Calibration by Automatic Differentiation

Govindaraj Indragoby and Olivier Pironneau *

September 20, 2004

Abstract

In this paper, we present a way to calibrate the volatility surface of the Black-Scholes and Dupire's financial model from market observed values of a set of call options. We use least-squares, a standard data assimilation technique, for this inverse problem; because we use splines to represent the volatility surface, no regularisation is required for the problem. The method has been implement with automatic differentiation by operator overloading in C++; the object of the paper is to report on the performance of this approach.

Keywords: Black-Scholes equation, control, calibration, automatic differentiation.

1 Introduction

In a market with interest rate r and volatility σ the Black-Scholes model for the evolution of the price S of a financial asset assumes that $t \mapsto S_t$ satisfies a stochastic ODE

$$[\text{ equ1 }]\ dS_t = S_t(\mu dt + \sigma dW_t) \qquad S_0 \text{ given} \tag{1}$$

where W_t is a Wiener process and μ is the tendency of S.

A European call option based on S with strike K at expiration date T and dividend q is priced by its expected profit, namely

$$[\text{ equ1bis }]\ C_{K,T}(S,t) = e^{-r(T-t)}\mathbf{E}(S-K)_+ \tag{2}$$

We will use a least square algorithm to calibrate the local volatility by fitting the prices of a set of European calls available on the market. Using Dupire's equation saves a lot of work for evaluating the least square functional. For computing the gradient of the cost function with respect to volatility, we propose to reduce the number of parameters by using bicubic splines. A Conjugate gradient algorithm will be used on the optimization problem and the

*Laboratoire Jacques-Louis Lions, Université Paris VI, University of Pondicherry - IUF (pironneau@ann.jussieu.fr)

gradients will be computed by automatic differentiation in the forward mode.

In previous studies such as in Coleman et al [**?**], Jackson et al [**?**] and Lagnado and Osher [**?**, **?**] a rather similar method is used with the splines of matlab and implicit computation of derivatives either by numerical differences or by Adol-C. In Achdou et al [**?**] Dupire's equation is used like here too, but not automatic differentiation. One advantage of our implementation over the one used in [**?**] is that we do not call external Matlab or Adol-C routines and yet the C++ implementation is very compact.

In the second part, the method is extended to American option and compared with Achdou et al [**?**].

2 The Black-Scholes Model

An Ito Calculus shows that $C_{K,T}(S,t)$ satisfies the Black-Scholes partial differential equation:

$$\text{[21]} \qquad \partial_t C + \frac{\sigma^2 S^2}{2}\partial^2_{SS} C + (r-q)S\partial_S C - rC = 0 \quad (t,S) \in Q, \qquad (3)$$

with $Q = R^+ \times (0,T)$ and the boundary conditions

$$\begin{array}{rcll} \text{[b]}\, C(S,T) & = & (S-K)_+ & S \in (0,\overline{S}), \\ C(0,t) & = & 0 & t \in (0,T], \\ \lim\limits_{\bar{S}\to\infty} (C(\bar{S},t) - \bar{S} + Ke^{-r(T-t)}) & = & 0 & t \in (0,T]. \end{array} \qquad (4)$$

Notice that $D(S,t) = C(S,t)e^{q(T-t)}$ satisfies the Black-Scholes equation above with q replaced by 0 and r replaced by $r-q$. So without loss of generality we can assume $r := r-q,\ q := 0$ and we will correct the result by $C := Ce^{-q(T-t)}$.

Similarly a put option $P(S,t)$ satisfies also (**??**) but with boundary conditions

$$\begin{array}{rcll} \text{[bb]}\, P(S,T) & = & (K-S)_+ & S \in (0,\overline{S}), \\ P(0,t) & = & e^{-r(T-t)}K & t \in (0,T], \\ \lim\limits_{\bar{P}\to\infty} (C(\bar{S},t) & = & 0 & t \in (0,T]. \end{array} \qquad (5)$$

Recall that both quantitities are constrained by the put-call parity relation:

$$\text{[putcall]}\, P(S,t) = Ke^{-r(T-t)} - S + C(S,t) \qquad (6)$$

3 Homogeneous Equation

To avoid numerical scaling problems we notice that only the variables $x = S/K$ and $\tau = T - t$ enter in the formulation. Indeed with $u(x,\tau) = C(Kx, T-\tau)/K$, we have

$$K\partial_\tau u = -\partial_t C \quad K\partial_x u = K\partial_S C \quad \partial_{xx} u = K\partial_{SS} C$$

so (**??**) is also

$$-K(\partial_\tau u - \frac{\sigma^2}{2}(\frac{S}{K})^2\partial_{xx}u - r\frac{S}{K}\partial_x u + ru) = 0$$

giving finally

$$\text{[homo]} \qquad \begin{aligned} &\partial_\tau u - \frac{\sigma^2 x^2}{2}\partial_{xx}u - rx\partial_x u + ru = 0 \\ &u(x,0) = (x-1)_+ \end{aligned} \tag{7}$$

The put-call parity is now

$$\text{[homo0]}\, v = u + e^{-r\tau} - x \tag{8}$$

with $v(x,\tau) = P(Kx, T-\tau)/K$.

