Development of a Black-Scholes solver

Daniel Hupp

October 30, 2012

Contents

1 Introduction

In times, where it is impossible to read the newspaper without stumbling over news about finance markets and their situation, it can be worthwhile to deal with certain aspects of them, for example trading of derivatives. Derivatives are contracts for buying or selling underlying assets in the future to a specified price. The asset can be a stock title, a currency or another tradable good. These contracts itself can be traded.[2]

Options are a special kind of derivatives, that give the buyer the possibility to sell or buy the asset at a special date, called the *date of expiry*, to a certain price, called the *strike price*. This was mainly intended to give people the possibility to insure them against fluctuating stock prices[5]. For example a company wants to takeover another enterprise in the future, by buying then a certain amount of stock titles. By buying an option the company can insure itself to not pay more than the *strike price*. This is also called hedging against fluctuations. Another example is that someone has to pay a bill in a foreign currency in the future, so again he can buy an option with the currency as asset. A second way to use options is for speculating, instead of speculating in stock titles itself. This can be more interesting, because the profit margin can be much higher, but one has to consider that the risk is also higher, then one can easily loose 100% of the investment. This will be easily seen after reading section 2.1. When traders are only interested in the profit, often the asset is not exchanged and only the according difference between market price and *strike price* is paid out[5].

Knowing the motivation for buying options, it rises the questions why and who sells them. Firstly every owner of an option can resell it, when he thinks he can make a profit out of it. But secondly there are the option writers, who have to deliver or buy the asset on the expiry date, when the option is exercised. They have similar motivations like the speculators, they bet on rising or sinking prices. Their hope is that the option is not exercised and then they have the money for which they have sold the option as profit. Now we have seen, there are buyer and seller of options. Where those two fraction meet there is a market and a price is built for the good, in this case the option. Now the question is what is a good and fair price for it, and for this *Black* and *Scholes* have made a mathematical construct to quantify it: the famous *Black-Scholes equation*[1]. The price calculated from it

often reflects the market equilibrium. But it can also be used, where no equilibrium happens due to lag of participants[5].

The purpose of this work is to have a standalone solver for the *Black-Scholes equation*, which will be presented and tested. This work is structured in the following way: in chapter 2 a short introduction to the *Black-Scholes equation* will be given, then in chapter 3 how it can be discretized and solved numerically. Afterward in chapter 4 the implementation in DIFFPACK and the *graphical user interface* will be presented. In chapter 5 various results and performance measurements will be shown. Finally in chapter 6 the whole work will be concluded.

2 Black-Scholes equation

In the previous chapter was already mentioned, that *Black* and *Scholes* found a equation to compute a good and fair price for an option. Before this is shown, first some variables and assumptions have to be introduced.

Only European options are considered, they give the right to buy or sell an asset, after the time of expiry T, to a strike price K. The value of this options is denoted with V and depends on time t and the price of the underlying asset S. The asset is most often a stock title, so in this work S will also be called stock price. When the asset is bought at the time of expiry, the option is called call option. If the asset is sold, then the option is called put option.

To derive the equation certain assumptions have to be made. Although the derivation will not be given here, it is important to know the assumption for not misusing the result of the *Black-Scholes equation*.

The first assumption is very practical, all variables have to be continuous in time and the assets have to be tradable in fractions. This means amongst others, that one cannot model stock crashes.

The next assumption is, if the underlying asset is a stock title, it is not allowed to pay dividend. Another assumption concerning the underlying asset is that it behaves like a *geometric Brownian motion* with a constant *drift* and *volatility*. Furthermore the market has to be frictionless, which means that financial transaction have no costs, there are no taxes on the profits. The market has also to be free of arbitrage, thus no-one can make profit from a transaction without risk. It is also assumed, that it is possible to invest money and get a certain constant *interest rate* r_0.

When these assumption are fulfilled, on can show, that the option price obeys the *Black-Scholes equation*

$$\frac{\partial V}{\partial t} = r_0 V - r_0 S \frac{\partial V}{\partial S} - \frac{1}{2}(\sigma S)^2 \frac{\partial^2 V}{\partial S^2} \quad , \tag{2.1}$$

the derivation can be read in the book of Hull[5].

Here σ, which has to be greater than zero, denotes the volatility, which is a measure for the strength of fluctuations of the the underlying asset. It is the only value, which cannot directly determined in practice, normally it has to be estimated from historical data.

2.1 Initial- and boundary conditions

To solve the *partial differential equation* (2.1), one needs also *initial-* and *boundary conditions*. The value of the option at time T can be easily calculated, making following consideration.

For a call option: if the stock price is higher then the strike price, one exercises the option, buys therefor the stock for the price K and immediately sells it on the market for the price of S. The profit is then $S - K$. If the stock price is lower than the strike price, one does not exercise the option and makes no profit. In the sense of arbitrage one is not allowed to make risk less profit, therefore the value of the option is equal to its profit. Thus, the option value can be written as,

$$V(S,T) = \max(S-K,0) \qquad . \tag{2.2}$$

The same consideration applies for put options, which leads to

$$V(S,T) = \max(K-S,0) \qquad . \tag{2.3}$$

This can be used as initial conditions at $t = T$. Because one is interested in the option value at $t = 0$ one has to calculate backwards in time from this initial condition.

One should use the boundaries at $S_{\min} = 0$ and $S_{\max} = \infty$, but for computations this is not possible, so one has to settle for $S_{\max}$ big enough, such that the boundary conditions have only a small influence in the area of the *strike price* and the expected area of possible stock prices. At the boundaries following linear conditions are used,

$$\frac{\partial V}{\partial t} = r_0 V - r_0 S \frac{\partial V}{\partial S} \qquad , \tag{2.4}$$

assuming that the second derivative of the option price is zero at the boundary.

3 Discretization

In this chapter the discretization of the *Black-Scholes formula* is given. In the first part the discretization in time is considered and in the following two parts the spatial discretization is explained for *finite difference* and *finite element method*.

3.1 Time-stepping

Starting from the Black-Scholes formula (2.1):

$$\frac{\partial V}{\partial t} = \underbrace{r_0 V - r_0 S \frac{\partial V}{\partial S} - \frac{1}{2}(\sigma S)^2 \frac{\partial^2 V}{\partial S^2}}_{:=f(V(S,t),S)} \quad ,$$

implicit and explicit Euler [3] is used, which gives following two equations:

$$\frac{V(S,t^{n-1}) - V(S,t^n)}{\Delta t} = -f(V(S,t^{n-1}),S) \qquad \text{and} \tag{3.1}$$

$$\frac{V(S,t^{n-1}) - V(S,t^n)}{\Delta t} = -f(V(S,t^n),S) \qquad . \tag{3.2}$$

From now on $V^n = V(S,t^n)$ is used, with the definition of $t^n = n\Delta t$, where n goes from 0 to $n_{\max} = \frac{T}{\Delta t}$.

Note that the calculation is done, like already mentioned backwards, in time. That is why a minus sign comes into play on the right hand side, moreover the option price at t^{n-1} depends on the option price of t^n.

