A GAUSS Implementation of Non Uniform Grids for PDE

The **PDE** library

Jérome Bodeau, Gaël Riboulet and Thierry Roncalli

Groupe de Recherche Opérationnelle
Bercy-Expo — Immeuble Bercy Sud — 4[è] étage
90quai de Bercy — 75613 Paris Cedex 12 — France

December 15, 2000

Contents

Chapter 1

Introduction

1.1 Installation

1. The file *libpde.zip* is a zipped archive file. Copy this file under the root directory of GAUSS, for example **C:\GAUSS**.

2. Unzip it with archive mode. It is automatically recognized by WinZip. With Unzip or PKunzip, use the -d flag

   pkunzip -d libpde.zip

   Directories will then be created and files will be copied over them:

target_path	*readme.pde*
*target_path***dlib**	*pde.dll* file
*target_path***examples****pde**	examples and tutorial files
*target_path***lib**	library file
*target_path***src****pde**	source code files

3. Run GAUSS. **Log on to the *src\pde* directory**[1] and add the path to the library file *pde.lcg* in the following way:

   lib pde /addpath

1.2 Getting started

Gauss 3.2 for OS/2, Windows NT/95 or Unix[2] is required to use the **PDE** routines.

1.2.1 The file *readme.PDE*

The file *readme.PDE* contains last minute information on the **PDE** procedures. Please read it before using them.

1.2.2 Setup

In order to use these procedures, the **PDE** library must be active. This is done by including **PDE** in the `LIBRARY` statement at the top of your program:

library PDE;

[1] You may use the commands `ChangeDir` or `chdir`. Note that you can verify that you are in the *src\pde* directory with the `cdir(0)` command.

[2] see however the paragraph 2.5.3 for using the library with Unix.

To reset global variables in subsequent executions of the program, the following instruction should be used:

PDEset;

If you plan to make any right-hand reference to the global variables, you will also need the statement:

#include *target_path***src\\pde\\pde.ext;**

The **PDE** library uses a dynamic link library *pde.dll*. This dll file contains a tridiagonal solver written in C in order to speed up computations. You have to declare this dll with the following command:

dlibrary PDE.dll;

Nevertheless, if you use the `PDEset` command at the top of your program, it is done automatically. Moreover, if you don't want to use this dll file, you can use the following compiler directive:

#declare not_DLLs;

The **PDE** version number is stored as a global variable:

_PDE_ver	3×1 matrix where the first element indicates the major version number, the second element the minor version number, and the third element the revision number

1.3 What is PDE ?

PDE is a GAUSS library for solving Parabolic and Elliptic Partial Differential Equations (PDE) with non uniform grids. It includes θ-schemes algorithms with finite difference methods.

PDE contains the procedures whose list is given below. See the command reference part for a full description.

- **derivePDE**: Computes the numerical first and second derivative of the solution of a PDE problem.
- **FindIndex**: Returns the indices of the elements of a vector x equal to the elements of a vector v.
- **generateGrid1**: Generates a uniform grid.
- **generateGrid2:** Generates a non uniform grid with the inverse distribution method.
- **generateGrid3:** Geneates a non uniform grid with the Tavella-Randall method.
- **loadGrid**: Loads the dataset `xFile`.
- **PDE**: Initializes the PDE problem.
- **PDEset**: Resets the global variables declared in *pde.dec*.
- **plotGrid**: Plots the (temporal) non uniform grid.
- **readPDE**: Extracts solution of the database `uFile` computed by `solvePDE`.
- **readPDE2**: Extracts solution of the database `uFile` computed by `solvePDE2`.
- **saveGrid**: Saves the dataset `xFile`.
- **solvePDE**: Solves the PDE problem with non uniform grids.
- **solvePDE2**: Solves the PDE problem with temporal non uniform grids.
- **solveTDG**: Solves a tridiagonal system.

1.4 Using Online Help

PDE library supports Windows Online Help. Before using the browser, you have to verify that the **PDE** library is activated by the `library` command.

Chapter 2

Partial Differential Equations

The library **PDE** is a GAUSS implementation of the use of non uniform grids for solving PDE in finance described in BODEAU, RIBOULET and RONCALLI [2000]. The reader may refer to this article to understand the notations used in this manual.