On a truncated domain the system for v is

$$\text{[homo1]} \qquad \begin{aligned} &\partial_\tau v - \frac{\sigma^2 x^2}{2}\partial_{xx}v - (r-q)x\partial_x v + (r-q)v = 0 \\ &v(x,0) = (1-x)_+ \\ &v(0,\tau) = e^{-(r-q)\tau} \\ &v(\bar{x},\tau) = 0 \end{aligned} \tag{9}$$

Here we have written the equation with $r-q$ instead of r because this is the value that must be used in the actual calculation, as it was explained in the beginning.

Then the call option is recovered by

$$\text{[homo3]}\, C_{K,T}(S,t) = e^{-q(T-t)}(Kv(\frac{S}{K}, T-t) - Ke^{-(r-q)(T-t)} + S) \tag{10}$$

with $v(x,\tau)$ solution of (**??**) at $x = S/K, \tau = T-t$.

3.1 Discretization

When $\sigma, r-q$ are smooth and σ is strictly positive, u solution of (**??**) exist and are unique. A good numerical method (see [**?**]) is obtained with the semi-implicit Euler time scheme which leaves the drift term explicit and a central finite difference scheme to discretize in x. This way at each time step the problem reduces to a positive definite linear system of equations. A direct and fast solution is obtained by Gauss factorization of the linear system because the linear system is tri-diagonal. More explicitly, the finite difference scheme is

$$\text{[3]}\, \frac{u_i^{m+1} - u_i^m}{\delta t} - \frac{\sigma_i^{m2} x_i^2}{2}\frac{u_{i+1}^{m+1} - 2u_i^{m+1} + u_{i-1}^{m+1}}{(\delta x)^2} - rx_i\frac{u_{i+1}^m - u_{i-1}^m}{2\delta x} + ru_i^{m+1} = 0, \tag{11}$$

The above discretized equation (**??**) can be written in matrix form in terms of $\{u_i^{m+1}\}_i$ as follows,

$$[14]\ (I+\delta t\mathbf{A}))\begin{bmatrix} u_2^{m+1} \\ u_3^{m+1} \\ \vdots \\ u_{i-1}^{m+1} \\ u_i^{m+1} \\ u_{i+1}^{m+1} \\ \vdots \\ u_{n-1}^{m+1} \end{bmatrix} = \begin{bmatrix} u_2^m \\ u_3^m \\ \vdots \\ u_{i-1}^m \\ u_i^m \\ u_{i+1}^m \\ \vdots \\ u_{n-1}^m \end{bmatrix} + \begin{bmatrix} b_2 \\ b_3 \\ \vdots \\ b_{i-1} \\ b_i \\ b_{i+1} \\ \vdots \\ b_{n-1} \end{bmatrix}$$

where,

$$\mathbf{A} = \begin{bmatrix} \beta_2 & \gamma_2 & 0 & 0 & . & . & . & 0 \\ \alpha_3 & \beta_3 & \gamma_3 & 0 & . & . & . & 0 \\ 0 & \alpha_4 & \beta_4 & \gamma_4 & 0 & . & . & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 0 & . & . & 0 & 0 & \alpha_{n-2} & \beta_{n-2} & \gamma_{n-2} \\ 0 & . & . & . & 0 & 0 & \alpha_{n-1} & \beta_{n-1} \end{bmatrix}$$

$$\begin{aligned} \alpha_i &= -\frac{{\sigma^m}_i^2 x_i^2}{2(\delta x)^2}, \\ \beta_i &= \frac{{\sigma^m}_i^2 x_i^2}{(\delta x)^2} + r, \\ \gamma_i &= -\frac{{\sigma^m}_i^2 x_i^2}{2(\delta x)^2} \\ b_1 &= \delta t(\tfrac{{\sigma^m}_i^2 x_2^2}{2\delta x} u_1^m + r x_2 \frac{u_3^m - u_1^m}{2\delta x}) \\ b_i &= \delta t(r x_i \frac{u_{i+1}^m - u_{i-1}^m}{2\delta x}) \\ b_n &= \delta t(\tfrac{{\sigma^m}_i^2 x_{n-1}^2}{2\delta x^2} u_n^m + r x_{n-1} \frac{u_n^m - u_{n-1}^m}{2\delta x}) \end{aligned}$$

A C++ implementation is given in Appendix A.

