeq.toList|>RV
                                      |>recombine
```

This is just the Monad Transformer structure except we add the recombine method

```
let crr i =RV [Rv(i+1,Prob(0.5),log 1.01);Rv(i-1,Prob(0.5),-log (1.01))]
```

- This is transition function in a CRR Tree i.e. of type int->RV<int>
- If we build a tree without the recombine function we get something like

```
let tree= rv{ let! a = crr 0
              let! b = crr a
              let! c = crr b
              return c}
```

```
val tree : RandomVar<int> =
  RV
    [Rv (1,Prob 0.125,0.009950330853); Rv (-1,Prob 0.125,-0.009950330853);
     Rv (-3,Prob 0.125,-0.02985099256); Rv (-1,Prob 0.125,-0.009950330853);
     Rv (-1,Prob 0.125,-0.009950330853); Rv (1,Prob 0.125,0.009950330853);
     Rv (1,Prob 0.125,0.009950330853); Rv (3,Prob 0.125,0.02985099256)]
```

which is a tree with 8 nodes and not recombining

- With the *recombine* we get the recombining tree

```
val tree : RandomVar<int> =
  RV
    [Rv (-3,Prob 0.125,-0.02985099256); Rv (1,Prob 0.375,0.009950330853);
     Rv (3,Prob 0.125,0.02985099256); Rv (-1,Prob 0.375,-0.009950330853)]
```

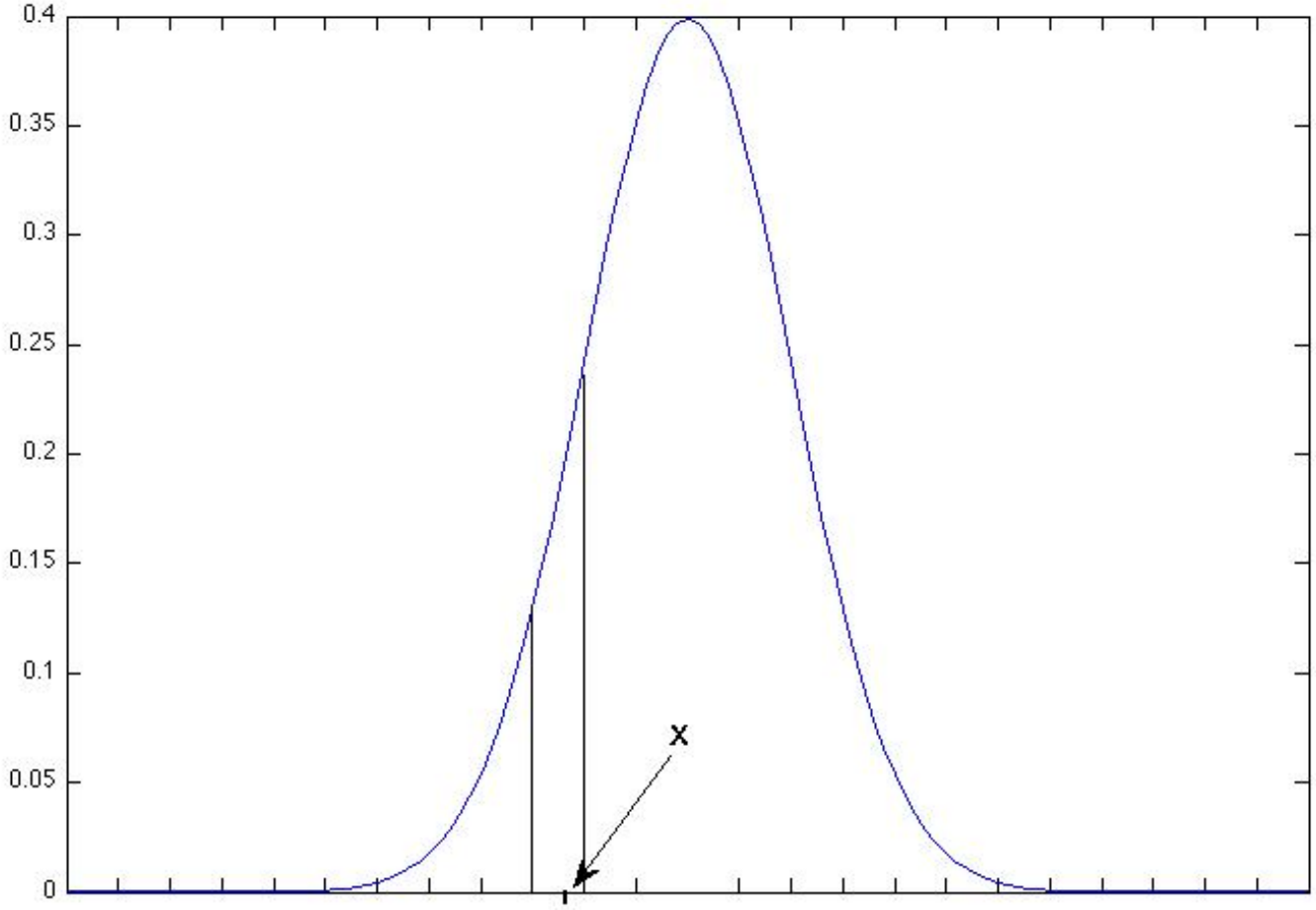
  with the correct probabilities
- This is actually a discretization of a normal distribution

- How can we in general create a discrete approximation to a Random Variable?
- If $X: \Omega \to \mathbb{R}$ is a RV, its CDF is the function $\Psi(t)=\text{Prob}\{X<t\}$. The PDF, ϕ, is the derivative of the CDF (if it exists)
- It has the property that

$$\text{Prob}\{a \leq X < b\} = \int_a^b \varphi(t)\,dt$$

for all $a<b$

