Documentation:

Option calibration of exponential Lévy models in R

Jakob Söhl Mathias Trabs

September 3, 2012

The calibration of exponential Lévy models based on European option prices was discussed in [4]. There the spectral calibration method was applied which was developed in [1] for Lévy processes of finite jump activity and which was extended to self–decomposable Lévy processes in [5]. We give here a detailed description of our implementation of the calibration method for the finite activity case. For the specification of the model and the definition of the estimators we refer to [4].

1 Global parameters

Some global parameters to adjust the code and some auxiliary functions are necessary. These are namely `Zeta` needed for simulations, `CFHat` and `ZetaHat` to reproduce observations from the estimated model, the Fourier transform function `FT` and a continuous logarithm `logc`. The calibration itself is done in the function `calibration` documented in Section 2. We use these functions either to calibrate simulations or to estimate real data in Section 3.

There are a few parameters to control the execution of the program:

- `model` permits the values "Merton", "Kou" and "real data". In the first two cases observations will be simulated with the corresponding model. Picking the last value, the user has to supply the necessary data in the workspace of R before running the script (vector of strikes `k`, vector of option prices `op`, interest rate `r` and maturity `T`).
- In case of simulations the user has to specify `noiseLevel`, the `sampleSize` and the number of Monte Carlo iterations in `monteCarlo`. Furthermore, `design` decides whether the observed strikes are sampled "deterministic" or "random".
- The choice of the cut–off parameter U is adjusted by `mode`. In [2] are three possibilities proposed which are (in modifications) also available in this implementation. Possible values for `mode` are:
  - "oracle": U^* and U_ν^* minimize the discrepancy between the estimators and the given values, separately for $(\sigma^2, \gamma, \lambda)$ and ν.
  - "flat": The cut–offs correspond to points where the estimators stabilize:

$$U^* = \operatorname{argmin}_U \left(\left| \frac{d}{dU} \hat{\sigma}_U \right| + \alpha U \right), \quad \alpha > 0,$$
$$U_\nu^* = \operatorname{argmin}_{U_\nu \leq U^*} \left\| \frac{d}{dU} \hat{\nu}_{U_\nu} \right\|_{L^2}.$$

    In our simulations and real data estimations we used $\alpha = 10^{-5}$.

– "PLS": The common cut–off U^* solves

$$\inf_U \left[\sum_{i=1}^{N} |C(K_i; \mathcal{T}_U) - Y_i|^2 + \alpha \int_{\mathbb{R}} |\hat{\nu}''_U(x)|^2 \, dx \right], \quad \alpha > 0,$$

where $C(K; \mathcal{T}_U)$ is the price at strike K computed using Levy triplet $\mathcal{T}_U = (\hat{\sigma}_U, \hat{\gamma}_U, \hat{\nu}_U,)$. In our simulations and real data estimations we use $\alpha = 0$.

– "fix": The same cut–off values `Ufix` and `UfixNu` are used to estimate the parameters and the jump density, respectively.

- If the variable `linear` is true then the observation will be interpolated linearly. Otherwise quadratic B–splines are used.
- The smoothness of the jump density is set in parameter `s` (in the context of [1] `s` implies a smoothness $s = 2*\texttt{s}$ of ν).

We remark that the library `cobs` is necessary to use this program. It provides the spline interpolation.

2 The function `calibration`

Given a vector of log–strikes `sk` and corresponding call option prices `snop` this function performs the calibration of the model and returns the cut–off values U, U_ν, the model parameters $\hat{\sigma}, \hat{\gamma}, \hat{\lambda}$ and the estimated jump density $\hat{\nu}$ on a grid `x`. If a simulation is performed then also the quadratic L^2–error of $\hat{\nu}$ is given back.

```
calibration<-function(sk, snop){
  ...
  # return
  list(U=U, UNu=UNu, sigmaHat=sigmaHat, gammaHat=gammaHat, lambdaHat=
      lambdaHat, x=x, nuHat=nuHat, nuError2=nuError2)
}
```

