The Bates volatility model

Maya Briani Roberto Natalini Marco Papi
Francesco Ferreri

Premia 14

1 Introduction

In the following we shall describe finite difference approximations for the Bates volatility model. We shall recall some financial backgrounds, we shall introduce the finite difference approximation and the setup for numerical tests. Some tests will be also described.

2 The Bates volatility model

Merton's and Heston's approaches were combined by Bates in 1996 [1], who proposed a stock price model with stochastic volatility and jumps:

$$\frac{dS_t}{S_t} = (r-\delta)dt + \sqrt{V_t}dW_t^{(1)} + dZ_t$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t}dW_t^{(2)}.$$

where r is the spot interest rate, δ is the dividend paid by the asset S, V is the value of the spot volatility, θ is the long-run volatility, σ is the volatility of volatility (vol-vol) and W^1, W^2 two stochastic processes correlated by ρ. Z_t is a compound Poisson process with intensity λ and independent jumps J with

$$\log(1+J) \sim N(\log(1+\chi) - \frac{1}{2}\alpha^2, \alpha^2).$$

The parameters χ and α determine the distribution of the jumps and the Poisson process is assumed to be independent of the Wiener processes.

Assuming that the previous dynamics represent the evolution of the state process (S_t, V_t) under a risk-neutral measure, then the pricing equation of a European contingent claim C on S is the following:

$$\begin{aligned}&\frac{\partial C}{\partial t} + \frac{1}{2}VS^2\frac{\partial^2 C}{\partial S^2} + \frac{1}{2}\sigma^2 V\frac{\partial^2 C}{\partial V^2} + \rho\theta VS\frac{\partial^2 C}{\partial S\partial V} + (r-\delta-\lambda\bar{k})S\frac{\partial C}{\partial S}\\ &+\kappa(\theta - V)\frac{\partial C}{\partial V} - rC + \mathcal{I}C = 0,\end{aligned} \tag{1}$$

where

$$\begin{aligned}
\mathcal{I}C(S,V,t) &= \lambda \int_0^{+\infty} [C(S\xi,V,t) - C(S,V,t)]q(\xi)d\xi,\\
q(\xi) &= \frac{1}{\sqrt{2\pi}\alpha\xi} e^{-\frac{1}{2\alpha^2}(\log(\xi)-m)^2}\\
m &:= \log(1+\chi) - \frac{1}{2}\alpha^2,\\
\bar{k} &:= e^{\alpha^2/2+m} - 1.
\end{aligned}$$

The equation holds for $S \in [0,+\infty)$. For the V variables, we assume that $V \in [0, V_{max}]$.
The PDE is usually given along with a proper payoff. In the case of vanilla European call option we have,

$$C(S,V) = \max\left[S-K,0\right], \tag{2}$$

where K is the strike price. It should be noted that closed-form solutions of problem (1) for vanilla-option payoff do exist, methods for integrating such solutions can be found in [4]. Nevertheless, direct numerical integration of (1) is important when dealing with non-trivial payoff functions.

2.1 Numerical approximation

We want to solve problem (1). Following [5], we convert the problem defined on an infinite domain into one defined on a finite domain, by using the following transformation:

$$x = \frac{S}{S+S_0},\ y = V,\ \tau = T-t,\ u(x,y,\tau) = \frac{C(S,V,t)}{S+S_0}, \tag{3}$$

where $x \in [0,1)$, $y \in [0, V_{max}]$, $\tau \in [0,T]$ and S_0 is the current asset price.
Since this transformation converts a point $S \in [0,+\infty)$ into a point $x \in [0,1)$, $u = u(x,y,t)$ is defined on the domain $(0,1)\times(0,y_{max})\times(0,T]$ and the Cauchy problem (1) can be rewritten as

$$\begin{aligned}
&\frac{1}{2}yx^2(1-x)^2u_{xx} + \rho\sigma yx(1-x)u_{xy} + \frac{1}{2}\sigma^2 y u_{yy} + (r-\delta-\lambda\bar{k})x(1-x)u_x\\
&+ \left(\rho\sigma yx + \kappa(\theta-y)\right)u_y + \left((r-\delta-\lambda\bar{k})x - r - \lambda\right)u\\
&+\lambda\int_0^{+\infty}\left(1+x(\xi-1)\right)u\left(\frac{x\xi}{x\xi+1-x},y,\tau\right)q(\xi)d\xi = u_\tau,
\end{aligned} \tag{4}$$

with initial data

$$u(x,y,0) = [x - K(1-x)/S_0]_+.$$

We observe that, in order to have a unique solution, the key is that the coefficients of the pde should satisfy the reversion conditions. Actually it is equivalent to check this conditions on the original equations.

