

Computational Database Technology Applied to Option Pricing Via Finite Differences

Jöns Åkerlund, Krister Åhlander, and Kjell Orsborn

Department of Information Technology, Uppsala University, Uppsala, Sweden
jons.akerlund@asia.apple.com,
krister.ahlander@it.uu.se, kjell.orsborn@it.uu.se

Abstract. Computational database technology spans the two research fields data-base technology and scientific computing. It involves development of database capabilities that support computational-intensive applications found in science and engineering. This includes support for representing and processing of mathematical models within the database environment without any significant performance loss compared to conventional implementations.

This paper describes how an existing database management system, AMOS II, is extended with capabilities to solve the Black–Scholes equation commonly used in option pricing. The numerical method used is finite differences, and a flexible database framework that can deal with complex mathematical objects and numerical methods is created. We describe how computational data representations and operations are adapted to the database management system and the approach is evaluated with respect to performance, extensibility, and ease of use.

1 Introduction

The numerical solution of partial differential equations (PDEs) is an important area of scientific computing, since there are so many processes in e.g. engineering, physics, biology, and even economics, that can be modeled by PDEs, but there are so few PDEs that are solvable analytically. This kind of applications usually require a very high performance, and a wish to solve PDEs numerically has often been an important force in the development of high performance hardware, such as the Earth Simulator [1]. Finding a suitable, general environment for numerical computations is also an ongoing concern in the field of scientific computing. Historically, the scientific computing community developed successful Fortran libraries for numerical linear algebra such as Linpack [8] and Lapack [9]. With the advent of object-oriented (OO) methods came better modeling tools for supporting complex data structures. Examples of OO projects are Cogito [36,35], Overture [10], and Diffpack [11]. OO frameworks have also been developed and Compose [3] presents a quite general PDE solver design, implemented on top of Overture. Pantazopoulos [26] presents Finanzia as an OO framework for financial modeling. Generally, we find that program packages are either optimized for speed, often written in C or Fortran, or developed in a high-level language such as Matlab to allow for fast development and more readable code. Most of these solutions miss out on one or some of the aspects of performance, maintenance, ease of use, and ability to analyze data.

The approach of using computational databases for numerical methods in engineering has been explored by Orsborn [24], where a FEM application has been integrated with the AMOS database management system (DBMS). The general idea with this approach is to make database technology with efficient data management and query capabilities accessible to computational-intensive applications. However, this will put new demands on the DBMS itself including support for new types of mathematical data and operations. If these requirements can be resolved, future database tools can be used for developing computational database applications found in advanced scientific and engineering applications and furthermore to extend their functionality with facilities like ad hoc query capabilities.

In this paper, a problem solving environment for PDEs is created by extending AMOS II [5] with suitable numerical mechanisms [4]. Specifically, support for finite difference approximations and methods for solving the resulting linear systems of equations are developed. We use the framework to develop a financial modeling application. We solve the Black–Scholes (BS) equation in one and two dimensions. The BS equation describes how the prices of options and financial instruments vary over a certain designated time. This application is very important in today financial markets. The issue to construct general software for modeling it has also been addressed by Skavhaug using Diffpack [32], and by the already mentioned project Finanzia.

We think, however, that a full-fledged computational environment for financial modeling must employ a database for evaluating simulations and for monitoring the market. We argue that a computational database is the appropriate way to design a useful software environment for this kind of applications, in the same way as OO analysis and design stress the importance of data before algorithms. While the computational problem chosen has been the BS equation, it should be emphasized that the overall objective is on providing a framework for scientific computing that is both effective and easy to use, rather than focusing on some specific equation or type of problem.

2 Database Technology for Computational Applications

Database systems have traditionally been positioned for administrative systems development. However, the current trend broadens this perspective to incorporate support for more advanced and complex data sets and applications. These advanced applications are often found in science and engineering where many applications involve large data sets of high complexity. Furthermore, many of these complex data sets originate from some mathematical model where data are generated by applying mathematical operators and algorithms of various complexity.

This work focus on this interdisciplinary research area of database technology and scientific computing that we term *computational database technology* earlier discussed in [24] and [23] that studies how to provide database support for computational-intensive database applications. A central idea is here to provide query-based computations and analysis of complex models within the DBMS while withholding computational performance competitive with conventional codes for scientific computing. To support these computational-intensive applications, a computational database management system (CDBMS) must be extensible on all levels [12] [24] that include:

- *Storage and access extensions* - it should be possible to create new storage structures and operations on them. Computational-intensive applications normally involve tailored and optimized data structures such as numerical matrix and vector representations. These tailored data representations also require specialized indexes and operators such as indexing of numerical matrices and numerical operations such as matrix multiplication and decomposition operations.
- *Query language extensions* - the possibility to create abstract data types and define operations on them, or overloading existing operations. Furthermore, the storage and access extensions should be transparently integrated into the query language to become accessible in query expressions.
- *Query processing extensions* - changing execution strategies should be an option, so that the database can choose between different operations. For instance, the most efficient execution plan for a set of complex arithmetical operations of a matrix expression. Here, the query processor needs to understand specialized indexes, cost models and possibly optimization algorithms.