To approximate the $\mathcal{O}$–function from the given discrete points (`sk`, `snop`), we switch from the log–scale to the normal one, because we can use the convexity of the function $K \mapsto C(K,T) = e^{-rT}\mathbb{E}[(S_T - K)^+]$ for the quadratic spline interpolation. Furthermore, we extrapolate the observations by adding boundary points whose position is set by `extrapolation`. To get an option value for the new strike near zero we use the slope of $C(K,T)$ in K at 0. In the case of quadratic spline interpolation the function `cobs` from package `cobs` is used to interpolate and evaluate the result on the logarithmic scale on a fine grid. We get the points (`knew`, `opnew`) and use the put–call parity to have a discrete version of the function $k \mapsto \tilde{\mathcal{O}}(k - rT)$.

```
extrapolation<-0.005

extraK<-rep(0,length(sK)+2)
extraK[1]<-extrapolation*sK[1]
extraK[2:(length(sK)+1)]<-sK
extraK[length(sK)+2]<-sK[length(sK)]*1/extrapolation

extraOp<-rep(0,length(sK)+2)
extraOp[1]<-1-exp(-r*T)*extraK[1]
extraOp[2:(length(sK)+1)]<-snop
extraOp[length(sK)+2]<-0

BI<-log(extraK[1])
BE<-log(extraK[length(extraK)])
```

```
knew<-seq(BI,BE,length=2^12)

if(linear){
  OpofK <- approxfun(extraK,extraOp, method="linear",rule=2,ties=
      mean)
  opnew <- OpofK(exp(knew))
}else{
  css<-cobs(extraK,extraOp,constraint="convex",nknots=min(100,length
      (sK)-2),degree=2)
  opnew<-predict(css,exp(knew))[,2]
}

opnew<-opnew-pmax(0.,1-exp(knew-r*T))
opnew<-pmax(0,opnew)
```

We can now apply the Fourier transform to get first the estimate of $v \mapsto \tilde{\varphi}_T(v-i) = 1 + iv(1+iv)\mathcal{F}\tilde{O}(v)$ and calculate then $\tilde{\psi}(v) = 1/T \log(\tilde{\varphi}(v-i))$ by profiting from symmetry. Now we have (`v`, `psi`) and define `L` as the index where `v` has value 0.

```
z<-FT(knew-r*T,opnew,TRUE)  #x=k-rT,  z=FO(v)

v<-z$u
phi<-z$fy*1i*v*(1+1i*v)+1

psi<-rep(0,length(phi))
psi[(length(v)/2):length(v)]<-1/T*logc(phi[(length(v)/2):length(v)])
psi[1:(length(v)/2-1)]<-Re(psi[(length(v)-1):(length(v)/2+1)])-1i*Im
    (psi[(length(v)-1):(length(v)/2+1)])

L<-length(v)/2
```

In the following we calculate for numerous cut–off values U the estimators $\hat{\sigma}^2, \hat{\gamma}$ and $\hat{\lambda}$ and save each in `sigma2HatCur`, `gammaHatCur` and `lambdaHatCur`, respectively. The cut–off values are multiples of the mesh of `v` and their number is given by `cutOffIterations`. Since the calculation time in the "PLS"–mode is much longer than in the other modes but the picked cut–off is smaller in general we choose less iterations in "PLS". If the mode is "fix", only one iteration with the fixed cut–off suffices. Because of a large bias for small U we start with `(v[2]-v[1])*11` and therefore end at `(v[2]-v[1])*(10+cutOffIterations)`. Later we need `bestError` for the decision in the oracle–mode. With use of symmetry it is enough to calculate (`v_cut`, `psi_cut`) on the positive half–axis.

```
if(mode=="oracle" | mode=="flat")
  cutOffIterations<-130
else if(mode=="PLS")
  cutOffIterations<-90
else if(mode=="fix")
  cutOffIterations<-1

sigma2HatCur<-rep(0,cutOffIterations)
gammaHatCur<-rep(0,cutOffIterations)
lambdaHatCur<-rep(0,cutOffIterations)
bestError<--1
for(i in 1:cutOffIterations){
  if(mode=="fix"){
    UCur<-Ufix
    gridU<-round(Ufix/(v[2]-v[1]))
```

```
    }else{
      UCur <- (v[2]-v[1])*(i+10)
      gridU <- i+10
    }
    v_cut<-v[L:(L+gridU)]
    psi_cut<-psi[L:(L+gridU)]
    ...
  }
```

