



UPPSALA  
UNIVERSITET

TVE 16 067 juni

Examensarbete 15 hp  
Juni 2016

# Discretization of the Dirac delta function for application in option pricing

---

Adam Lindell  
Håkan Öhrn



UPPSALA  
UNIVERSITET

Teknisk- naturvetenskaplig fakultet  
UTH-enheten

Besöksadress:  
Ångströmlaboratoriet  
Lägerhyddsvägen 1  
Hus 4, Plan 0

Postadress:  
Box 536  
751 21 Uppsala

Telefon:  
018 – 471 30 03

Telefax:  
018 – 471 30 00

Hemsida:  
<http://www.teknat.uu.se/student>

## Abstract

### **Discretization of the Dirac delta function for application in option pricing**

---

*Adam Lindell, Håkan Öhrn*

This paper compares two different approximations of the Dirac deltafunction used in a Fokker-Planck equation. Both methods deal with the singularity problem in the initial condition. The Dirac delta approximation, constructed in MATLAB with a method derived by Tornberg and Engquist [2], was compared to an already given method Aït-Sahalia [3]. The methods were implemented as the initial condition in the Fokker-Planck equation, e.g approximating a probability density function. In most cases Aït-Sahalia and Tornberg-Engquist were interchangeable. During specific circumstances one method was significantly more accurate than the other. Increasing the amount of time/spatial steps enhanced the differences in error while having less time/spatial steps made the difference in error converge. The Aït-Sahalia method produces slightly more accurate results in more cases.

# Contents

<b>1 Populärvetenskaplig sammanfattning</b>	<b>4</b>
<b>2 Introduction</b>	<b>4</b>
2.1 Introduction . . . . .	4
2.2 Problem description . . . . .	5
<b>3 Theory</b>	<b>5</b>
3.1 The Black-Scholes equation . . . . .	6
3.2 The Fokker-Planck equation . . . . .	6
3.3 Expected value of the option . . . . .	6
3.4 Approximating the Dirac delta function . . . . .	6
3.4.1 The Aït-Sahalia method . . . . .	6
3.4.2 Tornberg-Engquist method (as implemented by us) . . . . .	7
3.5 Implementation of Tornberg-Engquist method in script . . . . .	9
<b>4 Results and Discussion</b>	<b>9</b>
4.1 The error as a function of volatility $\sigma$ . . . . .	10
4.2 The error as function of the initial asset price $s_0$ . . . . .	16
<b>5 Conclusion</b>	<b>23</b>
<b>6 References</b>	<b>23</b>
<b>7 Appendix</b>	<b>24</b>

# 1 Populärvetenskaplig sammanfattning

På börsen kan man köpa och sälja värdepapper, till exempel aktier. Priset på aktierna fluktuerar över tid. En europeisk köpooption är ett finansiellt kontrakt mellan två parter där en part ges möjligheten att vid en bestämd tidpunkt (slutdatum, date of expiry) köpa en aktie till ett bestämt pris (strike price) av den andra parten.

När man behandlar optioner matematiskt använder man sig av någonting som kallas en avkastningsfunktion. Denna funktion beskriver hur stor vinst man gör på optionen som funktion av aktiepriset. För att beräkna vinsten simulerar man hur aktiepriset kommer att röra sig genom något som kallas Brownsk rörelse, eller slumpmässig rörelse med vissa uppmätta parametrar, som t.ex. hur mycket aktiepriset fluktuerar över en dag.

Ett sätt att estimera vad optionen ska kosta är att göra simuleringar baserade på partiella differentialfunktioner för sannolikhetsfördelningar. Detta innebär att man i simuleringen behandlar sannolikhetsfördelningar för vad chansen är att aktiepriset når ett visst värde om t.ex. en timme.

En sannolikhetsfördelning har egenskapen att arean under kurvan är 1. Ju kortare tid in i framtiden du försöker förutse desto spetsigare och smalare blir denna sannolikhetsfördelning. Denna sannolikhetsfördelning vid tidpunkt 0 är vad som kallas en Dirac delta funktion. Den har en kurva som är oändligt hög, oändligt smal och har arean 1. Dirac delta funktionen är väldigt svårt att använda numeriskt när man simulerar på grund av dess Singularitet. Problemet går att lösa på olika sätt.

Detta arbete går ut på att jämföra två sådana metoder, Aït-Sahalia och Tornberg-Engquist. I Aït-Sahalia gör man approximationer med hjälp av så kallade Hermite polynominals där man estimerar tätthetsfunktionen en bit fram i tiden medan i Tornberg- Engquist använder momentkriterier för att bygga en Dirac delta funktion vid tiden noll. Vi approximerar priset på optioner med så kallade PDEer (partiella differentialekvationer).

Det visade sig att de båda metoderna oftast är utbytbara medan under vissa förutsättningar var den ena bättre än den andra. Aït-Sahalia var överlag den bättre metoden.

## 2 Introduction

### 2.1 Introduction

In [5] a European call option is defined as a financial contract between two parties that states that a party A has the right (but not the obligation) to purchase a certain asset for a certain price at some point in the future from a party B. This asset (that may be a commodity such as gold or platinum, a stock or any number of other things) is called the underlying asset. An example of the dynamics of an option is as follows.

Party A holds 100 shares of a company. The stock of the company now lies at market price of 100 sek. Party B offers an option contract to party A that has a strike price of 120 sek, the time frame 1 year, and a small fee of 300 sek. The two parties now enter into this contract with each other. If the share reaches above the strike price, party B will buy the 100 shares from A at a price of 120

sek and thus makes a profit. Otherwise party B will not buy the shares from A since the market price is cheaper than the strike price.

Pricing an option has proven to be very difficult. One way of pricing a European call option is integrating the Fokker-Planck equation and multiplying it with the pay-off function [1]. The benefits of using this method is that you only have to solve the Fokker-Planck equation once and then you can multiply it with different pay-off functions for different options. This proves to be a very effective way of pricing European call options. A complication using this method is the singularity of the initial condition for the Fokker-Planck equation. The reason for this is that the distribution function for the value of a stock at  $t = 0$ , is a Dirac delta function placed at the stocks value at that time.

A Dirac delta function is a generalized function, or distribution on the real number line that is zero everywhere except at zero, where its infinity. It has a integral of 1 over the real entire real line [4].

The singularity of the initial condition creates computation problems that can be solved using different methods to estimate the Dirac delta function. In this paper we were given a MATLAB script pricing European call options using the Fokker-Planck equation and the Ait-Sahalia method to circumvent the singular nature of the Dirac delta function. In this paper we will compare it with another method derived by Tornberg-Engquist.

## 2.2 Problem description

The purpose of this paper is to compare two different methods of solving the singularity problem that arises at the initial condition for the Fokker-Planck equation. This requires that the Tornberg-Enquist method is to be implemented and replace the Ait-Sahalia method in a already functioning MATLAB script. The accuracy of the two different methods will then be displayed in this paper as the result.

## 3 Theory

The underlying theory stems from the Black-Scholes market model [5] with a deterministic bond and a stochastic asset with price-dynamics  $B_t$  and  $S_t$  respectively

$$dB_t = rB_t dt \quad (1)$$

$$dS_t = \mu S_t dt + \sigma S_t dT_t \quad (2)$$

where  $r$  is the risk-free interest rate,  $\mu$  is the drift and  $\sigma$  the volatility of the asset.  $S_t$  denotes the price of the stock and  $B_t$  is the riskless asset. The option has an expiration date at  $T$  and a strike price of  $K$ . We also introduce a so called pay-off function which describes the pay-off e.g. the profit of an option. It is denoted by  $\phi(s, K) = \max(s - K, 0) = (s - K)^+$  where

$$(s - K)^+ = \begin{cases} s - K, & s \geq K \\ 0, & s < K \end{cases} \quad (3)$$

We denote the value of the option today by  $u_0$  which is given by

$$u_0 = e^{-rT} E_Q(\phi(S_T, K)) \quad (4)$$

where  $E_Q$  is the risk-neutral expectation.

### 3.1 The Black-Scholes equation

Many times option pricing is done through solving the Black-Scholes equation [5]

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sigma^2 s^2 \frac{\partial^2 u}{\partial s^2} + rs \frac{\partial u}{\partial s} - ru = 0 \quad (5)$$

where we have  $u(s, T) = \phi(s, K)$  and  $s \in R^+, t < 0$ . There are different techniques to solve (5) e.g. adaptive finite differences and approximations with RBF [6]. The result gives the value for one option, the one with pay-off function  $\phi(s, K)$  but we get the value  $u$  for all values  $s > 0$ . This makes the method inefficient. The paper [1] which is the background for this paper instead uses the Fokker-Planck equation for the pricing of options.

### 3.2 The Fokker-Planck equation

The Fokker-Planck equation is originally an equation describing a moving particle exposed to random forces, resulting in a Brownian motion. If the underlying asset of the option follows the dynamic in (1) and (2) then the probability  $p(s, t)$  that the asset assumes the value  $s$  at a time  $t$  is described by the Fokker-Planck equation [1]

$$\frac{\partial p}{\partial t} - \frac{1}{2} \frac{\partial(\sigma^2 s^2 p)}{\partial s^2} + \frac{\partial(rsp)}{\partial s} = 0 \quad (6)$$

where we have  $p(s, 0) = \delta(s_0 - s)$  and  $\delta(s_0 - s)$  is the Dirac delta function.

### 3.3 Expected value of the option

To compute the expected value we can integrate over the probability function  $p(s, T)$  derived from (6) and multiply with the pay-off function (3). We then arrive at the function

$$u_0(K, T) = e^{-rT} \int_{s \in R^+} p(s, T) \phi(s, K) ds \quad (7)$$

which is the expected value of the option. The benefits of using this method is that one can solve the Fokker-Planck equation once and then use (7) to price different options with different pay-off functions.

### 3.4 Approximating the Dirac delta function

#### 3.4.1 The Aït-Sahalia method

The basic idea of the Aït-Sahalia method, introduced in [3], is to approximate  $p_x(x, \Delta t)$  from

$$dX_t = \mu_X(X_t)dt + \sigma_X(X_t)dW_t \quad (8)$$

given  $X_0 = x_0$  and using Hermite polynomials. To ensure convergence, the above equation is transformed into

$$Y \equiv \gamma(X) = \int_0^X \frac{1}{\sigma_X(u)} du \quad (9)$$

with the dynamics

$$\begin{aligned} dY_t &= \mu_Y(Y_t)dt + dW_t \\ \mu_Y(y) &= \frac{\mu_X(\gamma^{-1}(y))}{\sigma_X(\gamma^{-1}(y))} - \frac{1}{2} \frac{\partial \sigma_X}{\partial x} \end{aligned}$$

The next transformation is

$$Z \equiv \frac{Y - y_0}{\sqrt{\Delta t}}$$

where  $Z$  is close to being a  $N(0, 1)$ -variable for some fixed  $\Delta t$ . This is then used to approximate  $p_Z(z, \Delta t)$ . For the derivation and details, see [3]. The approximation for the Dirac delta function in the Fokker-Planck equation is

$$\begin{aligned} p(s, \Delta t) &= \frac{1}{\sigma s \sqrt{2\pi \Delta t}} \left( 1 - \frac{1}{2} \left( \frac{r}{\sigma} - \frac{\sigma}{2} \right)^2 \Delta t + \frac{1}{8} \left( \frac{r}{\sigma} - \frac{\sigma}{2} \right) \Delta t^2 \right) \times \\ &\quad \times \exp \left( -\frac{1}{2\Delta t} \left( \frac{\log(s) - \log(s_0)}{\sigma} \right)^2 + \left( \frac{r}{\sigma} - \frac{\sigma}{2} \right) \left( \frac{\log(s) - \log(s_0)}{\sigma} \right) \right) \end{aligned} \quad (10)$$

### 3.4.2 Tornberg-Engquist method (as implemented by us)

In [9] a moment is defined a specific quantitative measure used in both mechanics and statistics that gives information of the shape of a function on a set of points. In case of mechanics where the the points represent a mass, the zero moments is the total mass, the second moment is the rotational inertia. In the case where the points represent a probability density the zero moment represents the total probability, the first the mean. If centred around the first the second moment is the variance and the third the skewness. The n-th moment of a real value continuous function  $f(x)$  about a value  $c$  is

$$M_n = \int_{-\infty}^{\infty} (x - c)^n f(x) dx \quad (11)$$

The Tornberg-Enquist method [2] constructs an estimation of the Dirac delta function that satisfies these moment conditions. In the simplest case in which  $s_0$  is placed on a grid point (the computational grid) one can make the estimation that a one dimensional  $\delta_\epsilon$  that has compact support in  $[-\epsilon, \epsilon]$  where  $\epsilon = mh$ , satisfies  $q$  discrete moment conditions of

$$M_r(\delta_\epsilon, \bar{x}, h) = h \sum_{j=-\infty}^{\infty} \delta_\epsilon(x_j - \bar{x})(x_j - \bar{x})^r = \begin{cases} 1, & r = 0 \\ 0, & 1 \leq r < q \end{cases} \quad (12)$$

for any  $\bar{x} \in \mathbb{R}$ , where  $x_j = jh, h > 0, j \in \mathbb{Z}$ . The method requires that  $2\epsilon \geq qh$ . The first moment condition ( $r = 1$ ) makes sure the delta functions mass is equal

to 0 while the higher moment conditions are needed when the delta function is multiplied by a non-constant function. The method gives an error

$$E = \left| h \sum_{j=-\infty}^{\infty} \delta_\epsilon(x_j - \bar{x})(x_j - f(x_j)) - f(\bar{x}) \right| \leq Ch^q \quad (13)$$

For further information on how the size of error was calculated see [2].