Equation (3.1) is multiplied with θ and equation (3.2) with $(1-\theta)$, where θ has to be between zero and one. Adding these two equations, gives rise to

$$\frac{V^{n-1} - V^n}{\Delta t} = -\theta f(V^{n-1},S) - (1-\theta) f(V^n,S) \tag{3.3}$$

Note that in this work for all computations $\theta = \frac{1}{2}$ is used, which corresponds to Crank-Nicolson time-stepping[4].

3.2 Finite difference method

For discretizing equation (3.3) also in space, finite differences are used. Namely second order central differences for the second derivative of V and first order forward finite differences for the first derivative. For the right hand side function $f(V, S)$ one gets

$$f(V_i^n, S_i) = r_0 V_i^n - r_0 S_i \frac{V_i^n - V_{i+1}^n}{\Delta S} - \frac{1}{2}(\sigma S)^2 \frac{V_{i-1}^n - 2V_i^n + V_{i+1}^n}{\Delta S^2} \quad , \tag{3.4}$$

where V_i^n is equal to $V(S_i, t^n)$, the stock price coordinate is equidistant discretized with ΔS, such that $S_i = S_{min} + i\Delta S$. The index i goes from zero to $I = \frac{S_{max} - S_{min}}{\Delta S}$.

Combining equation (3.3) and (3.4) gives,

$$\begin{aligned} \frac{V_i^{n-1} - V_i^n}{\Delta t} = &+ \frac{1}{2}\frac{(\sigma S_i)^2}{\Delta S^2} && \left(\theta V_{i-1}^{n-1} + (1-\theta)V_{i-1}^n\right) \\ &- \left(\frac{(\sigma S_i)^2}{\Delta S^2} - \frac{r_0 S_i}{\Delta S} + r_0\right) && \left(\theta V_i^{n-1} + (1-\theta)V_i^n\right) \\ &+ \left(\frac{1}{2}\frac{(\sigma S_i)^2}{\Delta S^2} - \frac{r_0 S_i}{\Delta S}\right) && \left(\theta V_{i+1}^{n-1} + (1-\theta)V_{i+1}^n\right) \end{aligned} \quad . \tag{3.5}$$

For the *boundary conditions* (2.4) the same approach leads to

$$\begin{aligned} \frac{V_0^{n-1} - V_0^n}{\Delta t} = &- \left(\frac{r_0 S_{\min}}{\Delta S} + r_0\right) && \left(\theta V_0^{n-1} + (1-\theta)V_0^n\right) \\ &+ \frac{r_0 S_{\min}}{\Delta S} && \left(\theta V_1^{n-1} + (1-\theta)V_1^n\right) \end{aligned} \quad , \tag{3.6}$$

at $S_0 = S_{min}$, and for $S_I = S_{max}$ one gets

$$\begin{aligned} \frac{V_I^{n-1} - V_I^n}{\Delta t} = &\left(\frac{r_0 S_{\max}}{\Delta S} - r_0\right) && \left(\theta V_I^{n-1} + (1-\theta)V_I^n\right) \\ &- \frac{r_0 S_{\max}}{\Delta S} && \left(\theta V_{I-1}^{n-1} + (1-\theta)V_{I-1}^n\right) \end{aligned} \quad . \tag{3.7}$$

3.3 Finite element method

For the *weak formulation*, equation (3.3) is multiplied with the test function $U(S)$ and integrated over the domain $\Omega = [S_{min}, S_{max}]$. This gives

$$\int_\Omega (V^{n-1} - V^n) U \mathrm{d}S = -\Delta t \int_\Omega \left(\theta f(V^{n-1}, S) + (1-\theta) f(V^n, S)\right) U \mathrm{d}S \tag{3.8}$$

Note, that the test function is zero on the boundaries, the *boundary conditions* will be discussed separately in the end of this section.

For computing $\int_\Omega f(V,S) U \mathrm{d}S$, first the function $f(V,S)$ is manipulated. Namely the product rule is used for the second derivative term, which gives an additional term:

$$\begin{aligned} f(V,S) &= r_0 V - r_0 S \frac{\partial V}{\partial S} - \frac{\partial}{\partial S}\left(\frac{1}{2}(\sigma S)^2 \frac{\partial V}{\partial S}\right) + \frac{\partial}{\partial S}\left(\frac{1}{2}(\sigma S)^2\right)\frac{\partial V}{\partial S} \\ &= r_0 V - \left(r_0 S - \sigma^2 S\right)\frac{\partial V}{\partial S} - \frac{\partial}{\partial S}\left(\frac{1}{2}(\sigma S)^2 \frac{\partial V}{\partial S}\right) \quad . \end{aligned} \tag{3.9}$$

Now equation (3.9) is multiplied with the test function and integrated over the domain,

$$\int_\Omega f(V,S) U \mathrm{d}S = \int_\Omega r_0 V U \mathrm{d}S - \int_\Omega \left(r_0 - \sigma^2\right) S \frac{\partial V}{\partial S} U \mathrm{d}S - \int_\Omega \frac{\partial}{\partial S}\left(\frac{1}{2}(\sigma S)^2 \frac{\partial V}{\partial S}\right) U \mathrm{d}S \quad . \tag{3.10}$$

For the last term of the previous equation(3.10), *integration by parts* is used:

$$\begin{aligned} \int_\Omega f(V,S) U \mathrm{d}S = \int_\Omega r_0 V U \mathrm{d}S &- \int_\Omega \left(r_0 - \sigma^2\right) S \frac{\partial V}{\partial S} U \mathrm{d}S \\ &\underbrace{- \frac{1}{2}(\sigma S)^2 \frac{\partial V}{\partial S} U \Big|_{\partial\Omega}}_{=0} + \int_\Omega \frac{1}{2}(\sigma S)^2 \frac{\partial V}{\partial S}\frac{\partial U}{\partial S}\mathrm{d}S \quad . \end{aligned} \tag{3.11}$$

Now basis function $N_i(S)$ will be used, which will not be specified here, because they can be easily changed in the DIFFPACK implementation, but note that for the computation in this work first order continuous finite elements on a equidistant grid are used. The function of option value is approximated by $V^n(S) = \sum_i v_i^n N_i(S)$ and test functions N_j for all j are used.

So equation (3.8) becomes:

$$\int_\Omega \left(\sum_i v_i^{n-1} N_i - \sum_i v_i^n N_i \right) N_j \mathrm{d}S =$$
$$- \Delta t \int_\Omega \left(\theta f(\sum_i v_i^{n-1} N_i, S) + (1-\theta) f(\sum_i v_i^n N_i, S) \right) N_j \mathrm{d}S \quad . \quad (3.12)$$

Ordering the unknown terms to the left and the known terms to the right leads to,

$$\int_\Omega (\sum_i v_i^{n-1} N_i N_j \mathrm{d}S + \Delta t \theta \int_\Omega f(\sum_i v_i^{n-1} N_i, S) N_j \mathrm{d}S =$$
$$\int_\Omega \sum_i v_i^n N_i N_j \mathrm{d}S + \Delta t (1-\theta) \int_\Omega \left(f(\sum_i v_i^n N_i, S) \right) N_j \mathrm{d}S \quad . \quad (3.13)$$

