COMISEF WORKING PAPERS SERIES

WPS-030 08/03/2010

Calibrating Option Pricing Models with Heuristics

M. Gilli
E. Schumann

Calibrating Option Pricing Models with Heuristics

Manfred Gilli and Enrico Schumann[*]

University of Geneva, Department of Econometrics, and Swiss Finance Institute

University of Geneva, Department of Econometrics

March 8, 2010

Abstract

Calibrating option pricing models to market prices often leads to optimisation problems to which standard methods (like such based on gradients) cannot be applied. We investigate two models: Heston's stochastic volatility model, and Bates's model which also includes jumps. We discuss how to price options under these models, and how to calibrate the parameters of the models with heuristic techniques.

1 Introduction

Implied volatilities obtained by inverting the Black–Scholes–Merton (BSM) model vary systematically with strike and maturity; this relationship is called the volatility surface. Different strategies are possible for incorporating this surface into a model. We can accept that volatility is not constant across strikes and maturities, and directly model the volatility surface and its evolution. With this approach we assume that a single underlier has different volatilities; but still, it is the approach that is mostly used in practice. An alternative is to model the option prices such that the BSM-volatility surface is obtained, for instance by including locally-varying volatility (Derman and Kani, 1994; Dupire, 1994), jumps, or by making volatility stochastic. In this paper, we look into models that follow the latter two approaches, namely the models of Heston (1993) and Bates (1996).

As so often in finance, the success of the BSM model stems not so much from its empirical quality, but from its computational convenience. This convenience comes in two flavours. Firstly, we have closed-form pricing equations (the Gaussian distribution function is not available analytically, but fast and precise approximations exist). Secondly, calibrating the model requires only one parameter to be determined, the volatility, which can be readily computed from market prices with Newton's method or another zero-finding technique. For the Heston and the Bates model, both tasks become more difficult. Pricing requires numerical integration, and calibration requires to find five and eight parameters instead of only one for BSM.

In this paper, we will look into the calibration of these models. Finding parameters that make the models consistent with market prices means solving a non-convex

optimisation problem. We suggest to use optimisation heuristics, more specifically we show that Differential Evolution and Particle Swarm Optimisation are both able to give good solutions to the problem. The paper is structured as follows: In Section 2 we discuss how to price options under the Heston and the Bates model. Fast pricing routines are important since the suggested heuristics are computationally intensive; hence to obtain calibration results in a reasonable period of time, we need to be able to evaluate the objective function (which requires pricing) speedily. Section 3 details how to implement the heuristics for a calibration problem, Section 4 discusses several computational experiments and their results. Section 5 concludes.

2 Pricing with the characteristic function

There are several generic approaches to price options. The essence of BSM is a no-arbitrage argument; it leads to a partial differential equation that can be solved numerically or, in particular cases, even analytically. A more recent approach builds on the characteristic function of the (log) stock price. European options can be priced by the following equation (Bakshi and Madan, 2000; Schoutens, 2003):

$$C_0 = \mathrm{e}^{-q\tau} S_0 \Pi_1 - \mathrm{e}^{-r\tau} X \Pi_2 \tag{1}$$

where C_0 is the call price today (time 0), S_0 is the spot price of the underlier, and X is the strike price; r and q are the riskfree rate and dividend yield; time to expiration is denoted τ. The Π_j are calculated as

$$\Pi_1 = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \mathrm{Re}\left(\frac{\mathrm{e}^{-\mathrm{i}\omega \log(X)} \phi(\omega - \mathrm{i})}{\mathrm{i}\omega \phi(-\mathrm{i})} \right) \mathrm{d}\omega\,, \tag{2a}$$

$$\Pi_2 = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \mathrm{Re}\left(\frac{\mathrm{e}^{-\mathrm{i}\omega \log(X)} \phi(\omega)}{\mathrm{i}\omega} \right) \mathrm{d}\omega\,. \tag{2b}$$

We define $\Pi_j^* \equiv \pi(\Pi_j - 1/2)$ for the integrals in these equations. The symbol ϕ stands for the characteristic function of the log stock price; the function $\mathrm{Re}(\cdot)$ returns the real part of a complex number. For a given ϕ we can compute Π_1 and Π_2 by numerical integration, and hence obtain option prices from Equation (1).

2.1 Black–Scholes–Merton

In a BSM world, the stock price S_t under the risk-neutral measure follows

$$\mathrm{d}S_t = (r - q)S_t \mathrm{d}t + \sqrt{v}S_t \mathrm{d}z_t \tag{3}$$

where z is a Wiener process (Black and Scholes, 1973). The volatility $\sqrt{v}$ is constant. The well-known pricing formula for the BSM call is given by

$$C_0 = \mathrm{e}^{-q\tau} S_0 \, \mathrm{N}(d_1) - X\mathrm{e}^{-r\tau} \, \mathrm{N}(d_2) \tag{4}$$

with

$$d_1 = \frac{1}{\sqrt{v\tau}} \left(\log\left(\frac{S_0}{X}\right) + \left(r - q + \frac{v}{2}\right)\tau \right) \tag{5a}$$

$$d_2 = \frac{1}{\sqrt{v\tau}} \left(\log\left(\frac{S_0}{X}\right) + \left(r - q - \frac{v}{2}\right)\tau \right) = d_1 - \sqrt{v\tau} \tag{5b}$$

and $N(\cdot)$ the Gaussian distribution function.

Given the dynamics of S, the log price $s_\tau = \log(S_\tau)$ follows a Gaussian distribution with $s_\tau \sim \mathcal{N}\left(s_0 + \tau(r - q - \frac{1}{2}v), \tau v\right)$, where s_0 is the natural logarithm of the current spot price. The characteristic function of s_τ is given by

$$
\begin{aligned}
\phi_{\text{BSM}}(\omega) &= \mathsf{E}(e^{i\omega s_\tau}) \\
&= \exp\left(i\omega s_0 + i\omega\tau(r - q - \frac{1}{2}v) + \frac{1}{2}i^2\omega^2 v\tau\right) \\
&= \exp\left(i\omega s_0 + i\omega\tau(r - q) - \frac{1}{2}(i\omega + \omega^2)\tau v\right).
\end{aligned}
\tag{6}
$$

Inserting (6) into Equation (1) should, up to numerical precision, give the same result as Equation (4).

2.2 Merton's jump–diffusion model

Merton (1976) suggested to model the underlier's movements as a diffusion with occasional jumps; thus we have

$$
dS_t = (r - q - \lambda\mu_J)S_t dt + \sqrt{v}S_t dz_t + J_t S_t dN_t.
\tag{7}
$$

N_t is a poisson counting process, with intensity λ; the J_t is the random jump size (given that a jump occurred). In Merton's model the log-jumps are distributed as

$$
\log(1 + J_t) \sim \mathcal{N}\left(\log(1 + \mu_J) - \frac{\sigma_J^2}{2}, \sigma_J^2\right).
$$

The pricing formula is the following (Merton, 1976, p. 135):

$$
C_0 = \sum_{n=0}^{\infty} \frac{e^{-\lambda'\tau}(\lambda'\tau)^n}{n!} C_0'(r_n, \sqrt{v_n})
\tag{8}
$$

where $\lambda' = \lambda(1 + \mu_J)$ and C_0' is the BSM formula (4), but the prime indicates that C_0' is evaluated at adjusted values of r and v:

$$
\begin{aligned}
v_n &= v + \frac{n\sigma_J^2}{\tau} \\
r_n &= r - \lambda\mu_J + \frac{n\log(1 + \mu_J)}{\tau}
\end{aligned}
$$

The factorial in Equation (8) may easily lead to an overflow (`Inf`), but it is benign for two reasons. Firstly, we do not need large numbers for n, a value of about 20 is well-sufficient. Secondly (if we insist on large n), software packages like Matlab or R will evaluate `1/Inf` as zero, hence the summing will add zeros for large n. (Numerical analysts prefer to replace $n!$ by $\exp(\sum_{i=1}^n \log i)$ since this leads to better accuracy for large n. Again, for Merton's model this is not needed.) Depending on the implementation, working with large values for n may still lead to a warning or an error, and so interrupt a computation. In R for instance, the handling of such a warning will depend on the options setting:

3

```
1 > options()$warn
2 [1] 0
```

This is the standard setting. Computing the factorial for a large number will result in a warning; the computation continues.

```
1 > factorial(200)
2 [1] Inf
3 Warning message:
4 In factorial(200) : value out of range in 'gammafn'
```

But with `warn` set to two, any warning will be transformed into an error. Thus:

```
1 > options(warn=2)
2 > factorial(200)
3 Error in factorial(200) :
4   (converted from warning) value out of range in 'gammafn'
```

and our computation breaks. We may want to safeguard against such possible errors: we can for instance replace the function call `factorial(n)` by its actual calculation which produces:

```
1 > options(warn=2)
2 > exp( sum(log(1:200)) )
3 [1] Inf
4 > prod(1:200)
5 [1] Inf
```

Or even simpler, as in Matlab's implementation of `factorial`, we can check the given value of n; if it is too large, we have it replaced by a more reasonable value.

The characteristic function of Merton's model is given by

$$\phi_{\text{Merton}} = e^{A+B} \tag{9}$$

where

$$A = i\omega s_0 + i\omega\tau(r - q - \frac{1}{2}v - \mu_J) + \frac{1}{2}i^2\omega^2 v\tau$$

$$B = \lambda\tau\left(\exp\left(i\omega \log(1 + \mu_J) - \frac{1}{2}i\omega\sigma_J^2 - \omega^2\sigma_J^2 \right) - 1 \right),$$

see Gatheral (2006, ch. 5). The A-term in ϕ_{Merton} corresponds to the BSM dynamics with a drift adjustment to account for the jumps; the B-term adds the jump component. Like in the BSM case, we can compare the results from Equation (1) with those obtained from Equation (8).

2.3 The Heston model

Under the Heston (1993) model the stock price S and its variance v are described by

$$dS_t = rS_t dt + \sqrt{v_t}S_t dz_t^{(1)} \tag{10a}$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t}dz_t^{(2)}. \tag{10b}$$

The long-run variance is denoted θ, mean reversion speed is κ and σ is the volatility-of-volatility. The Wiener processes $z^{(\cdot)}$ have correlation ρ. For $\sigma \to 0$, the Heston

dynamics approach those of BSM. A thorough discussion of the model can be found in Gatheral (2006). The characteristic function of the log-price in the Heston model looks as follows, see Albrecher et al. (2007).

$$\phi_{\text{Heston}} = e^{A+B+C} \tag{11}$$

where

$$A = i\omega s_0 + i\omega(r-q)\tau$$

$$B = \frac{\theta\kappa}{\sigma^2}\left((\kappa - \rho\sigma i\omega - d)\tau - 2\log\left(\frac{1 - ge^{-d\tau}}{1-g}\right)\right)$$

$$C = \frac{\frac{v_0}{\sigma^2}\left(\kappa - \rho\sigma i\omega - d\right)\left(1 - e^{-d\tau}\right)}{1 - ge^{-d\tau}}$$

$$d = \sqrt{(\rho\sigma i\omega - \kappa)^2 + \sigma^2(i\omega + \omega^2)}$$

$$g = \frac{\kappa - \rho\sigma i\omega - d}{\kappa - \rho\sigma i\omega + d}$$

With only five parameters (under the risk-neutral probability), the Heston model is capable of producing a volatility smile, see Figure 1.

2.4 The Bates Model

This model, described in Bates (1996), adds jumps to the dynamics of the Heston model. The stock price S and its variance v are described by
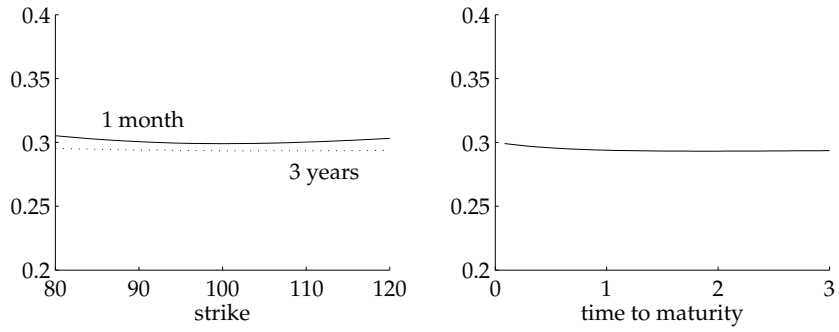
$$dS_t = (r - q - \lambda\mu_J)S_t dt + \sqrt{v_t}S_t dz_t^{(1)} + J_t S_t dN_t$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t}dz^{(2)}.$$

N_t is poisson count process with intensity λ, hence the probability to have a jump of size one is λdt. Like in Merton's model, the logarithm of the jump size J_t is distributed as a Gaussian, ie,
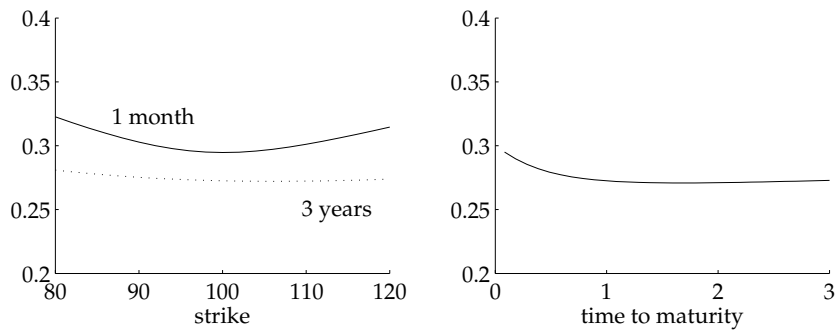
$$\log(1 + J_t) = \mathcal{N}\left(\log(1 + \mu_J) - \frac{\sigma_J^2}{2}, \sigma_J^2\right).$$