In the case where the Dirac delta function is placed at  $\bar{x} = x_p + \alpha h$  where  $x_p$  is the grid point closest to the left of the Dirac delta function and  $\alpha \in [0, 1]$ , the discrete moments as seen in (11) can be expressed as

$$M_r(\delta_j, x_p, h, \alpha) = h \sum_{j=k-(m-1)}^{k+m} \delta_j(x_j - x_p - \alpha)(x_j - x_p - \alpha)^r \quad (14)$$

where  $2m$  is the support of  $\delta_\epsilon$ .

This problem is difficult to solve. To simplify, we set the Dirac delta at  $\bar{x} = 0 + \alpha h$ .

$$M_r(\delta_\epsilon, x_p, h, \alpha) = h \sum_{j=-(m-1)}^m \delta_\epsilon(x_j - \alpha h)(x_j)^r \quad (15)$$

To solve (15) we use that

$$f(x_0) = \int_{-\infty}^{\infty} f(x) \delta(x - x_0) dx \approx h \sum f(x) \delta(x - x_0) \quad (16)$$

setting  $x_0$  to  $\alpha h$  where  $\alpha \in [0, 1]$ . The Dirac delta approximation has support over  $2m$  grid points.

$$f(h\alpha) = h \sum_{j=-m+1}^m \delta_j f(jh) \quad (17)$$

$\delta_j$  is the value of the Dirac delta estimation at grid point  $x_j$ . Setting  $f(x) = x^r$  and cancel out  $h^r$  on both sides we get

$$\alpha^r \approx h \sum_{j=-(m-1)}^m \delta_j j^r \quad (18)$$

for  $0 \leq r < q$  where  $q$  is the support of the  $\delta_j$ . This formula is used to estimate the Dirac delta function in cases where  $s_0$  is placed between two grid points.

**Example** Say one wants to create a Dirac delta function with a support over 6 grid points at two different points along the grid. One Dirac delta that is on one of the grid points at  $s_0 = 0$ . The second Dirac delta function is placed between two grid points with a displacement  $s_0 = \alpha h$  from 0. This would create a linear system of equations  $Ay = e_1$  where in the case of the delta function at  $s_0 = 0$

$$y = (y_{-2}, \dots, y_3)^T, \quad e_1 = \left( \frac{1}{h}, 0, \dots, 0 \right)^T$$

and

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 & 3 \\ 4 & 1 & 0 & 1 & 4 & 9 \\ -8 & -1 & 0 & 1 & 8 & 27 \\ 16 & 1 & 0 & 1 & 16 & 81 \\ -32 & -1 & 0 & 1 & 32 & 243 \end{bmatrix}$$

whereas at the case when  $s_0 = \alpha h$  the  $\mathbf{e}_1 = \frac{1}{h}(1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5)^T$

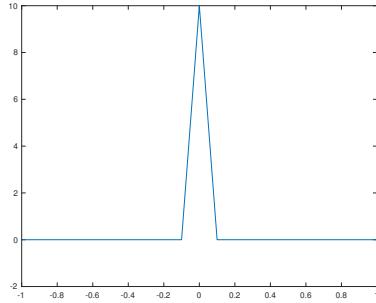


Figure 1: Dirac delta function centered at a grid point

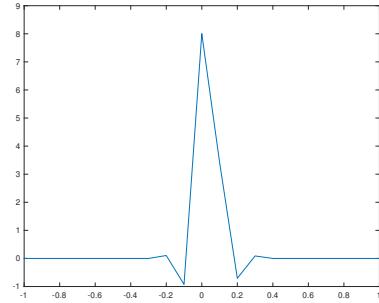


Figure 2: Dirac delta function centred between grid points

The Dirac delta functions estimations in Figure 1 and 2 are simulated with large spatial steps to illustrate their differences. Note the difference in the two plots when one is centred at a grid point and the other is not.

### 3.5 Implementation of Tornberg-Engquist method in script

In the original script solving the Fokker-Planck equation (6) a closed form solution using the Aït-Sahalia method was used to take the first time-step. The remaining time-steps used Backward Differentiation formula of order 2 (BDF-2) [7] and least-squares Radial Basis Function (RBF) [8] approximation in space. As the focus of this paper is to compare two solutions to the singularity problem in the initial condition these methods will not be explained further.

To implement the Tornberg-Engquist method the Aït-Sahalia solution was removed and replaced with the new approximation of the Dirac delta. Then the system was modified to start at time  $t_0$  instead of the original  $t_0 + \Delta t$  where  $\Delta t$  is the size of the first Aït-Sahalia time-step.

## 4 Results and Discussion

To get an understanding of how well the Tornberg-Engquist method compares to the Aït-Sahalia method let us conduct some tests. Let  $P$  be the approximation of the probability density function in (6) obtained using the RBF method with different support for the Tornberg-Engquist method as well as the Aït-Sahalia method. We set the  $P_{\text{exact}}$  to be the analytical solution to the Fokker-Planck equation (6)

$$P_{exact}(x, t) = \frac{1}{\sqrt{2\pi}t\sigma x} \exp\left(\frac{(\log(x/x_0) - (r - \sigma^2/2)t)^2}{2\sigma^2 t}\right) \quad (19)$$

The plots display the maximum error  $\max|P - P_{exact}|$  of the approximated solution compared to the analytical solution. The risk free interest rate is set to  $r = 0.05$ . The number of time and space discretization steps is set to 100 unless specified otherwise. The size of the Aït-Sahalia time-step is 0.01. All simulations in this paper use  $3N$  least squares-evaluation points which is more accurate than solving using collocation. These numerical simulations were run on a Macbook pro OS X 2.4 GHz Intel core i5 processor with 4 Gb ram.

#### 4.1 The error as a function of volatility $\sigma$

In this section we plot the error in the distribution function as a function of  $\sigma$  fixing  $s_0$  at 1, 1.5 and 2.

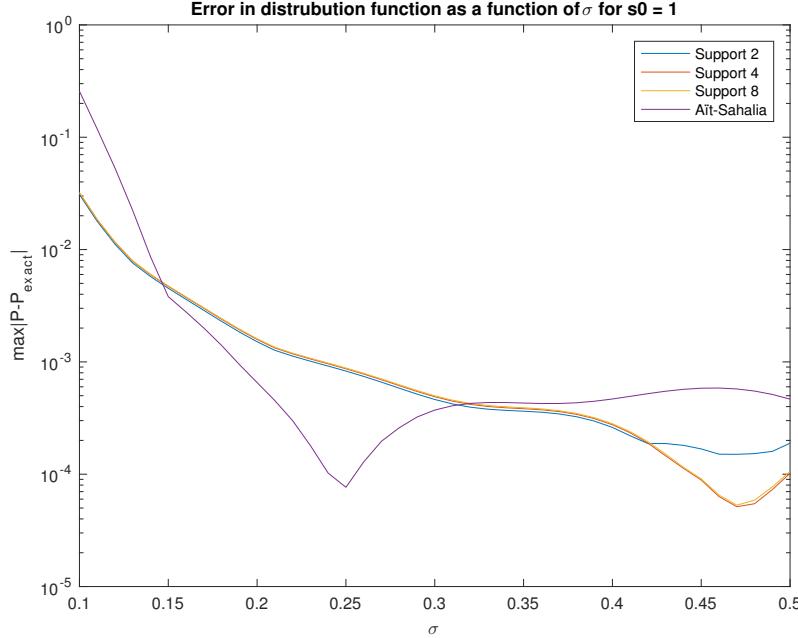


Figure 3: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1

In Figure 3 we can see that at  $0.1 \leq \sigma \leq 0.15$  the Tornberg-Engquist method surpasses the Aït-Sahalia and then again at  $\sigma > 0.3$ . It seems like a larger support is needed for very large volatilities.

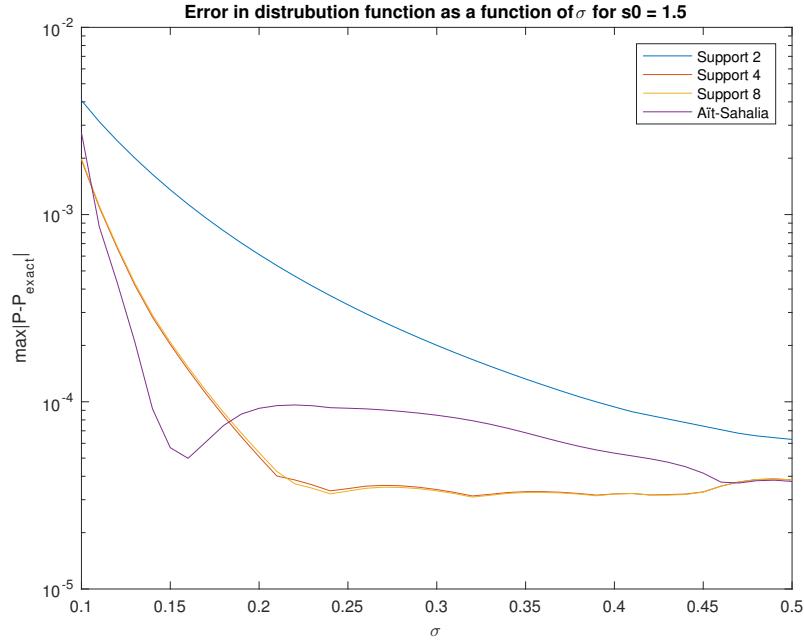


Figure 4: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1.5

In Figure 4 the Dirac delta approximations with support 4 and 8 grid points is better than the Aït-Sahalia method for  $0.19 \leq \sigma \leq 0.46$  and then approximately the same after that. The  $\delta_\epsilon$  with support of 2 grid points is consistently worse than the others. Aït-Sahalia's best value is as good as the Tornberg-Engquist, but the Dirac delta approximation is better for a larger range of  $\sigma$ .

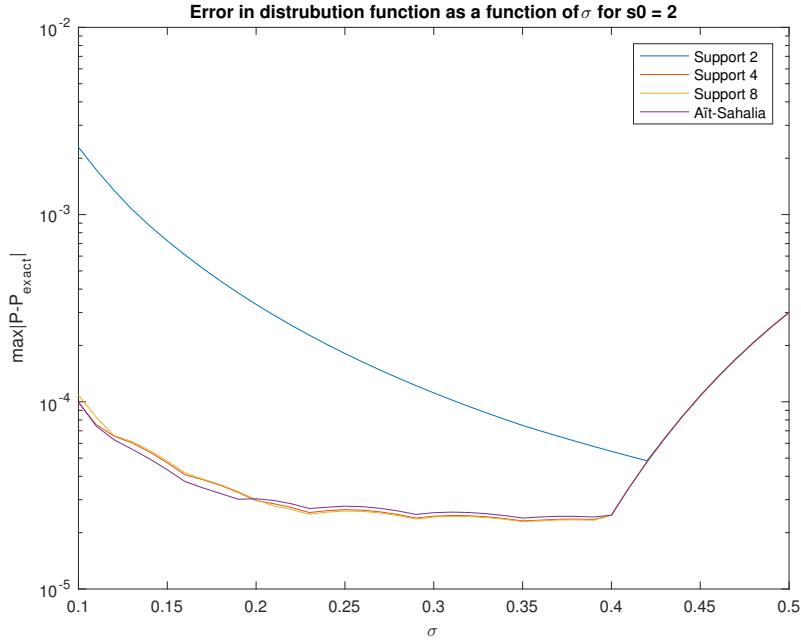


Figure 5: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 2.