Using the discrete function space on equation (3.11), one gets

$$\int_\Omega f(\sum_i v_i^n N_i, S) N_j \mathrm{d}S = \int_\Omega r_0 \sum_i v_i^n N_i N_j \mathrm{d}S - \int_\Omega \left(r_0 - \sigma^2 \right) S \frac{\partial}{\partial S} (\sum_i v_i^n N_i) N_j \mathrm{d}S$$
$$+ \int_\Omega \frac{1}{2} (\sigma S)^2 \frac{\partial}{\partial S} (\sum_i v_i^n N_i) \frac{\partial N_j}{\partial S} \mathrm{d}S \quad . \quad (3.14)$$

Because all terms are linear, one gets from equation (3.13) and (3.14) a linear equation system:

$$A v^{n-1} = b(v^n) \quad , \quad (3.15)$$

where b is depending on the previous time step. The system matrix A is given by

$$A_{ij} = \int_\Omega N_i N_j \mathrm{d}S + \Delta t \theta \Big[r_0 \int_\Omega N_i N_j \mathrm{d}S - \left(r_0 - \sigma^2 \right) \int_\Omega S \frac{\partial N_i}{\partial S} N_j \mathrm{d}S$$
$$+ \int_\Omega \frac{1}{2} (\sigma S)^2 \frac{\partial N_i}{\partial S} \frac{\partial N_j}{\partial S} \mathrm{d}S \Big] \quad . \quad (3.16)$$

And the right hand side is

$$b_j = \sum_i v_i^n \int_\Omega N_i N_j \mathrm{d}S - \Delta t (1-\theta) \Big[r_0 \sum_i v_i^n \int_\Omega N_i N_j \mathrm{d}S$$
$$- \left(r_0 - \sigma^2 \right) \sum_i v_i^n \int_\Omega S \frac{\partial N_i}{\partial S} N_j \mathrm{d}S + \sum_i v_i^n \int_\Omega \frac{1}{2} (\sigma S)^2 \frac{\partial N_i}{\partial S} \frac{\partial N_j}{\partial S} \mathrm{d}S \Big] \quad . \quad (3.17)$$

Note, that in the implementation *Gauss quadrature*[6] is used for evaluating the integrals.

The *boundary conditions*(2.4) are not treated in the *weak formulation*, so additional equations for the boundary nodes are needed. Instead the equation (2.4) are treated with *finite differences*, which leads to following entry in the system matrix,

$$A_{ii} = N_i\Big|_{S_{\min\,/\,\max}} + \theta\Delta t \left[r_0 N_i\Big|_{S_{\min\,/\,\max}} - r_0 S_{\min\,/\,\max} \frac{\partial N_i}{\partial S}\Big|_{S_{\min\,/\,\max}} \right] \quad . \tag{3.18}$$

The according entry in the right hand side vector, is

$$b(i) = v_i^n N_i\Big|_{S_{\min\,/\,\max}} -(1-\theta)\Delta t \left[r_0 v_i^n N_i\Big|_{S_{\min\,/\,\max}} - r_0 S_{\min\,/\,\max} \frac{\partial \sum_j v_j^n N_j}{\partial S}\Big|_{S_{\min\,/\,\max}} \right] \quad . \tag{3.19}$$

These expressions are only valid when N_i is a boundary node.

4 Implementation

In this chapter, the implementation of the discretization introduced in the previous chapter, will be shown. In the first part the numerical solvers are shown in the later part the *graphical user interface* is presented.

4.1 Solvers

In DIFFPACK to set up a simulation, the `MenuSystem` is used. It handles classes of type `SimCase`, and is therefor a flexible interface between console, input file or GUI class and the actual solver.

The self programmed solver has to implement the interface `SimCase` 4.1. The `MenuSystem` has two functions, which one has to use: `init()` and `multipleLoop()`. The top level code for a simulation could look like this:

```
menu.init ("", "");
SimCase *p; p = new Solver***();
menu.multipleLoop(*p);
```

Note in this text `Solver***` is used, when a code fragment is or can be used by `SolverFDM` or `SolverFEM`.

The `init()` method sets up the `MenuSystem` and gives it a title. The function `mutlipleLoop()` does the main work, namely it calls `adm()` and `solveProblem()` on its `SimCase`, so they have to be implemented in the self written solver class. Furthermore the `mutlipleLoop()` makes it possible to run the solver for various input parameter and there combination, which is very pleasant for example for convergence studies, because one has to handle only one input file and run the code only once. In case of this work `adm()` is the same for `SolverFDM` and `SolverFEM`.

```
void Solver***::adm(MenuSystem& menu) {
  attach(menu);
  define(menu);
  scan();
```

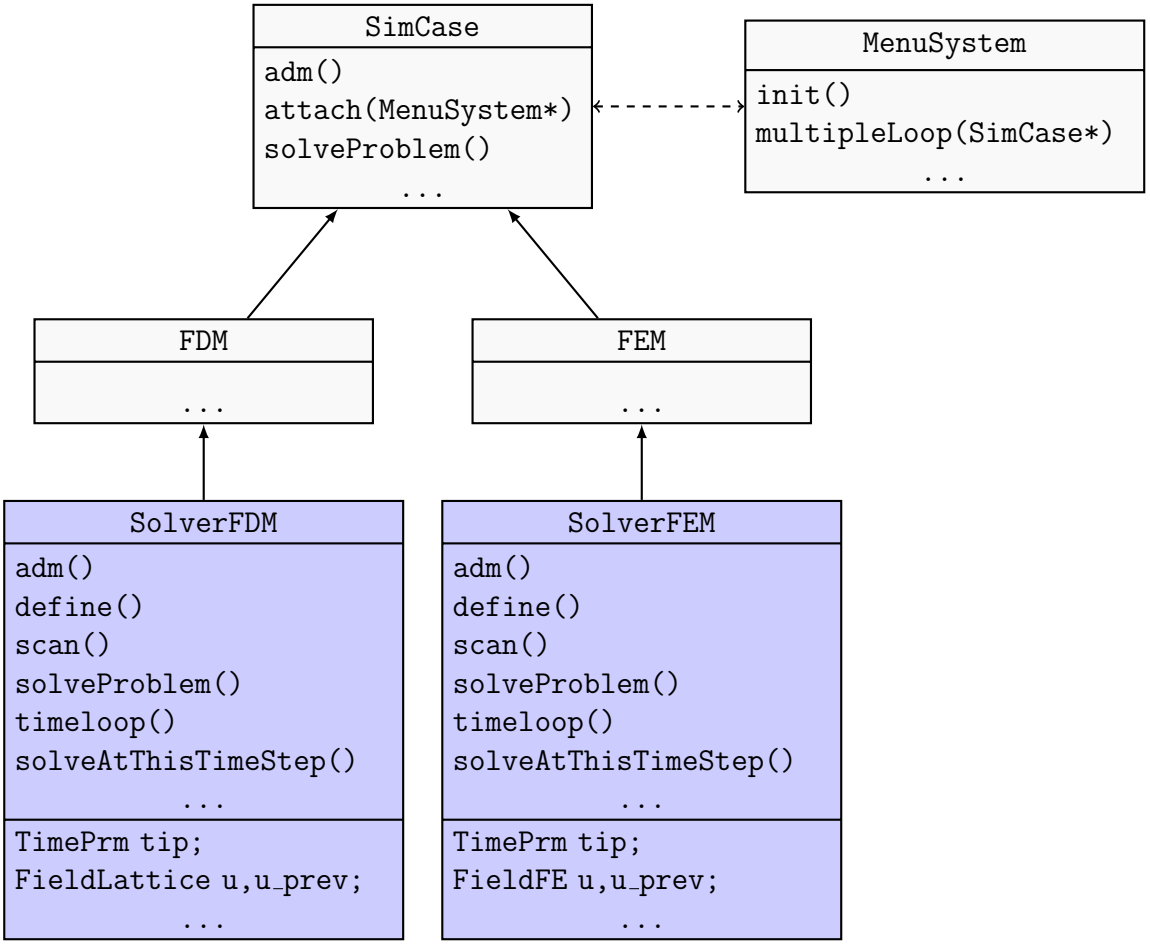

Figure 4.1: Class diagram of Solver classes: the blue classes are implemented for solving the *Black-Scholes equation* (2.1); the gray ones are provided by DIFFPACK. The dashed lines corresponds to has-a relation and the continuous line show inheritance relations.

```
}
```

`attach()` links the `SimCase` with the `MenuSystem`, whereas `define()`, creates different entries in the `MenuSystem` with their names and default values.