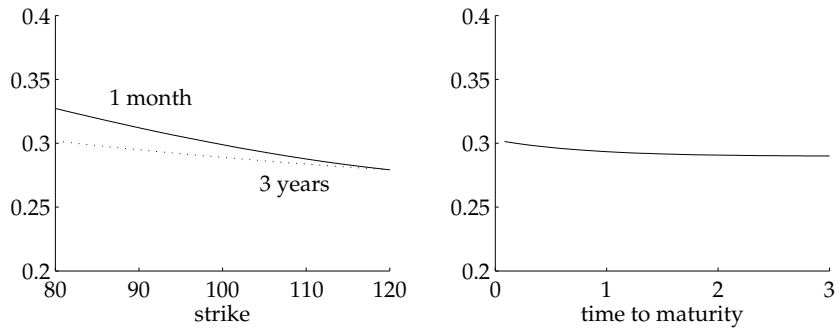The characteristic function becomes (Schoutens et al., 2004):

$$\phi_{\text{Bates}} = e^{A+B+C+D} \tag{12}$$
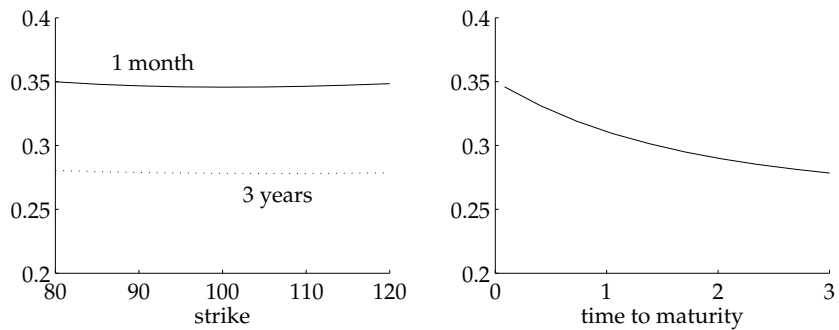
(a) The base case: $S = 100$, $r = 2\%$, $q = 2\%$, $\sqrt{v_0} = 30\%$, $\sqrt{\theta} = 30\%$, $\rho = 0$, $\kappa = 1$, $\sigma = 30\%$.

(b) $\sigma = 70\%$: short-term smile (the position of the kink is controlled by ρ); often we need substantial volatility-of-volatility

(c) $\rho = -0.5$: skew (a positive correlation induces positive slope).

(d) $v_0 = 35\%$, $\theta = 25\%$: term structure is determined by the difference between current and long-run variance, and κ.

Figure 1: Heston model: re-creating the implied volatility surface.
The graphics show the ʙꜱᴍ implied volatilities obtained from prices under the Heston model.

with

$$A = \mathrm{i}\omega s_0 + \mathrm{i}\omega(r-q)\tau$$

$$B = \frac{\theta\kappa}{\sigma^2}\left((\kappa - \rho\sigma\mathrm{i}\omega - d)\tau - 2\log\left(\frac{1 - g\mathrm{e}^{-d\tau}}{1-g}\right)\right)$$

$$C = \frac{\frac{v_0}{\sigma^2}\left(\kappa - \rho\sigma\mathrm{i}\omega - d\right)\left(1 - \mathrm{e}^{-d\tau}\right)}{1 - g\mathrm{e}^{-d\tau}}$$

$$D = -\lambda\mu_J\mathrm{i}\omega\tau + \lambda\tau\left((1+\mu_J)^{\mathrm{i}\omega}\mathrm{e}^{\frac{1}{2}\sigma_J^2\mathrm{i}\omega(\mathrm{i}\omega-1)} - 1\right)$$

$$d = \sqrt{(\rho\sigma\mathrm{i}\omega - \kappa)^2 + \sigma^2(\mathrm{i}\omega + \omega^2)}$$

$$g = \frac{\kappa - \rho\sigma\mathrm{i}\omega - d}{\kappa - \rho\sigma\mathrm{i}\omega + d}$$

Since the jumps are assumed independent, the characteristic function is the product of ϕ_{Heston} with the function for the jump part (D). Figure 2 shows that adding jumps makes it easier to introduce curvature into the volatility surface, at least for short maturities.

2.5 Integration schemes

Sample Matlab programs for pricing under the different models are given in the Appendix and can be downloaded from `http://comisef.eu`. The programs use Matlab's quad function, an adaptive algorithm based on Simpson's rule; it is reliable but slow. The pricing can be accelerated by precomputing a fixed number of nodes and weights under the given quadrature scheme. We use a Gauss–Legendre rule, see Davis and Rabinowitz (2007), Trefethen (2008), and the Appendix; we experimented with alternatives like Gauss–Lobatto as well, but no integration scheme was clearly dominant over another, given the required precision of our problem (there is no need to compute option prices to eight decimals). Thus, in what follows, we do not use quad, but compute nodes and weights, and directly evaluate the integrals in Equations (2).

To test our pricing algorithms, we first investigate the BSM model and Merton's jump–diffusion model. For these models, we can compare the solutions obtained from the classical formulæ with those from integration. Furthermore, we can investigate several polar cases: for Heston with zero volatility-of-volatility we should get BSM prices; for Bates with zero volatility-of-volatility we should obtain prices like under Merton's jump diffusion (and of course Bates with zero volatility-of-volatility and no jumps should again give BSM prices).

As an example of these tests, we compare here results from Equation (4) with results from Equation (1). The integrands in Equations (2) are well-behaved for BSM, see Figure 3 in which we plot $\Pi_j^* = \pi(\Pi_j - 1/2)$. The functions start to oscillate for options that are far away from the money, increasingly so for low volatility and short time to maturity. This is shown in the left panel of Figure 4, where we plot Π_1^* for varying strikes X and ω values (Π_2^* looks similar). The right panel of Figure 4 gives
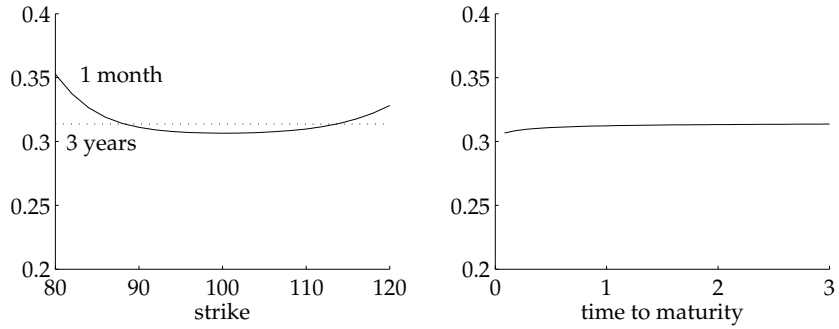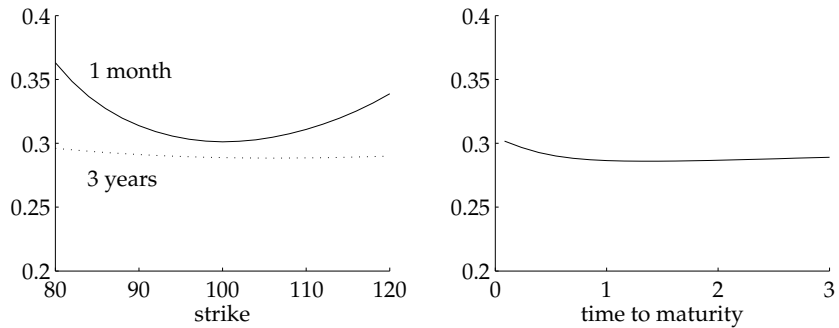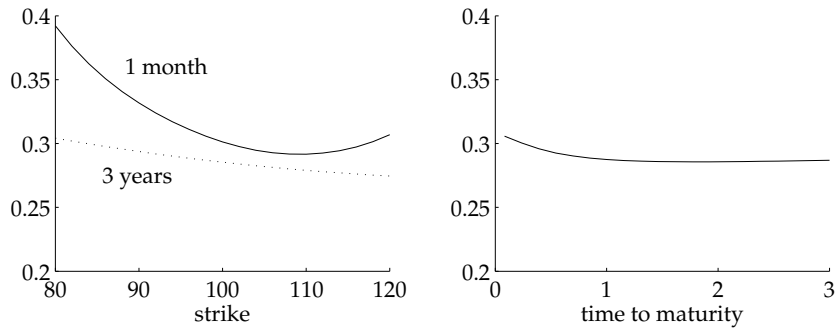
(a) The base case: $S = 100$, $r = 2\%$, $q = 2\%$, $\sqrt{v_0} = 30\%$, $\sqrt{\theta} = 30\%$, $\rho = 0$, $\kappa = 1$, $\sigma = 0.0\%$, $\lambda = 0.1$, $\mu_J = 0$, $\sigma_J = 30\%$. Volatility-of-volatility is zero, as is the jump mean.
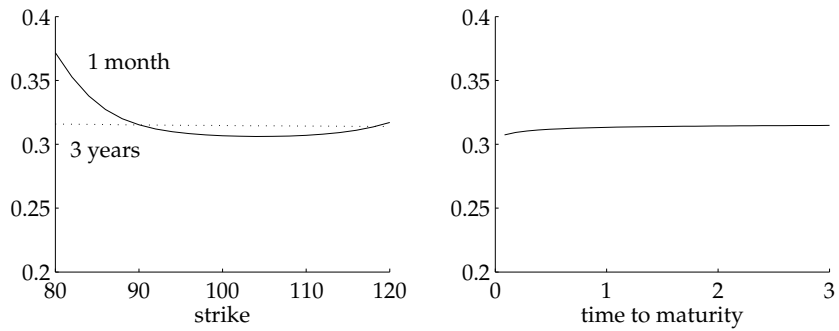
(b) $\mu_J = -10\%$: more asymmetry

(c) $\theta = 70\%$: stochastic volatility included.

(d) $\mu_J = -10\%$, $\theta = 70\%$, $\rho = -0.3$.

Figure 2: Bates model: re-creating the implied volatility surface
The graphics show the BSM implied volatilities obtained from prices under the Bates model.
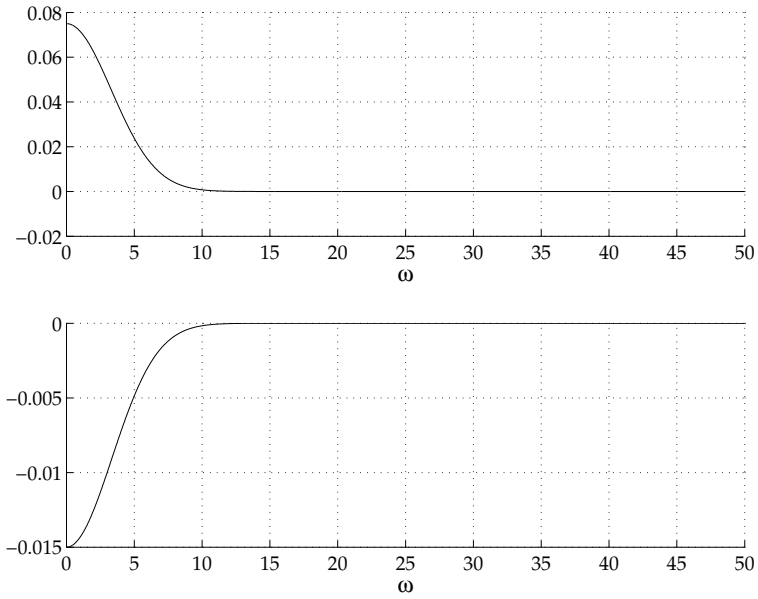
Figure 3: Π_1^* and Π_2^* for BSM ($S = 100$, $X = 100$, $\tau = 1$, $\sqrt{v} = 0.3$, $r = 0.03$).

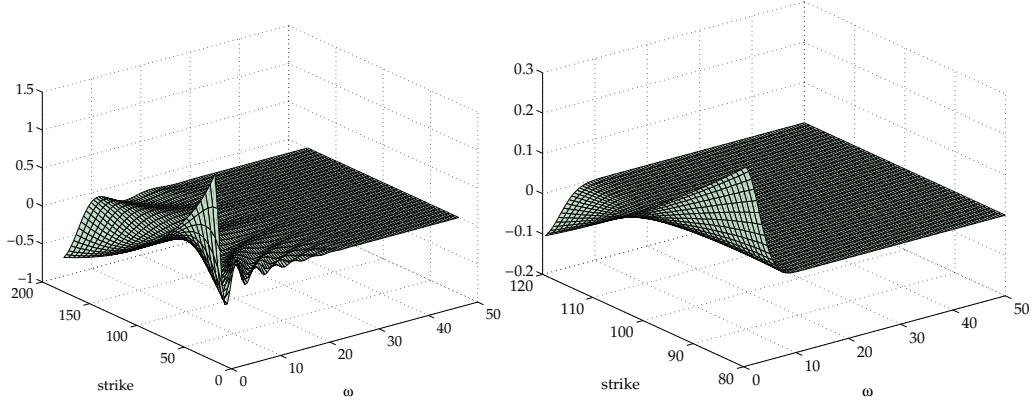the same plot, but shows only the strikes from 80 to 120; here little oscillation is apparent.

Figure 4: Π_1^* for BSM with varying strikes ($S = 100$, $\tau = 1/12$, $\sqrt{v} = 0.3$, $r = 0.03$).

Gauss–Legendre quadrature is applicable to finite intervals, but the integrals in Equations (2) decay so rapidly to zero that we can also evaluate them up to a cutoff point, which we set to 200. Figure 5 shows the relative pricing errors for the BSM case with 20 nodes (left) and 100 nodes (right). Note that here we are already pricing a whole matrix of options (different strikes, different maturities). This matrix is taken from the experiments described in the next section. Already with 100 nodes the pricing errors are in the range of 10^{-13}, ie, practically zero.