4 Calibration Setting

4.1 European Options

Suppose we know a set of prices $C_{d_i\, i\in I}$ of European call options with strike and expiration dates (K_i, T_i) then we transform them into values for v by (**??**)

$$v_{d_i} = \frac{1}{K}_i (C_{d_i} e^{q(T-t)} + K e^{-(r-q)(T_i-t)} - S) \tag{12}$$

We shall solve the problem

$$[equ6]\quad \min_{\tilde{\sigma}} J(\tilde{\sigma}) = \sum_{i\in I} \mid v_i(\frac{S}{K_i}, T_i) - v_{d_i} \mid^2$$
$$\text{subject to } v_i \text{ solution of (??) with } \sigma(x,\tau) = \tilde{\sigma}(xK_i, T_i - \tau) \tag{13}$$

Notice that one can also have the same unique σ for all the v_i in which case all v_i are equal. That corresponds to a $K-T$ dependent σ for each C_i. It means that one solves

$$\text{[equ16]} \min_{\sigma} J(\sigma) = \sum_{i \in I} \mid v(\frac{S}{K_i}, T_i) - v_{d_i} \mid^2 \text{ subject to } (\textbf{??}) \text{ with } \sigma(x,\tau) \tag{14}$$

4.2 Discretization

At each time step and for each option U_k we have to solve a linear system of the type:

$$(I + A(\sigma_k^m))U_k^{m+1} = b^m + U_k^m$$

Denote by $\mathcal{U}$ the vector of space-time values of option U_k based on K_k, T_k. he criteria for optimization is a discrete sum over the points of the mesh times instants in which market data are available

$$J = \sum_k (\mathcal{U}_k - \mathcal{U}_{kd})^T B^k (\mathcal{U}_k - \mathcal{U}_{kd})$$

Here B^k is a rectangular matrix with zeros and ones to select the mesh points and time instants at which market data u_s is available for $\mathcal{U}_k$.

4.3 Minimizing by Descent Method

To minimize $J(a)$ we use the conjugate gradient algorithm with a triple parabolic fit to compute the step size ρ:

The conjugate gradient method (Hestenes[**?**]) is a descent method in the direction d^m with stepsize ρ_m

$$a^{m+1} = a^m + \rho_m d^m$$

where the descent direction d^m is a linear combination of $\nabla_a J(a^m)$ and d^{m-1}:

$$\begin{aligned} \gamma_m &= \nabla J(a^m) \cdot (\nabla J(a^m) - \nabla J(a^{m-1})) \;/\; |\nabla J(a^{m-1})|^2 \\ d^m &= -\nabla J(a^m) + \gamma_m d^{m-1} \end{aligned} \tag{15}$$

and where the step size is the minimum ρ_m of the real variable function $\rho \mapsto f(\rho) := J(a^m + \rho d^m) : \rho_m = \text{argmin} f(\rho)$. However to implement the method one must compute an approximation of this optimal ρ. To do so we use Armijo rule whereby $\rho = \rho_0 2^{-p}$ is from the first p found (dichotomy) such that

$$f(\rho_0 2^{-p}) - f(0) \leq \frac{\rho_0 2^{-p}}{2} d^m \cdot \nabla J(a^m)$$

A C++ implementation is given in Appendix C.

4.4 American Options

With American option the strategy is similar. The added constraint, which is active only with put options,

$$P(S,t) > (K-S)^+ \tag{16}$$

is treated numerically by the Semi-Smooth-Newton algorithm introduced by Kunisch [**?**].

4.4.1 Semi-Smooth Newton Method

With an semi-implicit finite difference Euler time scheme the American option problem becomes

$$\begin{array}{l} \frac{u^m - u^{m-1}}{\Delta t} - \frac{\partial}{\partial x}(\frac{x^2\sigma^2}{2}\frac{\partial u^m}{\partial x}) + ru^m \geq x(r - \frac{\sigma^2}{2})\frac{\partial u^{m-1}}{\partial x}, \\ u^m(x) \geq \phi(x), \;\; x \in \mathcal{R}^+, \; t \in (0,T) \end{array} \tag{17}$$

with equality at each x on one of the two inequations, and initialized by $u^0(x) = \phi(x), \; x \in \mathcal{R}^+$.