Figure 5 shows that for  $s_0 = 2$  the two methods perform about equally well with the exception of the approximation with a support of 2 grid points. Both methods have their lowest error at a point  $\sigma = 0.4$ . In this case the methods are interchangeable. As  $\sigma$  and  $s_0$  increases the difference in error between the two methods tend to converge as can be seen in Figure 5, 4 and 3. For Aït-Sahalia, the first time step  $\Delta t$  taken was optimized for  $\sigma = 0.2$ . In Figure 3 the method performs best at  $\sigma = 0.25$  which is around that area. In Figure 4 with  $s_0 = 1.5$  the peak is at 0.15 and Figure 5 with  $s_0 = 2$  the method has none. Had the initial time step been implemented as a function of  $s_0$  and  $\sigma$  the method might be better over a larger range of values. Such a implementation is not part of this paper but should be done in the future to make the method better for a larger range of values for  $\sigma$  and  $s_0$ .

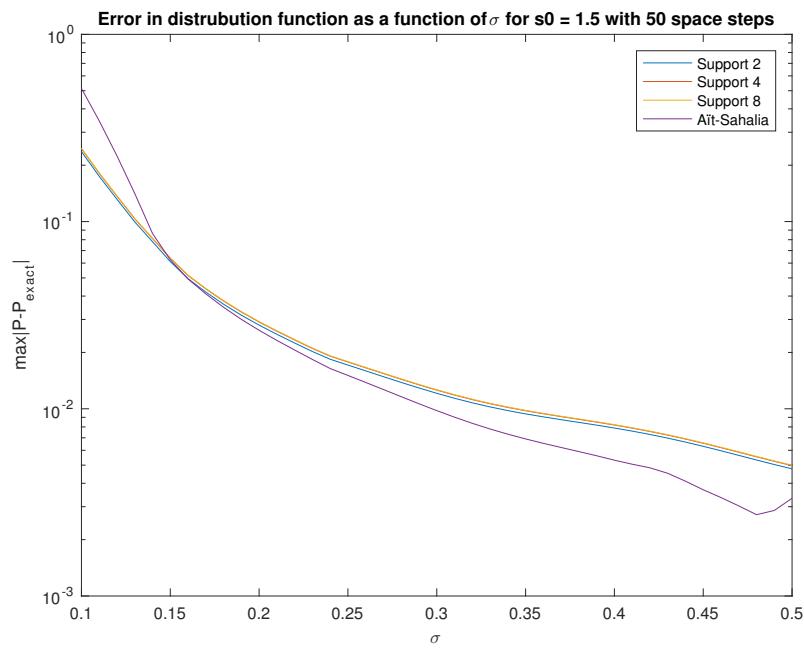


Figure 6: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1.5 and with 50 spatial steps.

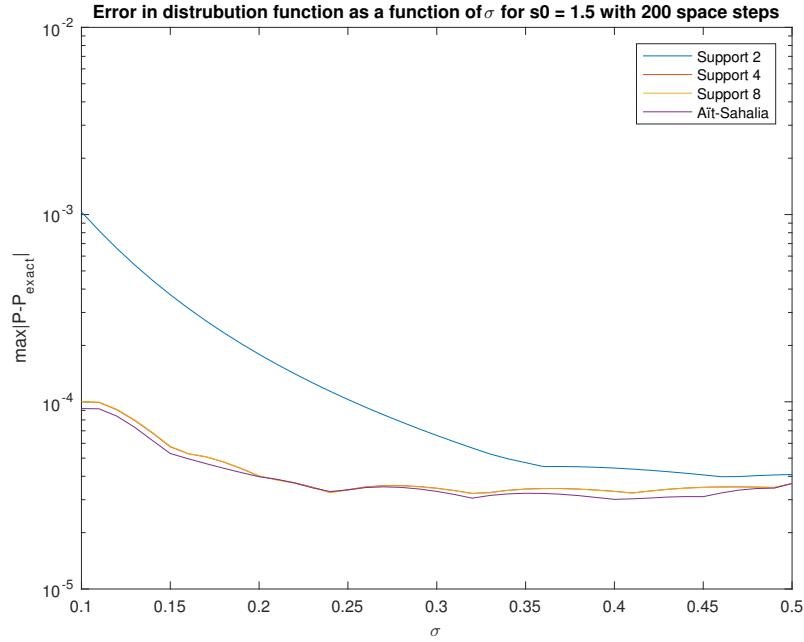


Figure 7: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1.5 and with 200 spatial steps.

To see how the different methods depend on the amount of spatial steps we test the methods using half the amount as well as twice as many (e.g 50 and 200). At 50 spatial steps the Ait-Sahalia method is consistently better for  $\sigma > 0.15$  and when the spatial steps is increased to 200 the two methods' performance is similar, apart from the Dirac delta estimation with support of 2 grid points.

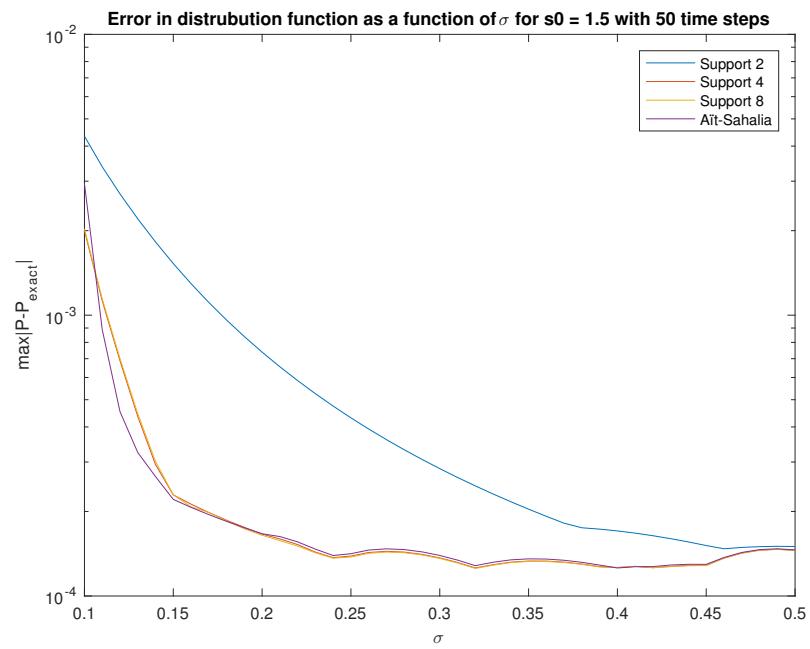


Figure 8: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1.5 with 50 time steps.

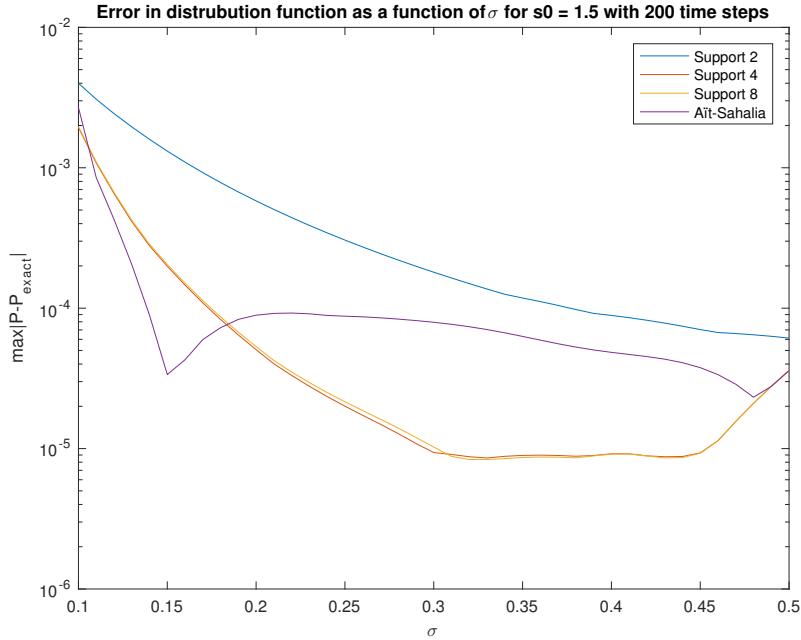


Figure 9: Plot showing how the error of the function depends on different volatilities  $\sigma$  when the initial asset price  $s_0$  is fixed to 1.5 with 200 time steps.

The same test was done for different amount of time steps. In Figure 8 where the time steps is decreased to 50 the Aït-Sahalia's and Tornberg-Engquist's performance is similar. However in Figure 9 where the time steps are increased to 200 Tornberg-Engquist surpasses Aït-Sahalia within  $0.19 < \sigma < 0.47$ . The result in Figure 9 is quite similar to Figure 4. The reason for this could be that the Tornberg-Engquist method becomes better when refined while Aït-Sahalia is not affected.

#### 4.2 The error as function of the initial asset price $s_0$

In this section of the result we plot the error in the distribution function as a function of  $s_0$  while fixating  $\sigma$  at 0.1, 0.3 and 0.5 respectively. We also vary the amount of time steps and well as spatial steps.

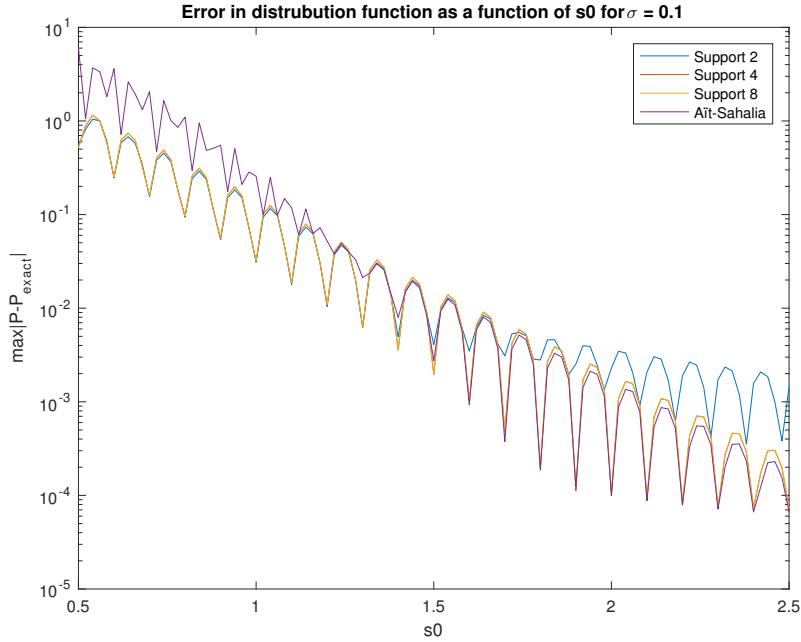


Figure 10: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.1$

Figure 10 shows that for a fixed  $\sigma = 0.1$  the Tornberg-Engquist method is more accurate for  $s_0 < 1.4$ . After that point the two methods have approximately the same error. The error also seems to decrease as the  $s_0$  increases. The Tornberg-Engquist with support of 2 grid points becomes less accurate compared to the others as  $s_0$  increases. Both methods have an error of oscillating nature for all values of  $\sigma$ . What causes oscillation is how  $s_0$  is placed on the computational grid. The Aït-Sahalia exhibits poor performance at very low  $\sigma$  and  $s_0$ .