Extensibility, have mainly been promoted for the object-relational class of database management systems [31] and by the release of the SQL:99 standard. Besides extensibility, also embeddable [29] and main-memory [16] database management systems are important enabling technologies for supporting computational database systems. The ability to embed, extend, compose and configure a DBMS into a tailored system for developing advanced scientific applications can really have the capability to leverage development of scientific software as well as scientific data management. Main-memory database technology is also critical since computational performance must compete with that of conventional implementations in C or Fortran. Earlier work have compared differences between secondary and primary memory storage techniques, especially with regard to speed and results indicates that this approach is feasible [24]. A more thorough discussion on the requirements on computational database systems is given in [24] and several authors have been discussing the need to develop database technology to support advanced applications [30] [6] [2] [17] [18].

The AMOS II DBMS [27] [5], used in this work, is an object-relational DBMS that combines object-oriented modeling with powerful query capabilities. AMOS II is a fully extensible system, covering all levels of extensibility discussed in the previous section, and can be composed and configured for specific needs. The AMOSQL database language of AMOS II can be extended by transparently integrating foreign functions implemented in a conventional programming language such as C/C++, Java or Lisp. Furthermore, AMOS II has a small footprint and can be embeddable into applications providing access to full DBMS capabilities within conventional applications. The final characteristic that makes AMOS II most suitable for developing computational database systems is that it is a main-memory DBMS making it possible to achieve computational performance on par with corresponding C or Fortran implementations.

3 Financial Derivatives and Finite Differences

As mentioned in the introduction, the Black–Scholes (BS) equation is commonly used in the financial field to value financial instruments, such as option pricing of financial

derivatives. These instruments usually depend on the more or less random fluctuation of an underlying value or asset. It is beyond the scope of this paper to discuss finance modeling in depth and we refer to standard financial textbooks for more details [21,38].

In Section 3.2, we mention a few numerical methods for the BS equation. In particular, we recall some basic ideas regarding the finite difference method, and we highlight some requirements on the software that this method imposes. For a thorough treatment on finite differences we refer to e.g. [19], and for a good description of how to solve the BS equation with finite differences we refer to Tavella [34].

3.1 Financial Derivatives and the Black–Scholes Equation

There are many variants of financial derivatives. Among the simplest are European call and put options. A call option is an agreement between two parties that the option holder has the right but not the obligation to buy a specified asset for a fixed price at a future date—i.e., to *exercise* the option. The asset is often a stock, but may be anything from gold to cattle. A put option instead gives the right to sell the asset. There are also American options with the difference that, while a European option can only be exercised at a specified future date, an American option can be exercised at any time prior to the expiry date.

Options can be used for mere speculation—if you think that the price of a stock will increase drastically at the open market, it might be a good idea to buy call options for this stock. If the stock is worth more at the market than the exercise price, you make a profit. Otherwise, it is no gain in exercising the option and you have lost worth the option cost. Another common use of options is to limit various risks for a company, i.e., *hedging*. The bottom-line is that it is important to have appropriate models for the pricing of options, whether you sell or buy them and whatever your purpose is.

In order to model financial markets, it is often assumed that the *Efficient Market Hypothesis* holds. Its weak form states that no excess returns can be earned by analyzing historical data, and the only factor that affects stock prices is the introduction of news, and the market responds immediately to it. Under various assumptions, see e.g. [21,38], the BS equation for the value V of an option based upon a single stock with price S is derived:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS \frac{\partial V}{\partial S} - rV = 0. \quad (1)$$

This PDE states how the time derivative of the value ($\partial V/\partial t$) depends upon the volatility σ , the interest-free rate r , and on the first and second derivatives of the option value with respect to the underlying stock, $\partial V/\partial S$ and $\partial^2 V/\partial S^2$, respectively. At termination time, the value of the option as a function of the underlying stock price is known. For example, a European call option is worth nothing if the actual stock price S is less than the exercise price V , and it is worth $V - S$ if $V > S$. The PDE described by (1) can then be used to compute backwards in time, in order to obtain an estimate of what the option is worth today.

The model is quite sensitive to the underlying data, and the parameter σ is very hard to estimate. The partial derivatives of V are important to consider when analyzing the computations. In financial modeling, they are often referred to as the “Greeks”. They are delta, gamma, rho, theta and vega (which is not actually a Greek letter):

$$\Delta = \frac{\partial V}{\partial S}, \Gamma = \frac{\partial^2 V}{\partial S^2}, \rho = \frac{\partial V}{\partial r}, \Theta = \frac{\partial V}{\partial t}, \mathcal{V} = \frac{\partial V}{\partial \sigma}. \tag{2}$$

Options may be based upon more than one asset, so called *basket options*. When modeling basket options on d underlying assets, BS equation in d dimensions are used. The value V now depends on the coefficient matrix $\sigma\sigma^T$, where the individual components represent volatilities or connection between different volatilities, as well as on partial derivatives with respect to each of the underlying assets:

$$\frac{\partial V}{\partial t} + \frac{1}{2} \sum_{i,j=1}^d [\sigma\sigma^T]_{ij} S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=1}^d r S_i \frac{\partial V}{\partial S_i} - rV = 0 \tag{3}$$

The “multi-dimensional” BS equation (3) is difficult to solve when d becomes large.