The behaviour of the integrals is similar for other characteristic functions. For the Heston model, examples for Π_j^* are given in Figure 6. If we let the volatility-of-volatility go to zero, the functions exactly resemble those of the corresponding BSM case. Figure 7 shows Π_1^* for different strikes, analogously to Figure 4.
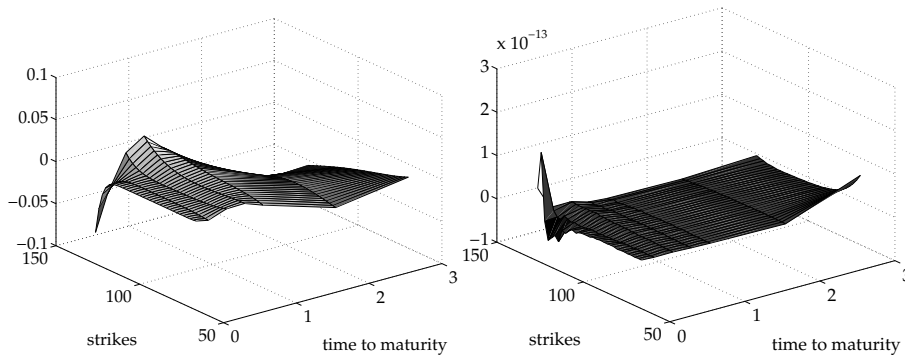
9

Figure 5: Relative pricing errors compared with analytical ʙsᴍ: a Gauss-rule with 20 nodes (left) and 100 nodes (right).

A remark: integration rules like Gauss–Legendre (or others, eg, Clenshaw–Curtis) prescribe to sample the integrand at points that cluster around the endpoints of the interval. This happens because essentially a Gauss rule approximates the function to be integrated by a polynomial, and then integrates this polynomial exactly. Gauss rules are even optimal in the sense that for a given number of nodes, they integrate exactly a polynomial of highest order possible. For an oscillating function, however, we may need a very-high–order polynomial to obtain a good approximation, hence alternative rules may be more efficient for such functions (Hale and Trefethen, 2008). In our experiments this oscillation never caused problems. Furthermore, in Figures 4 and Figures 7 the strikes we computed range from 20 to 180 (with spot at 100). The potentially difficult cases are options with delta zero or delta one, which are not the instruments that are difficult to price.

3 Calibrating model parameters

Calibrating an option pricing model means to find parameters such that the model's prices are consistent with market prices, leading to an optimisation problem of the form

$$\min \sum_{i=1}^{M} \frac{\left| C_i^{\text{model}} - C_i^{\text{market}} \right|}{C_i^{\text{market}}} \tag{13}$$

where M is the number of market prices. Alternatively, we could specify absolute deviations, use squares instead of absolute values, or introduce weighting schemes. The choice of the objective function depends on the application at hand; ultimately, it is an empirical question to determine a good objective function. Since here we are interested in numerical aspects, we will use specification (13). Figure 8 shows an example objective function for the Heston model (on a log scale) when varying two parameters – mean reversion κ and volatility-of-volatility σ – while holding the others fixed. The problem is not convex, and standard methods (eg, based on derivatives of the objective function) may fail. We deploy heuristic methods, Differential Evolution and Particle Swarm Optimisation, to solve problem (13).

When we evaluate (13), we price not just one option, but a whole array of different strikes and different maturities. But for a given set of parameters that describe the
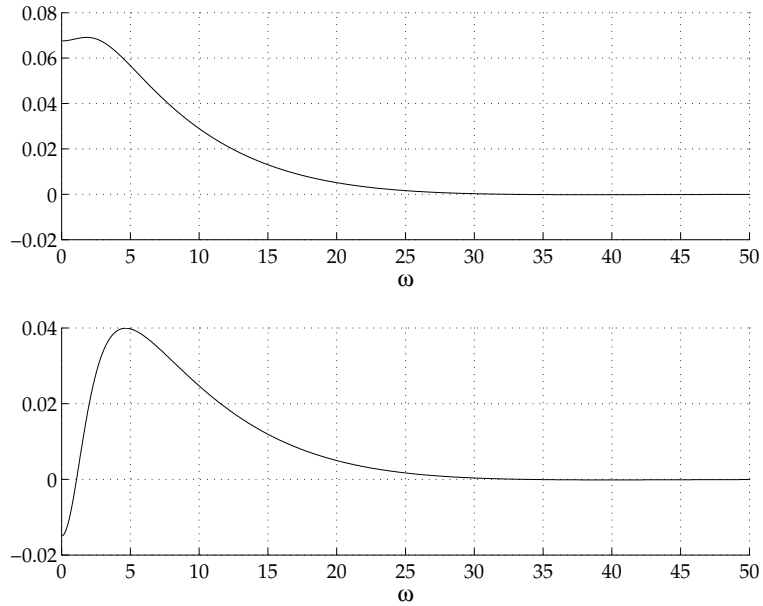
Figure 6: Π_1^* and Π_2^* for Heston model ($S = 100$, $X = 100$, $\tau = 1$, $\sqrt{v_T} = 0.3$, $\sqrt{v_0} = 0.3$, $r = 0.03$, $\kappa = 0.2$, $\sigma = 0.8$, $\rho = -0.5$).

underlying process of the model, the characteristic function ϕ only depends on the time to maturity, not on the strike price. This suggests that speed improvements can be achieved by preprocessing those terms of ϕ that are constant for a given maturity, and then compute the prices for all strikes for this maturity, see Kilin (2007) for a discussion, see Algorithm 1 for a summary.

3.1 Differential Evolution

Differential Evolution (DE) is described in detail in Storn and Price (1997). DE evolves a population of n_P solutions, stored in real-valued vectors of length p (p is five for
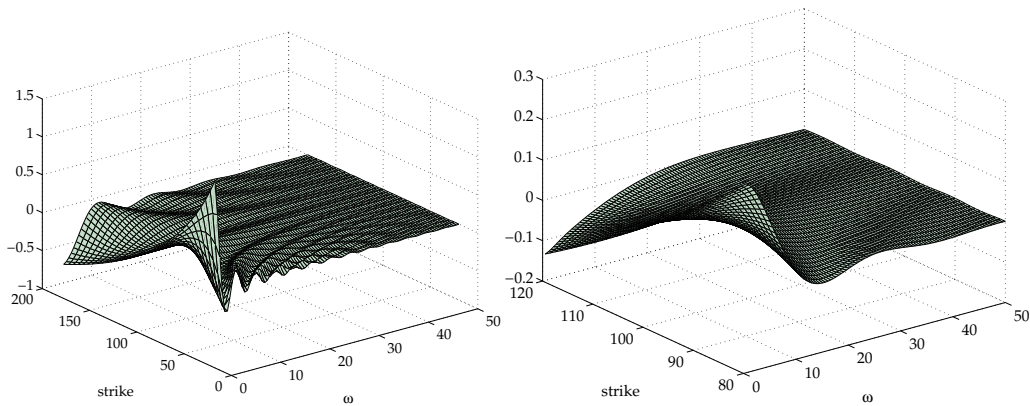

Figure 7: Π_1^* for Heston model with varying strikes ($S = 100$, $\tau = 1/12$, $\sqrt{v_T} = 0.3$, $\sqrt{v_0} = 0.3$, $r = 0.03$, $\kappa = 0.2$, $\sigma = 0.8$, $\rho = -0.5$).
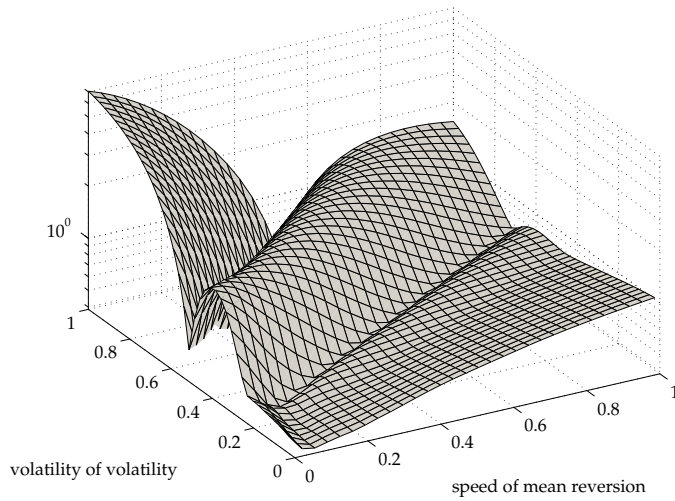
11

Figure 8: A search space for the Heston model.

Algorithm 1 Computing the prices for a given surface.

1: set parameters, set $\mathcal{T}$ = maturities, set $\mathcal{X}$ = strikes
2: **for** $\tau \in \mathcal{T}$ **do**
3:     compute characteristic function ϕ
4:     **for** $X \in \mathcal{X}$ **do**
5:         compute price for strike X, maturity τ
6:     **end for**
7: **end for**
8: compute objective function

Heston, eight for Bates). In every generation k, the algorithm creates n_{P} new candidate solutions; one new solution for each existing one. Such candidate solutions are constructed by taking the difference between two other solutions, weighting this difference, and adding the weighted difference to a third solution. Then an element-wise crossover takes place between the auxiliary solutions $P^{(v)}$ and the original solutions. If such a final candidate solution is better than the original solution, it replaces it; if not, the old solution is kept.

Algorithm 2 summaries the technique. (Notation: n_{G} - number of generations; F - weight parameter; CR - crossover probability; P - population (a matrix of size $p \times n_{\mathrm{P}}$); $\ell_{(.)}$ - integers; F - objective function; ζ - a random variate with uniform distribution on [0 1].)

3.2 Particle Swarm Optimisation

In Particle Swarm Optimisation (PS; Eberhart and Kennedy, 1995), we have again a population that comprises n_{P} solutions, stored in real-valued vectors. In every generation, a solution is updated by adding another vector called velocity v_i. We may think of a solution as a position in the search space, and of velocity as a direction into which the solution is moved. Velocity changes over the course of the optimisation, the magnitude of change is the sum of two components: the direction towards the best

---
Algorithm 2 Differential Evolution.

1: set parameters n_P, n_G, F and CR
2: initialise population $P_{j,i}^{(1)}$, $j = 1, \ldots, p$, $i = 1, \ldots, n_P$
3: **for** $k = 1$ to n_G **do**
4: $\quad P^{(0)} = P^{(1)}$
5: $\quad$ **for** $i = 1$ to n_P **do**
6: $\quad\quad$ generate $\ell_1, \ell_2, \ell_3 \in \{1, \ldots, n_P\}$, $\ell_1 \neq \ell_2 \neq \ell_3 \neq i$
7: $\quad\quad$ compute $P_{\cdot,i}^{(v)} = P_{\cdot,\ell_1}^{(0)} + \text{F} \times (P_{\cdot,\ell_2}^{(0)} - P_{\cdot,\ell_3}^{(0)})$
8: $\quad\quad$ **for** $j = 1$ to p **do**
9: $\quad\quad\quad$ **if** $\zeta < \text{CR}$ **then** $P_{j,i}^{(u)} = P_{j,i}^{(v)}$ **else** $P_{j,i}^{(u)} = P_{j,i}^{(0)}$
10: $\quad\quad$ **end for**
11: $\quad\quad$ **if** $F(P_{\cdot,i}^{(u)}) < F(P_{\cdot,i}^{(0)})$ **then** $P_{\cdot,i}^{(1)} = P_{\cdot,i}^{(u)}$ **else** $P_{\cdot,i}^{(1)} = P_{\cdot,i}^{(0)}$
12: $\quad$ **end for**
13: **end for**
14: find best solution $gbest = \text{argmin}_i \ F(P_{\cdot,i}^{(1)})$
15: solution $= P_{\cdot,gbest}^{(1)}$

---

solution found so far by the particular solution, *Pbest$_i$*, and the direction towards the best solution of the whole population, *Pbest*$_{\text{gbest}}$. These two directions are perturbed via multiplication with a uniform random variable ζ and constants $c_{(\cdot)}$, and summed, see Statement 7. The vector so obtained is added to the previous v_i, the resulting updated velocity is added to the respective solution. In some implementations, the velocities are reduced in every generation by setting the parameter δ, called inertia, to a value smaller than unity.

Algorithm 3 details the procedure. (Notation: n_G - number of generations; P - population (a matrix of size $p \times n_P$); F - objective function; F_i - objective function value associated with the ith solution; ζ - a random variate with uniform distribution on $[0\ 1]$.)

3.3 A simple hybrid

Population-based methods like PS and DE are often effective in exploration: they can quickly identify promising areas of the search space; but then these methods converge only slowly. In the literature we thus often find combinations of population-based search with local search (in the sense of a trajectory method that evolves only a single solution); an example are Memetic Algorithms (Moscato, 1989). We also test a simple hybrid based on this idea; it combines DE and PS with a direct search component. In the classification systems of Talbi (2002) or Winker and Gilli (2004), this is a high-level relay hybrid.

Preliminary tests suggested that the objective function is often flat, thus different parameter values give similar objective function values. This indicates that (i) our problem may be sensitive to small changes in the data when we are interested in precise parameter estimates; and that (ii) if we insist on computing parameters precisely, we may need either many iterations, or an algorithm with a large step size. Thus, as a local search strategy, we use the direct search method of Nelder and Mead (1965) as implemented in Matlab's `fminsearch`. This algorithm can change its step size; it is also robust in case of noisy objective functions (eg, functions evaluated by numerical

---
Algorithm 3 Particle Swarm.