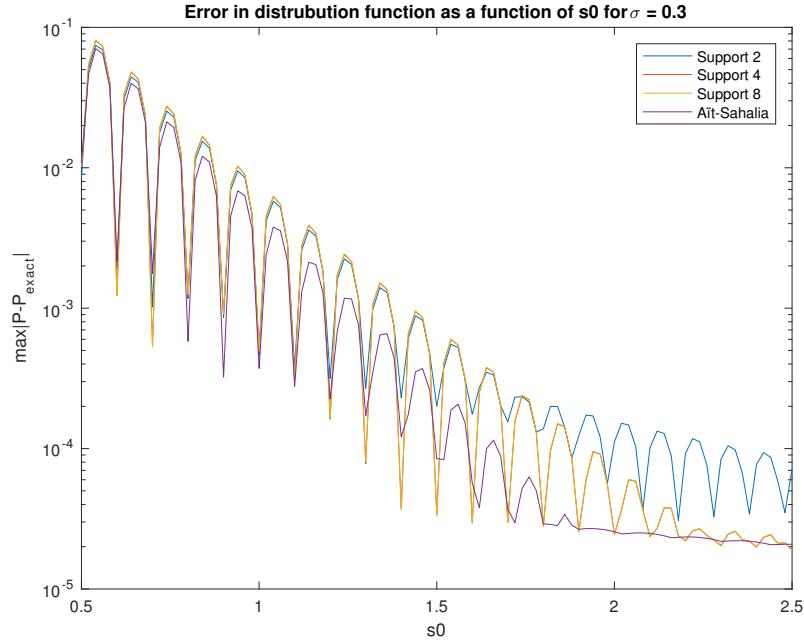


Figure 11: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.3$

In Figure 11 we see that between approximately  $s_0 = 0.5$  and  $s_0 = 2.2$  Aït-Sahalia has a smaller error than Tornberg-Engquist, with any of the given support sizes. The approximation using Aït-Sahalia stabilizes at about  $s_0 = 1.8$  while Tornberg-Engquist has yet to do so at  $s_0 = 2.5$

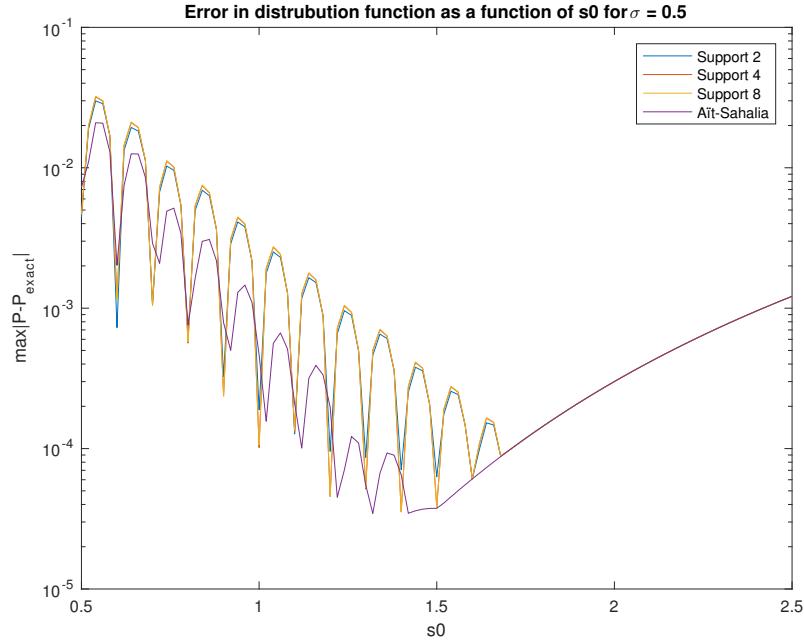


Figure 12: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.5$

For  $\sigma = 0.5$  Aït-Sahalia method is consistently better until  $s_0 = 1.5$  where the two methods are indistinguishable. At low values for  $\sigma$  and  $s_0$  the Tornberg-Engquist seem to perform better and at high  $\sigma$  and low  $s_0$  Aït-Sahalia performs better as can seen in Figure 10 and 12.

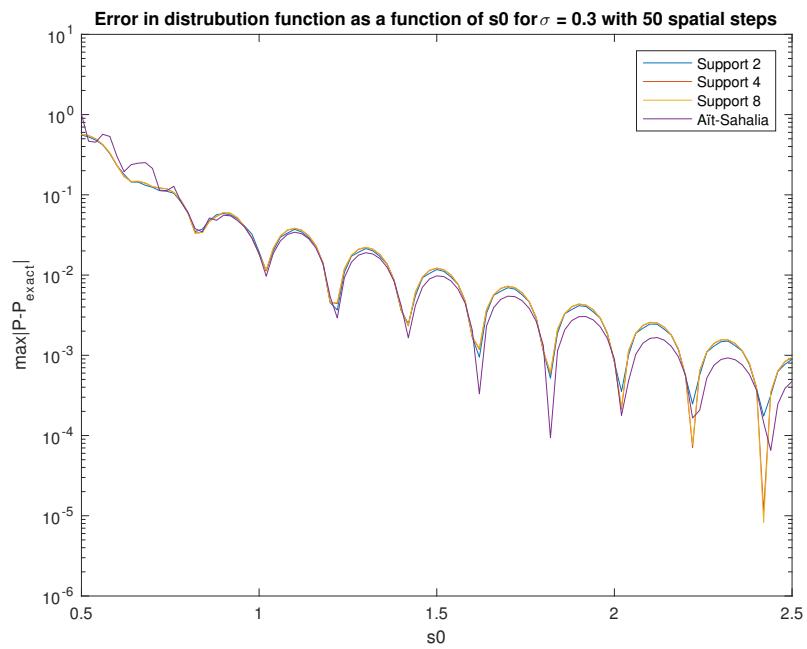


Figure 13: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.3$  using 50 steps in space

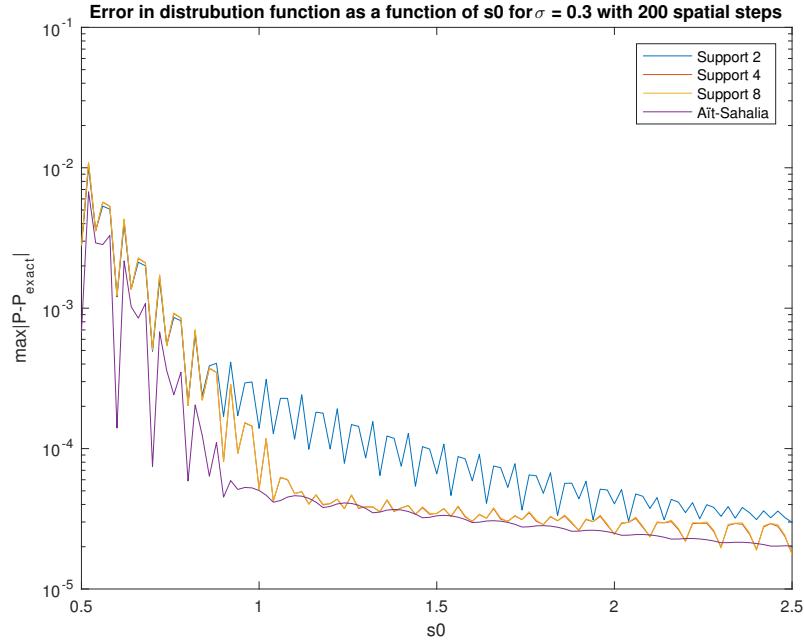


Figure 14: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.3$  using 200 steps in space

When lowering the amount of time/spatial steps the difference in error between the two methods becomes less while increasing the amount of time/spatial steps increases the difference in error. Generally one can say that as  $s_0$  increases the error in the two methods decreases, however this is the opposite when  $\sigma = 0.5$ , then the smallest error occur when  $s_0 = 1.5$  see Figure 12.

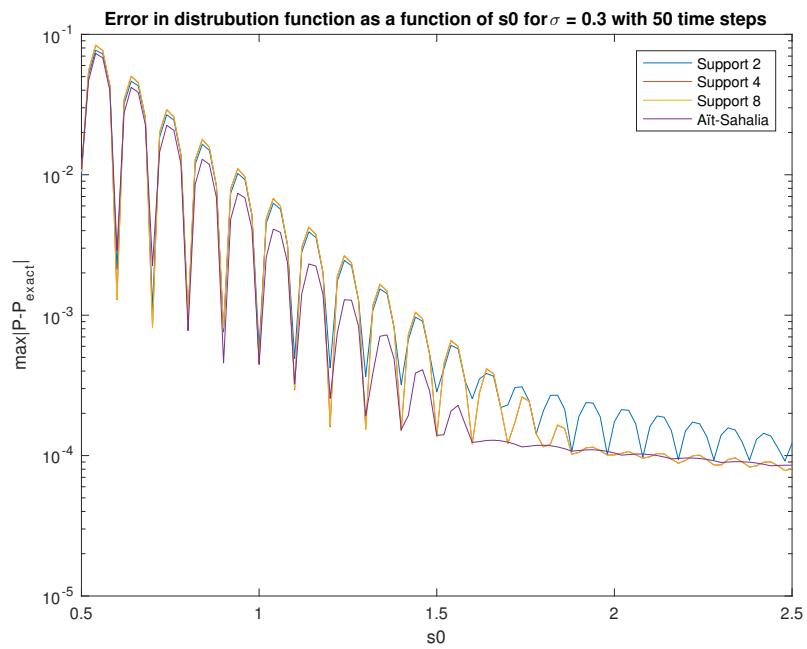


Figure 15: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.3$  using 50 steps in time

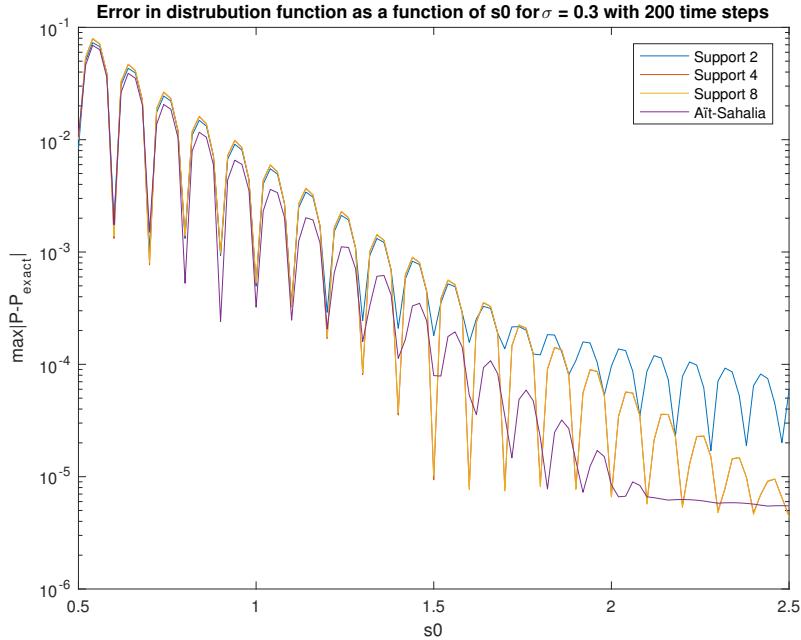


Figure 16: Plot showing the error as a function of the initial asset price with a fixed volatility  $\sigma = 0.3$  using 200 steps in time

## 5 Conclusion

The difference in error between the two methods is usually so small that the methods become interchangeable. However at some points one method gives better accuracy of some significance. At very low values for  $\sigma$  and  $s_0$  the Tornberg-Engquist perform better and at high  $\sigma$  and low  $s_0$  Aït-Sahalia performs better as seen in Figure 10 and 12. When  $\sigma$  and  $s_0$  increases the difference in error between the two methods tend to converge as can be seen in Figure 5. Increasing the amount of time/spatial steps enhanced the differences in error for the methods while having less time/spatial steps had the difference in error converge. If only one method could be used it should be Aït-Sahalia since it performer better than Tornberg-Engquist in more scenarios.

## 6 References

- 1.J Rad, J Höök, E Larsson, L von Sydow *Forward option pricing using Gaussian RBFs* (not yet published).
- 2.A Engquist, B Tornberg, *Numerical approximations of singular source terms in differential equations*, (2004).
3. Y Aït-Sahalia, *Maximum likelihood estimation of discretely sampled diffusion: A closed-form approximation approach*, *Econometrica*, 70(1):233-262, 2002.

4. H Sollervall, B Styf, *Transformteori för ingenjörer* 3.upplaga, 9789144022000 2006
5. F Black, M Scholes, *The pricing of options and corporate liabilities*, J. Polit. Econ., 81:637–654, 1973.
6. L. von Sydow, L. J. Höök, E. Larsson, E. Lindström, S. Milovanović, J. Persson, V. Shcherbakov, Y. Shpolyanskiy, S. Sirén, J. Toivanen, J. Waldén, M. Wiktorsson, J. Levesley, J. Li, C. W. Oosterlee, M. J. Ruijter, A. Toropov, Y. Zhao, BENCHOP-The BENCHmarking project in Option Pricing Int. J. Comput. Math., 92 (2015), pp. 2361-2379.
7. E Hairer, S Nørsett, G Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems.*, Solving Ordinary Differential Equations. Springer, 2008.
8. E Larsson, S Gomes, *A least squares multi-level radial basis function method with applications in finance*. manuscript in preparation, 2015.
9. A Spanos, *Probability Theory and Statistical Inference*, Cambridge University Press, ISBN 0-521-42408-9, 1999.