Moreover, to convert the infinite integral domain into a finite one, we set

$$z = \frac{x\xi}{x\xi + 1 - x}. \tag{5}$$

Now the integral term can be rewritten as

$$\mathcal{I}u(x,y,\tau) = \frac{\lambda}{\alpha\sqrt{2\pi}} \int_0^1 \frac{1-x}{z(1-z)^2} u(z,y,\tau) exp\left(-\frac{1}{2\alpha^2}\left(\log\left(\frac{z(1-x)}{x(1-z)}\right) - m\right)^2\right) dz.$$

We calculate the solution of problem (4) by using centred finite differences for first and second order terms. For mixed derivatives we use the following expansion suggested by Bouchut [2]:

$$\frac{\partial^2 u}{\partial x \partial y} \approx \frac{1}{2}\frac{1}{\Delta x \Delta y}\left(\Delta_x^0 \Delta_y^0 + \Delta_x^+ \Delta_x^- \Delta_y^+ \Delta_y^-\right) u^n.$$

As shown by Bouchut, not only does this expansion provide second-order accuracy under Crank-Nicolson schemes, but also is monotone where the naive expansion is not.

The integral term is solved by quadrature rules [3]. Notice that, after the change of variables (3) and (5), the x variables and the z variables belong both to $[0,1]$. Therefore, we may use as stencil for the integral term approximation the same as of the differential part, i.e. we compute the integral term on $(x_j)_{j=0,N_x}$ grid nodes, and we do not need any additional artificial conditions to solve the problem. For the integral term, we then get

$$\mathcal{I}u(x,y,\tau) \approx \lambda \sum_{l=0}^{N_x} \beta_l f(x, x_l) u(x_l, y, \tau),$$

where

$$f(x,z) = \frac{1}{\alpha\sqrt{2\pi}} \frac{1-x}{z(1-z)^2} e^{-\frac{1}{2\alpha^2}(log(\frac{z}{1-z}) + log(\frac{1-x}{x}) - m)^2}.$$

The coefficients β_l are defined by the quadrature rule. In the implementation code, we use the Simpson's rule,

$$\beta = (\beta_0, ..., \beta_{N_x}) = \frac{\Delta x}{3}(1, 4, 2, 4, 2, ..., 4, 1).$$

The above expansions for the space variables, were used in an explicit scheme in time in order to smooth out the initial condition for the first 20 steps. After that, a Crank-Nicolson scheme was implemented. To keep the scheme stable we used a CFL condition for the first explicit steps as,

$$\Delta T_{\text{EX}} \approx \min\left\{\Delta x^2/32, \Delta y^2 \sigma^2/4\right\},$$

and, for the following time steps, using Crank-Nicolson approximation,

$$\Delta T_{\text{CN}} = \sqrt{\Delta T_{\text{EX}}}.$$

Under this setup, at each time step, the finite-differences approximation generate, both for the explicit and Crank-Nicolson part, the following linear system:

$$D^{n+1} U^{n+1} = D^n U^n + F^n + B^n \tag{6}$$

where D^n are square matrices computed by iteratively applying stencils to each node of the underlying discrete grid, F^n is a vector containing the integral term approximation and B_c^n is a vector containing correction for boundaries.