1: set parameters n_P, n_G, δ, c_1 and c_2
2: initialise particles $P_i^{(0)}$ and velocity $v_i^{(0)}$, $i = 1, \ldots, n_P$
3: evaluate objective function $F_i = F(P_i^{(0)})$, $i = 1, \ldots, n_P$
4: *Pbest* $= P^{(0)}$, *Fbest* $= F$, *Gbest* $= \min_i(F_i)$, *gbest* $= \operatorname{argmin}_i(F_i)$
5: **for** $k = 1$ to n_G **do**
6:    **for** $i = 1$ to n_P **do**
7:       $\triangle v_i = c_1 \times \zeta_1 \times (Pbest_i - P_i^{(k-1)}) + c_2 \times \zeta_2 \times (Pbest_{gbest} - P_i^{(k-1)})$
8:       $v_i^{(k)} = \delta v^{(k-1)} + \triangle v_i$
9:       $P_i^{(k)} = P_i^{(k-1)} + v_i^{(k)}$
10:    **end for**
11:    evaluate objective function $F_i = F(P_i^{(k)})$, $i = 1, \ldots, n_P$
12:    **for** $i = 1$ to n_P **do**
13:       **if** $F_i < Fbest_i$ **then** $Pbest_i = P_i^{(k)}$ and $Fbest_i = F_i$
14:       **if** $F_i < Gbest$ **then** $Gbest = F_i$ and $gbest = i$
15:    **end for**
16: **end for**
17: solution $= P_{\cdot, gbest}^{(n_G)}$
---

techniques that may introduce truncation error, as could be the case here). The hybrid is summarised in Algorithm 4.

---
Algorithm 4 Hybrid search.

1: set parameters for population-based method
2: **for** $k = 1$ to n_G **do**
3:    do population-based search
4:    **if** local search **then**
5:       select n_S solutions as starting values for local search
6:       **for** each selected solution **do**
7:          perform local search
8:       **end for**
9:    **end if**
10: **end for**

For an implementation, we need to decide how often we start a local search, how many solutions we select, and how we select them. In the extreme, with just one generation and $n_S = n_P$, we would have a simple restart strategy for the local search method.

3.3.1 Nelder–Mead Search

Spendley et al. (1962) suggested to code a solution x as a simplex. A simplex of dimension p consists of $p + 1$ vertices (points), hence for $p = 1$ we have a line segment; $p = 2$ is a triangle, $p = 3$ is a tetrahedron, and so on. In the algorithm of Spendley et al. (1962), this simplex could be reflected across an edge, or it could shrink. Thus, the size of the simplex could change, but never its form. Nelder and Mead (1965) added two more operations: now the simplex could also expand and contract; hence the simplex could change its size and its form. The possible operations are illustrated in the following figure:

a simplex ($p = 2$) reflection expansion

outside contraction inside contraction shrinking
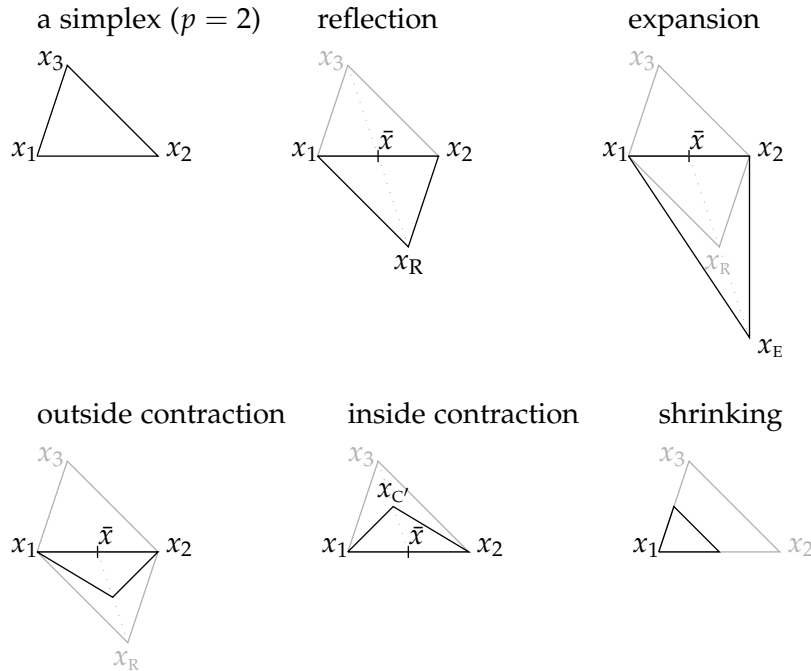
Algorithm 5 outlines the Nelder–Mead algorithm. The notation follows Wright (1996); when solutions are ordered as

$$x_1, \ x_2, \ \ldots, x_{p+1}$$

this means we have

$$F(x_1) < F(x_2) < \ldots < F(x_{p+1}).$$

We denote the objective values associated with particular solutions as $F_1, F_2, \ldots, F_{p+1}$.

Typical values for the parameters in Algorithm 5 are

$$\varrho = 1, \chi = 2, \ \gamma = 1/2, \ \text{and} \ \varsigma = 1/2;$$

these are also used in Matlab's `fminsearch`. Matlab transforms our initial guess x into

$$
\begin{matrix}
x^{(1)} & x^{(1)} + \varepsilon x^{(1)} & x^{(1)} & x^{(1)} & \ldots & x^{(1)} \\
x^{(2)} & x^{(2)} & x^{(2)} + \varepsilon x^{(2)} & x^{(2)} & \ldots & x^{(2)} \\
x^{(3)} & x^{(3)} & x^{(3)} & x^{(1)} + \varepsilon x^{(3)} & \ldots & x^{(3)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
x^{(p)} & x^{(p)} & x^{(p)} & x^{(p)} & \ldots & x^{(p)} + \varepsilon x^{(p)}
\end{matrix}
\tag{14}
$$

where the superscript $^{(i)}$ denotes the ith element of x. In the implementation used here (Matlab 2008a), ε is 0.05. If $x^{(i)}$ is zero, then $\varepsilon x^{(i)}$ is set to 0.00025.

The simplex adjusts to the contours of the objective function (ie, it can 'stretch' itself) and so can make larger steps into favourable directions. But this flexibility can

Algorithm 5 Nelder–Mead search.

1: set $\varrho, \chi, \gamma, \varsigma$
2: **while** stopping criteria not met **do**
3:     order points $x_1, \ldots, x_{p+1}$, compute $\bar{x} = 1/p \sum_{i=1}^{p} x_i$
4:     set *shrink* = false
5:     $x_{\mathrm{R}} = \bar{x} + \varrho(\bar{x} - x_{p+1})$, $F_{\mathrm{R}} = F(x_{\mathrm{R}})$ # reflect
6:     **if** $F_1 \leq F_{\mathrm{R}} < F_p$ **then**
7:         $x^* = x_{\mathrm{R}}$
8:     **else if** $F_{\mathrm{R}} < F_1$ **then**
9:         $x_{\mathrm{E}} = \bar{x} + \chi(x_{\mathrm{R}} - \bar{x})$, $F_{\mathrm{E}} = F(x_{\mathrm{E}})$ # expand
10:         **if** $F_{\mathrm{E}} < F_{\mathrm{R}}$ **then** $x^* = x_{\mathrm{E}}$ **else** $x^* = x_{\mathrm{R}}$
11:     **else if** $F_{\mathrm{R}} \geq F_p$ **then**
12:         **if** $F_p \leq F_{\mathrm{R}} < F_{p+1}$ **then**
13:             $x_{\mathrm{C}} = \bar{x} + \gamma(x_{\mathrm{R}} - \bar{x})$, $F_{\mathrm{C}} = F(x_{\mathrm{C}})$ # outside contract
14:             **if** $F_{\mathrm{C}} \leq F_{\mathrm{R}}$ **then** $x^* = x_{\mathrm{C}}$ **else** *shrink* = true
15:         **else**
16:             $x_{\mathrm{C'}} = \bar{x} - \gamma(\bar{x} - x_{p+1})$, $f_{\mathrm{C'}} = f(x_{\mathrm{C'}})$ # inside contract
17:             **if** $F_{\mathrm{C'}} < F_{p+1}$ **then** $x^* = x_{\mathrm{C'}}$ **else** *shrink* = true
18:         **end if**
19:     **end if**
20:     **if** *shrink* == true **then**
21:         $x_i = x_1 + \varsigma(x_i - x_1)$, $i = 2, \ldots, p+1$
22:     **else**
23:         $x_{p+1} = x^*$
24:     **end if**
25: **end while**

also be a disadvantage: try to visualise a narrow valley along which a long-stretched simplex ⬭ advances. If this valley were to take a turn, the simplex could not easily adapt (see Wright (1996) for a discussion). This phenomenon seems to occur in our problem. When we initialise the simplex, the maximum of a parameter value is 5% greater than its minimum, and this is true for all parameters by construction, see Equation (14). Thus the stretch in relative terms along any dimension is the same. When we run a search and compare this initial with the final simplex, we often find that the stretch along some dimensions is 200 times greater than along other dimensions; the condition number of a simplex often increases from 10^3 or so to 10^{12} and well beyond. This can be a warning sign, and here it is: it turns out that restarting the algorithm, ie, re-initialising the simplex several times, leads to much better solutions.

3.4 Constraints

We constrain all heuristics to favour interpretable values: thus we want non-negative variances, correlation between -1 and 1, and parameters like κ, σ and λ also non-negative. These constraints are implemented through penalty terms. For any violation a positive number proportional to the violation is added to the objective function.

4 Experiments and results

We create artificial data sets to test our techniques. The spot price S_0 is 100, the riskfree rate r is 2%, there are no dividends. We compute prices for strikes X from 80 to 120 in steps of size 2, and maturities τ of $1/12$, $3/12$, $6/12$, $9/12$, 1, 2 and 3 years. Hence our surface comprises $21 \times 7 = 147$ prices. Given a set of parameters, we compute option prices and store them as the true prices. Then we run 10 times each of our methods to solve problem (13) and see if we can recover the parameters; the setup implies that a perfect fit is possible.

The parameters for the Heston model come from the following table:

$\sqrt{v_0}$	0.3	0.3	0.3	0.3	0.4	0.2	0.5	0.6	0.7	0.8
$\sqrt{\theta}$	0.3	0.3	0.2	0.2	0.2	0.4	0.5	0.3	0.3	0.3
ρ	-0.3	-0.7	-0.9	0.0	-0.5	-0.5	0.0	-0.5	-0.5	-0.5
κ	2.0	0.2	3.0	3.0	0.2	0.2	0.5	3.0	2.0	1.0
σ	1.5	1.0	0.5	0.5	0.8	0.8	3.0	1.0	1.0	1.0

For the Bates model, we use the following parameter sets:

$\sqrt{v_0}$	0.3	0.3	0.3	0.3	0.4	0.2	0.5	0.6	0.7	0.8
$\sqrt{\theta}$	0.3	0.3	0.2	0.2	0.2	0.4	0.5	0.3	0.3	0.3
ρ	-0.3	-0.7	-0.9	0.0	-0.5	-0.5	0.0	-0.5	-0.5	-0.5
κ	2.0	0.2	3.0	3.0	0.2	0.2	0.5	3.0	2.0	1.0
σ	0.3	0.5	0.5	0.5	0.8	0.8	1.0	1.0	1.0	1.0
λ	0.1	0.1	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
μ_J	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1	-0.1
σ_J	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

With ten different parameter sets for every model and with ten optimisation runs (restarts) for each parameter set, we have 100 results for each optimisation method. For each restart we store the value for the objective function (the mean percentage error; Equation (13)), and the corresponding parameter estimates. For the latter we compute absolute errors, ie,

$$\text{error } = | \text{ estimated parameter } - \text{ true parameter } | .$$

Below we look at the distributions of these errors.

All algorithms are coded in Matlab, for the direct search we use Matlab's `fminsearch`. We ran a number of preliminary experiments to find effective parameter values for the algorithms. For DE, the F-parameter should be set to around 0.3–0.5 (we use 0.5); very low or high values typically impaired performance. The CR-parameter had less influence, but levels close to unity worked best; each new candidate solution is then likely changed in many dimensions. For PS, the main task is to accelerate convergence. Velocity should not be allowed to become too high, hence inertia should be below unity (we set it to 0.7); we also restricted maximum absolute velocity to 0.2. The stopping criterion for DE and PS is a fixed number of function evaluations (population size $\times$ generations); we run three settings,

$$
\begin{aligned}
1\,250 \quad & (25 \times 50), \\
5\,000 \quad & (50 \times 100), \\
20\,000 \quad & (100 \times 200).
\end{aligned}
$$

On an Intel P8700 single core at 2.53GHz with 2 GB of RAM one run takes about 10, 40, 160 seconds, respectively. (An alternative stopping criterion is to halt the algorithm once the diversity within the population – as measured for instance by the range of objective function or parameter values – falls below a tolerance level. This strategy works fine for DE where the solutions generally converge rapidly, but leads to longer run times for PS.)

For the hybrid methods, we use a population of 25 solutions, and run 50 generations. Every 10 generations, one or three solutions are selected, either the best solutions ('elitists') or random solutions. These solutions are then used as the starting values of a local search. This search comprises repeated applications of Nelder–Mead, restricted to 200 iterations each, until no further improvement can be achieved; 'further improvement' is a decrease in the objective function greater than 0.1% . One run takes between 10 and 30 seconds.

4.1 Goodness-of-fit

We first discuss the achieved objective function values, ie, the average percentage pricing error. Figures 9 to 16 show the empirical distributions of the pricing errors. For the pure population-based approaches, the grey scales indicate the increasing number of function evaluations ($1\,250$, $5\,000$, and $20\,000$).

For the Heston model, all methods converge quickly to very good solutions with pricing errors less than one percent; we can also achieve a perfect fit. (Not reported: we ran further tests for DE and PS with more function evaluations which increased running time to 3–5 minutes; then both algorithms converged on the true solutions without exception. However, for practical purposes, the precision achieved here is sufficient.) DE converges faster than PS. This becomes apparent for the hybrid algorithms. Here, for DE it makes little difference how we select solutions for local search (random or elitists) because all members of the population are similar. For PS, selecting solutions by quality works better, see Figures 11 and 12.