2.1 The PDE problem

The PDE problem consists of the linear parabolic equation

$$\frac{\partial\, u\,(t,x)}{\partial\, t} + c\,(t,x)\, u\,(t,x) = \mathcal{A}_t u\,(t,x) + d\,(t,x) \tag{2.1}$$

where $\mathcal{A}_t$ is the general two-space dimensions differential operator

$$\mathcal{A}_t u\,(t,x) = a\,(t,x)\,\frac{\partial^2\, u\,(t,x)}{\partial\, x^2} + b\,(t,x)\,\frac{\partial\, u\,(t,x)}{\partial\, x} \tag{2.2}$$

The **PDE** library solves the problem (2.1) in the region of the (t,x) space given by $\mathfrak{T} \times \mathfrak{X}$ with

$$\mathfrak{X} = \left[x^-, x^+\right] \tag{2.3}$$

and

$$\mathfrak{T} = \left[t^-, t^+\right] \tag{2.4}$$

We could impose Dirichlet or Neumann conditions:

$$\begin{aligned} &u\left(t^-, x\right) = u_{(t^-)}\,(x) \\ &u\left(t, x^-\right) = u_{(x^-)}\,(t) \quad \bigvee \quad \left.\frac{\partial\, u\,(t,x)}{\partial\, x}\right|_{x=x^-} = u'_{(x^-)}\,(t) \\ &u\left(t, x^+\right) = u_{(x^+)}\,(t) \quad \bigvee \quad \left.\frac{\partial\, u\,(t,x)}{\partial\, x}\right|_{x=x^+} = u'_{(x^+)}\,(t) \end{aligned} \tag{2.5}$$

To initialize the PDE problem, we use the `PDE` procedure

```
call PDE(&aProc,&bProc,&cProc,&dProc,&eProc,&tminBound,
         &xminBound,&xmaxBound,&yminBound,&ymaxBound);
```

The general form of the procedures is

```
proc (1) = aProc(t,x);
  local a;

  a =
```

```
  retp(a);
endp;

endp;
```

Remark 1 *e is a special function in order to solve variational inequalities. If it is not initialized to 0, the PDE algorithm use this function at each iteration m to modify the numerical solution*

$$\mathbf{u}_m := e\left(t, x, \mathbf{u}_m\right)$$

The form of the `eProc` procedure is also

```
proc (1) = eProc(t,x,u);
  local e;

  e =

  retp(e);
endp;
```

Remark 2 *In the **PDE** library, x is treated as a $N \times 1$ column vector and the procedures `*Proc` must return a $N \times 1$ vector.*

For each bound, you have to specify a boundary condition, either a Dirichlet or a Neumann condition. For example, if we have the following command line

```
call PDE(&aProc,&bProc,&cProc,&dProc,&eProc,&tminBound,
         0,&xmaxBound,&DxminBound,0);
```

Dirichlet conditions are imposed for $x = x^+$ and we put a user-defined Neumann condition on $x = x^-$.

Remark 3 *The `PDE` procedure prints information about the boundary nature of the PDE problem if `__output` is set to 1.*

For the precedent example, we have

```
==========================================================
Bound              Dirichlet             Neumann

xmin                                    *********
xmax               *********
==========================================================
```

2.2 The PDE algorithm

2.2.1 The case of non-uniform grids

The procedure `solvePDE` enables you to solve the PDE problem. Its syntax is

```
call solvePDE(t,x,uFile,theta);
```

The variables `t` and `x` correspond to the non uniform grid used for solving the PDE. They are respectively $M \times 1$ and $N \times 1$ vectors. The numerical solution of the PDE problem u_i^m is stored in the dataset `uFile` in the following way:

	$x_0 = x^-$	x_1	x_2	$\cdots$	x_{N-2}	$x_{N-1} = x^+$
$t_0 = t^-$	u_0^0	u_1^0	u_2^0	$\cdots$	u_{N-2}^0	u_{N-1}^0
t_1	u_0^1	u_1^1	u_2^1	$\cdots$	u_{N-2}^1	u_{N-1}^1
$\vdots$				$\vdots$		
$t_{M-1} = t^+$	u_0^{M-1}	u_1^{M-1}	u_2^{M-1}	$\cdots$	u_{N-2}^{M-1}	u_{N-1}^{M-1}

with

$$
\begin{aligned}
t_m &= t^- + \sum_{j=1}^{m-1} k_j \\
x_i &= x^- + \sum_{j=1}^{i-1} h_j
\end{aligned}
$$