In particular, the vector F^n of system (6) is defined by

$$F^n = (f_{00}^n, f_{10}^n, f_{20}^n, ..., f_{N_x 0}^n, f_{01}^n, f_{11}^n, ..., f_{N_x 1}^n, ..., f_{1 N_y}^n, ..., f_{N_x N_y}^n),$$

where, for $i, j = 0, ..., N_x$

$$f_{ij}^n = \lambda \sum_{l=0}^{N_x} \beta_l f(x_i, x_l) u_{lj}^n.$$

For the explicit steps the solution U_{n+1} can be directly calculated. For the Crank-Nicolson part, the solution U_{n+1} is calculated using a standard Stabilised BiConjugate Gradient (SBiCG) iterative algorithm with the Incomplete LU preconditioner. If the SBiCG algorithm doesn't converge to a solution with a specific accuracy after N steps, a Generalised Minimum Residuals (GMRES) algorithm is used.

2.2 Remark on the boundary approximation

Notice that artificial boundary conditions are introduced only on the y direction. For $y = 0$ and $y = y_{max}$, we assume that the solution of (4) follows the behaviour of the discounted payoff, then for all $x \in (0, 1)$ and $\tau \in (0, T]$, we set

$$u(x, 0, \tau) = u(x, y_{max}, \tau) = [x - Ke^{-r\tau}(1 - x)/S_0]_+.$$

While, for the x direction we don't need any artificial condition. In fact, at the boundary, for $x = 0$ and for $x = 1$ the equation (4) reduces respectively to

$$\frac{1}{2}\sigma^2 y u_{yy}(0, y, \tau) + (\kappa(\theta - y)) u_y(0, y, \tau) - ru(0, y, \tau) = u_\tau(0, y, \tau). \tag{7}$$

and to

$$\frac{1}{2}\sigma^2 y u_{yy}(1, y, \tau) + (\rho\sigma y + \kappa(\theta - y)) u_y(1, y, \tau) - \delta u(1, y, \tau) = u_\tau(1, y, \tau). \tag{8}$$

In both cases the solution is computed directly by the numerical scheme. The proposed transformation is actually quite effective and easy to be implemented. This way, it is possible to avoid the numerical errors coming from the artificial truncation, which is necessary in unbounded domain. Here we perform this change of variable only in the S direction, since the diffusion term is singular in V.

Let us notice also that for other problems, i.e.: one dimensional Merton's model, and the 2D model in [5], this change of variable yields an even greater improvement on the numerical tests, with respect to the troncature methods.

2.3 Numerical Test

For the purpose of testing the implementation, we fixed the following parameters:

- Current Date: 0.000000
- Spot: $S_0 = 100$
- Annual Dividend Rate: 0
- Annual Interest Rate: 10 (Instantaneous Interest Rate: $r = 0.095310$)
- Current Variance: $V_0 = 0.01$
- Mean Reversion: $\kappa = 2$
- Long-Run Variance: $\theta = 0.01$
- Volatility of Volatility: $\sigma = 0.2$
- Lambda: $\lambda = 0.1$
- Mean: $m = 0$
- Variance: $\alpha = 0.16$
- Rho: $\rho = 0.5$
- Strike: $K = 100$
- Maturity: $T = 1$

In the table the results for a European Call for different asset spot prices are shown: here N1 = 201 and N2 = 51. The results are compared with those calculated from the evaluation of the closed form formula as implemented in Premia:

S_0	P (\$)	Δ (\$)	time (s)	P_{CF} (\$)	Δ (\$)
90	3.020626	0.421302	3.919221	3.076410	0.422205
95	5.766727	0.689945	3.921054	5.792484	0.666280
100	9.829271	0.892125	4.261158	9.658327	0.863656
105	14.593351	0.971309	3.890816	14.253142	0.958117
110	19.533194	0.993057	4.361929	19.127745	0.988669

References

[1] DS Bates, Jumps and stochastic volatility: exchange rate processes implicit in deutsche mark options, Rev Fin 1996; 9:69-107 1

[2] F. Bouchut, H. Frid, Finite difference schemes with cross derivatives correctors for multidimensional parabolic systems. J. Hyperbolic Differ. Equ. 3 (2006), no. 1, 27–52. 3

[3] M. Briani, C. La Chioma, R. Natalini Convergence of numerical schemes for viscosity solutions to integro-differential degenerate parabolic problems arising in financial theory, Numer. Math. 98 (2004), no. 4, 607–646. 3

[4] SL Heston, S Nandi, A closed-form GARCH option valuation model, Rev Fin 2000; 13:585-625. 2

[5] Zhu, You-Lan; Li, Jinliang. Multi-factor financial derivatives on finite domains. Commun. Math. Sci. 1 (2003), no. 2, 343–359. 2, 4

3 Appendix: the PDE solver

Here we describe the general infrastructure of the solver stored in the directory

`$PREMIA/Src/mod/highdim_solver/`

3.1 Architecture

We tried to follow an object-oriented approach even if working with a procedural language (ANSI C), as a first step we identified the main data abstractions of our application domain.