## 7 Appendix

```

1
2  function [k,beta0,beta1,beta2]=BDF2coeffs(T,M)
3  %
4  % T is the final time, M is the number of timesteps.
5  % beta0 is constant throughout the time-stepping.
6  % betal and beta2 are vectors with coefficients for the M steps
7  %
8  c(1)=1;
9  k(1)=1;
10 for step=2:M
11     omega(step,1)=c(step-1) - 0.5 + 0.5*sqrt(4*c(step-1)^2+1);
12     k(step,1) = omega(step)*k(step-1);
13     c(step,1) = (1+omega(step))/(1+2*omega(step));
14 end
15 %
16 % Scale the steps to fit the final time
17 %
18 k = T/sum(k)*k;
19 %
20 % Compute the coefficients for the method. (Initial values are correct.)
21 %
22 beta0=k(1);
23 betal = (1+omega).^2./(1+2*omega);
24 beta2 =      omega.^2./(1+2*omega);

1 function f=BDF2rhs(step,betal,beta2,Ael,Aem,lambda_1,lambda_2,mu_1,mu_2);
2 %
3 % For the first step lambda_1 is supposed to be u^0, and for the second
4 % step, lambda_2 = u^0. The rest of the steps are normal.
5 %
6 if (step == 1)
7     f = betal(step)*lambda_1;
8 elseif (step == 2)
9     f = betal(step)*(Ael*lambda_1+Aem*mu_1) - beta2(step)*lambda_2;
10 else

```

```

11     f = [Ael Aem]* (beta1(step)*[lambda_1;mu_1]-beta2(step)*[lambda_2;mu_2]);
12 end

1 %
2 % Initial condition for the CIR model
3 %
4 function y=CIRInitialmm(x,t1, x0, t0, a,b,sigma)
5 y = ModelU22(x,x0,t1,[0,-a, sigma]);

1 function [Q,R,Lf,Uf,P,Cem,Abl,Abm,Ael,Aem,Esl,Esm,Albml,Albmm]= ...
2     BMLeastSquaresMatrices(method,a_p,b_p,sigma_p,beta0,phi,ep,xc,xb,xe,xs)
3 %
4 % Assuming that L is a cell-vector
5 %
6 L = length(method);
7 d = size(xc{1},2);
8 Ne = size(xe,1);
9 %
10 % Form some coefficients for the operator (same for all ell)
11 %
12 sig2=sigma_p*sigma_p';
13 for k=1:d
14     % NOTE: Making S sqrt form the start
15     S{k}=spdiags(sqrt(xe(:,k)),0,Ne,Ne);
16     I{k}=speye(Ne);
17 end
18 %
19 % Compute matrices for each level with the appropriate method
20 %
21 Albml=[]; Albmm=[]; % Default in case not needed
22 for ell = 1:L
23
24     N = size(xc{ell},1);
25     if (iscell(xb))
26         Nb = size(xb{ell},1);
27     else
28         Nb = size(xb,1);
29     end
30 %
31 % Form all the RBF-matrices that are needed
32 %
33 if (strcmp(method{ell}, 'qr'))
34     if (d==1)
35         error('QR not implemented')
36         [A,Psi] = RBFQRmat('0',[xe;xb],xc{ell},ep(ell));
37         A1{1} = RBFQRmat('x',xe,Psi);
38         A2{1,1} = RBFQRmat('xx',xe,Psi);
39         E = RBFQRmat('0',xs,Psi);
40     elseif (d==2)
41         tol = 2;
42         error('QR not implemented')
43
44     [A,Psi] = RBFQRmat_2D('1',[xe;xb{ell}],xc{ell},ep(ell),tol);
45     op = {'x','y','xx','xy','yy'};
46     B = RBFQRmat_2D(op,xe,Psi);
47     A1 = {B{1},B{2}};
48     A2{1,1} = B{3}; A2{1,2} = B{4}; A2{2,1} = B{4}; A2{2,2} = B{5};
49     clear B
50     E = RBFQRmat_2D('1',xs,Psi);
51     %

```

```

52      % Crossmatrices for evaluting this solution at all finer grids
53      %
54      for m=ell+1:L
55          Alb = RBFQRmat_2D('1',xb{m},Psi);
56          Albml{m}{ell} = Alb(:,1:N-Nb);
57          Albmm{m}{ell} = Alb(:,N-Nb+1:N);
58      end
59  else
60      error('RBF-QR for d>2 is not supported yet')
61  end
62 elseif (strcmp(method{ell}, 'dir'))
63     r_ec = xcdist(xe, xc{ell}, 1);
64     if (iscell(xb))
65         r_bc = xcdist(xb{ell}, xc{ell}, 1);
66     else
67         r_bc = xcdist(xb, xc{ell}, 1);
68     end
69     A = RBFmat(phi, ep(ell), [r_ec(:,:,1) ; r_bc(:,:,1)], '0');
70     for k=1:d
71         A1{k} = RBFmat(phi, ep(ell), r_ec, '1', k);
72         A1b{k} = RBFmat(phi, ep(ell), r_bc, '1', k);
73         for j=1:d
74             if (j==k)
75                 op = '2';
76                 dim = j;
77             else
78                 op = 'm2';
79                 dim = [j k];
80             end
81             A2{j,k} = RBFmat(phi, ep(ell), r_ec, op, dim);
82         end
83     end
84     r_sc = xcdist(xs, xc{ell}, 0);
85     r_sc2 = xcdist(xs, xc{ell}, 1);
86     E = RBFmat(phi, ep(ell), r_sc, '0');
87     %
88     % Crossmatrices for evaluting this solution at all finer grids
89     %
90     if (d>1)
91         for m=ell+1:L
92             r = xcdist(xb{m}, xc{ell}, 0);
93             Albml{m}{ell} = RBFmat(phi, ep(ell), r(:,1:N-Nb), '0');
94             Albmm{m}{ell} = RBFmat(phi, ep(ell), r(:,N-Nb+1:N), '0');
95         end
96     end
97 else
98     error('The method has to be dir or qr');
99 end
%
100 %
101 % Form the specific Black-Scholes related matrices
102 %
103 Ael{ell} = A(1:Ne, 1:N-Nb);
104 Aem{ell} = A(1:Ne, N-Nb+1:N);
105 %
106 % Flux conserving boundary condition section:
107 %
108 Abl{ell} = A(Ne+1:Ne+Nb, 1:N-Nb);
109 %size(A1{1})
110 %
111 % Flux conserving boundary condition on the lhs
112 Abl{ell}(2, :) = (a_p*( xb(2) - b_p ) + sig2*xb(2) )*Abl{ell}(2, :) ...
113     + sig2/2*(xb(2).^2)*A1b{1}(2, 1:N-Nb);

```

```

114 % Condition for conservation of probability, lhs
115 % Abl{ell}(2,:) = 1- normcdf(0, xc{ell}(1:N-Nb), 1/sqrt(2)/ep(ell));
116
117
118 Abm{ell} = A(Ne+1:Ne+Nb,N-Nb+1:N);
119 % Flux conserving boundary condition on the rhs
120 Abm{ell}(2,:) = (a_p*( xb(2) - b_p ) + sig2*xb(2) )*Abm{ell}(2,:) ...
121     + sig2/2*(xb(2).^2)*A1b{1}(2,N-Nb+1:N);
122
123 % Condition for conservation of probability, rhs
124 %Abm{ell}(2,:) = 1- normcdf(0, xc{ell}(N-Nb+1:N), 1/sqrt(2)/ep(ell));
125
126
127 % End flux section
128 ep2(ell) = ep(ell).^2;
129
130
131 Esl{ell} = E(:,1:N-Nb);
132 Esm{ell} = E(:,N-Nb+1:N);
133 %Esl{ell} = (0.5/ep2(ell))*E(:,1:N-Nb);
134 %Esm{ell} = (0.5/ep2(ell))*E(:,N-Nb+1:N);
135 %Esl{ell} = Esl{ell} - (sqrt(pi)/(2*sqrt(ep2(ell))))*r_sc2(:,1:N-Nb,2).* (1-erf(sqrt(ep2(ell)))*r_s
136 %Esm{ell} = Esm{ell} - (sqrt(pi)/(2*sqrt(ep2(ell))))*r_sc2(:,N-Nb+1:N,2).* (1-erf(sqrt(ep2(ell)))*r_
137
138 % Killing term
139 C = (a_p+sig2)*A(1:Ne,:);
140 for k=1:d
141     for j=1:d
142         % Diffusion coeff, sigma^2 x /2
143         % Found bug 141007, 0.2*sig2 shoudl be 0.5*sig2
144         % Added a 0.5 which should not be there
145         C = C + 0.5*sig2(j,k)*(S{j}.^2)*(S{k}.^2)*A2{j,k};
146         % C = C + 0.5*sig2(j,k)*S{k}*A2{j,k};
147
148     end
149     % Drift coeff for the CIR model (expanded): ( a*(x-b) + sigma^2 )
150     % C = C - 0.5*( a_p*(S{k}-b_p) + sig2)*A1{k};
151     C = C + ( -a_p*(b_p*I{k} - S{j}*S{k}) + 2*sig2*S{j}*S{k} )*A1{k};
152 end
153 if (nargout==15 & ell==L)
154     BV = C;
155 end
156 C = A(1:Ne,:)-beta0*C;
157 Cel{ell} = C(:,1:N-Nb);
158 Cem{ell} = C(:,N-Nb+1:N);
159
160 clear A A1 A2 E C
161 %
162 % LU-factorize the small square matrix
163 %
164 [Lf{ell},Uf{ell},P{ell}] = lu(Abm{ell});
165 %
166 % Form the Schur complement
167 %
168 Sel = Uf{ell}\(Lf{ell}\)\(P{ell}*Abl{ell}));
169 Sel = Cel{ell}-Cem{ell}*Sel;
170 %
171 % QR-factorize the Schur complement
172 %
173 [Q{ell},R{ell}]=qr(Sel,0);
174
175 clear Sel

```

```

176
177 end

1 function [maxnrm,u,v,xs,error,xls,res,time,timerror,timeres,ao,mem]= ...
2     BMMain(a_p,b_p,sigma_p,T,x0,start,ep,M,N,Ns,rg,nodetype,method,Nls)
3 %
4 % Main program for all the different variants of the method
5 %
6 % phi is the RBF 'mq', 'gs',...
7 % ep(1:L) is the (constant) shape parameter for each grid
8 % Note that for a one grid method ep is just given as a scalar
9 % M is the number of time steps
10 % N(1:L) is the number of center points for each grid
11 % Ns is the number of uniform points where the solution is evaluated
12 % rg(1:2) is the range over which we are computing, typically [0 4]
13 % ctype is 'cheb' or 'uni' for the distribution of node points
14 % etype has the same function for the points where equations are enforced
15 % method{1:L} is 'qr' or 'dir' for RBF-QR or RBF-Direct for each grid
16 % note only the lsml version allows different
17 % show is 'yes' or 'no' and tells if plots of the errors should be made
18 % col is the color to use for the error plots.
19 % Nls is _optional_. If present it indicates that least squares should be used.
20 %-----
21 %
22 % Some problem parameters that we do not change very often.
23 %
24 phi = 'gs';
25 % Initial time to get smooth initial condition, used by CIRinitial
26 t1 = start;%1e-1;
27 t0 = 0;
28 %x0 = 1; % initial position for the process
29 ctype=nodetype;
30 etype=ctype;
31 show='yes';
32 col='b';
33 %
34 %
35 % Determine if least squares are to be used
36 %
37 if ( nargin<=13)
38     ls = 0;
39 else
40     ls=1;
41 end
42 %
43 % Compute nodes, collocation/least squares points, and error evaluation points
44 % But this is only for 1-D, right?
45 %
46 L = length(N);
47 for ell=1:L
48     if (strcmp(ctype,'uni'))
49         xc{ell} = linspace(rg(1),rg(2),N(ell))';
50     elseif (strcmp(ctype,'cheb'))
51         xc{ell} = ChebPts(rg,N(ell));
52     elseif (strcmp(ctype,'adap'))
53         xc{ell} = AdaptivePts(rg,N(ell));
54     else
55         error('Incorrect value of mode (node distribution)')
56     end
57 %
58 % Put the boundary points or the extreme points last.