`scan()` gets the simulation parameter from the `MenuSystem`, according to the defined values in the `defined()` function, and initials the data members accordingly.

The `solveProblem()` routine is called afterward, it is separately implemented in the `SolverFDM` and `SolverFEM`, although it is almost the same. It can be excerpt as:

```
void Solver***::solveProblem() {
  // ...
  timeLoop();
  // ...
}
```

Also the `timeloop()` function is more or less the same for both classes, and can be summarized as:

```
void Solver***::timeloop() {
  // ...
  tip->initTimeLoop();
  // ...
  while( !tip->finished() ) {
    // ...
    tip->increaseTime();
    solveAtThisTimeStep();
    *u_prev = *u;
    // ...
  }
  // ...
}
```

The crucial differences in the two solver classes is in the implementation of SolveAtThisTimeStep(), which will discussed separately in the next 2 subsections.

4.1.1 Finite difference solver

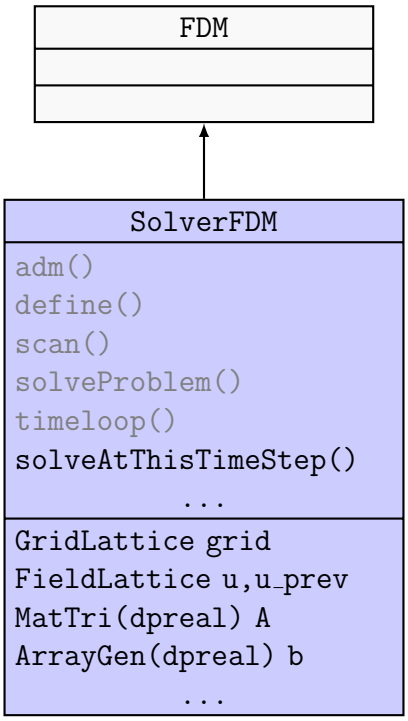

Figure 4.2: Class Diagram of `SolverFDM`, the gray functions, were already discussed in the previous section.

For better understanding it is worthwhile to have a look at the underling used data members: `GridLattice` is used, and gives access to the underlying asset variable, it is initialized in the `scan()` function. For the field, which one wants to compute the according `FieldLattice` is used. For the tridiagonal system matrix with the according right hand side vector `MatTri` and `ArrayGen` are used, they are initialized at every time-step in the `solveAtThisTimeStep()` routine, according to the equations (3.5), (3.6) and (3.7):

```
void SolverFDM::solveAtThisTimeStep() {
  // ...
  A(1,0)   = 1 + Δtθ (r_0 + r_0 S_i / ΔS) ;
  A(1,1)   = −Δtθ (r_0 S_i / ΔS) ;
  b(1) = u_prev->values()(1) * (1 − Δt(1 − θ)(r_0 + r_0 S_i / ΔS))
       - u_prev->values()(2) *  Δt(1 − θ)(− r_0 S_i / ΔS) ;
  for(i = 2; i <= n-1; i++) {
    A(i,-1) = - Δtθ (1/2) (σ S_i / ΔS)^2 ;
```

```
    A(i,0)   = 1 + Δtθ ( (σS_i/ΔS)^2 + r_0 + r_0S_1/ΔS ) ;
    A(i,1)   = −Δtθ ( 1/2 (σS_i/ΔS)^2 + r_0S_1/ΔS ) ;
    b(i) = - u_prev->values()(i-1) * Δt(1 − θ) ( −1/2 (σS_i/ΔS)^2 )
         + u_prev->values()(i) *
            ( 1 − Δt(1 − θ) ( (σS_i/ΔS)^2 + r_0 + r_0S_i/ΔS ) )
         - u_prev->values()(i+1) *  Δt(1−θ) ( −1/2 (σS_i/ΔS)^2 − r_0S_i/ΔS ) ;
  }
  A(n,-1)   = Δtθ ( r_0S_n/ΔS ) ;
  A(n,0)   = 1 + Δtθ ( r_0 − r_0S_n/ΔS ) ;
  b(n) = - u_prev->values()(n-1) *Δt(1 − θ) ( r_0S_n/ΔS )
         + u_prev->values()(n) * ( 1 − Δt(1 − θ) ( r_0 − r_0S_n/ΔS ) ) ;
  // ...
  A.factLU();
  A.forwBack(b, u->values());
}
```

Afterwards the matrix `A` is factorized in its LU-decomposition, which is used to solve the linear system of equation by forward backward substitution, thereby the solution is also transferred to the lattice field `u`. Note that a tridiagonal matrix `MatTri` is used, because in section 3.2, it has been seen that discretization leads to a tridiagonal matrix. This matrix implementation is is very efficient in storage and solving. The first index denotes the row, and the second index has to be between minus one and one, denoting the sub- and main diagonals.

4.1.2 Finite element solver

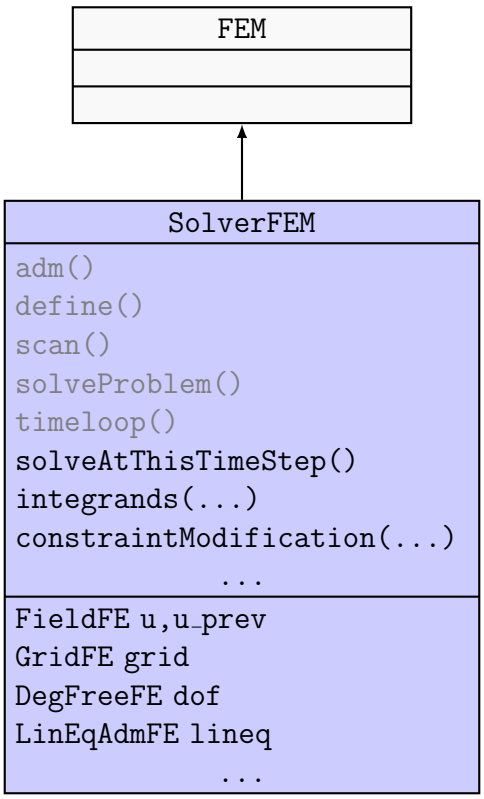

Figure 4.3: Class Diagram for `SolverFEM`, the gray functions were already discussed in the previous section.