Thus, at each time step one must solve a problem of the type

$$Au \geq f, \quad u \geq \phi \text{ in } \mathcal{R}^+ \tag{18}$$

where A is the strongly elliptic symmetric operator

$$u \rightarrow (r + \frac{1}{\Delta t})u - \frac{\partial}{\partial x}(\frac{x^2\sigma^2}{2}\frac{\partial u}{\partial x}) \quad \text{and } f = \frac{u^{m-1}}{\Delta t} + x(r - \frac{\sigma^2}{2})\frac{\partial u^{m-1}}{\partial x}$$

The problem is also

$$\min_{u \in H^1(\mathcal{R}^+), u \geq \phi} \{\frac{1}{2}a(u,u) - (f,u)\}$$

with

$$a(u,v) = \int_0^\infty (\alpha uv + \mu \frac{\partial u}{\partial x}\frac{\partial v}{\partial x}), \quad \alpha = r + \frac{1}{\Delta t} \quad \text{and } \mu = \frac{x^2\sigma^2}{2}.$$

Recall that an equation like $F(x) = 0$ (with $F : \mathcal{R}^n \rightarrow \mathcal{R}^n$) can be solved by Newton's method:

$$x_{k+1} = x_k - G(x_k)^{-1}F(x_k)$$

with $G = F'$ the jacobian of F. Hintermuller et al[**?**] observed that Newton's algorithm converges even if F is not differentiable provided that there exists G such that

$$\text{for all x} \quad \lim_{\|h\| \rightarrow 0} \|F(x+h) - F(x) - G(x+h)h\| = 0$$

a property which is satisfied by $F(x) = \max\{0, x\}$ for instance with $G(x) = \max\{0, x\}/x$.

Ito et al[**?**] suggested to apply the idea to (**??**) reformulated as

$$\begin{array}{l} a(u,v) - (\lambda, v) = (f,v) \;\; \forall v \in H^1(\mathcal{R}^+), \;\; i.e. Au - \lambda = f \\ \lambda - \min\{0, \lambda + c(u - \phi)\} = 0, \end{array} \tag{19}$$

The last equality is equivalent to $\lambda \leq 0, \; \lambda \leq \lambda + c(u - \phi)$ i.e. $u \geq \phi, \; \lambda \leq 0$, with equality on one of them for each x. This problem is equivalent to (**??**) for an real constant $c > 0$ because λ is the Lagrange multiplier of the constraint.

4.4.2 Algorithm

Newton's algorithm applied to (**??**) gives

1. Choose $c > 0,\ , u_0,\ \lambda_0$, set $k = 0$.
2. Determine
$$A_k := \{x\ :\ \lambda_k(x) + c(u_k(x) - \phi(x)) < 0\}$$
3. Set
$$u_{k+1} = \arg \min_{u \in H^1(\mathcal{R}^+)} \{\frac{1}{2}a(u,u) - (f,u)\ :\ u = \phi \text{ on } A_k\}$$
4. Set
$$\lambda_{k+1} = f - Au_{k+1}$$

5 Volatility Surface with Splines

Here the unknown is the volatility surface $S, t \rightarrow \sigma(S,t)$ and what is known is some observed financial data. Here we assume that some American options are known, namely

$$[\text{eq1}]\, p_i := P_{K_i,T_i}(S,t), \quad i = 1, ..., n_d \tag{20}$$

A change in volatility will make a visible difference in the price except near $S = 0$ and for S large where the solution is rather insensitive to σ. Nevertheless the problem is to find the best σ so as to satisfy (**??**).

To reduce the number of parameters we consider a volatility surface in the space of bicubic splines, i.e. for each fixed t, $S \rightarrow \sigma(S,t)$ is a cubic spline in S and for each fixed S, $t \rightarrow \sigma(S,t)$ is a cubic spline in t. But to account for the fact that σ is too difficult to calibrate near 0 and for S large, we limit the interval for the splines to $S \in (0, \overline{S})$ and take σ constant and equal to the implied volatility outside.

The splines are C^1 curves defined by control points $\{S_i\}_1^N$ and it is assume that the tangent at S_i is parallel to $S_{i+1} - S_{i-1}$ except at the first and last points where it is given, zero in our case.

$$\begin{aligned}[24]\sigma(S,t) \quad &= \quad \sigma(S_{i+1},t)\frac{S - S_i}{S_{i+1} - S_i} + \sigma(S_i,t)\frac{S_{i+1} - S}{S_{i+1} - S_i} \\ &\quad + A_i(S - S_i)(S_{i+1} - S)^2 + B_i(S - S_i)^2(S_{i+1} - S)\end{aligned} \tag{21}$$

$\forall \quad S_i \in (0, \overline{S})$ and $\forall \quad S_i < S < S_{i+1}$

Where A_i and B_i $\forall \quad i = 1, 2, 3, ..., N-1$ are constant coefficients, that has to be calculated as follows.