3.2 Finite Differences

There are many different ways to numerically solve PDEs. Finite elements, finite differences, and finite volume methods are well-known general purpose methods. For the BS equation, Monte-Carlo methods are often used, particularly when the number of dimensions grow large. Another standard method is based upon trees; see e.g. Hull for an introduction to so-called *lattice methods* [21]. Finite elements are applied to the BS equation in [32]. A thorough description on finite differences for the BS equation is found in Tavella [34].

Ideally, a CDBMS for the BS equation should support a variety of numerical methods. To start with, we have chosen to use finite differences, because it is a fairly simple method to implement and because it is generally applicable. In this section, we recall the basics of finite differences. We also discuss how to solve the resulting linear system of equations by means of iterative methods.

When using the finite difference method, a solution to a certain equation is approximated over a number of discrete points, generally referred to as a grid or mesh, which might be in any number of dimensions depending on the equation. For some problems it is useful to have non-uniform grids, with individual points more densely placed in areas that require higher precision.

In the finite difference method, all partial derivatives in the PDE are approximated by finite difference operators. As an example, the partial derivative $\partial u / \partial t$ is defined as follows:

$$\frac{\partial u}{\partial t}(x, t) = \lim_{\delta t \rightarrow 0} \frac{u(x, t + \delta t) - u(x, t)}{\delta t} \tag{4}$$

The discretization is now made by setting δt to a small but nonzero number:

$$\frac{\partial u}{\partial t}(x, t) \approx \frac{u(x, t + \delta t) - u(x, t)}{\delta t} \tag{5}$$

It is this step, which involves a number of small differences that are not infinitesimal, that is referred to as a *finite difference*. The example given above makes a forward step in time and is for this reason called a *forward difference*, often denoted $D_{+,t}u(x, t)$. Similarly,

$$\frac{\partial u}{\partial t}(x, t) \approx D_{-,t}u(x, y) = \frac{u(x, t) - u(x, t - \delta t)}{\delta t}, \tag{6}$$

is referred to as a *backward step*. These approximations are of first order, which means that the error is proportional to the time step δt . A more accurate approximation is

$$\frac{\partial u}{\partial t}(x, t) \approx D_{0,t}u(x, y) = \frac{u(x, t + \delta t) - u(x, t - \delta t)}{2\delta t}, \quad (7)$$

which is a second order approximation. Combinations of these and other finite differences are used for approximating other partial derivatives. For example,

$$\frac{\partial^2 u}{\partial x^2}(x, t) \approx D_{+,x}D_{-,x}u(x, t) \quad (8)$$

which is also a second order approximation.

For our application, we have chosen the following interior discretization of the BS equation. In one dimension,

$$\begin{aligned} D_{+,t}V(S, t) + \frac{1}{2}\sigma^2 S^2 D_{+,S}D_{-,S} \frac{1}{2}(V(S, t) + V(S, t + \delta t)) + \\ rSD_{0,S} \frac{1}{2}(V(S, t) + V(S, t + \delta t)) - rV(S, t) = \frac{1}{2}(V(S, t) + V(S, t + \delta t)), \quad (9) \end{aligned}$$

with obvious generalizations to higher dimensions, see Tavella [34]. Tavella also presents the boundary conditions that we use.

As seen above, finite differences may be applied in both space and time, leading to a discrete approximation of the PDE in every interior space point. This approximation is often referred to as a *stencil*. If the time discretization is a forward difference, the values at each time level can be calculated from the values at the previous level—i.e., the method is *explicit*. With a backward difference, the values at a new time level are dependent on each other. The method is *implicit*, which implies that a sparse linear system of equations must be solved at each time level. The implications for a general-purpose software that should support finite differences, is that there should be an easy interface to construct different finite difference stencils, and it should be possible to use them both in explicit and implicit settings. However, since (9) is implicit, we have here focussed on this case.

In order to solve linear systems of equations, we can basically choose between direct methods and iterative methods. Iterative methods are often advantageous for sparse systems. We have chosen to implement the generalized minimal residual (GMRES) iterative method, which is a well-known and robust method for this application. There are also several sparse matrix formats to choose from, such as the Compressed Sparse Row (CRS) format and the Ellpack-Eispack format [28]. Initially, we have chosen to support the Ellpack-Eispack format. Even though it is less flexible than CRS, it is appropriate for our application.

4 Implementation

Constructing a suitable database environment for a finite difference solver requires that the suitable numerical methods described are implemented. While these different parts

have all been referred to as extensions, they are regarded as a whole, and all contribute to the solution.