- A possible way to discretize X would be to take a large enough interval I such that the probability of X taking a value outside I is very small i.e.
  $\int I\uparrow\varphi(t)dt \approx 1$
- Then divide I into sub-intervals $\{I_k\}$, $k=0,1,..n$
- For each k let $P_k = \text{Prob}\{X \in I_k\}$ and choose a point x_k in I_k
- We can then discretize to a RandomVar<int> $[Rv(0,P_0,x_0);..;Rv(k,P_k,x_k),..;Rv(n,P_n,x_n)]$


0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
x

- The question is how to pick the points x_k
- For instance we could pick the midpoint or one of the endpoints
- Is there an optimal (in some sense) choice

Gaussian Quadrature Rules

- The Gaussian Quadrature Rule says that for a given density function φ there exist points $\xi_1, \xi_2, \ldots, \xi_n$ and weights $w_1, w_2, \ldots, w_n$ such that for any polynomial f of degree $\leq 2n-1$,
$$\int_a^b f(t)\varphi(t)dt = w_1 f(\xi_1) + w_2 f(\xi_2) + \ldots + w_n f(\xi_n)$$

- In particular for $f=1$ we get
$$\int_a^b \varphi(t)dt = \sum_1^n w_i$$

- The ξ_i s are roots of a certain degree n polynomial which occurs in an orthogonal sequence of polynomials depending on the particular density function e.g. Hermite polynomials for the normal density
- We have found that using a Gaussian Quadrature rule with two points in each interval gives a good balance between numerical precision and performance
- Using Gaussian rules with many points gets pretty complicated, we have tried it with up to 80 points but the numerical results are not as good

- In each sub-interval we approximate the integral using Simpson's rule

Simpson's rule on each sub-interval