By definition one should get $\sigma(S,t)\mid_{S=S_i} = \sigma(S_i,t)$ and $\sigma(S,t)\mid_{S=S_{i+1}} = \sigma(S_{i+1},t)$. The first derivative with respect to S_i and S_{i+1} are given by

$$\begin{aligned}\sigma'(S_i,t) &= \lim_{S\to S_i}\frac{\sigma(S,t)-\sigma(S_i,t)}{S-S_i} \quad \text{and}\\ \sigma'(S_{i+1},t) &= \lim_{S\to S_{i+1}}\frac{\sigma(S,t)-\sigma(S_{i+1},t)}{S-S_{i+1}}\end{aligned}$$

which implies from (7),

$$\begin{aligned}\sigma'(S,t)\mid_{S=S_i} = \sigma'(S_i,t) &= \frac{\sigma(S_{i+1},t)-\sigma(S_i,t)}{S_{i+1}-S_i} + A_i(S_{i+1}-S)^2\\ \textit{therefore } A_i &= \frac{\sigma'(S_i,t)}{(S_{i+1}-S_i)^2} - \frac{\sigma(S_{i+1},t)-\sigma(S_i,t)}{(S_{i+1}-S_i)^3}) \quad \text{and}\\ \sigma'(S,t)\mid_{S=S_{i+1}} = \sigma'(S_{i+1},t) &= \frac{\sigma(S_{i+1},t)-\sigma(S_i,t)}{S_{i+1}-S_i} - B_i(S-S_i)^2\\ \textit{therefore } B_i &= \frac{\sigma'(S_{i+1},t)}{(S_{i+1}-S_i)^2} + \frac{\sigma(S_{i+1},t)-\sigma(S_i,t)}{(S_{i+1}-S_i)^3}\end{aligned}$$

The spline is unique if $\{x_i,\sigma(S_i,t),\sigma'(S_i,t)\}_{i=1}^N$ are given. So as said above our suggestion for the first derivative for each σ_i is as follows,

$$\sigma'(S_i,t) = \frac{\sigma(S_{i+1},t)-\sigma(S_{i-1},t)}{S_{i+1}-S_{i-1}}$$

except at i = 1 where $\sigma'(S_1,t)$ = 0, therefore

$$\begin{aligned}A_1 &= -\frac{\sigma(S_2,t)-\sigma(S_1,t)}{(S_2-S_1)^3}) \quad \text{and}\\ B_1 &= \frac{\sigma(S_2,t)-\sigma(S_1,t)}{(S_2-S_1)^3}\end{aligned}$$

The same is done with respect to t for each given S, so on each patch $(S_i,S_{i+1})\times(t_j,t_{j+1})$ the spline is the product of a cubic polynomial $P(S)$ in S and a cubic polynomial $Q(t)$ in t:

$$\sigma(S,t) = \sum_{i,j} P_{i,j}(S)Q_{i,j}(t)I_{(S_i,S_{i+1})\times(t_j,t_{j+1})}$$

where I_D is the indicator function of the set D. The total number of parameter in this representation is the number of control points in S multiplied by the number of control points in t.

6 Automatic Differentiation

Derivatives of functions defined by their computer implementations can be calculated automatically and exactly. Several techniques are available but we have used the *forward*

mode [**?**] only. The basic idea is that each line of a computer program can be differentiated automatically, except perhaps branching statement but since there are only a finite number of them in a computer program differentiability will be obtained almost everywhere at worst.

6.1 The Direct Mode

Derivatives of a function can be computed from its differential form, this observation is easy to understand on the following example:
Let $J(u) = |u - u_d|^2$, then its differential is

$$\delta J = 2(u - u_d)(\delta u - \delta u_d) \tag{22}$$

and obviously the derivative of J with respect to u is obtained by putting $\delta u = 1,\ \delta u_d = 0$ in (**??**):

$$\frac{\partial J}{\partial u} = 2(u - u_d)(1.0 - 0.0)$$

Now suppose that J is programmed in C/C++ by

```
    double J(double u, double u_d){
        double z = u-u_d;
        z = z*(u-u_d);
        return z;
     }

    int main(){
        double u=2,u_d = 0.1;
        cout << J(u,u_d) << endl;
     }
```