It is more difficult to calibrate the Bates model. Figures 13 to 16 show that convergence is slower here. The hybrid methods perform very well; in particular so when based on DE, or on PS with solutions chosen by quality. For both the Heston and the Bates model, the hybrids performed best, with a small advantage for the DE-based algorithm.

4.2 Parameters

It is also of interest to know how fast the parameter estimates converge to their true values – or if they do. To save space here, we only present results for DE; convergence for other methods looked similar. Figures 17 and 18 show that for the Heston model, the parameters converge roughly in-line with the objective function; see for instance Figure 9. Still, good fits (say, less than one percent pricing error) can be achieved with quite different parameter values.

For the Bates model, the results are worse: Figures 19, 20, and 21 give results for DE. Those parameters that are also in the Heston model are estimated less precisely; but for the jump parameters (λ, μ_J and σ_J) there is essentially no convergence. No convergence in the parameters does not mean we cannot get a good fit; compare Figures 13 to 16 and the discussion below. This non-convergence is to some extent
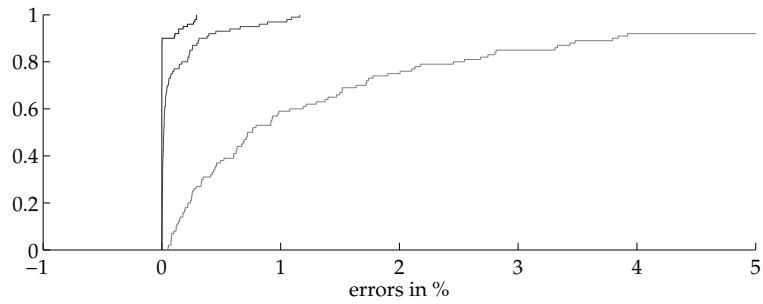
Figure 9: Heston model: Distributions of errors for Differential Evolution. Darker grey indicates more iterations.
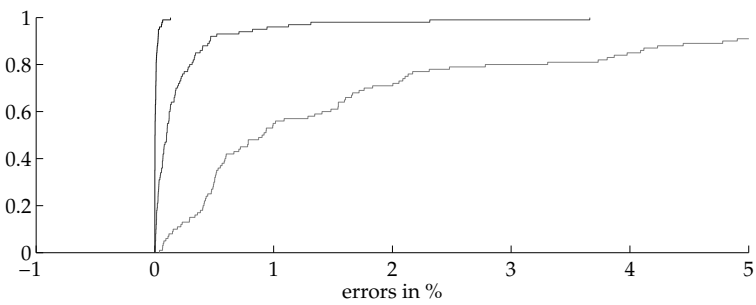

Figure 10: Heston model: Distributions of errors for Particle Swarm Optimisation. Darker grey indicates more iterations.
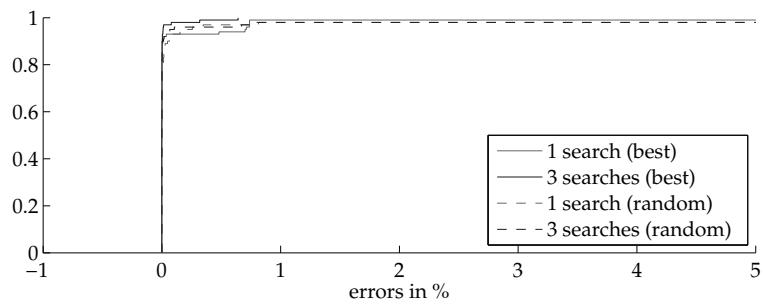

Figure 11: Heston model: Distributions of errors for hybrid based on Differential Evolution.
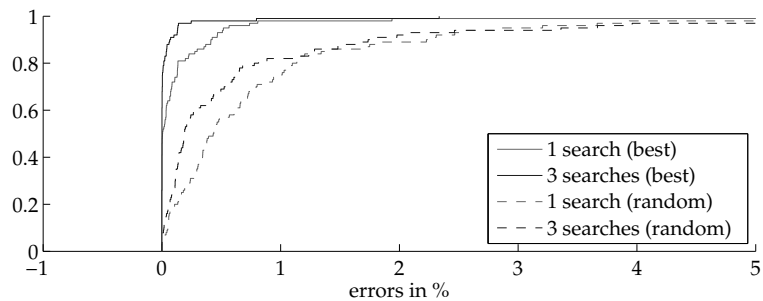

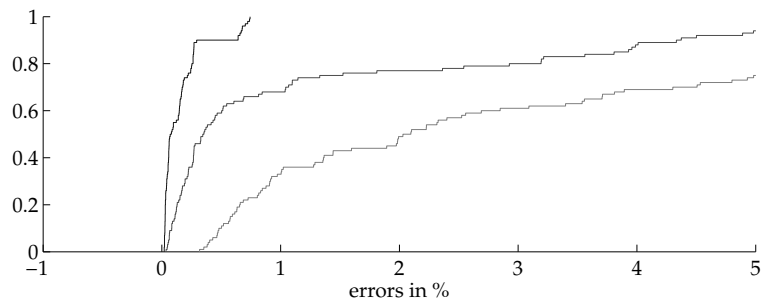Figure 12: Heston model: Distributions of errors for hybrid based on Particle Swarm Optimisation.

Figure 13: Bates model: Distributions of errors for Differential Evolution. Darker grey indicates more iterations.
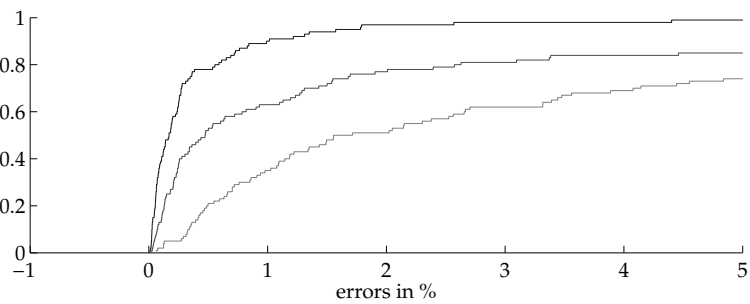

Figure 14: Bates model: Distributions of errors for Particle Swarm Optimisation. Darker grey indicates more iterations.
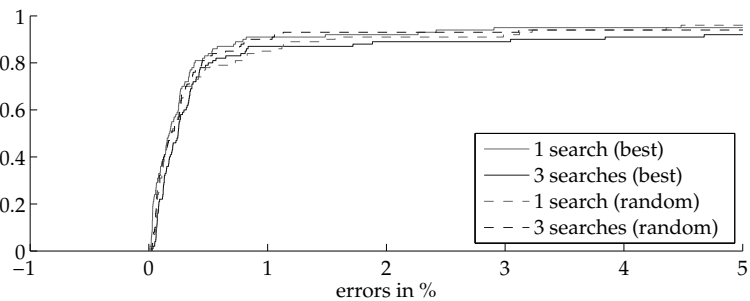

Figure 15: Bates model: Distributions of errors for hybrid based on Differential Evolution.
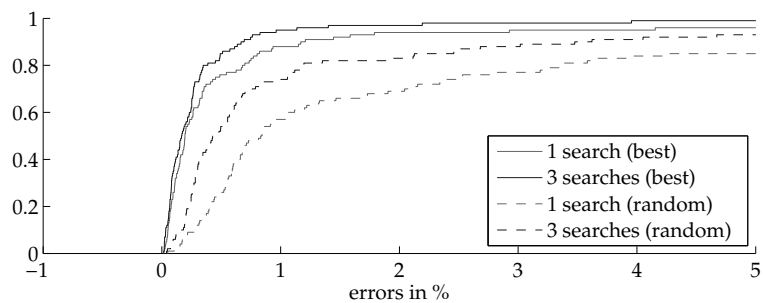

Figure 16: Bates model: Distributions of errors for hybrid based on Particle Swarm Optimisation.
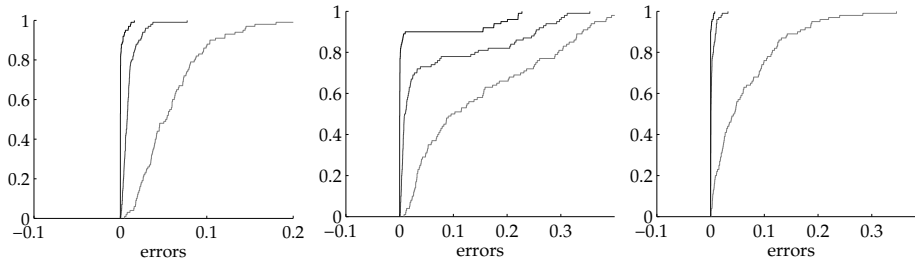
Figure 17: Convergence of parameter estimates for the Heston model with DE.
The figure shows the distributions of absolute errors in the parameters for $\sqrt{v_0}$ (left), $\sqrt{\theta}$ (middle), and ρ (right).
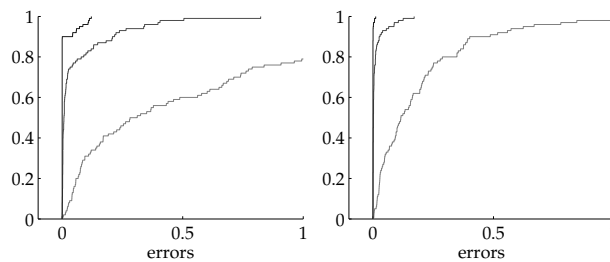

Figure 18: Convergence of parameter estimates for the Heston model with DE.
The figure shows the distributions of absolute errors in the parameters for κ (left), and σ (right).

owed to our choice of parameters. Experiments with Merton's model (not reported) showed that for 'small' mean jumps μ_J of magnitude -10% or -20%, it is difficult to recover parameters precisely because many parameter values give price errors of practically zero. In other words, the optimisation is fine, we can well fit the prices, but we cannot accurately identify the different parameters. In any case, parameter values of the magnitude used here have been reported in the literature (eg, Schoutens et al., 2004, or Detlefsen and Härdle, 2007). The precision improves for large jumps. This is consistent with other studies: He et al. (2006) for instance report relatively-precise estimates for a mean jump size -90%. (At some point, this begs the question how much reason we should impose on the parameters. An advantage of theoretical models over simple interpolatory schemes is the interpretability of parameters. If we can only fit option prices by unrealistic parameters, there is little advantage in using such models.)

The convergence of parameters is also pictured in Figures 22 to 26. The graphics show the parameter errors of a solution compared with the overall price error of the respective solutions. In the middle panel of Figure 22, for example, we see that in the Heston model we can easily have a price error of less than one percent, but a corresponding error in long-run volatility of 0.2 (ie, 20 percentage points). Most remarkable is Figure 26: we see that for μ_J and σ_J, practially any value can be compatible with a low price error.

Figure 19: Convergence of parameter estimates for the Bates model with DE. The figure shows the distributions of absolute errors in the parameters for $\sqrt{v_0}$ (left), $\sqrt{\theta}$ (middle), and ρ (right).

Figure 20: Convergence of parameter estimates for the Bates model with DE. The figure shows the distributions of absolute errors in the parameters for κ (left), σ (middle), and λ (right).
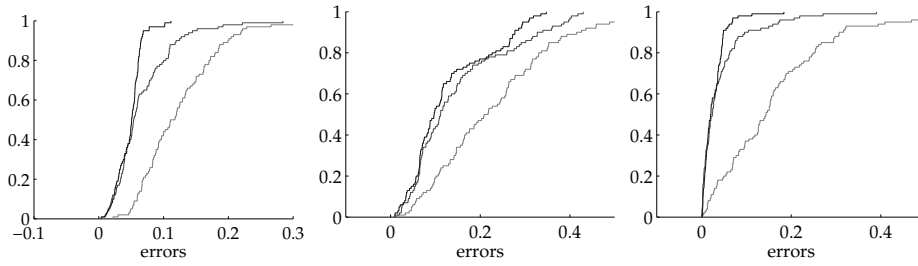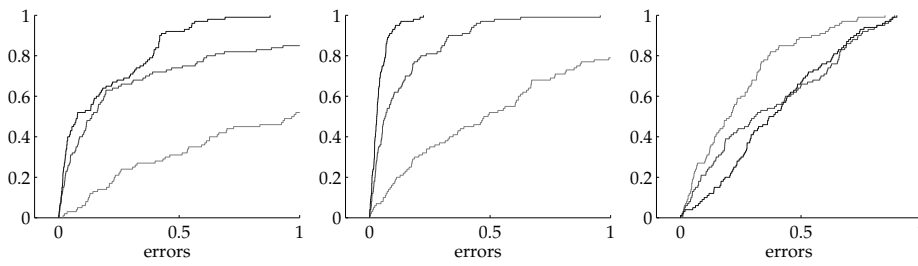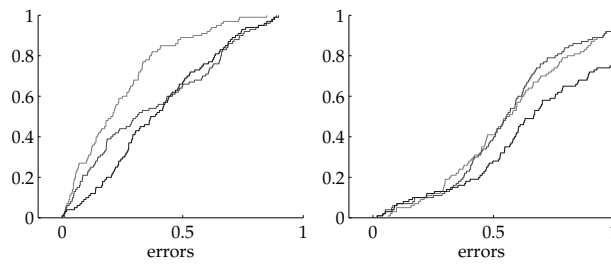

Figure 21: Convergence of parameter estimates for the Bates model with DE. The figure shows the distributions of absolute errors in the parameters for μ_J (left), and σ_J (right).

Figure 22: Goodness of fit vs parameter errors for the Heston model with DE.
The *x*-scale gives the objective function value (average error in percentage points) for solutions.
The *y*-scale shows the associated absolute errors in the parameters for $\sqrt{v_0}$ (left), $\sqrt{\theta}$ (middle), and ρ (right).