```

```

59      %
60      xc{ell} = [xc{ell}(2:end-1);xc{ell}(1);xc{ell}(end)];
61  end
62
63  if (ls)
64    if (strcmp(etype,'cheb'))
65      xls = ChebPts(rg,Nls);
66    elseif (strcmp(etype,'uni'))
67      xls = linspace(rg(1),rg(end),Nls)';
68    end
69    %
70    % Allow a special case for Nls=N-2 with collocation and QR-factorization
71    %
72    if (L==1 & Nls==N(1)-2)
73      xls = xc{1}(2:end-1);
74    end
75  else
76    xls = 0;
77  end
78  %
79  % Boundary points
80  %
81 xb= rg(:); % For the 1-D case
82 Nb = 2;
83 %
84 % The evaluation points are always uniform Ns should be larger than N
85 %
86 xs = linspace(rg(1),rg(2),Ns)';
87 %xs=5
88 %
89 % BDF2-coefficients to have constant matrices.
90 %
91 %[k,beta0,beta1,beta2]=BDF2coeffs(T-t1,M);%%%%%%%%%%%%%
92 [k,beta0,beta1,beta2]=BDF2coeffs(T,M);
93 %
94 % Initiate all matrices or matrix factors that we need for time-stepping.
95 %
96 if (ls)
97
98   [Q,R,Lf,Uf,P,Cem,Abl,Abm,Ael,Aem,Esl,Esm,Albml,Albmm]= ...
99     BMLeastSquaresMatrices(method,a_p,b_p,sigma_p,beta0,phi,ep,xc,xb,xls,xs);
100 else
101   if (strcmp(method{1}, 'dir'))
102     [L2,U2,P2,L1,U1,P1,Cel,Cem,Abl,Abm,Ael,Aem,Esl,Esm]= ...
103       BMMatricesesm(a_p,b_p,sigma_p,beta0,phi,ep,xc,xb,xs);
104   elseif (strcmp(method{1}, 'qr'))
105     error('QR not implemented, yet')
106   % [L2,U2,P2,L1,U1,P1,Cel,Cem,Abl,Abm,Ael,Aem,Esl,Esm]= ...
107   % BSMatrices_QR(ir,sigma,beta0,phi,ep,xc,xb,xs);
108   else
109     error('The method argument should be dir or qr')
110   end
111 end
112 %
113 % The time-stepping loop
114 %
115 res = 0; timeres=0;
116 for step = 1:M
117   %time(step) = t1+sum(k(1:step));%%%%%%%%%%%%%
118   time(step) = sum(k(1:step));
119   %
120   % Start with the right hand sides that do not change with the level

```

```

121 %
122 g = BMrhsm(xb,time(step), ep);
123 %
124 % Loop over the levels, solving for one at a time
125 %
126 for ell = 1:L
127 %
128 % For the first two time-steps, lambda_1 and lambda_2 have special meaning.
129 %
130 if (step==1)
131   for m=1:L
132     mu_1{m} = 0;
133     lambda_2{m} = 0;
134     mu_2{m} = 0;
135     lambda_1{m} = 0;
136   end
137   if (ls)
138     lambda_1{L} = BMInitialmm(xls(2:end),t1, x0, t0, a_p,b_p,sigma_p);
139     lambda_1{L}=[0;lambda_1{L}];
140   else
141     lambda_1{ell} = BMInitialmm(xc{ell}(1:end-Nb,:),
142                               t1, x0, t0, a_p,b_p,sigma_p);
143   end
144
145 elseif (step==2 & ~ls)
146   for m=1:L
147     lambda_2{m} = 0;
148     mu_2{m} = 0;
149   end
150   if (~ls)
151     lambda_2{ell} = BMInitialmm(xc{ell}(1:end-Nb,:),
152                               t1, x0, t0, a_p,b_p,sigma_p);
153   end
154 end
155 %
156 % The least squares rhs
157 %
158 if (ls & ell==1) % Is the residual in later steps
159   f = zeros(Nls,1);
160   for m=1:L
161     f = f + BDF2rhs(step,beta1,beta2,Ael{m},Aem{m}, ...
162                      lambda_1{m},lambda_2{m},mu_1{m},mu_2{m});
163   end
164 end
165 %
166 % The right-hand side for the collocation case
167 %
168 if (~ls)
169   f = zeros(size(xc{ell},1)-Nb,1);
170   for m=1:L
171     f = f + BDF2rhs(step,beta1,beta2,Ael{ell,m},Aem{ell,m}, ...
172                      lambda_1{m},lambda_2{m},mu_1{m},mu_2{m});
173   end
174   for m=1:ell-1
175     f = f - Cel{ell,m}*lambda{m} - Cem{ell,m}*mu{m};
176   end
177 end
178 if (ls)
179   [lambda{ell},mu{ell},res(1:Nls,ell)]= ...
180   LSSolve(Q{ell},R{ell},Lf{ell},Uf{ell},P{ell}, ...
181             Cem{ell},Ab1{ell},Abm{ell},Ael{ell},Aem{ell},f,g);
182 timeres(step,ell) = rg(2)/(N(ell))*sum(abs(res(:,ell)));%max(abs(res(:,ell)));%sqrt(rg(2)/(N

```

```

183     else
184         [lambda{ell},mu{ell}]=MLSolve(L2{ell},U2{ell},P2{ell}, ...
185                                         L1{ell},U1{ell},P1{ell}, ...
186                                         Cem{ell,ell},Ab1{ell},Abm{ell}, ...
187                                         Ael{ell,ell},Aem{ell,ell},f,g);
188     end
189     %
190     % At all levels except the first g=0 in 1-D
191     %
192     g(:) = 0;
193     if (ls)
194         % The new right hand side is the residual
195         f = res(:,ell);
196     end
197 end
198 coeff_l= normcdf(0, xc{1}(1:N-Nb), 1/sqrt(2)/ep(1));
199 coeff_m= 1- normcdf(0, xc{1}(N-Nb+1:N), 1/sqrt(2)/ep(1));
200 pos_l=find(lambda{1}<0); pos_m=find(mu{1}<0);
201 negmass(step)=sqrt(pi)/ep(1)*(sum(coeff_l(pos_l).*lambda{1}(pos_l))+...
202                               sum(coeff_m(pos_m).*mu{1}(pos_m)));
203 mass(step)=sqrt(pi)/ep(1)*(sum(coeff_l.*lambda{1})+sum(coeff_m.*mu{1}));
204 % [lambda{1};mu{1}]
205 % pause
206 % end
207 %
208 % Try Crazy approach, normalize after each step
209 %
210 %lambda{1}=(1/mass(step))*lambda{1};
211 %mu{1}=(1/mass(step))*mu{1};
212 %
213 % Move the values from the previous step
214 %
215 lambda_2 = lambda_1;
216 mu_2 = mu_1;
217 lambda_1 = lambda;
218 mu_1 = mu;
219 %
220 % Compute the maximum error as a function of time.
221 %
222 %u = cirpdf(xs, time(step), x0, t0, a_p, b_p, sigma_p);
223 u = bmpdfm(xs, time(step), x0, t0, -a_p, sigma_p);
224 u(1)=0;
225 % u = Exact1D(xs,time(step),sigma,ir);
226 v = EvalV(Esl{1},Esm{1},lambda_1{1},mu_1{1});
227 %
228 %figure(44)
229 %plot(xs, v, 'k', xs, u, 'r--')
230 %legend('RBF sol', 'Exact')
231 %
232 %
233 for ell=2:L
234     v = v + EvalV(Esl{ell},Esm{ell},lambda_1{ell},mu_1{ell});
235 end
236 v;
237 timerror(step) = max(abs(u-v));
238 % if (min(lambda{1})<0 | min(mu{1})<0)
239 %     figure(38),clf
240 %     plot([lambda{1};mu{1}], 'o')
241 %
242 %
243 end
244 end

```

```

245 %
246 % Compute the errors at the final time
247 %
248 error = u-v;
249 %ngauss=5;
250 %K=0.5;
251 %[xp,wp]=lgwt(ngauss,K,10);
252 %sum=0;
253 %for l=1:ngauss
254 %    sum=sum+;
255 %end
256 %
257 % Compute both maximum norm and financial norm
258 %
259 maxnrm = max(abs(error));
260 pos = find(xs >= 1/3 & xs <= 5/3);
261 finnrm = max(abs(error(pos)));
262 %
263 % Compute the integrated timeerror
264 %
265 h = time(2:end)-time(1:end-1);
266 inrm=sum(h/2.*((log10(timererror(1:end-1))+log10(timererror(2:end))))) ;
267 %
268 % Compute the computationl costs
269 %
270 if (ls)
271     [ao,mem]=LSMLcosts(M,N,Nb*ones(size(N)),Nls); % OK also for LS
272 else
273     [ao,mem]=CLMLcosts(M,N,Nb*ones(size(N))); % OK also for collocation
274 end
275 %
276 % Plot the errors. The figures are not cleared, in case overlay is desired.
277 %
278 % if (~strcmp(show,'no'))
279 %     figure(1)
280 %     H=plot(xs,error,col);
281 %     hold on
282 %     if (ls)
283 %         H=[H; plot(xls,res,'r--')];
284 %     end
285 %     %H=semilogy(xe,abs(error),col);
286 %     set(gca,'FontSize',18,'LineWidth',2)
287 %     set(H,'LineWidth',2)
288 %     xlabel('x')
289 %     ylabel('|Error|')
290 %
291 %     figure(2)
292 %     H=plot(time,timeres,'r--');
293 %     hold on
294 %     H=[H; plot(time,timeres,'r--')];
295 %     %H=semilogy(time,timeres,col);
296 %     set(gca,'FontSize',18,'LineWidth',2)
297 %     set(H,'LineWidth',2)
298 %     xlabel('time')
299 %     ylabel('Max error')
300 %
301 %     figure(3)
302 %     plot(time,negmass)
303 %
304 %     figure(4)
305 %     plot(time,mass)
306 % end

```

```

1  %
2  %
3  function res = bmpdfm(E,T,s,t,r,sigma)
4
5  %d1= 1/(sigma*sqrt(T-t))*(log(s./E)+(r+0.5*sigma^2)*(T-t));
6  %d2= 1/(sigma*sqrt(T-t))*(log(s./E)+(r-0.5*sigma^2)*(T-t));
7  %res = E.*exp(-r*(T-t)).*normcdf(-d2)-s.*normcdf(-d1);
8
9  f=log(E./s)-(r-(sigma^2)/2)*T;
10 f2=-f.^2./(2*(sigma^2)*T);
11 f3=exp(f2);
12 f4=(1./sqrt(2*pi*T)*sigma*E));
13 res=f4.*f3;
14 end

1  function g=BMrhs(x,t, epsilon)
2  %
3  % Boundary conditions
4  %
5  %
6  g = zeros(size(x, 1),1);
7  g(2)=0;
8  %g(2) = epsilon/sqrt(pi);

1  clc
2  clear
3  phi='gs';
4
5  a=-0.05;
6  b=0;
7  s0=1;
8  sig0=0.2;
9
10 %mu = @(t) a*(b-s0)*t;
11 %sigma = @(t) sig0*sqrt(s0*t);
12
13 %t1=1*(1e-2);
14 t1=0.01;
15 %t1=0.01;
16 T=1;
17
18 %KS=0.5;
19 %
20 % Shape parameter values for the first and the last time
21 %
22 %ep(1)=1/sqrt(2)/sigma(t1);
23 %ep(2)=1/sqrt(2)/sigma(T);
24
25 %ep=ep/4;
26 %
27 % How many nodes should we use to have phi(h/2)=1/2?
28 %
29 %h=2*sqrt(log(1/0.7))./ep;
30
31 %L=11.70;
32 r=-a;
33 sigma=sig0;
34 threshold=10^(-16);
35 L=s0*exp((r-1.5*(sigma^2))*T+sigma*sqrt(2*(sigma^2)*(T^2)-2*T*(r*T+log(s0*sqrt(2*pi*T)*threshold)));
36 %L=5.39;