An important part of modelling has been to construct a general storage format for sparse matrices that commonly occur in finite difference computations. The general framework for the matrix extension is first described followed by a description of the scientific computing extension.

4.1 The Matrix Extension

In order to extend the database query language with matrix and vector functionality, a foreign data source must be created that creates numerical objects and methods that can be used with them. Such an extension has been written in C, thereby enabling the external code to use the same physical storage as the database.

The most important points that the extended query language can do is listed in [24], notably the following:

- Make queries involving matrix types in combination with other types of heterogeneous data.
- Express more complex matrix operations in terms of simpler ones.
- Understand domain-specific operators and thereby choose algorithms based on cost measures. This is important in conjunction with solving the BS equation where different solvers for equation systems will be considered based on matrix size and dimension of the problem.
- Enables specific algorithms to be written for specific combinations of matrices and vectors.

As a basis for the matrix extension used in this package, a new version of the same basic representation that is described in [24] has been used. It builds upon the object hierarchy shown in Fig. 1. Part of this hierarchy has been implemented previously by Orsborn [24].

While a full-fledged implementation should contain all different matrix types, the implementation focuses on the core types that are needed for most PDE solvers - sparse matrices and dense vectors. The row type has been implemented due to its similarity with the column type, and dense matrices have been included as a proof of concept (partly of how different algorithms can be chosen for different types of matrices).

The matrix package also distinguishes what sort of numerical representation is used, by keeping the types *imatrix*, *dmatrix* and *fmatrix*, where *i*, *d* and *f* stands for int, double and float, respectively. It should be noted that in the implementation described here, only the double type is supported.

The basic operations that the query language must be able to perform on the matrices include the following matrix and vector operations: (this is a combination of a subset of the operations required by sparse matrix kits as described in [33], with vector operations)

- sparse matrix times dense vector
- sparse matrix plus sparse matrix
- sparse matrix minus sparse matrix
- sparse times constant
- sparse matrix times diagonal matrix ($C = AD$) and vice versa ($C = DA$)

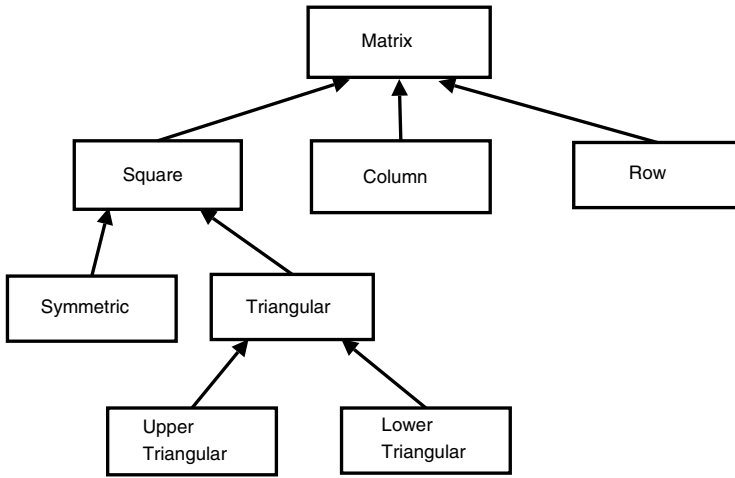


Fig. 1. Type taxonomy for the matrix package

- vector plus vector
- vector minus vector
- vector times constant
- cross product
- dot product
- Euclidean norm

As can be seen from the above, the polymorphism employed in the database allow users to choose many kinds of different operations depending on the chosen formats.

4.2 The Scientific Computing Extension

The scientific computing extension contains a few useful tools commonly used in general PDE solvers. In the present project, capabilities for easily constructing a typical finite difference coefficient matrix as well as two different solution methods, have been added.

Stencil to Matrix. As many computational problems require setting up some sort of banded matrix, especially when stencils are concerned, the function `stencil_to_matrix` has been designed to do that. It takes as argument two AMOS vectors, where the first vector describes positions relative to the main diagonal, and the other the coefficients the respective diagonals should have.

For example, to construct the matrix

$$\begin{pmatrix} 1. & -2. & 0. & 0. & 0. & 0. \\ -2. & 1. & -2. & 0. & 0. & 0. \\ 0. & -2. & 1. & -2. & 0. & 0. \\ 0. & 0. & -2. & 1. & -2. & 0. \\ 0. & 0. & 0. & -2. & 1. & -2. \\ 0. & 0. & 0. & 0. & -2. & 1. \end{pmatrix}$$

one could simply type:

```
set :s = square_sparse_dmatrix(6, 3);  
stencil_to_matrix(:s, {-1,0,1}, {-2.0, 1.0, -2.0});
```

Numerical Solvers. The ability to solve linear systems of equations is important in most PDE solvers. Generally, there are two approaches: direct solution methods such as Gaussian elimination (LU decomposition) or iterative solution methods. We have implemented a direct tridiagonal solver, which is often useful for one-dimensional problems [38], as well as GMRES, an iterative solver, since this usually is a better approach for PDEs in higher dimensions [28]. Both solvers are optimized for the present application.