A program which computes J and its differential can be obtained by writing above each differentiable line its differentiated form:

```
double JandDJ(double u, double u_d, double du,
                              double du_d, double *pdz)
{   double dz = du - du_d;
    double z = u-u_d;
    double dJ = dz*(u-u_d) + z*(du - du_d);
    z = z*(u-u_d);
    *pdz = dz;
    return z;
} int main()
     {      double u=2,u_d = 0.1;
            double dJ;
            cout << J(u,u_d,1,0,&dJ) << endl;
     }
```

Except for the embarrassing problem of returning both `z,dz` instead of z, the procedure is fairly automatic. It can be automatized more systematically by introducing a structured type of differentiable variable to hold the value of the variable and the value of its derivative:

```
    struct {double val[2];} ddouble;
```

and rewrite the above as

```
ddouble JandDJ(ddouble u, ddouble u_d) {
    ddouble z;
    z.val[1] = u.val[1]-u_d.val[1];
    z.val[0] = u.val[0]-u_d.val[0];
    z.val[1] = z.val[1]*(uval[0]-u_d.val[0])
                    + z.val[0]*(uval[1]-u_d.val[1]);
    z.val[0] = z.val[0]*(uval[0]-u_d.val[0]);
    return z;
} int main() {
    ddouble u;
    u.val[0]=2; u_d.val[0] = 0.1; u.val[1]=1; u_d.val[1] = 0.;
    ddouble dJ;
    cout  << J(u,u_d).val[0]<<'\t'<< J(u,u_d,1,0).val[1]<< endl;
}
```

In C++ the program can be simplified further by redefining the operators =, - and *. Then a class has to be used instead of a struct:

```
class ddouble{ public:
    double val[2];
ddouble(double a, double b=0)
       { v[0] = a; v[1]=b;}    // constructor
ddouble operator=(const ddouble& a)
  {
    val[1] = a.val[1]; val[0]=a.val[0];
    return *this;
  }
friend dfloat operator - (const dfloat& a, const dfloat& b)
       {    dfloat c;
            c.v[1] = a.v[1] - b.v[1];   // (a-b)'=a'-b'
            c.v[0] = a.v[0] - b.v[0];
            return c;
       }
friend dfloat operator * (const dfloat& a, const dfloat& b)
       {    dfloat c;
            c.v[1] = a.v[1]*b.v[0] + a.v[0]* b.v[1];
            c.v[0] = a.v[0] * b.v[0];
```

```
            return c;
        }
};
```

As before a differentiable variable has two data fields, its value and the value of its derivative. Then we need a constructor to initialize such a variable and also the operator = to assign them to another one, so that `u=v` triggers `u.val[1]=v.val[1]` and `u.val[0]=v.val[0]`. The operator minus does the usual minus operation on the value of the variables and also on the value of their differentials. For the product the rule for the differentiation of products is used. Finally the function and its calling program are

```
ddouble JandDJ(ddouble u, ddouble u_d) {
    ddouble z= u-u_d
    z = z*(u-u_d);
    return z;
} int main() {
    ddouble u(2,1), u_d=0.1;
    cout  << J(u,u_d).val[0]<<'\t'<< J(u,u_d,1,0).val[1]<< endl;
}
```

Note that `<<` is an operator which can be redefined also inside the class `ddouble`.

The conclusion is that a C program can be differentiated simply by replacing the keyword `double` *by* `ddouble`*.*

Of course C programs are not only assignments and it remains to check that branching statements, loops and function calls etc have the same property, for more details see [**?**].

7 Numerical Result

Four problems have been solved. The first two use the data set defined in Coleman [**?**] for European calls:

$T \setminus \frac{K}{S}$	85	90	95	100	105	110	115	120	130	140
0.175	94.1334	65.5304	37.6595	14.539	2.628	0.169989	0.0511616	0.0216428	0.00219192	0.000664435
0.425	102.86	75.3296	49.5575	27.5782	11.0531	3.24331	0.675185	0.30758	0.0450049	0.0138481
0.695	112.242	85.9846	61.451	39.8781	21.564	8.81898	3.1426	1.08904	0.144728	0.0472063
0.94	120.609	95.2522	71.6456	49.9289	31.6987	16.616	7.68197	3.22097	0.403788	0.102891
1	122.632	97.4059	74.0471	52.3337	33.9402	18.6642	8.95278	3.86012	0.51361	0.12858
1.5	138.631	114.652	91.8786	70.704	51.5963	35.0751	22.7599	13.35	3.88789	0.808319
2	153.824	130.726	108.626	87.7861	68.5092	51.4607	37.6972	25.4531	11.2701	4.17138
3	181.689	159.929	139.102	119.185	100.331	82.7077	67.2662	53.0715	32.3146	19.0952
4	207.228	186.732	167.015	147.983	130.061	112.711	96.7382	81.892	57.2925	39.2772
5	230.781	211.719	192.905	174.576	157.457	141.094	125.126	109.95	84.2147	62.5172

Values C_i of European calls with maturity T_i and strike K_i when $S = 590$.