Figure 23: Goodness of fit vs parameter errors for the Heston model with DE.
The *x*-scale gives the objective function value (average error in percentage points) for solutions.
The *y*-scale shows the associated absolute errors in the parameters for κ (left), and σ (right).

Figure 24: Goodness of fit vs parameter errors for the Bates model with DE.
The *x*-scale gives the objective function value (average error in percentage points) for solutions.
The *y*-scale shows the associated absolute errors in the parameters for $\sqrt{v_0}$ (left), $\sqrt{\theta}$ (middle), and ρ (right).

Figure 25: Goodness of fit vs parameter errors for the Bates model with DE.
The *x*-scale gives the objective function value (average error in percentage points) for solutions.
The *y*-scale shows the associated absolute errors in the parameters for κ (left), σ (middle), and λ (right).
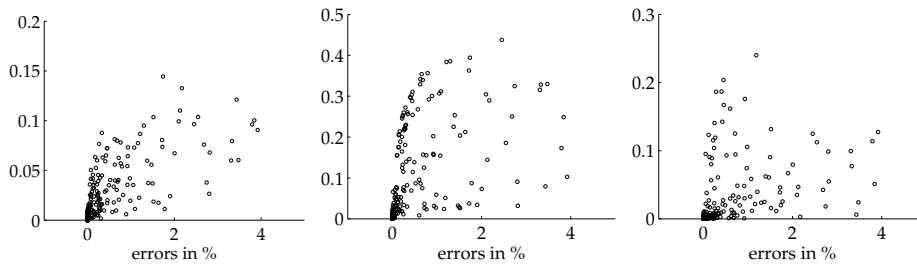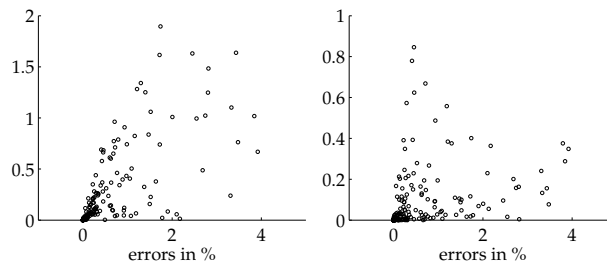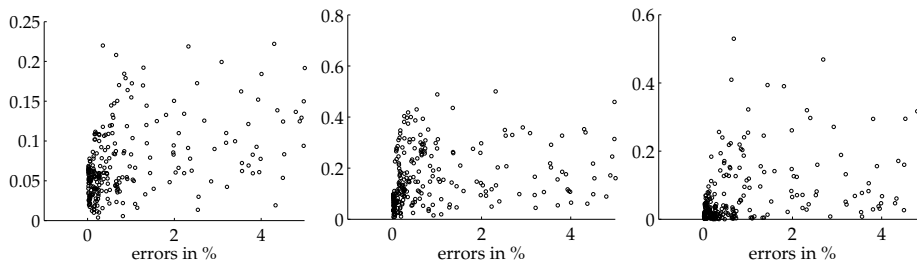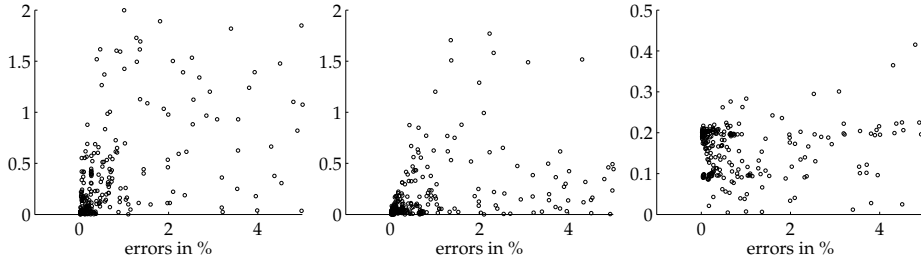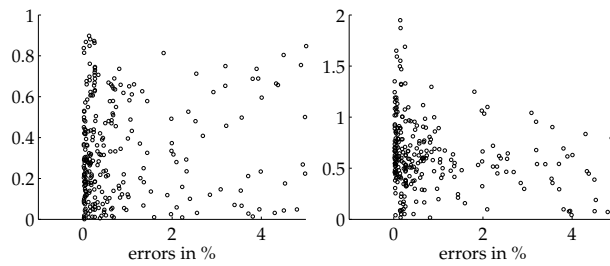

Figure 26: Goodness of fit vs parameter errors for the Bates model with DE.
The *x*-scale gives the objective function value (average error in percentage points) for solutions.
The *y*-scale shows the associated absolute errors in the parameters for μ_J (left), and σ_J (right).

4.3 Conditioning of the problem

We have seen that good results in terms of the objective function – ie, low price errors – can go along with sometimes large errors in the estimated parameters. This can have to do with specific parameter settings as discussed above for Merton's jump diffusion model; it also reflects the fact that both stochastic volatility and jumps can generate the volatility smile (though stochastic volatility models are better at this for long maturities, and jump models are better for short expirations, see Das and Sundaram, 1999). The optimisation procedure hence cannot clearly attribute the smile to either cause. This is an identification problem, a problem of the model, not of the numerical technique.

Our objective function (13) can be rewritten as a system of nonlinear equations

$$\frac{\left|C_i^{\text{model}} - C_i^{\text{market}}\right|}{C_i^{\text{market}}} = 0 \tag{15}$$

where $i \in 1, \ldots, M$. The number of market prices M is greater than the number of parameters, so the system is overidentified; it can only be solved by minimising a norm of the residual. The conditioning of a system of equations does not necessarily affect the size of the residual: even badly-conditioned equations may result in small residuals; but the parameters can then not be accurately determined. This seems to be the case here.

To test the conditioning, we explicitly evaluate the Jacobian matrix $\mathcal{J}$ at every step of the optimisation. $\mathcal{J}$ is a matrix of size $M \times \#\{x_0\}$ where $\#\{x_0\}$ is number of parameters of a solution x_0, h is a small number, and e_j is the jth unit vector. We denote c_0 a vector of length M that stores the relative price errors. Algorithm 6 describes how to approximate the Jacobian of (15) by a forward difference.

Algorithm 6 Computing the Jacobian matrix.

1: set h
2: compute c_0: the left-hand side of Equation (15) for parameters x_0
3: **for** $j = 1$ **to** $\#\{x_0\}$ **do**
4:     compute $x_j = x_0 + he_j$
5:     compute c_j: the left-hand side of Equation (15) for parameters x_j
6:     $\mathcal{J}_{\cdot,j} = (c_j - c_0)/h$
7: **end for**
8: compute condition number of $\mathcal{J}$

We find that mostly the condition number of the models is numerically acceptable for double precision (say, of order 10^5 or 10^6), even though there are steps where the conditioning deteriorates dramatically. An example for the Bates model ($\sqrt{v_0} = 0.3$, $\sqrt{\theta} = 0.3$, $\rho = -0.3$, $\kappa = 2$, $\sigma = 0.3$, $\lambda = 0.10$, $\mu_J = -0.10$, $\sigma_J = 0.10$) is given in Figure 27. For the Heston model, the conditioning is better.

Just because the condition number is numerically acceptable does not mean the model is fine. For intuition: in a linear regression model, the Jacobian is the data matrix. The condition number of this matrix can be acceptable even though high correlation between the columns may prohibit any sensible inference. The following Matlab script sets up a linear regression model with extremely correlated regressors.
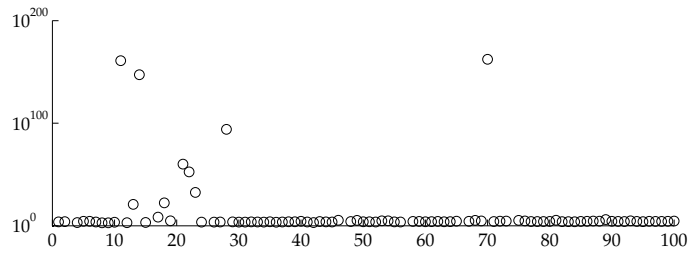
```
% set number of observations, number of regressors
```

Figure 27: Condition number of Jacobian for one solution over 100 generations.

```
nObs = 150; nR = 8;

% set up correlation matrix
C = ones(nR,nR) * 0.9999; C( 1:(nR+1):(nR*nR) ) = 1;

% create data
X = randn(nObs,nR); C = chol(C); X = X*C; bTrue = randn(nR,1);
y = X*bTrue + randn(nObs,1)*0.2;
```

The regression X\y can be computed without numerical problems, though we had better not interpret the coefficients.

The conditioning of the Bates model does not matter much if we just aim to interpolate current option prices; it is a problem if we want to compute parameters accurately from option prices – which we cannot do for this model.

5   Conclusion

In this paper we have investigated the calibration of option pricing models. We have shown how to calibrate the parameters of a model with heuristic techniques, Differential Evolution and Particle Swarm Optimisation, and that we can improve the performance of these methods by adding a Nelder–Mead direct search. While good price fits could be achieved with all methods, the convergence of parameter estimates was much slower; for the jump parameters of the Bates model, there was no convergence. This, it must be stressed, is not a problem of the optimisation technique, but it stems from the model. In comparison, parameters of the Heston model could be estimated more easily.

In empirical studies on option pricing models (eg, Bakshi et al., 1997, or Schoutens et al., 2004), the calibration is often taken 'for granted'; it is rarely discussed whether, for instance, restarts of an optimisation routine with different starting values would have resulted in different parameter estimates, and how such different esimates would have had influenced the studies' results. (In Gilli and Schumann (2010b) we showed that standard gradient-based methods often fail for the kinds of calibration problems discussed in this paper, and that restarts with different starting values can lead to very different solutions.) Different parameter values may lead to good overall fits in terms of prices, but these different parameters may well imply very different Greeks, or have a more marked influence on prices of exotic options. Hence empirical studies that look for example into hedging performance should take into account the sensitivity of their results with respect to calibration. With luck, all that is added is another layer of noise; but the relevance of optimisation is to be investigated by empirical testing, not by conjecturing. Such testing is straightforward: we just need to rerun

our empirical tests many times, each time also rerunning our calibration with alternative starting values, and hence can get an idea of the sensitivity of outcomes with respect to optimisation quality. Ideally, optimisation quality in-sample should be evaluated jointly with empirical, out-of-sample performance of the model, see Gilli and Schumann (2009).

Our findings underline the point raised in Gilli and Schumann (2010a) that modellers in quantitative finance should be sceptical of purely-numerical precision. Model risk is a still under-appreciated aspect in quantitative finance (and one that had better not be handled by rigorous mathematical modelling). For instance, Schoutens et al. (2004) showed that the choice of an option pricing model can have a large impact on the prices of exotic options, even though all models were calibrated to the same market data. (Unfortunately, different calibration criteria lead to different results, see Detlefsen and Härdle, 2007). In the same vein, Jessen and Poulsen (2009) find that different models, when calibrated to plain vanilla options, exhibit widely-differing pricing performance when used to explain actual prices of barrier options. Our results suggest that the lowly numerical optimisation itself can make a difference. How important this difference is needs to be assessed empirically.

A Matlab programs

A.1 Black–Scholes–Merton

Classic formula

```matlab
function call = callBSM(S,X,tau,r,q,sigma)
% callBSM Pricing function for European calls
% callprice = callBSM(S,X,tau,r,q,sigma)
% ---
% S     = spot
% X     = strike
% tau   = time to mat
% r     = riskfree rate
% q     = dividend yield
% sigma = volatility
% ---
% Manfred Gilli and Enrico Schumann, version 2010-02-15
% http://comisef.eu
%
d1 = ( log(S/X) + (r-q+sigma^2/2)*tau ) / (sigma*sqrt(tau));
d2 = d1 - sigma*sqrt(tau);

call = S*exp(-q*tau)*normcdf(d1,0,1) - X*exp(-r*tau)*normcdf(d2,0,1);
```

With the characteristic function

```matlab
function call = callBSMcf(S,X,tau,r,q,vT)
% callBSMcf Pricing function for European calls
% callprice = callBSMcf(S,X,tau,r,q,vT)
% ---
% S     = spot
% X     = strike
% tau   = time to mat
% r     = riskfree rate
% q     = dividend yield
% vT    = variance (volatility squared)
% ---
% Manfred Gilli and Enrico Schumann, version 2010-02-05
% http://comisef.eu
%
vP1 = 0.5 + 1/pi * quad(@P1,0,200,[],[],S,X,tau,r,q,vT);
vP2 = 0.5 + 1/pi * quad(@P2,0,200,[],[],S,X,tau,r,q,vT);
call = exp(-q * tau) * S * vP1 - exp(-r * tau) * X * vP2;
end
%
function p = P1(om,S,X,tau,r,q,vT)
p = real(exp(-1i*log(X)*om) .* cfBSM(om-1i,S,tau,r,q,vT) ./ (1i * om * S * exp((r
    -q) * tau)));
end
%
function p = P2(om,S,X,tau,r,q,vT)
p = real(exp(-1i*log(X)*om) .* cfBSM(om   ,S,tau,r,q,vT) ./ (1i * om));
end
%
function cf = cfBSM(om,S,tau,r,q,vT)
cf = exp(1i * om * log(S) + 1i * tau * (r - q) * om - 0.5 * tau * vT * (1i * om +
    om .^ 2));
end
```

A.2 Merton

Classic formula

```matlab
function call = callMerton(S,X,tau,r,q,sigma,lambda,muJ,vJ,N)
```