```

```

37 rg=[0 L];
38
39 %N=10+round(L./h+1);
40
41 % Time steps.
42 M= 500;
43
44 % Evaluation points
45 Ns=100;
46
47 ctype='uni';
48 etype='uni';
49 method={'dir'};
50
51 show='yes';
52 col='k';
53 %
54 e=[];
55 epv=[];
56 j=1;
57 tic
58 for s=0.1:0.01:.1
59 %s=0.7;
60 a=-r;
61 %s=0.1;
62 %sigma=sig0;
63 %threshold=10^(-16);
64 %s0=exp((r-1.5*(sigma^2))*T+sigma*sqrt(2*(sigma^2)*(T^2)-2*T*(r*T+log(s0*sqrt(2*pi*T)*(threshold));
65 %L=5.39;
66 %rg=[0 L];
67
68 %Nvec=[100:16:200];
69 %Nvec=20;
70 %
71 Nvec=floor(L/0.03)+1;
72 %Nvec=[100:16:200];
73 % Nvec=200;
74 %
75 % Computethe corresponding epsilon
76 %
77 d = L./(Nvec-1);
78
79 epvec=2*sqrt(-log(s))./d;
80
81 for k=1:length(Nvec)
82 N=Nvec(k);
83 %ep(j)=epvec(k);
84 ep(j)=0.81/d+0.15;
85 % 28.8310
86 % Least squares points.
87 %
88
89 Nls=4*max(N);
90 % Rung CIRMLMain instead of LeastSquares.
91 [maxnrm(k),u,v,xs,error]=BMMain(a,b,sig0,T,s0,t1,ep(j),M,N,Ns,rg,ctype,method,Nls);
92 end
93 time_final=toc
94 tic
95 i=1;
96 for KS=0:size(xs,1)-1
97 uBL(j,i)=exp(a*T)*trapz(xs,u.*max(xs-xs(KS+1),0));
98 %vBL(i)=exp(a*KS)*trapz(xs,v.*max(xs-KS,0));

```

```

99         vBL(j,i)=exp(a*T);
100        BL(j,i)=exactBL(s0,xs(KS+1),T,sig0,-a);
101        i=i+1;
102    end
103    size(vBL);
104    uNL(j,:)=(vBL(j,:)'.*v)';
105    time_final_u=toc
106    clc
107    [uBL(j,:)' BL(j,:)' uNL(j,:)];
108    %jj=plot(xs,abs(BL(j,:)-uNL(j,:))./BL(j,:));
109    jj=plot(s0./xs,log10(abs(BL(j,:)-uNL(j,:))));  

110    %jj=plot(s0./xs,log10(error));
111    %jj=plot(xs,uNL(j,:));
112    %max(uBL(j,:)-uNL(j,:))
113    set(gca,'FontSize',18,'LineWidth',2)
114    set(jj,'LineWidth',2)
115    xlabel('Strike price')
116    ylabel('u-u_{exact}')
117
118    pause(0.001)
119    k=k+1;
120
121    epv=[epv epvec];
122    %e=[e max(abs(BL(j,:)-uNL(j,:))]];
123    e=[e max(abs(BL(j,:)-uNL(j,:))]];
124    %e=[e max(error)];
125    j=j+1;
126 end
127 [in1,in2]=min(e);
128 figure(2)
129 jj=plot(xs,BL(in2,:)-uNL(in2,:));
130
131    %max(uBL(j,:)-uNL(j,:))
132    %set(gca,'FontSize',18,'LineWidth',2)
133    %set(jj,'LineWidth',2)
134    xlabel('Strike price')
135    ylabel('u-u_{exact}')
136    title(['The optimal value of shape parameter is ',num2str(epv(in2))])
137 figure(3)
138 plot(epv,log10(e),'--rs','LineWidth',3,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',10)
139 xlabel('Shape parameter')
140 ylabel('log10(max{|u-u_{exact}|})')
141 title('Logarithmic graph of maximum error')
142 %set(gca,'FontSize',18,'LineWidth',2)
143 log10(in1)
144 ep(in2)
145 time_final
146 time_final_u
147 time_final+time_final_u
148 figure
149 plot(xs,u);
150 % s=0.1:0.01:0.99
151 % d = L./(Nvec-1);
152 % epvec=2*sqrt(-log(s))./d;
153 % %surf(s,xs,BL(:,:,));
154 % contourf(xs,epvec(70:80),BL(70:80,:)-uNL(70:80,:))

1 function [delta] = delta_function(alpha,h,B)
2
3 %building A
4
```

```

5 A = zeros(B,B);
6
7 if mod(B,2) == 0
8     upper = round((B-1)/2);
9     lower = -upper+1;
10 else
11     upper = (B-1)/2;
12     lower = -upper;
13 end
14 i=1;
15
16 for k=lower:1:upper
17
18 for j=1:B;
19 A(j,i)=k^(j-1);
20 end
21 i=i+1;
22 end
23
24 %%% A complete %%%
25
26 %%%Building b-matrix%%%%%
27
28 b = zeros(B,1);
29 for t=1:1:B
30     b(t,1) = alpha^(t-1)/h;
31 end
32
33 %%%b-matrix done %%%
34
35 delta = A\b;
36
37 end

1 function v=EvalV(Ael,Aem,lambda,mu)
2
3 v = Ael*lambda + Aem*mu;

1 function exactBL=exactBL(xx,E,deltat,sigma,r)
2
3 d1= 1/(sigma*sqrt(deltat))*(log(xx/E)+(r+0.5*sigma^2)*deltat);
4 d2= 1/(sigma*sqrt(deltat))*(log(xx/E)+(r-0.5*sigma^2)*deltat);
5
6 exactBL = xx*normcdf(d1)-E*exp(-r*deltat)*normcdf(d2);
7
8 end

1 function [phi]=gs(epsil,r,nprime,dim)
2 %
3 % NPRIME is a string defining which operator to use on the basis function
4 %
5 % DIM is the dimension for the partial derivative if nprime is '0','1'...,'4'
6 % DIM(1:2) are the dimensions for the mixed second derivative if nprime
7 % is 'm2'
8 % and DIM is the number of space dimensions if nprime is 'L' or 'L2'
9 %
10 %
11 % Assume a one-dimensional problem if no dimension is given

```

```

12 %
13 if nargin<=3
14 dim=1;
15 end
16 %
17 % For the case of L or L2 operators, we need to know the number of dimensions
18 %
19 if (nprime(1)=='L')
20 if (size(r,3)==1)
21 nd=dim;
22 else
23 nd=size(r,3)-1;
24 end
25 end
26 %
27 % For the mixed derivative, the dimensions must be given even in 2D
28 %
29 if (nprime(1)=='m')
30 if (length(dim)~=2)
31 error('For the mixed derivative, dim=dim(1:2)')
32 elseif (dim(1)==dim(2))
33 error('For mixed derivatives, dim(1) must be other than dim(2)')
34 end
35 end
36 %
37 % epsil can be either just one value or a vector of N values
38 %
39 esz = size(epsil);
40 if (prod(esz)~=1)
41 if (min(esz)==1 & max(esz)==size(r,2))
42 %
43 % Make epsil into a matrix with constant columns
44 %
45 epsil = ones(size(r,1),1)*epsil(:)';
46 else
47 error('The size of epsil does not match the columns in r')
48 end
49 end
50
51 phi=zeros(size(r,1),size(r,2));
52
53 tmp = exp(-(epsil.*sq(r(:,:,1))).^2);
54
55 if nprime(1)=='0'
56 phi = tmp;
57
58 elseif nprime(1)=='1'
59 phi = -2*epsil.^2.*sq(r(:,:,dim+1)).*tmp;
60
61 elseif nprime(1)=='2'
62 phi = (-2*epsil.^2+4*epsil.^4.*sq(r(:,:,dim+1))).^2.*tmp;
63
64 elseif nprime(1)=='3'
65 phi = (12*epsil.^4.*sq(r(:,:,dim+1)) - 8*epsil.^6.*sq(r(:,:,dim+1))).^3.*tmp;
66
67 elseif nprime(1)=='4'
68 phi = (12*epsil.^4 - 48*epsil.^6.*sq(r(:,:,dim+1))).^2 + ...
69 16*epsil.^8.*sq(r(:,:,dim+1)).^4).*tmp;
70
71 elseif nprime(1)=='L' & length(nprime)==1
72 phi = (-2*nd*epsil.^2 + 4*epsil.^4.*sq(r(:,:,1))).^2.*tmp;
73

```

```

74 elseif nprime(1:2)=='L2'
75     phi = (4*nd*(nd+2)*epsil.^4 - 16*(nd+2)*epsil.^6.*sq(r(:,:,1)).^2 + ...
76         16*epsil.^8.*sq(r(:,:,1)).^4).*tmp;
77
78 elseif nprime(1:2)=='m2'
79     phi = 4*epsil.^4.*sq(r(:,:,dim(1)+1)).*sq(r(:,:,dim(2)+1)).*tmp;
80
81 else
82     error('Error in input argument nprime to function gauss')
83 end
84
85 function r=sq(r)
86 r=squeeze(r);

1 function [ao,mem]=LSMLcosts (M,N,Nb,Ne)
2 %
3 % N and Nb can be matrices (for many different runs).
4 % N(:,:,j) is for level j
5 %
6 %
7 % First check if N is just a vector over levels or a structure
8 %
9 sz = size(N);
10 if (prod(sz) == max(sz)) % A one-dimensional vector
11     [L,lev] = max(sz);
12 elseif (length(sz)==3)
13     lev = 3;
14     L = sz(3);
15 else
16     warning('N is a matrix, treated as one level')
17     lev = 3;
18     L = 1;
19 end
20
21 mem = 0.5*sum(N.^2,lev) + 2*Ne.*sum(N,lev) + 0.5*sum(Nb.^2,lev);
22
23 a_fac = 2*Ne.*sum((N-Nb).*N,lev) - (2/3)*sum((N-Nb).^3,lev) ...
24     + (2/3)*sum(Nb.^3,lev) + 2*sum(Nb.^2.*(N-Nb),lev);
25 a_sc = sum(N.^2,lev) + 3*sum(Nb.^2,lev) + 2*Ne.*sum(N,lev);
26 a_rhs = 2*Ne.*sum(N,lev);
27 a_res = 0;
28
29 if (L>1)
30     if (length(sz) == 3)
31         for ell=1:L-1
32             a_res = a_res + 2*Ne.*N(:,:,ell) ...
33                 + 2*(Nb(:,:,L)-Nb(:,:,ell)).*N(:,:,ell);
34         end
35     else
36         for ell=1:L-1
37             a_res = a_res + 2*Ne.*N(ell) ...
38                 + 2*(Nb(L)-Nb(ell)).*N(ell);
39         end
40     end
41 end
42
43 ao = a_fac + M.* (a_rhs + a_sc + a_res);

1 function [lambda,mu,res]=LSSolve(Q,R,L,U,P,Cem,Abl,Abm,Ael,Aem,f,g);

```

```

2 %
3 % Compute the solution using the Schur complement algorithm.
4 %
5
6 w = U \ (L \ (P * g));
7
8 ft = f - Cem * w;
9 b = Q' * ft;
10 lambda = R \ b;
11
12 v = U \ (L \ (P * (Abl * lambda)));
13
14 mu = w - v;
15 %
16 % Compute the QR-residual
17 %
18 res = ft - Q * b;

1 clc
2 %clear
3 %%%%%%%%VAR FIL%%%%%%%
4 phi='gs';
5
6 a=-0.05;
7 b=0;
8 %0=1;
9 sig0=0.1;
10 rakna = 1;%%%%%%%%%
11 e=[];
12 %for sig0=0.1:0.004:0.5%%%%%
13
14 for s0 = 0.5:0.02:2.5;
15
16
17 t1=1e-2;
18 T=1;
19
20 %L=5.39;
21 L=10;
22 rg=[0 L];
23
24
25 % Time steps.
26 M= 100;
27
28 % Evaluation points
29 Ns=100;
30
31 ctype='uni';
32 etype='uni';
33 method={'dir'};
34
35 show='yes';
36 col='k';
37 %
38
39 epv=[];
40 j=1;
41 %for s=0.1:0.01:0.99
42 %Nvec=[100:16:200];
43 Nvec=100;

```

```

44      %
45      % Compute the corresponding epsilon
46      %
47      d = L./(Nvec-1);
48
49      % epvec=2*sqrt(-log(s))./d;
50
51      for k=1:length(Nvec)
52          N=Nvec(k);
53          ep(j)=epvec(k);
54          ep(j)=0.81/d+0.15
55
56      % Least squares points.
57      %
58
59      Nls=4*max(N);
60      [maxnrm(k), u, v, xs]=BMMain(a,b,sig0,T,s0,t1,ep(j),M,N,Ns,rg,ctype,method,Nls);
61
62      end
63
64      i=1;
65      for KS=0:size(xs,1)-1
66          vBL(j,i)=exp(a*T);
67          BL(j,i)=exactBL(s0,xs(KS+1),T,sig0,-a);
68          i=i+1;
69      end
70      size(vBL);
71      uNL(j,:)=(vBL(j,:)'.*v)';
72      clc
73      %jj=plot(xs,BL(j,:)-uNL(j,:));%(ta baort)
74
75      %set(gca,'FontSize',18,'LineWidth',2)
76      %set(jj,'LineWidth',2);%(ta bort)
77      % xlabel('Strike price')
78      % ylabel('u-u_{exact}')
79
80      %pause(0.001)%%%%%%%%%%%%%
81      k=k+1;
82
83      % epv=[epv epvec];
84      e=[e max(abs(BL(j,:)-uNL(j,:)))] ;
85      j=j+1;
86      %end
87      [in1,in2]=min(e);
88      %figure(2);%(ta bort)
89      %jj=plot(xs,BL(in2,:)-uNL(in2,:));%(ta bort)
90
91      test(rakna)= max(abs(u-v));%%%%%
92      rakna=rakna+1;
93      %end
94
95
96      end%%%%%%%%%%%%%
97
98      %plot(sigiz,e);%%%%%
99
100     % xlabel('Strike price')
101     % ylabel('u-u_{exact}')
102     % title(['The optimal value of shape parameter is ',num2str(epv(in2))])
103     %figure(3);%(ta bort)
104     %plot(epv,log10(e),'--rs','LineWidth',3,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',10
105     % xlabel('Shape parameter')