The `SolverFEM` class uses a lot of routines, which are implemented in the base class `FEM`, therefore it is worthwhile to have a look in the book of Langtangen [6], and see also their implementation. Here only a rough overview will be given.

```
void SolverFEM::solveAtThisTimeStep() {
  // ...
  makeSystem(*dof,*lineq);
  // ...
  constraintModification(VecLinEqConstr, lineq->b(),
     essential_dof, lineq->A());
  lineq->solve();              // solve linear system
  dof->vec2field(linsol, *u); // load linsol into the
     field u
}
```

The first step in the `solveAtThisTimeStep()` is assembling of the system matrix and its right hand side vector. It is done in the function `makeSystem()`, which is inherit by the base class `FEM`. It contains the loop over the elements, for computing the system matrix. For this a object `DegFreeFE` is used, which has access to the `GridFE` and `FieldFE` and so maps the finite elements to the entries of the linear system of equations and vice versa. It uses the `calcElmMatVec()` routine, which calculates the element matrix and vector by numerical integration. For the numerical integration a loop over the integration points are done. For every integration point the function `integrands()` is called. This one is overwritten in the `SolverFEM` class, for simplicity and illustration purpose only one dimension is shown:

```
void SolverFEM::integrands(ElmMatVec& elmat, const
   FiniteElement& fe) {
  dpreal detJxW = fe.detJxW();
  const int  nbf = fe.getNoBasisFunc();

  Ptv(dpreal) gradup_pt;
  dpreal up_pt;
  dpreal S;

  ElmItgRules rules1 (GAUSS_POINTS, -1);
  fe.getGlobalEvalPt(S);

  // u and grad(u) at the previous time level at the
     current itg.pt:
  up_pt = u_prev->valueFEM (fe);
  u_prev->derivativeFEM (gradup_pt, fe);

  for(i = 1; i <= nbf; i++) {
    for(j = 1; j <= nbf; j++) {
      elmat.A(i,j)  =  fe.N(i)*fe.N(j)*detJxW;
      elmat.A(i,j) +=  Δtθ*½σ²S*fe.dN(i)*fe.dN(j)*detJxW;
      elmat.A(i,j) += -Δtθ*(r₀*S-σ²S)*fe.N(i)*fe.dN(j)*detJxW ;
      elmat.A(i,j) +=  Δtθ*r₀*fe.N(i)*fe.N(j)*detJxW;
    }
    elmat.b(i)  =  up_pt*fe.N()*detJxW;
    elmat.b(i) += -Δt(1-θ)*r₀*up_pt*fe.N(i)*detJxW;
    elmat.b(i) +=  Δt(1-θ)*(r₀*S-σ²S)*gradup_pt*fe.N()*detJxW;
    elmat.b(i) += -Δt(1-θ)*½σ²S*gradup_pt*fe.dN(i)*detJxW;
  }
```

```
}
```

The functions `integrands4side()` and `fillEssBC()` are empty implemented to not interfere with the boundary conditions (3.18), (3.19). For the boundary condition the function `constraintModification()` is called after the assembling procedure, which erases the entries of the equation for the boundary nodes and inserts the values according to (3.18) and (3.19) instead.

Afterwards the linear equation is solved by Gauss elimination and the solution is transferred by the `dof` object back to the field `u`.

Note that for all computation linear finite elements in one dimension are used (`ElmB2n1D`).

4.2 Graphical user interface

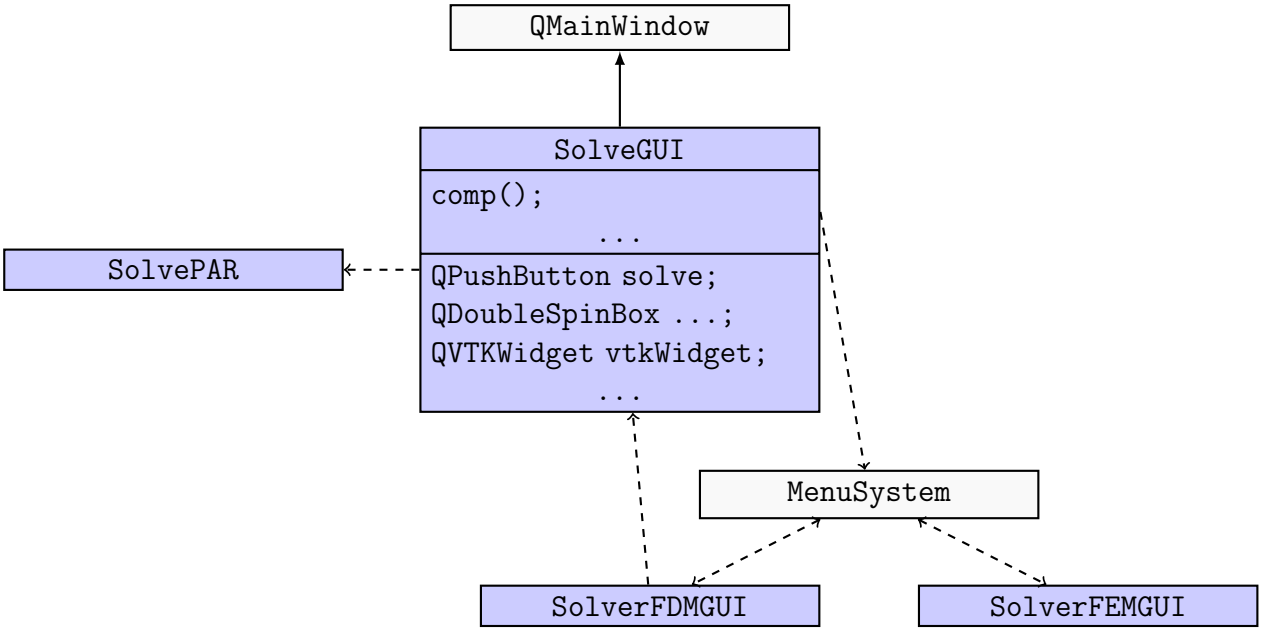

Figure 4.4: Class Diagram for `SolveGUI`. Here the gray class is implemented by QT or DIFFPACK. For clearness the has relation between `SolverFEMGUI` and `SolveGUI` is not drawn, then it is the same as for `SolverFDMGUI`.

The *graphical user interface*(abbr. GUI), which can be seen in figure 4.5, is implemented in QT. The main purpose of it is to calculate the solution of the *Black-Scholes equation* for European options, therefore one has an interface, where one can change all necessary parameters(q.v. chapter 2), choose between the finite element and finite difference method.

The main class is `SolveGUI`(cf. figure 4.4), here all QT objects are initialized and controlled, for example the `QDoubleSpinBox`, `QPushButton`, `QVTKWidget` and many more.

When the button *solve* is pressed the function `comp()` is called. There a local `MenuSystem` is initialized which then uses `SolverFDMGUI` or `SolverFEMGUI` for computing the solution of the Black-Scholes equation. The `Solver***GUI` classes are derived from `Solver***` and add responsibility for plotting their result in the `vtkWidget`, because they know best, how to plot their underlying data.

Further more the user has the opportunity to save and load the parameters shown in the GUI. For this the serializable class `SolvePAR` is used. After this one has the opportunity to save the plot to a JPEG file.