Within this loop we approximate the integrals in the definition of the estimators $\hat{\sigma}^2, \hat{\gamma}$ and λ [cf. 4, Sect. 3] with a composite trapezoidal rule and use the polynomial weight functions from above.

```
    w<-rep(1,length(psi_cut))
    w[length(psi_cut)]<-0.5

    wFkt<-function(x){(2*s+1)*x^(2*s)-4*(2*s+3)*x^(2*s+2)+6*(2*s+5)*x
        ^(2*s+4)-4*(2*s+7)*x^(2*s+6)+(2*s+9)*x^(2*s+8)}
    weight<-wFkt(v_cut/UCur)
    weight[length(weight)] <- weight[length(weight)]-2*sum(w*weight)
    sigma2HatCur[i]<-sum(w*weight*Re(psi_cut))
    sigma2HatCur[i]<--2*sigma2HatCur[i]/sum(w*abs(v_cut)^{2}*weight)

    wFkt<-function(x){x^(2*s+1)-3*x^(2*s+3)+3*x^(2*s+5)-x^(2*s+7)}
    weight<-wFkt(v_cut/UCur)
    gammaHatCur[i]<-sum(w*weight*Im(psi_cut))
    gammaHatCur[i]<-gammaHatCur[i]/sum(w*weight*v_cut)-sigma2HatCur[i]

    wFkt<-function(x){(2*s+3)*x^(2*s)-4*(2*s+5)*x^(2*s+2)+6*(2*s+7)*x
        ^(2*s+4)-4*(2*s+9)*x^(2*s+6)+(2*s+11)*x^(2*s+8)}
    weight<-wFkt(v_cut/UCur)
    weight[length(weight)] <- weight[length(weight)]-2*sum(w*weight*v_
        cut^2)/v[L+gridU]^2
    lambdaHatCur[i]<-sum(w*Re(psi_cut)*weight)
    lambdaHatCur[i]<--lambdaHatCur[i]/sum(w*weight)+gammaHatCur[i]+
        sigma2HatCur[i]/2
```

The choice of `U` depends on the mode. In "oracle" in every iteration step we calculate the distance between the estimators and the true values as an `error` term and select the minimizing U. In the case of "fix" the cut–off is unique.

```
    if(mode=="oracle"){
      error<-1*abs(sigma2HatCur[i]-sigma^2)+1*abs(gammaHatCur[i]-gamma
          )+1*abs(lambdaHatCur[i]-lambda)
      if((i==1) | error<bestError){
        U<-UCur
        sigmaHat<-sqrt(pmax(0,sigma2HatCur[i]))
        gammaHat<-gammaHatCur[i]
        lambdaHat<-lambdaHatCur[i]
        bestError<-error
      }
    }
    if(mode=="fix"){
      U<-UCur
      sigmaHat<-sqrt(pmax(0,sigma2HatCur[i]))
      gammaHat<-gammaHatCur[i]
      lambdaHat<-lambdaHatCur[i]
```

```
    bestError<-error
  }
```

After the loop and in case of the "flat"–mode we calculate a moving average of order two of the differences in the discrete curve $U \mapsto \hat{\sigma}_U$ and select the minimum of these values with respect to a penalty for big U.

```
if(mode=="flat"){
  diff<-diff(sigma2HatCur[1:length(sigma2HatCur)])
  ma<-rep(0,length(diff))
  maOrder<-2
  for(i in (1:length(diff))){
    ma[i]<-sum(abs(diff[max(i-maOrder,1):min(i+maOrder-1,length(diff
        ))]))/(min(i+maOrder,length(diff))-max(i-maOrder,1))
  }

  Us<-(v[2]-v[1])*(10+(1:(cutOffIterations-1)))
  alpha<-1e-5
  i<-which.min(ma+alpha*Us)
  U<-Us[i]
  sigmaHat<-sqrt(pmax(0,sigma2HatCur[i]))
  gammaHat<-gammaHatCur[i]
  lambdaHat<-lambdaHatCur[i]
}
```