As such, an instance of a `PDE_PROBLEM` is composed of:

- a `PDE` describing our Partial Differential Equation
- a `BOUNDARY_DESCRIPTION` for the given problem
- a `GRID` containing all control logic related to discretization of the problem domain
- a `PROBLEM_SOLVER` managing solution algorithms and related functions

A `PDE` object is composed of sub-objects representing both differential and integral terms, each with a related `STENCIL_OPERATOR` which is responsible for the application of proper discretization schemes.

3.2 Operational overview

In order to use our solver, one has to setup a proper `PDE_PROBLEM` instance, with related equation, boundary description and grid description.

```
pde_problem_create(&problem);
problem->max_explicit_steps = 20;
pde_problem_set_desired_accuracy(problem, 10e-10);
pde_problem_set_equation(problem, equation);
pde_problem_set_grid(problem, grid);
pde_problem_set_boundary(problem, boundary);
pde_problem_set_plotfile(problem, "heston_matrix");
pde_problem_set_plotting(problem, 0);
```

then problem solution is triggered by the invocation of the `pde_problem_solve()` function:

```
pde_problem_setup(problem);
pde_problem_solve(problem);
pde_problem_get_solution(problem,&solution);
```

behind the lines, a `PROBLEM_SOLVER` object is created, the solver performs the following steps:

3.2.1 Matrix and boundary setup

In the most generic case, at each time iteration our solver has to perform the following:

$$D^{n+1}U^{n+1} = D^nU^n + B^n_c \tag{9}$$

where D^j are square matrices computed by iteratively applying stencils to each node of the underlying discrete grid, and B^j_c is a vector containing correction for boundaries, source and integral terms (when needed). Several iteration methods are available through the `GRID` object, these are key functions for the overall application.

3.2.2 Iteration

The solver performs a fixed number of iterations (usually 20) by applying an explicit scheme, then solution goes on by means of an implicit Crank-Nicolson scheme. At each implicit iteration, a stabilized biconjugate gradient algorithm is firstly applied, falling back to a generalized minimal residual algorithm when desired accuracy is not reached.

3.3 Implementation

As stated before, code is organized according to an object-oriented fashion, each component of the architecture fits in a separated C module, in order to promote correctedness and robustness of the application, a minimal support for *Design by Contract* methodologies has been added by using a set of macro to support assertions (preconditions and postconditions) declaration in each C function[1]. Moreover, simple support for debugging output has been added, both debugging and assertions support can be enabled or disabled at compile time.

3.4 Grid definition

A grid is defined as follows:

```
/****************************
*     GRID and TUNER       *
****************************/
grid_tuner_create(&tuner);
grid_tuner_set_argument(tuner,&model);
grid_tuner_set_tuner(tuner, EXPLICIT_TUNER, explicit_tuner_proc);
grid_tuner_set_tuner(tuner, IMPLICIT_TUNER, implicit_tuner_proc);
grid_tuner_set_tuner(tuner, RESCALE_TUNER, focus_rescaler_proc);

grid_create(&grid);
grid_set_space_dimensions(grid,2);
grid_set_tuner(grid,tuner);
grid_set_min_value(grid,T_DIM,0.0);
grid_set_max_value(grid,T_DIM,model.T);
```

[1] see `http://archive.eiffel.com/doc/manuals/technology/contract/` for a general overview of these design techniques

```
  grid_set_ticks(grid,X_DIM,model.Ns);
  grid_set_ticks(grid,Y_DIM,model.Nv);
  grid_set_iterator(grid, X_DIM, ITER_PLAIN);
  grid_set_iterator(grid, Y_DIM, ITER_CORE);

  /* focus */
  grid_set_focus(grid,X_DIM,model.S0);
  grid_set_focus(grid,Y_DIM,model.V0);
  grid_rescale(grid);
```

firstly, a `GRID_TUNER` is needed to properly set all grid parameters, `GRID_TUNER` objects are just encapsulations of user-defined tuning routines; iterators specify how to run through the grid, then a grid focus is specified (i.e. the node where our final solution is in) and grid is re-scaled around that.