28

```
2  % callMerton Pricing function for European calls
3  % callprice = callMerton(S,X,tau,r,q,sigma,lambda,muJ,vJ,N)
4  % ---
5  % S     = spot
6  % X     = strike
7  % tau   = time to mat
8  % r     = riskfree rate
9  % q     = dividend yield
10 % sigma = volatility
11 % lambda= intensity of poisson process
12 % muJ   = mean jump size
13 % vJ    = variance of jump process
14 % N     = number of jumps to be included in sum
15 % ---
16 % Manfred Gilli and Enrico Schumann, version 2010-02-19
17 % http://comisef.eu
18 %
19 lambda2 = lambda*(1+muJ); call = 0;
20 for n=0:N
21     sigma_n = sqrt(sigma^2 + n*vJ/tau);
22     r_n = r - lambda*muJ+ n*log(1+muJ)/tau;
23     call = call + ( exp(-lambda2*tau) * (lambda2*tau)^n ) * ...
24         callBSM(S,X,tau,r_n,q,sigma_n)/ exp( sum(log(1:n)) );
25 end
```

With the characteristic function

```
1  function call = callMertoncf(S,X,tau,r,q,v,lambda,muJ,vJ)
2  % callMertoncf Pricing function for European calls
3  % callprice = callMertoncf(S,X,tau,r,q,v,lambda,muJ,vJ)
4  % ---
5  % S     = spot
6  % X     = strike
7  % tau   = time to mat
8  % r     = riskfree rate
9  % q     = dividend yield
10 % v     = variance (volatility squared)
11 % lambda= intensity of poisson process
12 % muJ   = mean jump size
13 % vJ    = variance of jump process
14 % ---
15 % Manfred Gilli and Enrico Schumann, version 2010-02-19
16 % http://comisef.eu
17 %
18 vP1 = 0.5 + 1/pi * quad(@P1,0,200,[],[],S,X,tau,r,q,v,lambda,muJ,vJ);
19 vP2 = 0.5 + 1/pi * quad(@P2,0,200,[],[],S,X,tau,r,q,v,lambda,muJ,vJ);
20 call = exp(-q * tau) * S * vP1 - exp(-r * tau) * X * vP2;
21 end
22 %
23 function p = P1(om,S,X,tau,r,q,v,lambda,muJ,vJ)
24 p = real(exp(-1i*log(X)*om) .* cfMerton(om-1i,S,tau,r,q,v,lambda,muJ,vJ) ./ (1i *
       om * S * exp((r-q) * tau)));
25 end
26 %
27 function p = P2(om,S,X,tau,r,q,v,lambda,muJ,vJ)
28 p = real(exp(-1i*log(X)*om) .* cfMerton(om  ,S,tau,r,q,v,lambda,muJ,vJ) ./ (1i *
       om));
29 end
30 %
31 function cf = cfMerton(om,S,tau,r,q,v,lambda,muJ,vJ)
32 A = 1i*om*log(S) + 1i*om*tau*(r-q-0.5*v-lambda*muJ) - 0.5*(om.^2)*v*tau;
33 B = lambda*tau*( exp(1i*om*log(1+muJ)-0.5*1i*om*vJ-0.5*vJ*om.^2) -1);
34 cf = exp(A + B);
35 end
```

A.3 Heston

In Matlab

```matlab
function call = callHestoncf(S,X,tau,r,q,v0,vT,rho,k,sigma)
% callHestoncf Pricing function for European calls
% callprice = callHestoncf(S,X,tau,r,q,v0,vT,rho,k,sigma)
% ---
% S    = spot
% X    = strike
% tau  = time to mat
% r    = riskfree rate
% q    = dividend yield
% v0   = initial variance
% vT   = long run variance (theta in Heston's paper)
% rho  = correlation
% k    = speed of mean reversion (kappa in Heston's paper)
% sigma = vol of vol
% ---
% Manfred Gilli and Enrico Schumann, version 2010-02-05
% http://comisef.eu
%
vP1 = 0.5 + 1/pi * quadl(@P1,0,200,[],[],S,X,tau,r,q,v0,vT,rho,k,sigma);
vP2 = 0.5 + 1/pi * quadl(@P2,0,200,[],[],S,X,tau,r,q,v0,vT,rho,k,sigma);
call = exp(-q * tau) * S * vP1 - exp(-r * tau) * X * vP2;
end
%
function p = P1(om,S,X,tau,r,q,v0,vT,rho,k,sigma)
i=1i;
p = real(exp(-i*log(X)*om) .* cfHeston(om-i,S,tau,r,q,v0,vT,rho,k,sigma) ./ (i *
    om * S * exp((r-q) * tau)));
end
%
function p = P2(om,S,X,tau,r,q,v0,vT,rho,k,sigma)
i=1i;
p = real(exp(-i*log(X)*om) .* cfHeston(om   ,S,tau,r,q,v0,vT,rho,k,sigma) ./ (i *
    om));
end
%
function cf = cfHeston(om,S,tau,r,q,v0,vT,rho,k,sigma)
d   = sqrt((rho * sigma * 1i*om - k).^2 + sigma^2 * (1i*om + om .^ 2));
g2  = (k - rho*sigma*1i*om - d) ./ (k - rho*sigma*1i*om + d);
cf1 = 1i*om .* (log(S) + (r - q) * tau);
cf2 = vT * k / (sigma^2) * ((k - rho*sigma*1i*om - d) * tau - 2 * log((1 - g2 .*
    exp(-d * tau)) ./ (1 - g2)));
cf3 = v0 / sigma^2 * (k - rho*sigma*1i*om - d) .* (1 - exp(-d * tau)) ./ (1 - g2
    .* exp(-d * tau));
cf  = exp(cf1 + cf2 + cf3);
end
```

In R

```r
callHestoncf <- function(S,X,tau,r,q,v0,vT,rho,k,sigma)
{
        # callHestoncf Pricing function for European calls
        # ---
        # S    = spot
        # X    = strike
        # tau  = time to mat
        # r    = riskfree rate
        # q    = dividend yield
        # v0   = initial variance
        # vT   = long run variance (theta in Heston's paper)
        # rho  = correlation
        # k    = speed of mean reversion (kappa in Heston's paper)
        # sigma = vol of vol
        # ---
        # Manfred Gilli and Enrico Schumann, version 2010-02-05
```

```
17          # http://comisef.eu
18          #
19
20          # -- functions --
21          P1 <- function(om,S,X,tau,r,q,v0,vT,rho,k,sigma)
22          {
23                  i <- 1i
24                  p <- Re(exp(-i*log(X)*om) * cfHeston(om-i,S,tau,r,q,v0,vT,rho,k,
                        sigma) / (i * om * S * exp((r-q) * tau)))
25                  return(p)
26          }
27          P2 <- function(om,S,X,tau,r,q,v0,vT,rho,k,sigma)
28          {
29                  i <- 1i
30                  p <- Re(exp(-i*log(X)*om) * cfHeston(om  ,S,tau,r,q,v0,vT,rho,k,
                        sigma) / (i * om))
31                  return(p)
32          }
33          cfHeston <- function(om,S,tau,r,q,v0,vT,rho,k,sigma)
34          {
35                  d   <- sqrt((rho * sigma * 1i*om - k)^2 + sigma^2 * (1i*om + om ^
                         2))
36                  g2  <- (k - rho*sigma*1i*om - d) / (k - rho*sigma*1i*om + d)
37                  cf1 <- 1i*om * (log(S) + (r - q) * tau)
38                  cf2 <- vT * k / (sigma^2) * ((k - rho*sigma*1i*om - d) * tau - 2
                        * log((1 - g2 * exp(-d * tau)) / (1 - g2)))
39                  cf3 <- v0 / sigma^2 * (k - rho*sigma*1i*om - d) * (1 - exp(-d *
                        tau)) / (1 - g2 * exp(-d * tau))
40                  cf  <- exp(cf1 + cf2 + cf3)
41                  return(cf)
42          }
43          # -- pricing --
44          vP1 <- 0.5 + 1/pi * integrate(P1,lower=0,upper=200,S,X,tau,r,q,v0,vT,rho,
                k,sigma)$value
45          vP2 <- 0.5 + 1/pi * integrate(P2,lower=0,upper=200,S,X,tau,r,q,v0,vT,rho,
                k,sigma)$value
46          call <- exp(-q * tau) * S * vP1 - exp(-r * tau) * X * vP2;
47          return(call)
48  }
```

A.4   Bates

In Matlab

```
1  function call = callBatescf(S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ,vJ)
2  % callBatescf Pricing function for European calls
3  % callprice = callBatescf(S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ,vJ)
4  % ---
5  % S      = spot
6  % X      = strike
7  % tau    = time to mat
8  % r      = riskfree rate
9  % q      = dividend yield
10 % v0     = initial variance
11 % vT     = long run variance (theta in Heston's paper)
12 % rho    = correlation
13 % k      = speed of mean reversion (kappa in Heston's paper)
14 % sigma  = vol of vol
15 % lambda = intensity of jumps;
16 % muJ    = mean of jumps;
17 % vJ     = variance of jumps;
18 % ---
19 % Manfred Gilli and Enrico Schumann, version 2010-02-05
20 % http://comisef.eu
21 %
```

```matlab
22  vP1 = 0.5 + 1/pi * quadl(@P1,0,200,[],[],S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ
       ,vJ);
23  vP2 = 0.5 + 1/pi * quadl(@P2,0,200,[],[],S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ
       ,vJ);
24  call = exp(-q * tau) * S * vP1 - exp(-r * tau) * X * vP2;
25  end
26  %
27  function p = P1(om,S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ,vJ)
28  i=1i;
29  p = real(exp(-i*log(X)*om) .* cfBates(om-i,S,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ
       ,vJ) ./ (i * om * S * exp((r-q) * tau)));
30  end
31  %
32  function p = P2(om,S,X,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ,vJ)
33  i=1i;
34  p = real(exp(-i*log(X)*om) .* cfBates(om  ,S,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ
       ,vJ) ./ (i * om));
35  end
36  %
37  function cf = cfBates(om,S,tau,r,q,v0,vT,rho,k,sigma,lambda,muJ,vJ)
38  d   = sqrt((rho * sigma * 1i*om - k).^2 + sigma^2 * (1i*om + om .^ 2));
39  %
40  g2  = (k - rho*sigma*1i*om - d) ./ (k - rho*sigma*1i*om + d);
41  %
42  cf1 = 1i*om .* (log(S) + (r - q) * tau);
43  cf2 = vT * k / (sigma^2) * ((k - rho*sigma*1i*om - d) * tau - 2 * log((1 - g2 .*
       exp(-d * tau)) ./ (1 - g2)));
44  cf3 = v0 / sigma^2 * (k - rho*sigma*1i*om - d) .* (1 - exp(-d * tau)) ./ (1 - g2
       .* exp(-d * tau));
45  % jump
46  cf4 = -lambda*muJ*1i*tau*om + lambda*tau*(  (1+muJ).^(1i*om) .* exp( vJ*(1i*om/2)
       .* (1i*om-1) )-1  );
47  cf  = exp(cf1 + cf2 + cf3 + cf4);
48  end
```

B Numerical integration

In this appendix we give a brief introduction to numerical integration. For a textbook exposition see for instance Heath (2005, ch. 8). Davis and Rabinowitz (2007) gives a detailed discussion. A highly-recommended paper is Trefethen (2008); it is one of the rare occasions where actual convergence – as opposed to theoretical optimality – is discussed for specific rules.

The essence of numerical integration, or quadrature, is to replace an integral

$$\int_a^b f(x)\mathrm{d}x \tag{16}$$

by the sum

$$\sum_{i=1}^n w_i f(x_i)\,. \tag{17}$$

The x_i are called the nodes or abscissas, the w_i are weights. We either assume there are n nodes, or that the interval $[a,b]$ is subdivided into m partitions. Quadrature rules detail how to choose these nodes and weights. A rule is called closed if it requires to evaluate the endpoints a and b; otherwise the rule is called open.

An intuitive approach is to follow Riemann's original idea and replace the integral (16) by the sum of the area of m rectangles. Such a Riemann sum is defined as follows. Assume

$$a = x_1 < x_2 < \ldots < x_m < x_{m+1} = b\,,$$

then any collection of nodes $v_k \in [x_k, x_{k+1}]$, for $k = 1, \ldots, m$, defines a Riemann sum

$$\sum_{k=1}^m (x_{k+1} - x_k) f(v_k)\,. \tag{18}$$

We define $h \equiv (b-a)/m$; then some possible quadrature rules based on Riemann sums are:

rectangular rule R $R_m = h \sum_{k=0}^{m-1} f(a+kh)$ (evaluation on the left side), or
$R_m = h \sum_{k=1}^{m} f(a+kh)$ (evaluation on the right side).

midpoint rule M We evaluate the rectangle in the middle, hence
$M_m = h \sum_{k=0}^m f\left(a + \left[k + \frac{1}{2}\right] h\right)$.

trapezoidal rule T $T_m = h\left(\sum_{k=1}^{m-1} f(a+kh) + \frac{f(a)}{2} + \frac{f(b)}{2}\right)$
for $m \geq 2$; or
$T_m = h\left(\frac{f(a)+f(b)}{2}\right)$ for $m = 1$.