The last two are based on observations at the same T_i, K_i of American puts computed with a volatility surface shown on figure **??**, i.e. given by

```
double sigvalue(int n, int m)
 { int nn = n*60, mm = m*30;
   if(nn <= 1320)
```

Figure 1: Test volatility surface for calibration of American options. On the right the same surface interpolated on a 5×5 spline is shown.

[voldat]

Figure 2: Test volatility surface obtained by calibration on European options using one homoegeneous equation only. On the right the observed and measured $x - t$ data prices surfaces are shown.

[voldat]

```
      if((nn-600)*(nn-600)+(mm-750)*(mm-750) < 400*400)
        return 0.5;
   else
      if((nn-1800)*(nn-1800)+(mm-750)*(mm-750) < 400*400)
        return 0.3;
   return 0.4;
 }
```

7.1 Calibration from European Calls Data with a Single PDE

We use the homegeneous equation (**??**) so that only one solution is used to match the scaled market data. The solution is shown on figure **??**.

There is almost no visible error between the observation data and the simulated one.

Appendix A

A C++ code to compute a call option with Dupire's equation

```
void Dupire::factLU() {
    cm[1] /= bm[1];
    for(int i=2;i<nK-1;i++)
    {   bm[i] -= am[i]*cm[i-1]; cm[i] /= bm[i]; }
}

void Dupire::solveLU(ddouble *z) {
    z[1] /= bm[1];
    for(int i=2;i<nK-1;i++) z[i] = (z[i] - am[i]*z[i-1])/bm[i];
    for(int i=nK-2;i>0;i--) z[i] -= cm[i]*z[i+1];
}

void Dupire::calc( ) {
  for(int i=0;i<nK;i++) u[i] = K>i*dK ? K-i*dK : 0; // init
  for(int j=0;j<nT;j++) // time loop
   {
     for(int i=1;i<nK-1;i++)  // rhs of PDE
       uold[i] = u[i] + dt*r*i*(u[i+1]-u[i-1])/2;
     u[nK-1]=0;
     u[0] = S*exp(-r*(j+0.5)*dt);
     uold[1] += u[0]*sigma[j][1]*sigma[j][1]*dt/2;

     for(int i=1;i<nK-1;i++) // build matrix
      {  ddouble aux=i*sigma[j][i]*i*sigma[j][i]*dt/2;
         bm[i] = (1+ r*dt + 2*aux); am[i] = -aux; cm[i] = -aux;
      }
     factLU();
     for(int i=1;i<nK-1;i++) u[i]=uold[i];
     solveLU(u);
   }
}
```

Appendix B

Derivation of Dupire's equation when σ is a given function of S/K and $T-t$ only.

Consider the following problem for an American put in $Q := R^+ \times (0,T)$:

$$\partial_t P + \frac{\sigma^2 S^2}{2}\partial_{SS}P + rS\partial_S P - rP \geq 0 \text{ in } Q$$
$$P(S,T) = (K-S)_+ \text{ in } R^+ \qquad P(S,t) \geq (K-S)_+ \text{ in } Q \qquad \text{[a2]} \quad (23)$$

We observe that $(K-S)_+ = K(1-S/K)_+$ so that $u := P/K$ is a function of τ, x, σ with $\tau := T-t,\ x := S/K$. Even if σ is not constant, as long as it is $\sigma(x,\tau)$, the change of variable to x, τ shows that u is the solution at $\kappa = K, \theta = T$ of

$$\partial_\tau u - \frac{\sigma^2 x^2}{2}\partial_{xx}u - rx\partial_x u + ru \geq 0 \text{ in } Q$$
$$u(x,0) = (1-x)_+ \text{ in } R^+ \qquad u(x,\tau) \geq (1-x)_+ \text{ in } Q \qquad \text{[a3]} \quad (24)$$