5 Performance

For the database to be a viable alternative to other applications and problem solving environments, it is not only important that it is easy to use but performance is critical as well. Since most of the time is spent in the time-marching process, the speed of the solvers becomes one of the most important measures. Three factors are important in this case: the time it takes to add and subtract sparse matrices and vectors, the speed of the solvers and the overhead of the database operations. The speed of the solvers have been tested with both the tridiagonal solver and GMRES (which is not the native Matlab GMRES but the same version implemented in the database).

In both tests dummy problems with tridiagonal matrices were used. Both solvers were given sparse matrices with 1.0 on the main diagonal. In the tridiagonal case, the side diagonals had the value 0.000001, and for GMRES, the value of 0.0001 was used.

As can be seen in Fig. 2, the differences between a pure C implementation and the database are too slight to be noticeable. Further, the database system is around 4 times faster than Matlab. In the example with GMRES, Fig. 3, the difference is even more emphasized, the database implementation being around 8 times faster. This should not be seen as saying that Matlab is really that much slower, since an interpreted function for GMRES is used. The time difference between the tridiagonal solver, which is an atomic function in Matlab, says more about real temporal differences.

It is safe to say, however, that the database implementation can provide quite competitive performance in comparison to the native C implementation which is a most promising result for the computational database approach.

In the case with the complete solvers for the BS equation, the difference needs some extra interpretation, as can be seen from Fig. 4, in which the database is around 9 times faster. The combination of the fact that Matlab is an interpreted language, and that both the inner and the other loop is in the foreign function *threeddiag_solve*, gives an explanation for the big difference.

In the case with the two-dimensional solver differences are much smaller, even though the database is faster, see Fig. 5. Since GMRES has been shown to be around 4 times faster in the database, and care has been taken to see that they achieve the same results, either the overhead of the database or the large number of sparse additions and multiplications in the database plays a role. For such operations it is most probable that

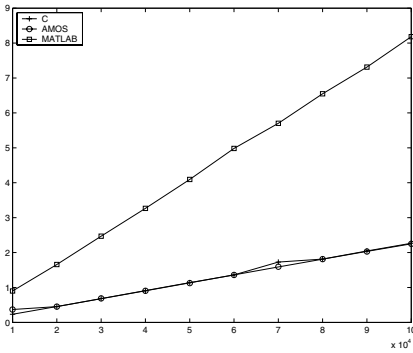


Fig. 2. Comparison between tridiagonal solver in C, AMOS and MATLAB

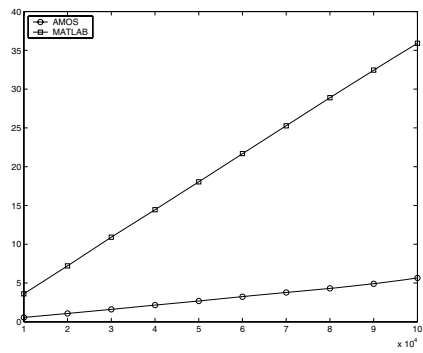


Fig. 3. Comparison between GMRES solver in AMOS and MATLAB

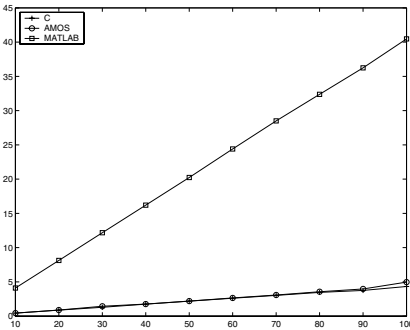


Fig. 4. Comparison of one-dimensional solver in AMOS and MATLAB

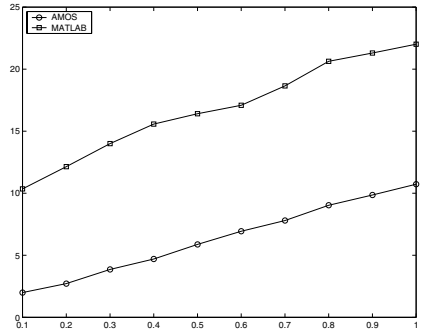


Fig. 5. Comparison of two-dimensional solver in AMOS and MATLAB

Matlab and the database have the same performance. It should further be noted that the two-dimensional database operations do not currently have an optimal implementation and should be exchanged with improved representations in future work.

For the one-dimensional solver, the values used were $S = 120$, $K = 20.0$, $r = 0.555555$, number of grid points = 48, $dt = 0.00005$, $\sigma = 0.3$.

For the two-dimensional solver, $S_1 = 120.0$, $S_2 = 120.0$, $K = 20.0$, $r = 0.555555$, number of grid points = $31 * 31$, $dt = 0.00005$, $\sigma_1 = 0.3$, $\sigma_2 = 0.3$, $\sigma_c = 0.05$ (where σ_c is the relation between the two volatilities).