3.4.1 Function definitions

Functions are needed to specify PDE terms and payoff boundaries, each function is represented by a `FUNCTION` object, the functional part is defined in a static C function, which take as arguments a `FUNCTION` object and a `GRID_NODE`:

```
static double func_uxy(const function *f, const grid_node *node){
  REQUIRE("function_not_null", f != NULL);
  REQUIRE("node_not_null", node != NULL);

  bates_model *model = (bates_model *)f->args;

  double x = node->value[X_DIM];
  double y = node->value[Y_DIM];

  double rho = model->rho;
  double sigma = model->sigma;

  double result = rho * sigma * y * x * (1.0-x);

  return result;
}
```

each `FUNCTION` object is created as follows:

```
  function_create(&f_uxy);
  function_set_args(f_uxy, &model);
  function_set_body(f_uxy, func_uxy);
```

3.4.2 PDE definition

PDEs are defined by defining each PDE term and then putting it all together:

```
  /***************************
  *     EQUATION            *
  ***************************/
  pde_create(&equation);

  /* 1: Uxx */

  function_create(&f_uxx);
  function_set_body(f_uxx,func_uxx);
  stencil_operator_create(&stnop,STENCIL_OP_UXX);

  pde_term_create(&pterm, UXX_TERM, f_uxx, stnop);
  pde_add_term(equation, pterm);

  /* 2: Uxy */
  function_create(&f_uxy);
  function_set_args(f_uxy, &model);
  function_set_body(f_uxy, func_uxy);
  stencil_operator_create(&stnop,STENCIL_OP_UXY);

  pde_term_create(&pterm, UXY_TERM, f_uxy, stnop);
  pde_add_term(equation, pterm);
...

  if(model.lambda != 0.0){
    pde_integral_term_create(&iterm);
    pde_integral_term_set_lambda(iterm, model.lambda);
    pde_integral_term_set_alpha(iterm, model.alpha);
    pde_integral_term_set_m(iterm, model.m);
    pde_integral_term_set_grid(iterm,grid);

    pde_set_integral_term(equation, iterm);
  }
```

3.4.3 Boundary definition

Boundaries are quite simple to setup, we must specify proper functions for left, right and initial values of our unknown solution:

```
  /***************************
  *      BOUNDARY           *
  ***************************/

  function_create(&f_payoff);
  function_set_args(f_payoff,&model);
  function_create(&f_boundary);
  function_set_args(f_boundary,&model);

  if(option_type == PUT_OPTION){
    function_set_body(f_payoff,func_put_payoff);
    function_set_body(f_boundary,func_put_boundary);
  }
  else{
```

```
  function_set_body(f_payoff,func_call_payoff);
  function_set_body(f_boundary,func_call_boundary);
}

boundary_description_create(&boundary);
boundary_description_set_left(boundary,X_DIM, f_boundary);
boundary_description_set_left(boundary,Y_DIM, f_boundary);
boundary_description_set_right(boundary, X_DIM, f_boundary);
boundary_description_set_right(boundary, Y_DIM, f_boundary);
boundary_description_set_initial(boundary, f_payoff);
```

3.4.4 Problem setup and solution

Last step is putting all components into a problem and solve it:

```
/***************************
*      PROBLEM            *
***************************/
pde_problem_create(&problem);
problem->max_explicit_steps = 20;
pde_problem_set_desired_accuracy(problem, 10e-10);
pde_problem_set_equation(problem, equation);
pde_problem_set_grid(problem, grid);
pde_problem_set_boundary(problem, boundary);

/***************************
*         SOLUTION        *
***************************/
pde_problem_setup(problem);
pde_problem_solve(problem);
pde_problem_get_solution(problem,U0);
pde_problem_get_delta_x(problem, D0);
(*U0) *= 2.0 * model.S0;
```