To estimate ν, we first have to build $\tilde{\psi}_\nu$:

```
z<-FT(knew-r*T,exp(-knew+r*T)*opnew,TRUE)
phiNu<-1-v*(v+1i)*z$fy
psiNu<-rep(0,length(phiNu))
psiNu[(length(v)/2):length(v)]<-1/T*logc(phiNu[(length(v)/2):length(
    v)])
psiNu[1:(length(v)/2-1)]<-Re(psiNu[(length(v)-1):(length(v)/2+1)])-1
    i*Im(psiNu[(length(v)-1):(length(v)/2+1)])
```

As before we use a loop to calculate $\hat{\nu}$ for different cut–off values. Since we have a separate cut–off `UNu` in the cases of "oracle" and "flat", the number of iterations is restricted to the selected cut–off `U` of the three parameters (in that way we save calculations, because $\hat{\nu}$ needs smaller cut–offs in general). The resulting $\hat{\nu}_U$ are stored in a matrix `nuHatCur`. Furthermore, we initiate some auxiliary variables and the `flatTopKernel`, as defined in (13) of [4].

```
if(mode=="oracle" | mode=="flat")
  cutOffIterations<-max(ceiling(U/(v[2]-v[1])),20)-10
nuHatCur<-matrix(0,cutOffIterations,length(v))
nuError2<--1
if(mode=="oracle")
  nuError2Cur<-rep(0,cutOffIterations)
if(mode=="PLS")
  objective<--1
flatTopKernel<-function(x){(abs(x)<=0.05)+(abs(x)<1 & abs(x)>0.05)*(
    exp(-exp(-(abs(x)-0.05)^(-2))/(abs(x)-1)^2))}
for(i in 1:cutOffIterations){
  if(mode=="fix")
    UCur<-UfixNu
  else
    UCur <- (v[2]-v[1])*(i+10)
  ...
```

```
}
```

The first step in this loop is to calculate $\hat{\nu}_U$ with an inverse Fourier transform as in (12). Depending on the mode we use the final estimators $\hat{\sigma}, \hat{\gamma}$ and $\hat{\lambda}$ or the parameters per cut–off (since we have in the "PLS" mode a common cut–off). `xCur` denotes the grid on which the jump density is provided. It will be the same in every iteration step because it only depends on the grid `v` of `psiNu`.

```
if(mode=="oracle" | mode=="flat" | mode=="fix"){
  lam<-lambdaHat
  fnuHat<-(psiNu-1i*gammaHat*v+lam+sigmaHat^2*v^{2}/2)*
      flatTopKernel(v/UCur)
}
else if(mode=="PLS"){
  lam<-lambdaHatCur[i]
  fnuHat<-(psiNu-1i*gammaHatCur[i]*v+lam+sigma2HatCur[i]*v^{2}/2)*
      flatTopKernel(v/UCur)
}
fxy_nu <- FT(v,fnuHat, noInverse=FALSE)
xCur <- fxy_nu$u
nuHatCur[i,] <- Re(fxy_nu$fy)
```

Up to now the condition $\hat{\nu}_U \geq 0$ is not ensured and needs a correction of $\hat{\nu}_U$. Referring to [2], we seek for a ξ such that the integral $\int_{\mathbb{R}} \max\{0, \hat{\nu}_U(x) - \xi\}\, \mathrm{d}x$ equals $\hat{\lambda}$ or $\hat{\lambda}_U$, respectively. We approximate the difference between the integral and `lam` as a function `eq` of `xi`, approximate its root, using the secants method, and redefine $\hat{\nu}_U$ as the corrected version.

```
wNu<-rep(xCur[2]-xCur[1],length(xCur))
wNu[1]<-0.5*wNu[1]
wNu[length(wNu)]<-0.5*wNu[length(wNu)]

if(lam>0){
  eq<-function(xi){sum(wNu*(pmax(0,nuHatCur[i,]-xi)))-lam}
  xi0<-0
  fxi0<-eq(xi0)
  xi1<-1
  fxi1<-eq(xi1)
  while(abs(fxi1)>0.001){
    xi2<-xi1-(xi1-xi0)/(fxi1-fxi0)*fxi1
    xi0<-xi1
    xi1<-max(0,xi2)
    fxi0<-fxi1
    fxi1<-eq(xi1)
  }
}else
  xi2<-0
nuHatCur[i,]<-pmax(0,nuHatCur[i,]-xi2)
```