```

```

106 ylabel('log10(max{|u-u_{exact}|})')
107 %title('Logarithmic graph of maximum error')
108 %set(gca,'FontSize',18,'LineWidth',2)
109 log10(in1)
110 %%%%%
111 % s=0.1:0.01:0.99
112 % d = L./(Nvec-1);
113 % epvec=2*sqrt(-log(s))./d;
114 % surfc(s,xs,BL(:,:,::))
115 % contourf(xs,epvec(70:80),BL(70:80,:)-uNL(70:80,:))

1 function output = ModelU2(x,x0,del,param)
2
3 % mu(x_) = a + b*x;
4 %
5 % s(x_) = d*x;
6 %
7 % g(x_) = log(x)/d;
8 %
9 %
10 % lnpX(del_, x_, x0_, 2, exact) = -(1/2))*log(2*Pi*del) - log(s(x)) + cY(g(x), g(x0), -1, exact)/c
11 %      cY(g(x), g(x0), 1, exact)*del + cY(g(x), g(x0), 2, exact)*(del^2/2);
12 %
13 %
14 % cY(y, y0, -1, exact) = -(1/2))*(y - y0)^2;
15 %
16 % cY(y, y0, 0, exact) = (E^((-d)*y) - E^((-d)*y0))*(-(a/d^2)) + (y - y0)*(b/d - d/2);
17 %
18 % cY(y, y0, 1, exact) = (a^2/(4*d^3))*(E^(-2*d*y) - E^(-2*d*y0))/(y - y0) + ((a*b)/d^3 - a/d)*((E^(-2*d*y) - E^(-2*d*y0))/(y - y0));
19 %      (2*b - d^2)^2/(8*d^2);
20 %
21 % cY(y, y0, 2, exact) = (-(a^2/(2*d^3)))*((E^(-2*d*y) - E^(-2*d*y0))/(y - y0)^3) +
22 %      ((2*a)/d - (2*a*b)/d^3)*((E^((-d)*y) - E^((-d)*y0))/(y - y0)^3) + (-(a^2/(2*d^2)))*((E^(-2*d*y) - E^(-2*d*y0))/(y - y0)^2);
23 %
24 %
25 % (* Expressions below to be used for coefficients of order 1 and 2 at y0=y: *)
26 %
27 %
28 % cY(y, y, 1, exact) = (-4*a^2 - 8*a*(b - d^2)*E^(d*y) - (-2*b +
29 % d^2)^2*E^(2*d*y))/(E^(2*d*y)*(8*d^2));
30 %
31 % cY(y, y, 2, exact) = ((1/6)*a*(-2*a + (-b + d^2)*E^(d*y)))/E^(2*d*y);
32 %
33 a = param(1);
34 b = param(2);
35 d = param(3);
36
37 y = log(x)/d;
38 y0 = log(x0)/d;
39
40 E = exp(1);
41
42 sx = d*x;
43
44 cYml = (-(1/2))*(y - y0).^2;
45 cY0 = (exp((-d)*y) - E^((-d)*y0)).*(-(a/d^2)) + (y - y0).*(b/d - d/2);
46
47 if y~=y0
48     cY1 = (a^2/(4*d^3))*((exp(-2*d*y) - E^(-2*d*y0))./(y - y0)) + ((a*b)/d^3 - a/d)*((exp((-d)*y) - E^(-2*d*y0))./(y - y0));
49     cY2 = (-(a^2/(2*d^3)))*((exp(-2*d*y) - E^(-2*d*y0))./(y - y0).^3) + ...

```

```

50      ((2*a)/d - (2*a*b)/d^3)*((exp((-d)*y) - E^((-d)*y0))./(y - y0).^3) + (-a^2/(2*d^2))*((exp((-d)*y) - E^((-d)*y0))./(y - y0).^2);
51      (a - (a*b)/d^2)*((exp((-d)*y) + E^((-d)*y0))./(y - y0).^2);
52  else
53      cY1 = (-4*a^2 - 8*a*(b - d^2)*exp(d*y) - (-2*b + d^2)^2*exp(2*d*y))./(exp(2*d*y)*(8*d^2));
54      cY2 = ((1/6)*a*(-2*a + (-b + d^2)*exp(d*y)))./exp(2*d*y);
55  end
56 output = (-(1/2))*log(2*pi*del) - log(sx) + cYm1/del + cY0 + cY1*del + cY2*(del^2/2);
57 %output = (-(1/2))*log(2*pi*del) - log(sx) + cYm1/del + cY0;
58 output=exp(output);

1 function output = ModelU22(x,x0,del,param)
2
3 %%%%%% Base bestammer supporten pa dirac funktionen%%%%%%%%%%%%%
4 Base = 6;
5 %%%%%%%%%%%%%%%%
6
7 if mod(Base,2) == 0
8     upper = round((Base-1)/2);
9     lower = -upper+1;
10 else
11     upper = (Base-1)/2;
12     lower = -upper;
13 end
14
15 a = param(1);
16 b = param(2);
17 d = param(3);
18
19
20 dx = x(2)-x(1);
21
22 Ns = length(x)-1;
23
24
25 %----- Delta function-----
26
27 % Center of dirac
28 xp = x0;
29 %find ax6
30
31 ss1 = x(1);
32 %Find location of nearest grid point in the computational domain.
33
34 not_found = 1;
35 for i = 1:Ns
36     if ((ss1+(i-1)*dx-xp) >=-5*eps && not_found)
37         xn = x(i-1);
38         xni = i-1;
39         not_found = 0;
40     end
41 end
42 alpha_x = (xp-xn)/dx;
43 delta = delta.function(alpha_x,dx,Base);
44
45 Deltal = zeros(Ns+1,1);
46 Deltal(xni+lower:xni+upper,1) = delta;
47
48 v(:,1) = Deltal;
49
50 output = v;

```

```

1 function A=RBFmat(phi,ep,r,nprime,dim);
2
3 if (nargin==5)
4     A = feval(phi,ep,r,nprime,dim);
5 elseif (nargin==4)
6     A = feval(phi,ep,r,nprime);
7 else
8     error('Wrong number of arguments to RBFmat')
9 end

10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
```

function z = simps(x,y,dim)

%SIMPS Simpson's numerical integration.

% The Simpson's rule for integration uses parabolic arcs instead of the straight lines used in the trapezoidal rule.

%

% Z = SIMPS(Y) computes an approximation of the integral of Y via the Simpson's method (with unit spacing). To compute the integral for spacing different from one, multiply Z by the spacing increment.

%

% For vectors, SIMPS(Y) is the integral of Y. For matrices, SIMPS(Y) is a row vector with the integral over each column. For N-D arrays, SIMPS(Y) works across the first non-singleton dimension.

%

% Z = SIMPS(X,Y) computes the integral of Y with respect to X using the Simpson's rule. X and Y must be vectors of the same length, or X must be a column vector and Y an array whose first non-singleton dimension is length(X). SIMPS operates along this dimension.

%

% Z = SIMPS(X,Y,DIM) or SIMPS(Y,DIM) integrates across dimension DIM of Y. The length of X must be the same as size(Y,DIM).

%

Examples:

-----

% The integration of sin(x) on [0,pi] is 2

% Let us compare TRAPZ and SIMPS

x = linspace(0,pi,6);

y = sin(x);

trapz(x,y) % returns 1.9338

simps(x,y) % returns 2.0071

%

If Y = [0 1 2  
3 4 5  
6 7 8]  
then simps(Y,1) is [6 8 10] and simps(Y,2) is [2; 8; 14]

-- Damien Garcia -- 08/2007, revised 11/2009

website: <a href="matlab:web('http://www.biomecardio.com')">www.BiomeCardio.com</a>

See also CUMSIMPS, TRAPZ, QUAD.

Adapted from TRAPZ

%% Make sure x and y are column vectors, or y is a matrix.

perm = []; nshifts = 0;

if nargin == 3 % simps(x,y,dim)

perm = [dim:max(ndims(y),dim) 1:dim-1];

yp = permute(y,perm);

[m,n] = size(yp);

elseif nargin==2 && isscalar(y) % simps(y,dim)

dim = y; y = x;

```

52     perm = [dim:max(ndims(y),dim) 1:dim-1];
53     yp = permute(y,perm);
54     [m,n] = size(yp);
55     x = 1:m;
56 else % simps(y) or simps(x,y)
57     if nargin < 2, y = x; end
58     [yp,nshifts] = shiftdim(y);
59     [m,n] = size(yp);
60     if nargin < 2, x = 1:m; end
61 end
62 x = x(:);
63 if length(x) ~= m
64     if isempty(perm) % dim argument not given
65         error('MATLAB:simps:LengthXmismatchY',...
66             'LENGTH(X) must equal the length of the first non-singleton dimension of Y.');
67 else
68     error('MATLAB:simps:LengthXmismatchY',...
69         'LENGTH(X) must equal the length of the DIM''th dimension of Y.');
70 end
71 end
72
73 %-- The output size for [] is a special case when DIM is not given.
74 if isempty(perm) && isequal(y,[])
75     z = zeros(1,class(y));
76     return
77 end
78
79 %-- Use TRAPZ if m<3
80 if m<3
81     if exist('dim','var')
82         z = trapz(x,y,dim);
83     else
84         z = trapz(x,y);
85     end
86     return
87 end
88
89 %-- Simpson's rule
90 y = yp;
91 clear yp
92
93 dx = repmat(diff(x,1,1),1,n);
94 dx1 = dx(1:end-1,:);
95 dx2 = dx(2:end,:);
96
97 alpha = (dx1+dx2)./dx1/6;
98 a0 = alpha.* (2*dx1-dx2);
99 a1 = alpha.* (dx1+dx2).^2./dx2;
100 a2 = alpha.*dx1./dx2.* (2*dx2-dx1);
101
102 z = sum(a0(1:2:end,:).*y(1:2:m-2,:)+...
103     a1(1:2:end,:).*y(2:2:m-1,:)+...
104     a2(1:2:end,:).*y(3:2:m,:),1);
105
106 if rem(m,2) == 0 % Adjusting if length(x) is even
107     state0 = warning('query','MATLAB:nearlySingularMatrix');
108     state0 = state0.state;
109     warning('off','MATLAB:nearlySingularMatrix')
110     C = vander(x(end-2:end))\y(end-2:end,:);
111     z = z + C(1,:).* (x(end,:).^3-x(end-1,:).^3)/3+...
112         C(2,:).* (x(end,:).^2-x(end-1,:).^2)/2+...
113         C(3,:).*dx(end,:);

```

```

114     warning(state0, 'MATLAB:nearlySingularMatrix')
115 end
116
117 %-- Resizing
118 siz = size(y); siz(1) = 1;
119 z = reshape(z,[ones(1,nshifts),siz]);
120 if ~isempty(perm), z = ipermute(z,perm); end

1 function r=xcdist(x,c,all)
2 %
3 % Evaluation/collocation points and center points are not always the
4 % same. We compute r=||xi-cj|| together with componentwise signed differences.
5 %
6 if (nargin==1)
7     all=0;
8     C=x;
9 elseif (nargin==2)
10    all=0;
11 end
12
13 [np,nd] = size(x);
14 nc = size(c,1);
15 nr = 1 + all*nd;
16
17 % r contains r and each ri, i=1...nd
18 r = zeros(np,nc,nr);
19 for d=1:nd
20     [pi,pj] = meshgrid(c(:,d),x(:,d));
21     r(:,:,:,1) = r(:,:,:1) + (pi-pj).^2;
22     if (all)
23         r(:,:,:d+1) = pj(pi);
24     end
25 end
26 r(:,:,:1) = sqrt(r(:,:,:1));

```