```
let fromPDF pdf a b N= let lambda=(b-a)/float(2*N)
                       let pdfSamples=List.init (2*N+1) (fun n->pdf(a+float n*lambda))
                       let simpsonFunctionals= [for k in 0..2..(2*N-2) do
                                                let F g =
                                                          (lambda/3.0)*(((g(a+float (k)*lambda))*pdfSamples.[k])
                                                                    +(4.0*(g(a+float(k+1)*lambda))*(pdfSamples.[k+1]))
                                                                      + (g(a+float(k+2)*lambda))*(pdfSamples.[k+2]))
                                                yield F ]
                       let moments= [ for k in 0..N-1 do
                                      yield Array.init 4 (fun i->simpsonFunctionals.[k] (fun x->(x**(float (i))))) ]
                       let gaussPoints = [ for k in 0..N-1 do
                                           let mu=moments.[k]
                                           let x=mu.[0]
                                           let y=mu.[1]
                                           let u=mu.[2]
                                           let v=mu.[3]
                                           let y1=((y*u-x*v)-sqrt((y*u-x*v)**2.0 - 4.0*(y**2.0-x*u)*(u**2.0-y*v)))/(2.0*(y**2.0-x*u))

                                           let y2=((y*u-x*v)+sqrt((y*u-x*v)**2.0 - 4.0*(y**2.0-x*u)*(u**2.0-y*v)))/(2.0*(y**2.0-x*u))

                                           yield [y1;y2]|>List.sort ]

                       let weights= [ for k in 0..N-1 do
                                      let Legendre= [fun x->(x-gaussPoints.[k].[1])/(gaussPoints.[k].[0]-gaussPoints.[k].[1]);
                                                     fun x->(x-gaussPoints.[k].[0])/(gaussPoints.[k].[1]-gaussPoints.[k].[0])]
                                      yield! [simpsonFunctionals.[k](Legendre.[0]);simpsonFunctionals.[k](Legendre.[1])] ]

                       let distr = (List.map2 (fun n p->Pr(n,Prob p)) [0..2*N-1] weights)|>PS
                       let ls=(List.zip distr.List (gaussPoints|>List.concat))
                               |> List.sortBy(fun (Pr(n,P),x)->x)
                             |> List.map(fun (Pr(n,P),x)->Rv(n,P,x))
                       RV(ls)
```

The quadrature points are roots of a quadratic polynomial

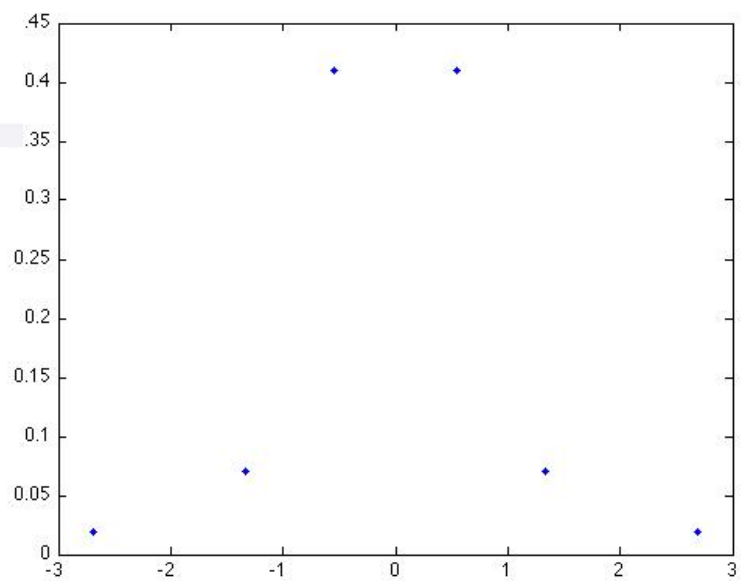
```
let normalPDF t=(1.0/(sqrt(2.0*System.Math.PI)))*exp(-0.5*t**2.0)

let X=RV.fromPDF normalPDF -4.0 4.0 1
```

```
val X : RandomVar<int> =
  RV [Rv (0,Prob 0.5,-0.05180010577); Rv (1,Prob 0.5,0.05180010577)]
```