6 Use Cases

As has been seen earlier on, PDE problems generally involve finding a problem domain, selecting initial values and then time-marching to a specific solution. In any case, regardless of dimensions, this amounts to the following:

1. Create the contract function as initial values.
2. Create the system matrix that is used for solving the system of linear equations.
3. Use a solver for the time-marching.

Both the 1d and 2d solvers are functions that already exist in the database, with predefined functions for the boundary values in 1d and 2d.

This section first gives an account of how the one- and two-dimensional cases work, followed by examples of queries a user might want to perform, including determining some of the Greeks in Eqs. (2).

6.1 1d Solver

While the contract function always looks the same, the system matrix used in this particular implementation consists of finite difference operators for the first- and second-order derivatives in the equation, as well as the time derivative and the r value. While the latter two only affects the main diagonal of the matrix, the operators for the other two (called D_0 and D_+D_- , respectively) use stencils that require information in directly adjacent gridpoints.

The boundary conditions used in both the 1d and the 2d solvers enforce the second derivative to zero (as described in [34]). In the one-dimensional case, this means that the D_+D_- -operator is not used in the first and the last point of the domain. The first-order derivative also looks slightly different in the end points.

The one-dimensional operators have been set up in a sparse matrix by using the *stencil_to_matrix* function described in Section 4.2. Either defined through a function or typed directly at the command line, the user can create the D_+D_- operator by the following set of commands, where $:s$ is an instance of a sparse matrix, and $c1$ and $c2$ are the different values in the stencil:

```
stencil_to_matrix(:s, {-1,0,1}, {c1, c2, c1}, 1, size-2);
```

This says that the matrix should be filled in all places except on the first and the last position (i.e. the first and the last rows in the matrix) However, if a user choosed to fill the whole matrix by these values and wants to change to the boundary condition described above, the following will suffice:

```
stencil_to_matrix(:s, {0, 1}, {0.0, 0.0}, 0);
stencil_to_matrix(:s, {-1,0}, {0.0, 0.0}, size-1);
```

The values in the first and the last rows in the matrix are now replaced by zero values.

When the difference operators have been created (this has been stored in the current implementation of the database as the $d0$ and the $d0dm$ functions, respectively), the system matrix can be expressed as follows:

```
sysmatrix = i(n) - (diag(x) * d0 * r2 +
                  diag(x2) * d0dm * pow(sigma, 2.0) -
                  i(n) * r2) * dt_scaled;
```

Here, $i(n)$ creates an identity matrix, $x2$ is computed as $pow(x, 2.0)$, n is the size of the computed matrix, $dt = \frac{\sigma^2}{2}$, $dt_scaled = dt * 0.5 * pow(sigma, 2.0)$ and $r2$ is the r value multiplied by 2. All the computations involved use the sparse functionality.

The Matlab command for the same expression would be:

```
sysmatrix = eye(n) - (sparse(diag(x)) * d0 * r2 +
                    sparse(diag(x.*x)) * dpdm * sigma^2
                    - eye(n) * r2) * dt_scaled;
```

As can be seen from this example, the complexity of the matrix and vector expressions in the database system can be at least as simple as the corresponding Matlab expression. There is no additional complexity introduced in using the database and numerical data can be freely combined with any other data in the database.

Finally, since the resulting system matrix is tridiagonal, that solver can be used. An example of calling the one-dimensional solver function is the following:

```
set :c = bs_solve_1d(120.0, 20.0, 0.555555, 50, 0.00005,
                   0.3, 0.2)
```

where the parameters given is S_{max} , E (exercise price), r , number of grid points, dt , volatility and the time step the user is interested in. The `:c` object is a vector containing the answer.

6.2 2d Solver

The specific equation for the two-dimensional problem is obtained from (3). The contract function used is:

$$\max\left(\frac{S_1 + S_2}{2} - K, 0\right) \quad (10)$$

The main difference between the one-dimensional and two-dimensional solvers is that the latter require a two-dimensional mesh and have more complicated boundary values. As there is still no specific mesh object available in this implementation (an issue that will be addressed in future), extra functionality must be added to treat a one-dimensional vector that is used in the solution process as two-dimensional.

For this purpose, the structure `size_2d` is created:

```
create type size_2d properties (y_size integer,
                              x_size integer);
```

This structure is then used as a basis for performing two-dimensional operations. The function `abs_pos` calculates the corresponding one-dimensional position of a two-dimensional position:

```
create function abs_pos(size_2d c, integer y,
                      integer x) -> integer as
  select y * y_size(c) + x;
```

The derived function `stencil_to_matrix_2d` uses this functionality to set boundary conditions in the correct rows in the system matrix. For the stencils used in the two-dimensional case, there are eight boundary conditions (four for the corners, and four for the upper, lower, left and right boundaries, respectively).

In order to specify that only the inner part of the square should be filled with stencil coefficients for the D_+D_- operator in the x dimension (as stated above, the second-order derivatives are zero), the following is sufficient. In this example, $:s$ is the system matrix, $:s2d$ is a $size_2d$ structure, $c1$ and $c2$ are constants, and y_size2 and x_size2 are the sizes in the x and y dimensions minus 2, respectively:

```
stencil_to_matrix_2d(:s, :s2d, 1, y_size2, 1, x_size2,
                    {-1,0,1}, {c1, c2, c1});
```

To add or remove a certain boundary condition, one expression like the above is needed. That makes testing different boundary conditions a rather simple task.