Now we are in position to do the mode depending choice of $\hat{\nu}$. In the "PLS" mode we also select the other three parameters. In "oracle" we calculate for every `UCur` the L^2–distance to the true ν and pick the cut–off `UNu` with the smallest one. In "fix"–mode there is only one estimation.

```
if(mode=="oracle"){
  nuTrue<-Nu(xCur)
  nuError2Cur[i]<-sum(wNu*(nuHatCur[i,]-nuTrue)^2)
  if(nuError2<0 | nuError2Cur[i]<nuError2){
    UNu<-UCur
    x<-xCur
```

```
      nuHat<-nuHatCur[i,]
      nuError2<-nuError2Cur[i]
    }
  }
  if(mode=="fix"){
    UNu<-UCur
    x<-xCur
    nuHat<-nuHatCur[i,]
    if(model!="real_data"){
      nuTrue<-Nu(xCur)
      nuError2<-sum(wNu*(nuHatCur[i,]-nuTrue)^2)
    }
  }
```

Still within the loop we deal with the "PLS" mode. Since it is driven by a least squares distance between the observations and the estimated model, we have to compute $\mathcal{O}$ from $\hat{\sigma}_U, \hat{\gamma}_U, \hat{\lambda}_U$ and $\hat{\nu}_U$. It is important that the martingale condition [4, equation (3)] holds for each U, otherwise the right–hand site of the pricing formula [4, equation (5)] can have a singularity. Therefore, we correct $\hat{\lambda}_U$ and $\hat{\nu}_U$ simultaneously. Remark that this is not done in the other modes (and actually cannot be done before the calculation of $\hat{\nu}$ is finished). The option pricing of (`kE`, `opE`) itself is described in detail in section 3. From all available strikes `kE` we chose the ones next to the given observations `sk` and get (`kHat`, `opHat`).

```
  if(mode=="PLS"){
    beta<-(0.5*sigma2HatCur[i]+gammaHatCur[i])/(lambdaHatCur[i]-sum(
        wNu*exp(xCur)*nuHatCur[i,]))
    lambdaHatCur[i]<-beta*lambdaHatCur[i]
    nuHatCur[i,]<-beta*nuHatCur[i,]

    vE<--2^9/2+(0:(2^9-1))*2^9/(2^9-1)
    yE<-sapply(vE, function(v){ZetaHat(v, sigma2HatCur[i],
        gammaHatCur[i], lambdaHatCur[i], xCur, nuHatCur[i,])})
    fxy<-FT(vE,yE,FALSE)
    kE<-fxy$u
    opE<-Re(fxy$fy)
    kHat<-rep(0,length(sk))
    opHat<-rep(0,length(sk))
    for(j in (1:length(sk))){
      index<-which.min(abs(kE-sk[j]))
      kHat[j]<-kE[index]
      opHat[j]<-opE[index]
    }
    ...
  }
```

To penalize large second derivatives of ν, the second order differences of `nuHatCur` are provided in `nuDeriv`. The term to be minimized is given in `objectiveCur`. In every iteration step we check whether the new objective is smaller then the last best.

```
    nuDeriv<-diff(nuHatCur[i,],2)
    wE<-rep(xCur[2]-xCur[1],length(nuDeriv))
    wE[1]<-0.5*wE[1]
    wE[length(wE)]<-0.5*wE[length(wE)]
    alpha<-0 #1e-8
    objectiveCur<-sum((opHat-snop+pmax(0.,1-exp(sk-r*T)))^2)+alpha*
        sum(wE*(nuDeriv/(xCur[2]-xCur[1])^2)^2)
```

```
    if(!is.nan(objectiveCur) & (objective<0 | objectiveCur<objective
       )){
      U<-UCur
      UNu<-UCur
      sigmaHat<-sqrt(pmax(0,sigma2HatCur[i]))
      gammaHat<-gammaHatCur[i]
      lambdaHat<-lambdaHatCur[i]
      x<-xCur
      nuHat<-nuHatCur[i,]
      objective<-objectiveCur
      if(model!="real_data"){
        nuTrue<-Nu(xCur)
        nuError2<-sum(wNu*(nuHat-nuTrue)^2)
      }
    }
```