```
let X=RV.fromPDF normalPDF -4.0 4.0 3
```

```
val X : RandomVar<int> =
  RV
    [Rv (0,Prob 0.01923219993,-2.682155294);
     Rv (1,Prob 0.07027096844,-1.335457634);
     Rv (2,Prob 0.4104968316,-0.5505670906);
     Rv (3,Prob 0.4104968316,0.5505670906);
     Rv (4,Prob 0.07027096844,1.335457634);
     Rv (5,Prob 0.01923219993,2.682155294)]
```



- Many derivatives pricing problems come down to computing the expectation of a pay-out function i.e. an integral of the form
  $E(f(X))=\int_{-\infty}^{\infty} f(t)\varphi(t)dt$
  where f is the pay-out function and φ is the density function of the Random Variable X
- Once we have a discretization this simply becomes
  $E(f(X))=\sum f(x_i)P_i$

Example

- Under the Black-Scholes model the risk neutral price of a stock at time T is described by the Random Variable
  $S_T = S_0 \exp((r - \sigma^2/2)T + \sigma\sqrt{T}X)$
  where X is a standard normal Random Variable
- The Black-Scholes price of a call option with strike K and expiration at T is

$$C = \exp(-rT)E(\max(S_T - K, 0))$$

- We can now compute this from our discretized normal distribution

```
let optionPrices S0 sigma r T= let strikes=[1.0..(S0/50.0)..2.0*S0]
                               let prices=strikes|>PSeq.map(fun K->(K,C S0 sigma r T K))|>PSeq.sort