Since the stencils used in the two-dimensional case are more complicated than in the one-dimensional problem, the tridiagonal solver can no longer be used. Instead, GMRES is used for the solution.

An example of calling the two-dimensional solver function is

```
set :c = bs_solve_2d(120.0, 120.0, 20.0, 0.555555,
                    30, 0.00005, 0.3, 0.3, 0.05, 0.01);
```

As before, $:c$ is the vector containing the solution, and the parameters are S_{max}^1 , S_{max}^2 , E (exercise price), r , number of grid points (the function currently only handles quadratic grids), dt , volatilities for the two underlying assets, the relation between the two volatilities, and the time amount.

6.3 Query Examples

One specific feature of the database that sets it apart from other systems with a similar aim is the ability to perform ad hoc queries, i.e. finding new information from previously calculated data (or computing new data) that it was not specifically designed for. Queries provide a simple way to find numerical information.

As an example of the latter, the following query is used to get the values of the most commonly used Greek, namely Delta (where $:c$ is the result of the one-dimensional solver).

```
select r from real r, gridpoint_1d g,
           integer size, integer pos
  where size = size(:c) - 1
        and g = unload(:c, 1, size)
        and pos = pos(g) - 1
        and r = (val(g) - val(unload_pos(:c, pos))) / 2.0;
```

The *unload* and *unload_pos* are functions that return a vector as a collection of gridpoints, or as a similar gridpoint for one position.

For Gamma, the expression is only slightly more complicated:

```
select r from real r, gridpoint_1d g, integer size,
           integer pos1, integer pos2
  where size = size(:c) - 2
```

```

and g = unload(:c, 1, size)
and pos1 = pos(g) - 1 and pos2 = pos(g) + 1
and r = val(unload_pos(:c, pos1)) +
      val(unload_pos(:c, pos2)) - 2.0 * val(g);

```

While the Greeks are an important use of queries, there are other examples that really show their capabilities. This example, in two dimensions, show how to get all values in the final solution that are above 1.0 (:s is a result from the two-dimensional solver, and :s2d is a size_2d structure):

```

select yp, xp, r from real r, gridpoint_2d g,
      integer xp, integer yp
where g = unload(:s, :s2d) and r = val(g) and r > 1.0
      and xp = xpos(g)
      and yp = ypos(g);

```

Here, by explicitly including x and y positions in the select statement, the result will show not only the requested values but also their positions.

As another example, say that a user wants to get the numerical values at the upper boundary. This is simply done by asking for the values at the upper gridpoints:

```

select r from real r, gridpoint_2d g
where g = unload(:c, :s2d)
      and ypos(g) = 0 and r = val(g);

```

7 Concluding Remarks

We have developed an extension to an existing database system, AMOS II, that can handle complex numerical models. The extension transparently handles sparse matrices and corresponding operations including numerical solvers, and we demonstrate how a PDE solver for the Black–Scholes equation is developed and used within the extension.

The computational database approach integrates advanced numerical capabilities within a database environment that can avoid unnecessary data duplication and transformation while making queries and other database facilities accessible to computational applications. Using a call-out interface with precompiled database functions and foreign functions allows a user to approach a problem with the same ease of use as for example using the Matlab system. Furthermore, these capabilities can be provided without any significant performance loss making the computational performance comparable with native C implementations. This combination of core functionality written as foreign functions, combined with the high-level efficiency and ease of use of the query language, shows that this approach has the capability to perform as well as other problem solving environments.

The ability to pose queries over numerical results is one of the most attractive features of our computational database environment and one that also distinguishes it from other environments. Future work include improved representations of some financial and mathematical concepts and operations and extending the approach to other application areas. Related work studies visualization of numerical data and, to address huge

data sets and high performance, we are currently developing support for parallel algorithms on distributed platforms.