We have

$$k_m = t_m - t_{m-1}$$

and

$$h_i = x_i - x_{i-1}$$

The first row of the dataset contains the N values $\{x_i\}$. The t_m and u_i^m values are stored in the next N rows. Let $\mathbf{u}^m$ be the vector with the (i) entry (u_i^m). Then, the storage method corresponds to the following stacking method

$$\left[\begin{array}{cc} t_m & \operatorname{vec}(\mathbf{u}^m)^\top \end{array}\right]$$

Remark 4 *You could use the* `_PDE_Elliptic` *variable to specify that the PDE problem is an elliptic problem. In this case,* `solvePDE` *stops iterations if the following condition is verified*

$$\mathbf{u}_{m+1} = \mathbf{u}_m$$

Note that `solvePDE` *uses the fuzzy comparison function* `feq` *to perform the test. You could also modify the value taken by* `_fcmptol`*.*

Remark 5 *If you would to save only the last iteration solution, you could use the following syntax*

```
_PDE_SaveLastIter = 1
```

In this case, the `uFile` *dataset becomes*

	$x_0 = x^-$	x_1	x_2	$\cdots$	x_{N-2}	$x_{N-1} = x^+$
$t_{M-1} = t^+$	u_0^{M-1}	u_1^{M-1}	u_2^{M-1}	$\cdots$	u_{N-2}^{M-1}	u_{N-1}^{M-1}

Remark 6 *You could print the number of iterations accomplished with the variable* `_PDE_PrintIters`*.*

Remark 7 *The* `solvePDE` *procedure uses the approximation method for the second derivatives described in* Bodeau, Riboulet *and* Roncalli *[2000]. We have*

$$
\begin{aligned}
h_i^+ &= \frac{2}{h_{i+1}\left(h_{i+1}+h_i\right)} \\
h_i^- &= \frac{2}{h_i\left(h_{i+1}+h_i\right)}
\end{aligned}
\tag{2.6}
$$

If you want the approximation method described in the footnote

$$
\begin{aligned}
h_i^+ &= 4\frac{h_i}{\left(h_{i+1}^2+h_i^2\right)\left(h_{i+1}+h_i\right)} \\
h_i^- &= 4\frac{h_{i+1}}{\left(h_{i+1}^2+h_i^2\right)\left(h_{i+1}+h_i\right)}
\end{aligned}
\tag{2.7}
$$

you can specify `_PDE_approx = 2`*.*

2.2.2 The case of temporal non-uniform grids

In this case, you must use the `solvePDE2` procedure:

```
call solvePDE2(xFile,uFile,theta);
```

The variable `xFile` is a dataset which contains the values of the nodes t_m and $x_i^{(m)}$. The storage is the following:

$$
\begin{array}{|cccccccc|}
\hline
t_0 = t^- & x_0^{(0)} & x_1^{(0)} & x_2^{(0)} & \cdots & x_{N^{(0)}-2}^{(0)} & x_{N^{(0)}-1}^{(0)} & \cdot\ \cdot\ \cdot \\
t_1 & x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & \cdots & x_{N^{(1)}-2}^{(1)} & x_{N^{(1)}-1}^{(1)} & \cdot\ \cdot\ \cdot \\
\vdots & & & & \vdots & & & \cdot\ \cdot\ \cdot \\
t_{M-1} = t^+ & x_0^{(M-1)} & x_1^{(M-1)} & x_2^{(M-1)} & \cdots & x_{N^{(M)}-2}^{(M-1)} & x_{N^{(M)}-1}^{(M-1)} & \cdot\ \cdot\ \cdot \\
\hline
\end{array}
$$

The symbol $\cdot$ indicates a missing values. Let $N = \max N^{(m)}$ denotes the maximum number of the discretization points. Because $N^{(m)}$ may change with m, we adopt this stacking method

$$
\begin{bmatrix} t_m & \operatorname{vec}\left(\begin{bmatrix} \mathbf{x}^{(m)} \\ \mathbf{e}^{(m)} \end{bmatrix} \right)^\top \end{bmatrix}
$$

with $\mathbf{e}^{(m)}$ a vector of missing values of dimension $N - N^{(m)}$. The dimension of the database is then $M \times (N+1)$. The dataset `uFile` is built in the same way. We have

$$
\begin{array}{|cccccccc|}
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\ \cdot\ \cdot \\
t_0 = t^- & u_0^0 & u_1^0 & u_2^0 & \cdots & u_{N^{(0)}-2}^0 & u_{N^{(0)}-1}^0 & \cdot\ \cdot\ \cdot \\
t_1 & u_0^1 & u_1^1 & u_2^1 & \cdots & u_{N^{(1)}-2}^1 & u_{N^{(1)}-1}^1 & \cdot\ \cdot\ \cdot \\
\vdots & & & & \vdots & & & \cdot\ \cdot\ \cdot \\
t_{M-1} = t^+ & u_0^{M-1} & u_1^{M-1} & u_2^{M-1} & \cdots & u_{N^{(M)}-2}^{M-1} & u_{N^{(M)}-1}^{M-1} & \cdot\ \cdot\ \cdot \\
\hline
\end{array}
$$

Note that the values of x are not stored, and the first row contains only missing values.