                               Seq.iter (fun p -> printfn "Strike: %f, Price: %f" (fst p)(snd p)) prices
```

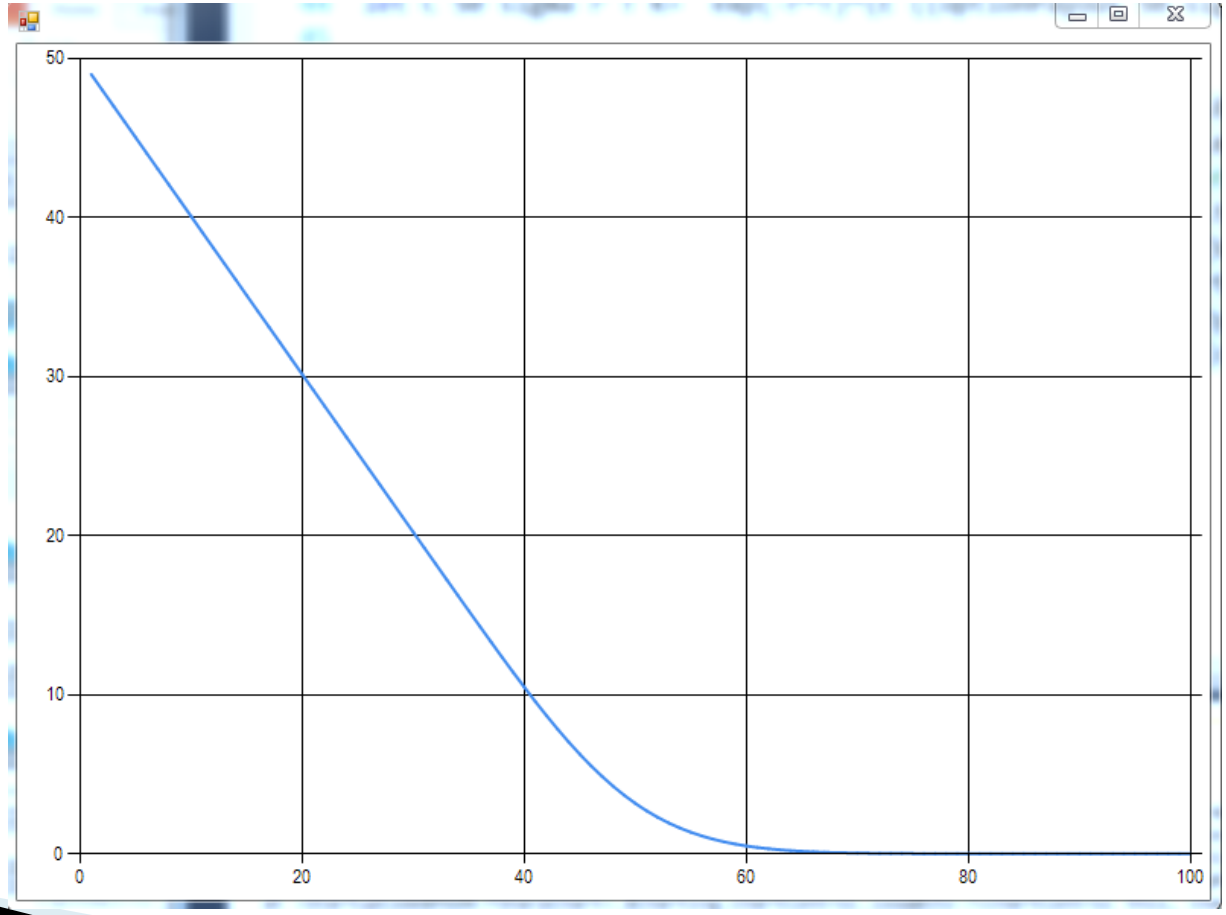
```
Strike: 33.000000, Price: 17.[illegible]0879
Strike: 34.000000, Price: 16.262464
Strike: 35.000000, Price: 15.277107
Strike: 36.000000, Price: 14.296555
Strike: 37.000000, Price: 13.323093
Strike: 38.000000, Price: 12.359681
Strike: 39.000000, Price: 11.409924
Strike: 40.000000, Price: 10.478211
Strike: 41.000000, Price: 9.569089
Strike: 42.000000, Price: 8.687667
Strike: 43.000000, Price: 7.839239
Strike: 44.000000, Price: 7.028495
Strike: 45.000000, Price: 6.260641
Strike: 46.000000, Price: 5.539130
Strike: 47.000000, Price: 4.867740
Strike: 48.000000, Price: 4.247852
Strike: 49.000000, Price: 3.681379
Strike: 50.000000, Price: 3.168780
Strike: 51.000000, Price: 2.708525
Strike: 52.000000, Price: 2.299035
Strike: 53.000000, Price: 1.938694
Strike: 54.000000, Price: 1.623960
Strike: 55.000000, Price: 1.351447
Strike: 56.000000, Price: 1.117510
Strike: 57.000000, Price: 0.918332
Strike: 58.000000, Price: 0.750065
Strike: 59.000000, Price: 0.608957
Strike: 60.000000, Price: 0.491770
```

Black-Scholes price is 3.16861

Time to compute 100 strikes