The following code shows how to implement such rules in Matlab. We include a call to Matlab's quad function.

```matlab
% Riemann sums - example
Fun1 = @(x)(exp(-x));
m = 5; a = 0; b = 5; h = (b-a)/m;


% rectangular rule -- left
w = h; k = 0:(m-1); x = a + k * h;
fprintf('rectangular (left)  with %i rectangles:\t %f\n',m,sum(w * Fun1(x)))

% rectangular rule -- right
w = h; k = 1:m; x = a + k * h;
fprintf('rectangular (right) with %i rectangles:\t %f\n',m,sum(w * Fun1(x)))

%midpoint rule
w = h; k = 0:(m-1); x = a + (k + 0.5)*h;
fprintf('midpoint           with %i rectangles:\t %f\n',m,sum(w * Fun1(x)))

%trapezoidal rule
w = h; k = 1:(m-1); x = [a a + k*h b];
aux = w * Fun1(x); aux([1 end]) = aux([1 end])/2;
fprintf('trapezoidal        with %i rectangles:\t %f\n',m,sum(aux))

%adaptive Simpson
fprintf('Adaptive Simpson (Matlab):\t\t\t\t %f\n',quad(Fun1,a,b))
```

Interpolatory rules

The three rules stated above partition the interval $[a\ b]$ arbitrarily into subintervals and approximate the integrand in each subinterval by a rectangle or a trapezium. The accuracy of this approach improves as the number of subintervals increases. But rectangles or trapezia may not be natural candidates to approximate a function; if the function is smooth we can do better. Given $n + 1$ nodes, we can fit a polynomial of order n that interpolates the function values at these nodes. This polynomial can then be integrated exactly as an approximation of the true integral. This approach is equivalent to setting the weights w such that the monomials x^0, x^1, ..., x^n are integrated exactly; see Davis and Rabinowitz (2007, ch. 2) for a proof. For equidistant nodes the resulting quadrature schemes are called Newton–Cotes rules.

Assume we wish to determine a k-point Newton–Cotes rule: we fix $x_1, x_2, \ldots, x_k$, then choose the $w_1, w_2, \ldots, w_k$ such that the resulting rule integrates the polynomials $x^0 = 1, x^1, \ldots, x^{k-1}$ exactly on the interval $[a, b]$. We obtain

$$w_1 \cdot 1 + w_2 \cdot 1 + w_3 \cdot 1 + \ldots + w_k \cdot 1 = \int_a^b 1 \, dx = b - a$$

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_k x_k = \int_a^b x \, dx = \frac{1}{2}(b - a)^2 \tag{19}$$

$$w_1 x_1^2 + w_2 x_2^2 + w_3 x_3^2 + \ldots + w_k x_k^2 = \int_a^b x^2 \, dx = \frac{1}{3}(b - a)^3$$

$$\vdots$$

$$w_1 x_1^{k-1} + w_2 x_2^{k-1} + w_3 x_3^{k-1} + \ldots + w_k x_k^{k-1} = \int_a^b x^{k-1} \, dx = \frac{1}{k}(b - a)^k.$$

This can be rewritten conveniently as

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{k-1} & x_2^{k-1} & x_3^{k-1} & \dots & x_k^{k-1} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{pmatrix} = \begin{pmatrix} b-a \\ \frac{1}{2}(b-a)^2 \\ \vdots \\ \frac{1}{k}(b-a)^k \end{pmatrix} \tag{20}$$

and then solved for w. (Compare this with the rules based on Riemann sums: for equidistant nodes, all function values were equally weighted there.) We define $h = (b-a)/m = (b-a)/(n-1)$; then we have the following closed rules:

n	name	x	w
2	trapezoidal rule	a, b	$\frac{1}{2}h$, $\frac{1}{2}h$
3	Simpson's rule	a, $a+h$, b	$\frac{1}{3}h$, $\frac{4}{3}h$, $\frac{1}{3}h$
4	Simpson's 3/8-rule	a, $a+h$, $a+2h$, b	$\frac{3}{8}h$, $\frac{9}{8}h$, $\frac{9}{8}h$, $\frac{3}{8}h$
5	Boole's rule	a, $a+h$, $a+2h$, $a+3h$, b	$\frac{14}{45}h$, $\frac{64}{45}h$, $\frac{24}{45}h$, $\frac{64}{8}h$, $\frac{14}{45}h$

Newton–Cotes rules of very high order are rarely used, though, since convergence is not guaranteed for $n \to \infty$, and Equations (20) become ever more badly conditioned as n increases. Instead, the interval of integration is subdivided into smaller subintervals, and to each a low-order rule is applied. Such an implementation is called a composite (or compound) rule.

The reasoning of Equations (19) can be taken one step further by also freely choosing the x_i: this will leave us $2n$ variables, the w and the x. Choosing them such that they integrate x^0, x^1, $x^2, \dots, x^{2n-1}$ exactly leads to Gauss rules. In principle, we could use the approach (19), but this leads to non-linear equations that are much harder to solve. Fortunately, nodes and weights can also be computed in alternative ways, for instance as the zeros of certain polynomials.

Finding the nodes for Gauss rules

Let φ_0, φ_1, $\varphi_2, \dots, \varphi_n$ be a sequence of orthogonal polynomials (the subscript indicates the order), ie,

$$\int_a^b \varphi_i(x)\varphi_j(x)\omega(x)\mathrm{d}x = 0 \text{ , for all } i \neq j.$$

Orthogonality holds with respect to a weight function ω, and for an interval $[a\,b]$. The zeros of $\varphi_n(x)$, that is, the x that satisfy $\varphi_n(x) = 0$, are the nodes of an n-point Gauss rule for the interval $[a\,b]$. If the sequence is also normalised, so that we have

$$\int_a^b \varphi_i(x)\varphi_i(x)\omega(x)\mathrm{d}x = 1,$$

we call the polynomials orthonormal. For orthogonal polynomials, the following three-term recurrence holds:

$$\varphi_n(x) = (\alpha_n x + \beta_n)\varphi_{n-1}(x) - \gamma_n \varphi_{n-2}(x), \quad n \geq 1. \tag{21}$$

α_n, β_n and γ_n are functions of the coefficients of the polynomials. For normalised polynomials, γ_n is equal to α_n/α_{n-1}, We can rearrange (21) to

$$x\varphi_{n-1}(x) = \frac{1}{\alpha_n}\varphi_n(x) + \underbrace{\left(-\frac{\beta_n}{\alpha_n}\right)}_{\delta_n}\varphi_{n-1}(x) + \frac{1}{\alpha_{n-1}}\varphi_{n-2}(x) \tag{22}$$

and put it into matrix notation (Wilf, 1978), where $\varphi_n(x) \equiv 0$ for $n < 0$:

$$x\underbrace{\begin{bmatrix} \varphi_0(x) \\ \varphi_1(x) \\ \varphi_2(x) \\ \vdots \\ \varphi_{n-1}(x) \end{bmatrix}}_{\Phi(x)} = \underbrace{\begin{bmatrix} \delta_1 & \frac{1}{\alpha_1} & 0 & 0 & \dots & 0 \\ \frac{1}{\alpha_1} & \delta_2 & \frac{1}{\alpha_2} & 0 & \dots & 0 \\ 0 & \frac{1}{\alpha_2} & \delta_3 & \frac{1}{\alpha_3} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \frac{1}{\alpha_{n-1}} & \delta_n \end{bmatrix}}_{A}\underbrace{\begin{bmatrix} \varphi_0(x) \\ \varphi_1(x) \\ \varphi_2(x) \\ \vdots \\ \varphi_{n-1}(x) \end{bmatrix}}_{\Phi(x)} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \frac{1}{\alpha_n}\varphi_n(x) \end{bmatrix} \tag{23}$$

The α and δ terms relate to Equation (22). Now assume we insert x^* into (23), and x^* is a zero of φ_n. Then the last term in (23) vanishes, and we are left with

$$x^*\Phi(x^*) = A\Phi(x^*)\,.$$

This equation can only hold if x^* is an eigenvalue of A: hence the zeros of φ_n, and thus the nodes of an n-point Gauss rule, are the eigenvalues of A. Having computed the nodes, we could compute weights with Equations (19) and (20); but the weights can also be obtained from the eigenvectors of A, see Wilf (1978); Golub and Welsch (1969). More specifically, the weight corresponding to an eigenvalue/node is given by

$$q_1^2 \int_a^b \omega(x)\mathrm{d}x$$

where q_1 is the first element of the eigenvector that belongs to the particular eigenvalue.

For many polynomials, the α, β and γ from Equations (21) are known, and hence A can be set up. For instance, the Legendre polynomials P_n (used in this paper) have weight function $\omega(x) \equiv 1$ and are defined on the interval $[-1\ 1]$. For the normalised polynomials we have $\delta_n = 0$, and $\alpha_n = (4 - \frac{1}{n^2})^{0.5}$. In Matlab:

```
n = 6;                 % number of nodes
aux = 1./sqrt(4-(1:(n-1)).^(-2));
A = diag(aux,1)+diag(aux,-1);
[V,D] = eig(A);
x = diag(D);
[x,i] = sort(x);       % Matlab does not guaranty sorted eigenvalues
w = (2*V(1,i).^2);     % Legendre: w(x)=1; integral from -1 to 1 = 2
```

A Gauss rule for an interval $[a_0\ b_0]$ can be transferred to an interval $[a\ b]$ as follows (Heath, 2005, pp. 352–353):

$$x' = \frac{(b-a)x + ab_0 - ba_0}{b_0 - a_0}$$

$$w' = \frac{b-a}{b_0 - a_0}w$$

```
1  % change interval of integration
2  a0 = -1; b0 =  1;                      % interval for Gauss rule
3  x = ((b-a)*x + a*b0-b*a0)/(b0-a0);
4  w = w * (b-a)/(b0-a0);
5  fprintf('Gauss-Legendre  with %i rectangles:\t %f\n',m,w * Fun1(x))
```

References

Hansjörg Albrecher, Philipp Mayer, Wim Schoutens, and Jurgen Tistaert. The Little Heston Trap. *Wilmott*, pages 83–92, January 2007.

Gurdip Bakshi and Dilip B. Madan. Spanning and Derivative-Security Valuation. *Journal of Financial Economics*, 55(2):205–238, 2000.

Gurdip Bakshi, Charles Cao, and Zhiwu Chen. Empirical Performance of Alternative Option Pricing Models. *Journal of Finance*, 52(5):2003–2049, December 1997.

David S. Bates. Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options. *Review of Financial Studies*, 9(1):69–107, 1996.

Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

Peter Carr and Dilip B. Madan. Option Valuation Using the Fast Fourier Transform. *Journal of Computational Finance*, 2(4):61–73, 1999.

Sanjiv Ranjan Das and Rangarajan K. Sundaram. Of Smiles and Smirks: A Term Structure Perspective. *The Journal of Financial and Quantitative Analysis*, 34(2):211–239, 1999.

Philip J. Davis and Philip Rabinowitz. *Methods of Numerical Integration*. Dover, 2nd edition, 2007.

Emanuel Derman and Iraj Kani. The volatility smile and its implied tree. *Goldman Sachs Quantitative Strategies Research Notes*, January 1994.

Kai Detlefsen and Wolfgang K. Härdle. Calibration Risk for Exotic Options. *Journal of Derivatives*, 14(4):47–63, 2007.

Bruno Dupire. Pricing with a Smile. *Risk*, 7(1):18–20, 1994.

Russell C. Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micromachine and Human Science*, pages 39–43, Nagoya, Japan, 1995.

Jim Gatheral. *The Volatility Surface*. Wiley, 2006.

Manfred Gilli and Enrico Schumann. Optimal enough? *COMISEF Working Paper Series No. 10*, 2009.

Manfred Gilli and Enrico Schumann. Optimization in Financial Engineering – An essay on 'good' solutions and misplaced exactitude. *Journal of Financial Transformation*, in press, 2010a.

Manfred Gilli and Enrico Schumann. Calibrating the Heston Model with Differential Evolution. In Cecilia Di Chio et al., editor, *EvoApplications 2010, Part II*, number 6025 in Lecture Notes in Computer Science, pages 242–250. Springer, 2010b.

Gene H. Golub and John H. Welsch. Calculation of Gauss Quadrature Rules. *Mathematics of Computation*, 23(106):221–230+s1–s10, 1969.

Nicholas Hale and Lloyd N. Trefethen. New quadrature formulas from conformal maps. *SIAM Journal of Numerical Analysis*, 46(2):930–948, 2008.

C. He, J.S. Kennedy, Thomas F. Coleman, Peter A. Forsyth, Y. Li, and K.R. Vetzal. Calibration and Hedging under Jump Diffusion. *Review of Derivatives Research*, 9(1): 1–35, 2006.

Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, 2nd edition, 2005.

Steven L. Heston. A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bonds and Currency options. *Review of Financial Studies*, 6(2):327–343, 1993.

Cathrine Jessen and Rolf Poulsen. Empirical Performance of Models for Barrier Option Valuation. Technical report, University of Copenhagen, 2009. URL `http://www.math.ku.dk/~rolf/papers.html`.

Fiodar Kilin. Accelerating the Calibration of Stochastic Volatility Models. *Centre for Practical Quantitative Finance Working Paper Series No. 6*, 2007.

Robert C. Merton. Option Pricing when Underlying Stock Returns are Discontinuous. *Journal of Financial Economics*, 3(1–2):125–144, 1976.

Pablo Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts – Towards Memetic Algorithms. Technical Report 790, CalTech California Institute of Technology, 1989.

John A. Nelder and Roger Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7(4):308–313, 1965.

Wim Schoutens. *Lévy Processes in Finance: Pricing Financial Derivatives*. Wiley, 2003.

Wim Schoutens, Erwin Simons, and Jurgen Tistaert. A Perfect Calibration! Now What? *Wilmott*, March 2004.

W. Spendley, G.R. Hext, , and F.R. Himsworth. Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation. *Technometrics*, 4(4):441–461, 1962.

Rainer M. Storn and Kenneth V. Price. Differential Evolution – a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

El-Ghazali Talbi. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics*, 8(5): 541–564, 2002.

Lloyd N. Trefethen. Is Gauss Quadrature Better than Clenshaw–Curtis? *SIAM Review*, 50(1):67–87, 2008.

Herbert S. Wilf. *Mathematics for the Physical Sciences*. Dover, 1978.

Peter Winker and Manfred Gilli. Applications of optimization heuristics to estimation and modelling problems. *Computational Statistics and Data Analysis*, 47:211–223, 2004.

Margaret H. Wright. Direct search methods: Once scorned, now respectable. In D.F. Griffiths and G.A. Watson, editors, *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis*, pages 191–208. Addison Wesley Longman, 1996.