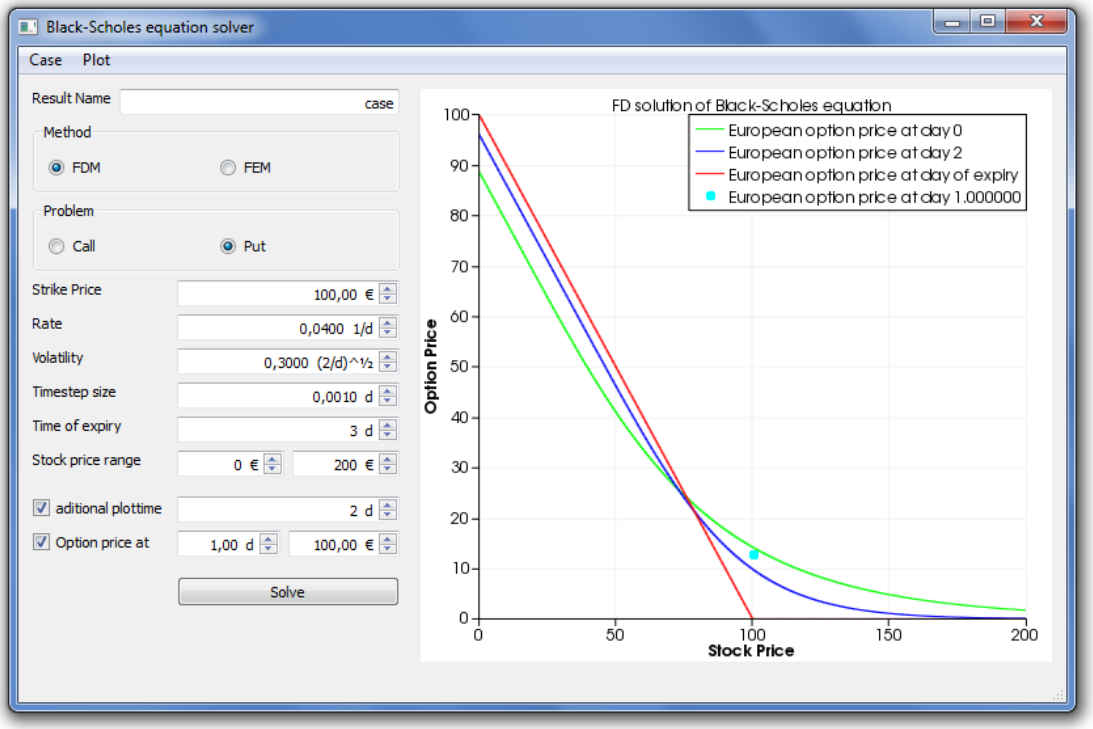

Figure 4.5: This is the *graphical user interface*.

5 Results

In this chapter, some examples solutions are presented and then the performance of the solvers is examined in terms of convergence and computation time.

5.1 Examples

The parameters used for these example cases are extracted from the book of Günther[2]. The two different implementations FDM and FEM lead to the same results. For all the following examples the expiration date of the option is given by $T = 1$ and for the time discretization $\Delta t = 0.01$ is used. The spatial discretization parameter $\Delta S = 1$ and $h = 1$ are used. Due to clearness, the option price V is shown only at $t = 0$ and $t = T$ as a function of the underlying stock price S.

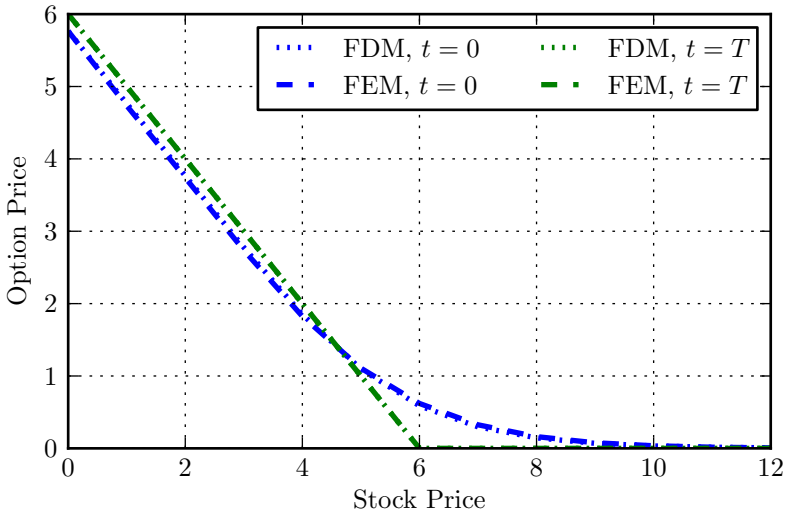

Figure 5.1: Put-option price at $t = 0$ and $t = T$ computed with FDM and FEM, using $K = 6$, $r_0 = 0.04$ and $\sigma = 0.3$.

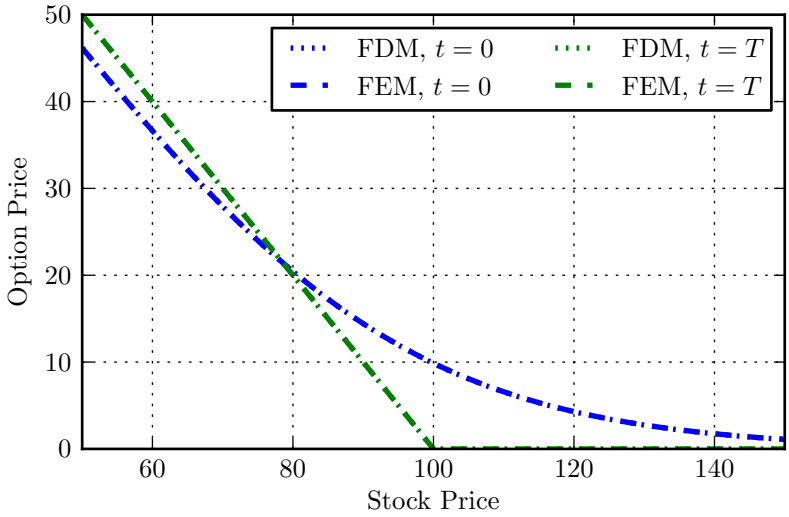

Figure 5.2: Put-option price for with $K = 100$, $r_0 = 0.04$, $\sigma = 0.3$.

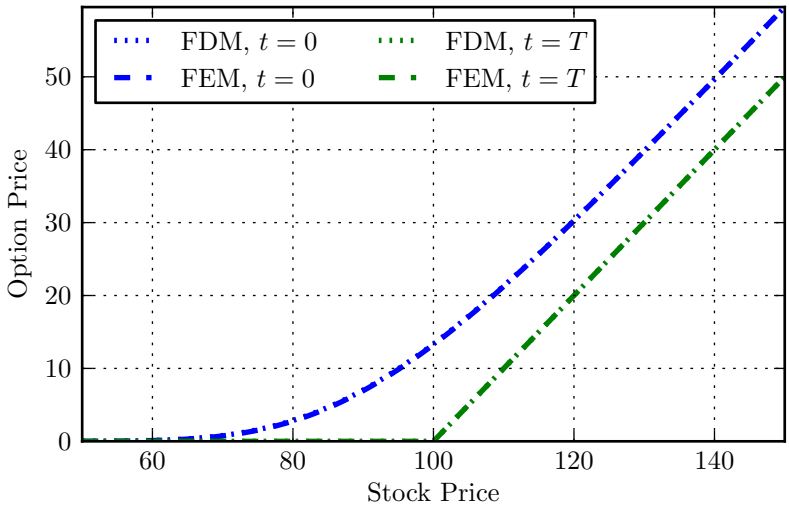

Figure 5.3: Call-option price considering $K = 100$, $r_0 = 0.04$ and $\sigma = 0.3$.