References

1. The Earth Simulator web site: <http://www.es.jamstec.go.jp/esc/eng/>. Jun 2006.
2. S Abiteboul et al: *The Lowell database research self-assessment*. Commun. ACM, 48(5), May 2005, 111–118.
3. K Åhlander: *An Object-Oriented Framework for PDE Solvers*, PhD Thesis, Thesis No. 423, Uppsala University, Uppsala 1999.
4. J Åkerlund: *A Computational Database for Black-Scholes Equation*, MSc Thesis, Department of Information Technology, Uppsala University, Uppsala, June 2005.
5. The Amos II web site: <http://user.it.uu.se/~udbl/amos/>, Mar 2006.
6. P Bernstein et al: *The Asilomar report on database research*. SIGMOD Record, 27(4), Dec 1998, 74–80.
7. T Björk: *Arbitrage Theory in Continuous Time*. Oxford University Press, 1998.
8. The Linpack web site at NetLib: <http://www.netlib.org/linpack/>, Jun 2006.
9. The Lapack web site at NetLib: <http://www.netlib.org/lapack/>, Jun 2006.
10. D Brown et al: *Overture: An object-oriented framework for solving partial differential equations on overlapping grids*. In M E Henderson, C R Anderson, S L Lyons (eds), *Object-oriented Methods for Interoperable Scientific and Engineering Computing*, SIAM Philadelphia, 1999.
11. A M Bruaset, H P Langtangen: *Object-oriented design of preconditioned iterative methods in Diffpack*. ACM Transactions on Mathematical Software, 23, 1997, 50–80.
12. M Carey, L Haas: *Extensible Database Management Systems*, SIGMOD Record, 19(4), Dec 1990, 54–60.
13. T Conolly, C Begg: *Database Systems - A Practical Approach to Design, Implementation and Management*, 3rd ed, Addison-Wesley 2002.
14. V Eijkhout: *LAPACK working note 50 - distributed sparse data structures for linear algebra operations*. 1992. Available from <http://www.cs.utk.edu/~library/TechReports/1992/ut-cs-92-169.ps.z>, 10 Feb 2005.
15. R Eggermont: *Sparse Matrix Compression Formats*. Available from http://ce.et.tudelft.nl/~robbert/sparse_matrix_compression.html, 10 Feb 2005.
16. H Garcia-Molina, K Salem: *Main memory database systems: an overview*. IEEE Transactions on Knowledge and Data Engineering, 4(6), 1992, 509–516.
17. J Gray, M Compton: *A call to arms*. Queue, 3(3), Apr 2005, 30–38.
18. J Gray et al: *Scientific data management in the coming decade*. SIGMOD Record, 34(4), Dec 2005, 34–41.
19. B Gustafsson et al: *Time Dependent Problems and Difference Methods*. John Wiley & Sons, Inc., 1995.
20. M T Heath: *Scientific Computing - An Introductory Survey*, 2nd ed, McGraw Hill, 2002.
21. J C Hull: *Options, Futures and other Derivatives*, 4th ed, Prentice-Hall International, Inc., 2000.
22. H Löf: *Parallelizing the method of conjugate gradients for shared memory architectures*, Uppsala University, Department of Information Technology, Uppsala 2004.
23. M Nyström, K Orsborn: *Computational Database Technology for Component Mode Synthesis*. Advances in Engineering Software, 35(10-11), Oct-Nov 2003, 735–745.

24. K Orsborn: *On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications*. PhD Thesis, Thesis No. 452, Linköping University, Linköping 1996.
25. K Orsborn et al: *Representing matrices using multi-directional foreign functions*. In *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. P M D Gray, L Kerschberg, P J H King, A Poulouvassilis (eds), Springer-Verlag, 2004.
26. K N Pantazopoulos, E N Houstos: *Modern software techniques in computational finance*. In E Arge, A M Bruaset and H P Langtangen (eds): *Modern Software Tools for Scientific Computing*, Birkhäuser, 1997, 227–246.
27. T Risch et al: *Functional data Integration in a distributed mediator system*, In *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. P M D Gray, L Kerschberg, P J H King, A Poulouvassilis (eds), Springer-Verlag, 2004.
28. Y Saad: *Iterative Methods for Sparse Linear Systems*, 2nd ed, 2000. Available from <ftp://ftp.cd.umn.edu/dept/users/saad/ITBOOK.tar.gz>
29. M Seltzer: *Beyond relational databases*. *Queue* 3(3), Apr 2005, 50–58.
30. A Silberschatz, S Zdonik: *Strategic directions in database systems – breaking out of the box*. *ACM Computing Surveys*, 28(4), Dec 1996, 764–778.
31. M Stonebraker, P Brown: *Object-Relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers, Inc., 1999.
32. O Skavhaug: *Numerical Methods and Software with Applications in Computational Finance*. PhD Thesis, Thesis No. 338, University of Oslo, Oslo 2004.
33. Y Saad: *SPARSKIT: a basic tool kit for sparse matrix computations, version 2*, 1994. Available from <http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps>, 10 Feb 2005.
34. D Tavella and C Randall: *Pricing Financial Instruments - The Finite Difference Method*, John Wiley & Sons, Inc., 2000.
35. M Thuné et al: *Object-oriented modeling of parallel PDE solvers*. In R F Boisvert and P T P Tang (eds), *The Architecture of Scientific Software*, Kluwer Academic Publishers, Boston, 2001, 159–174.
36. M Thuné, et al: *Object-oriented construction of parallel PDE solvers*. In E Arge, A M Bruaset, and H P Langtangen (eds), *Modern Software Tools for Scientific Computing*, Birkhäuser, 1997, 203–226.
37. G Wiederhold: *Information systems that really support decision-making*, *Journal of Intelligent Information Systems*, 14, 2000, 85–94.
38. P Willmott, J Dewynne and S Howison: *Option pricing - mathematical models and computation*, Oxford Financial Press, 2000.