```
Real: 00:00:00.039, CPU: 00:00:00.140, GC gen0: 2, gen1: 0, gen2: 0
val it : unit = ()
```

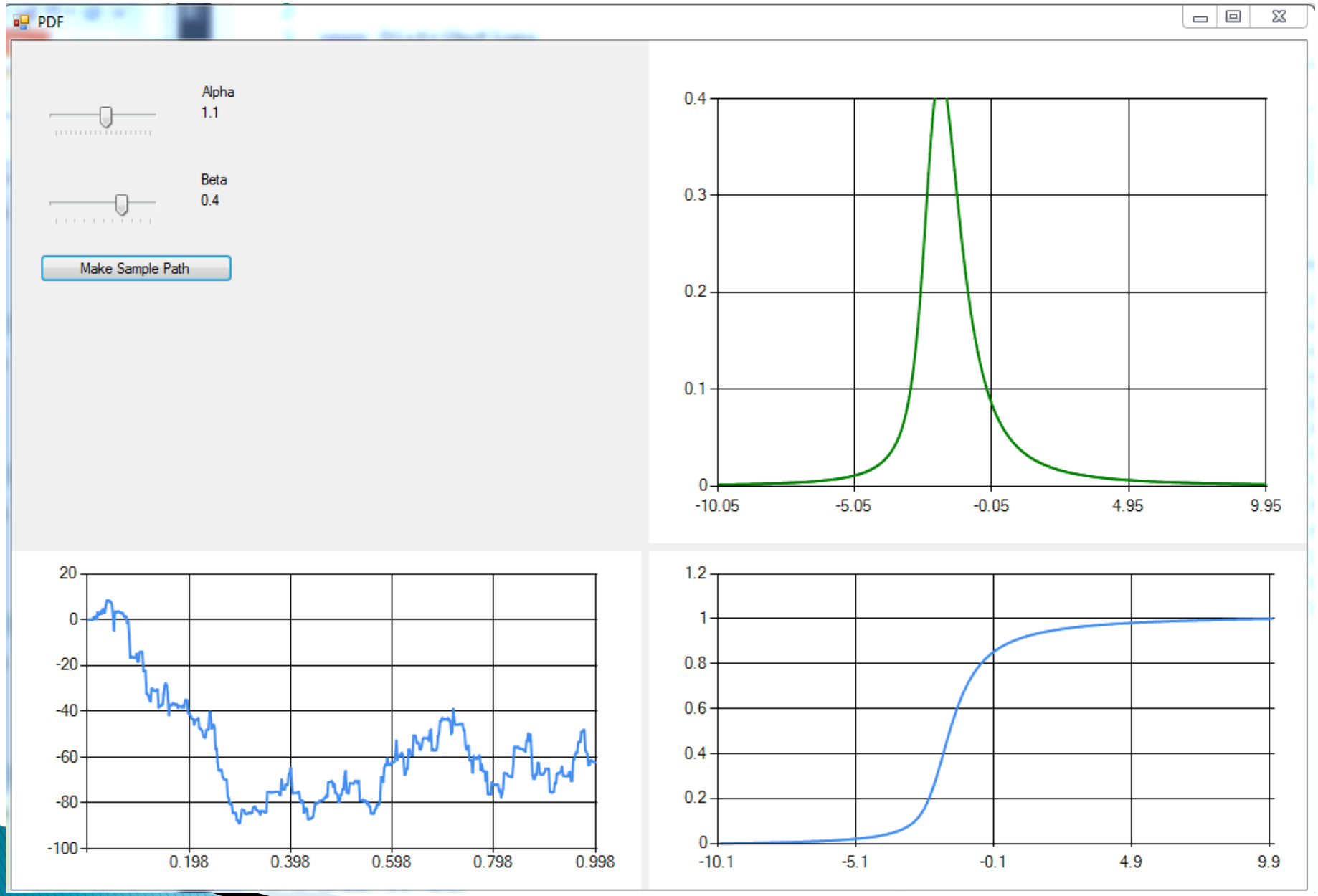
Price as a function of Strike



- We next look at using other distributions, here we look at stable distributions, the stock price follows a Levy process
- Stable distributions have the property that a linear combination of independent Random Variables with a particular stable distribution again has that distribution i.e. if $X_1 \; X_2 \ldots X_n$ are S distributed, where S is stable, then the Random Variable $a_1 X_1 + a_2 X_2 + .. = cX$ where X is S-distributed, e.g. Normal distributions are stable

- Stable distributions normally have infinite variance
- Stable distributions are parameterized by 4 parameters $(\alpha,\beta,\gamma,\delta)$, $0<\alpha\leq 2$, $-1\leq\beta\leq 1$
- When $\alpha=2$ the distribution is Normal
- Except for $\alpha=1,2$ *and* $\alpha=1/2$, $\beta=1$ there is no closed form for the density function, only the Characteristic Function i.e. the Fourier Transform of the density function is known

- To compute the density function we use an algorithm known as Fractional Fast Fourier Transform
- The usual FFT has the problem that the product of the spacings of the input and the output satisfies $\tau\omega=2\pi/N$ where N is the number of points
- Thus to get good precision for both input and output we need a lot of points, FFFT allows us to specify $\tau\omega=\lambda$ where we can choose λ independently of the number of points


PDF
Alpha
1.1
Beta
0.4
Make Sample Path
0.4
0.3
0.2
0.1
0
-10.05
-5.05
-0.05
4.95
9.95
20
0
-20
-40
-60
-80
-100
0.198
0.398
0.598
0.798
0.998
1.2
1
0.8
0.6
0.4
0.2
0
-10.1
-5.1
-0.1
4.9
9.9