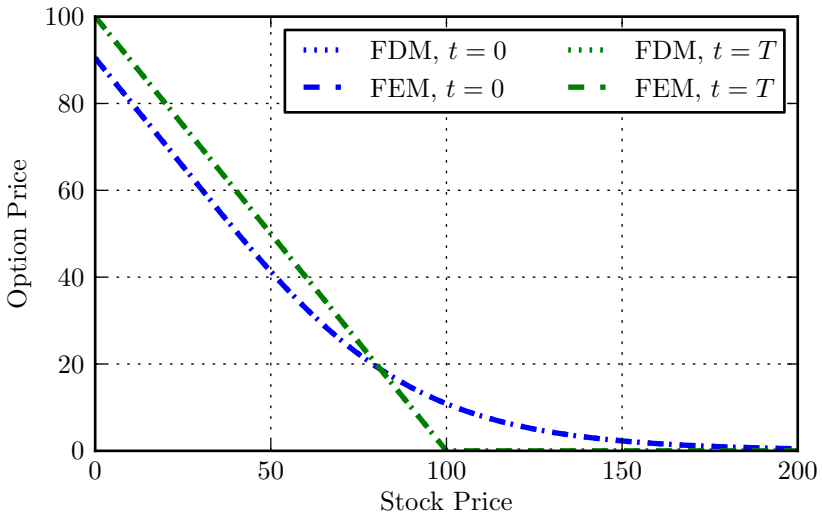

Figure 5.4: Put-option price for $K = 100$, $r_0 = 0.1$ and $\sigma = 0.4$. The result agrees nicely with the solution from the book[2]

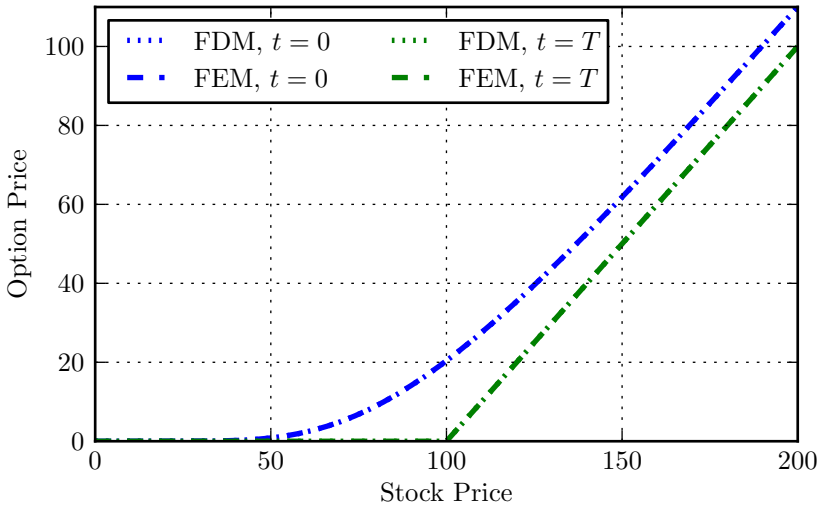

Figure 5.5: Call-option price computed with $K = 100$, $r_0 = 0.1$ and $\sigma = 0.4$. Also this result agrees with the solution from the book[2].

5.2 Convergence

Now the performance of the two different solvers will be compared. For this the convergence is examined. The analytical solution for equation (2.1) with the initial conditions (2.2) considering a call option is given in the book of Hull[5]:

$$V(S,0) = S\Phi(d_1) - K\exp(-rT)\Phi(d_2) \quad , \tag{5.1}$$

where Φ is the cumulative probability distribution function for a standardized normal distribution, and

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \quad , \tag{5.2}$$

$$d_2 = \frac{\ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \quad . \tag{5.3}$$

This solution is used to calculate the error $e(S) = V^0(S) - V(S,0)$. To measure the error, the L_∞-, L_2-norm and the H^1-semi-norm[4] are used. As test case, the same parameters are considered as in the previous section for figure 5.5.

The L^∞-norm is defined as

$$||e||_{L^\infty} := \sup_{S\in\Omega} ||e(S)|| \quad . \tag{5.4}$$

The result for this error norm for various grid sizes h can be seen in figure 5.6. *Algebraic* convergence is archived with a measured order of 1.015 for FDM and for FEM it is 1.023.

Following definition is used for the L^2-norm,

$$||e||_{L^2} := \left(\int_\Omega ||e(S)||^2 \mathrm{d}S\right)^{\frac{1}{2}} \quad . \tag{5.5}$$

For evaluating this integral, an adaptive Gauss quadrature rule is used. The result can be observed in figure 5.7. Here the orders are 0.989 for FDM and for FEM, it is 0.999.

The H^1-semi-norm is given by:

$$||e||_{H^1} := \left(\int_\Omega \left|\frac{\mathrm{d}e(S)}{\mathrm{d}S}\right|^2 \mathrm{d}S\right)^{\frac{1}{2}} \quad . \tag{5.6}$$

For evaluating this norm the analytical derivative of the solution (5.1) is used, and for integration Simpson quadrature rule is used. Which leads to the result in figure 5.8. The orders are 1.825 for FDM and 2.030 for FEM.

As overall observation one can conclude, that the error in general for FEM is slightly higher but therefor the order is a little bit better, which can be especially seen in the H^1-semi-norm.

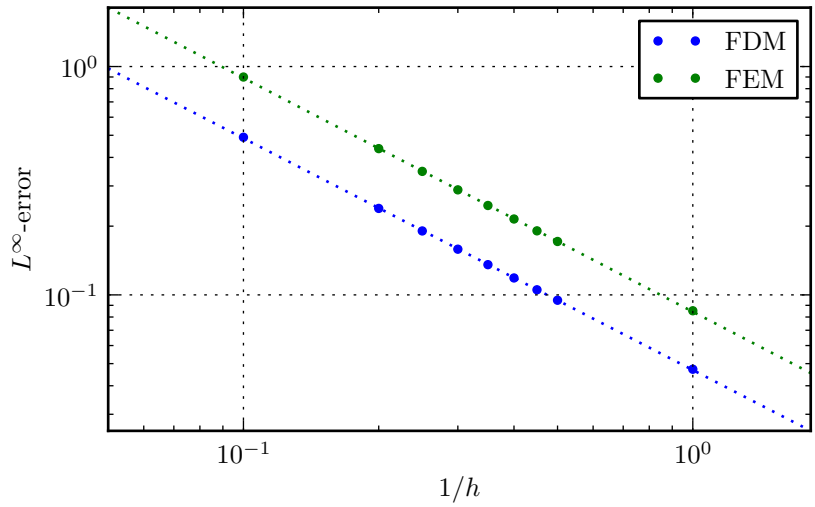

Figure 5.6: Here the dots corresponds to the measured error norm, and the dotted line corresponds to their fit. The measured order is 1.015 for FDM. For FEM, it is 1.023.