Now K and T are dummy parameters which appear only after u is computed by (**??**). The correct mathematical formulation for (**??**) is with the variational inequality:

$$u \in V := \{u \ :\ u(x,\tau) \geq (1-x)_+ \text{ in } Q\} \quad (25)$$
$$\forall \hat{P} \in V, \int_Q [-\frac{\partial \hat{P}}{\partial \tau}u + \frac{\partial}{\partial x}(\frac{\sigma^2 x^2}{2}\hat{P})\frac{\partial u}{\partial x} - rx\frac{\partial u}{\partial x}\hat{P} + ru\hat{P}]d\tau dx \quad (26)$$
$$\geq \int_{R^+}(1-x)_+\hat{P}(0)dx \qquad \text{[a3001]} \quad (27)$$

Now let us change variables under the integrals and work with T and K:

$$\frac{\partial u}{\partial T} = \frac{\partial u}{\partial \tau} \qquad d\tau = dT \qquad \frac{\partial u}{\partial x} = -\frac{K}{x}\frac{\partial u}{\partial K} \qquad dx = -\frac{x}{K}dK \quad (28)$$

Plugging these into (**??**) yields

$$\int_Q [u\frac{\partial \hat{P}}{\partial \tau} + \frac{\sigma^2 x^2}{2}[\frac{2K}{x^2}\frac{\partial u}{\partial K} + \frac{K^2}{x^2}\frac{\partial^2 u}{\partial K^2}]\hat{P} + [rK\frac{\partial u}{\partial K} - ru]\hat{P}\frac{x}{K}dKdT$$
$$\geq -\int_{R^+}(1-x)_+\hat{P}(0)\frac{x}{K}dK \qquad \text{[a5]} \quad (29)$$

It remains to set $P = uK$ into (**??**). Indeed

$$\frac{\partial P}{\partial K} = u + K\frac{\partial u}{\partial K} \qquad \frac{\partial^2 P}{\partial K^2} = 2\frac{\partial u}{\partial K} + K\frac{\partial^2 P}{\partial K^2} \text{ and so}$$
$$\int_Q [\frac{P}{K}\frac{\partial \hat{P}}{\partial T} + [\frac{K\sigma^2}{2}\frac{\partial^2 P}{\partial K^2} + r\frac{\partial P}{\partial K}]\hat{P}]\frac{x}{K}dKdT \geq -\int_{R^+}\frac{1}{K}(K-S)_+\hat{P}(0)\frac{x}{K}dK \quad (30)$$

This shows that P is the solution at $\kappa = K, \theta = T$ of

$$-\partial_T P + \frac{\sigma^2 K^2}{2}\partial_{KK}P - rK\partial_K P \geq 0 \text{ in } Q$$
$$P(K,0) = (K-S)_+ \text{ in } R^+ \qquad P(K,t) \geq (K-S)_+ \text{ in } Q \qquad \text{[a8]} \quad (31)$$

with the condition that both inequalities can't be strict at the same point.

Appendix C

A C++ code to compute the minimum of a function $E(xx)$ by a conjugate gradient algorithm with a single parabolic fit to compute an approximate step size.

```
int Min3problem::descent(){
  double E1, normg2old = 1e60;
  double* a=new double[nu];
  double* h=new double[nu];
  for(int i =0; i<nu; i++) h[i]=0;
  for(int p = 0; p < pmax; p++)
   { finishing = p==(pmax-1);
     double E0 = E().val[0];
     dE(a);  // computes the gradient a[]
     double normg2 = 0;
     for(int i =0; i<nu; i++) normg2 += a[i]*a[i];
     double g = normg2/normg2old;
     for(int i =0; i<nu; i++) h[i] = -a[i] + g*h[i];
     normg2old = normg2;
     double dE2 = 0;
     for(int i =0; i<nu; i++) dE2 += a[i]*h[i];
     double rhom=-1;
     do {
         for(int i =0; i<nu; i++) xx[i] += rho*h[i];
         E1 = E().val[0];
         if(fabs(E1-E0)<1e-60) return -1; // no variation of cost
         rhom = -(rho/2)/((E1-E0)/(dE2*rho)-1); // parabolic fit
         if(rhom>1e10) cout<< "rho>>1: ";
         if(rhom<0)rho/=10;
         if(rho<1e-60) {cout<< "rho<<1: "; rhom=1e-60;}
         if(rho<1e-60) break;
         for(int i =0; i<nu; i++) xx[i] += (rhom - rho)*h[i];
         }  while(rhom<0);
   return 0;
}
```

The following computes the gradient of E with respect to xx by automatic differentiation

```
  void Min3problem::dE(double* a)
 { for(int i =0; i<nu; i++)
    {  for(int j =0; j<nu; j++) xx[j].val[1]= 0;
       xx[i].val[1]=1;
       a[i] = E().val[1];
    }
 }
```