In the "flat" mode the choice of U_ν is made at the end of the loop. We approximate the L^2–norm of the derivative of the map $U \mapsto \hat{\nu}_U$ and select its minimum.

```
  if(mode=="flat"){
    derivativU<-diff(nuHatCur)/(v[2]-v[1])
    derivativL2<-rep(0,length(derivativU[,1]))
    for(i in (1:length(derivativU[,1])))
      derivativL2[i]<-sum(wNu*derivativU[i,]^2)
    i<-which.min(derivativL2)
    UNu<-(v[2]-v[1])*(i+10)
    x<-xCur
    nuHat<-nuHatCur[i,]
    if(model!="real_data"){
      nuTrue<-Nu(xCur)
      nuError2<-sum(wNu*(nuHat-nuTrue)^2)
    }
  }
```

Now we have all estimators and can return the results.

3 Option pricing from Lévy triplet

Given values of σ, γ, λ and ν we want to provide the $\mathcal{O}$–function. This happens in two situations of the script. On the one hand we simulate observations from a specific model and apply the calibration to the noised data and on the other hand, we have to calculate option prices from the estimated parameters in the least squares method. In [3, p. 361 et seq.] a good description of this option pricing can be found.

We define the function

$$\zeta(v) := e^{ivrT} \frac{\phi_T(v-i) - 1}{iv(1+iv)}$$

which is implemented as `Zeta(v)` and `ZetaHat(v, sigma2Hat, gammaHat, lambdaHat, x, nuHat)`, respectively. Then $\mathcal{O}$ as function of log–strike k is obtained by inverse Fourier transform of ζ. Hence, we evaluate ζ on a discrete grid `x` to get pairs of log–strikes and option prices (`k`, `op`). Because of the FFT algorithm a finer grid `x` leads to a larger limits of `k` and the other way around a higher range of `x` implies a smaller mesh of `k`.

```
A<-2^10
M<-12
N<-2^M
```

```
l<-0:(N-1)
Delta<-A/(N-1)
x<--A/2+l*Delta
y<-Zeta(x)
fxy<-FT(x,y,FALSE)
k<-fxy$u
op<-Re(fxy$fy)+pmax(0.,1-exp(k-r*T))
```

For simulations we noise and sample all pairs (`k`,`op`) and apply the calibration procedure in every Monte Carlo iteration step. Here the sampling is done according to a normal distribution centered at at–the–money strikes. Depending on `design`, the strikes a distributed deterministic or random.

```
for(mi in 1:monteCarlo){
  nop<-op+rnorm(length(op),0,(noiseLevel*abs(Re(fxy$fy))))

  M1<-round(length(op)/2)-which.min(abs(op[1:round(length(op)/2)
      ]-10^(-6)))
  M2<-which.max(op)+M1
  M1<-which.max(op)-M1

  if(design=="random"){
    prob<-exp(-(k[M2:M1]-r*T)^2)
    prob<-prob/sum(prob)
    ind_sub<-sort(sample(M2:M1,sampleSize,prob=prob))
    sk<-k[ind_sub]
    snop<-nop[ind_sub]
  }else if(design=="deterministic"){
    quantils<-qnorm(c(1:sampleSize)/(sampleSize+1),0,2^(-0.5))
    ind_sub<-sapply(quantils,function(x){which.min(abs(k-r*T-x))})
    sk<-k[ind_sub]
    snop<-nop[ind_sub]
  }
  snop<-snop+pmax(0.,1-exp(sk-r*T))
  ...
}
```

References

[1] Belomestny, D. and M. Reiß (2006a). Spectral calibration of exponential Lévy models. *Finance Stoch 10*, 449–474.

[2] Belomestny, D. and M. Reiß (2006b). Spectral calibration of exponential Lévy models [2]. SFB 649 Discussion Paper 2006-035, Sonderforschungsbereich 649, Humboldt Universität zu Berlin, Germany. Available at http://sfb649.wiwi.hu-berlin.de/papers/pdf/SFB649DP2006-035.pdf.

[3] Cont, R. and P. Tankov (2004). Non-parametric calibration of jump-diffusion option pricing models. *J. Comput. Finance 7*(3), 1–49.

[4] Söhl, J. and M. Trabs (2012). Option calibration of exponential Lévy models: Confidence intervals and empirical results. arXiv:1202.5983.

[5] Trabs, M. (2011). Calibration of self-decomposable Lévy models. arXiv:1111.1067.