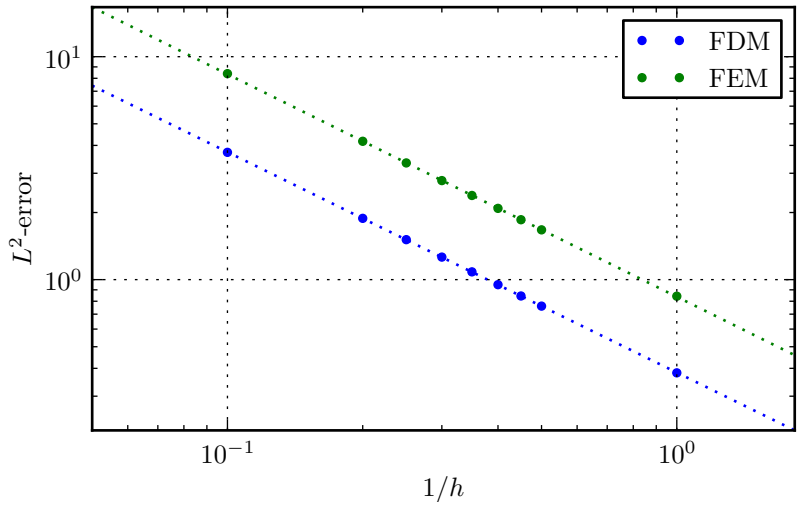

Figure 5.7: The measured order is 0.989 for FDM and for FEM, it is 0.999.

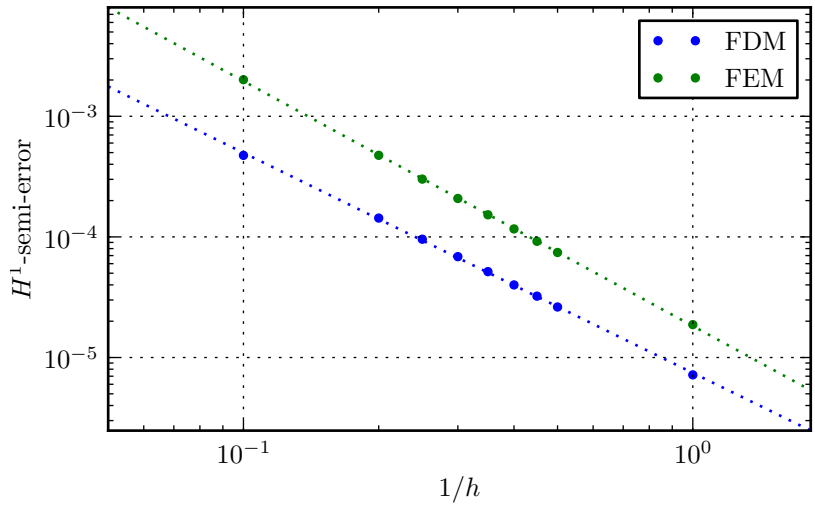

Figure 5.8: Here the measured order is 1.825 for FDM. For FEM, it is 2.030.

5.3 Timing

For further comparison of the two solvers, the computation times are compared. The same test case is used as for the convergence, with various system sizes, quantized by the number of *degrees of freedom*. For every amount of *degrees of freedom* the computation is done several times and only the minimum time is considered. The result of this can be see in figure 5.9.

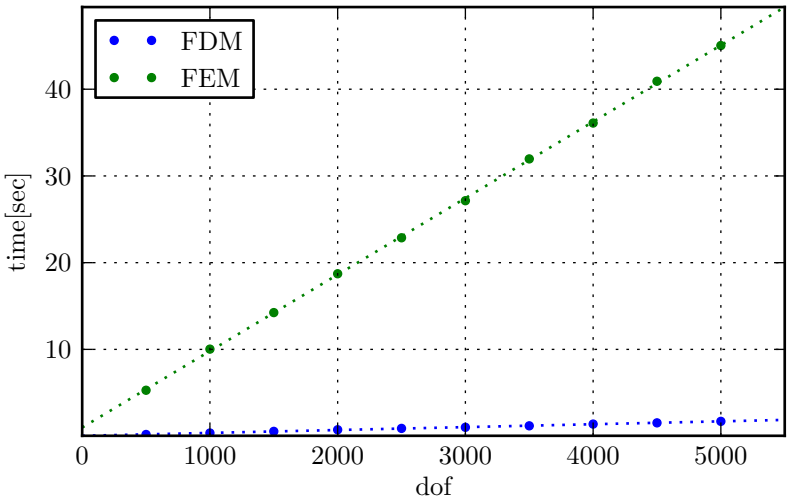

Figure 5.9: Here the computation time in seconds can be seen depending of the amount of *degrees of freedom*.

Obviously the FDM is much faster than the FEM. This has mainly two reasons. The first reason is that the assembly step in the FEM does a lot more work than the FDM. Furthermore the FDM uses a tridiagonal matrix format, which suits very good for this problem and has a very efficient implementation of solving the system of equation.

In favor of FEM, one has to mention that its code is much more flexible and can change the system solver to iterative solver at run-time. Also one has much more flexibility by changing the order of finite element. The computation time grows only linear nevertheless.

As general remark one can add, that the performance could be increased by assembling the system matrix only once and not at every time step.

6 Conclusion

In this work, we could mange to make a DIFFPACK solver to stand alone, having a nice graphical interface. For this purpose `SolverFEM` and `SolverFDM` were extended to get the same MenuSystem, which builds the interface for the GUI.

The GUI is endued with different features, for example the possibility to change the parameters, choose between the two solver methods, save and load the parameters to files. Event the result is plotted in the GUI. The plot can also be save to a JPEG file.

All this is done with the aid of QT and VTK libraries, that after being linked correctly work very smoothly together with DIFFPACK.

Performance studies were made, leading to the result that the convergence rates are almost the same for both solvers, but that the computation time for `SolverFDM` is much smaller than for `SolverFEM`. So `SolverFDM` can be seen as method of choice for these simple set ups. But if one is interested to extend the solver for example to higher dimensional asset spaces, the `SolverFEM` has many advantages.

As outlook in the future, one could also implement a progress bar, with the possibility to abort a computation. Furthermore including three dimensional visualization could be considered. Moreover computation times and efficiency could be improved. Also studies of `SolverFEM` doing p-refinement would be of interest.

Bibliography

[1] Fishcer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.

[2] Michael Günther and Ansgar Jüngel. *Finanzderivate mit Matlab. Mathematische Modellierung und Numerische Simulation.* Vieweg Verlag, Wiesbaden, 2010.

[3] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations I (2nd revised. ed.): nonstiff problems.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[4] Prof. R. Hiptmair, Prof. Ch. Schwab, Prof. H. Harbrecht, Dr. V. Gradinaru, Dr. A. Chernov, and Prof. P. Grohs. Numerical methods for partial differential equations. Course Material, March 2012.

[5] John C. Hull. *Options, Futures, and Other Derivatives with Derivagem CD (7th Edition).* Prentice Hall, 7 edition, May 2008.

[6] Hans Petter Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2 edition, 2003.