2.3 Solution extracting

You could of course use the GAUSS commands to extract solution from the dataset `uFile`. The `readPDE` and `readPDE2` procedures are provided to make it easier. Their syntax are

```
data = readPDE(uFile,cn);
```

and

```
{x,u} = readPDE2(xFile,uFile,t);
```

The variable `cn` could take differents values. If `cn` is the string `''t''`, then `data` corresponds to the column vector $\{t_m\}$. We obtain the column vector $\{x_i\}$ by setting `cn` to `''x''`. We could also extract specific solutions u_i^m by using a 2×1 vector. We have

`cn`	`data`
`''t''` | `tm`	$N \times 1$ vector with the (i) entry (u_i^m)
`''x''` | `xi`	$M \times 1$ vector with the (m) entry (u_i^m)

For the `readPDE2` procedure, `t` can be a scalar (a specific value of t_m) or a vector (different values of t_m). If the dimension of `t` is $E \times 1$, the dimension of `x` and `u` is $N \times E$.

2.4 Using non-uniform grids

There exist different procedures for the management of non-uniform grids. For example, to generate the vector $\{x_i\}$, we can use the `generateGrid*` procedures. Uniform grids are obtained with the `generateGrid1` procedure:

```
x = generateGrid1(xmin,xmax,N);
```

`generateGrid2` can be used to obtain a non uniform grid with the second method of Bodeau, Riboulet and Roncalli [2000]:

```
x = generateGrid2(xmin,xmax,N,&invcdf);
```

`invcdf` is a procedure wich compute the quantile of the distribution $\mathbf{F}(x)$. The last method which is called the Tavella-Randall method corresponds to the `generateGrid3` method:

```
x = generateGrid3(xmin,xmax,N,xstar,alpha);
```

with `xstar` and `alpha` the value of the parameters $x^\star$ and α.

We can use the previous procedures directly to define the variable `x` for the procedure `solvePDE`. For `solvePDE2`, we have to build the dataset `xFile`. We can do that with the commands of GAUSS, but we have included a procedure `saveGrid` to make it easier. Its syntax is

```
call saveGrid(cn,&gridProc,xFile);
```

If `cn` is a scalar, `cn` corresponds to the number M of discretization points in t. In this case, the procedure `gridProc` takes the following form:

```
proc (2) = gridProc(m);
  local t,x;

  t = ...;  /* the value of $t_m$ */
  x = ...;  /* the vector of the values $x_i^{(m)}$ */

  retp(t,x);
endp;
```

If `cn` is a vector, `cn` corresponds to the vector $\{t_m\}$. In this case, the form of the procedure `gridProc` is

```
proc (1) = gridProc(tm,m);
  local x;

  x = ...;  /* the vector of the values $x_i^{(m)}$ */

  retp(x);
endp;
```

The procedure `gridProc` is called M times to build the dataset `xFile`. Note that `saveGrid` can be used with the `generateGrid*` procedures. Here is an example:

```
proc (1) = gridProc(t,m);
  local x;
  local xstar;

  xstar = 0.5*(xmin+xmax);

  if t < 0.2;
    x = generateGrid1(xmin,xmax,N);
  elseif t < 0.5;
    x = generateGrid1(xmin,xmax,2*N);
  elseif t < 1;
```

```
    x = generateGrid3(xmin,xmax,N,xstar,20);
  else;
    x = generateGrid3(xmin,xmax,N,xstar,20/t);
  endif;

  retp(x);
endp;
```

Note also that we can load the dataset `xFile` with the `loadGrid` procedure:

```
{t,x} = loadGrid(xFile);
```

To plot a grid, we employ the command `plotGrid`:

```
{psym,pline} = plotGrid(t,x,symbol,line,rotate);
```

`symbol` indicates the type of symbol to mark the nodes. If it is equal to 0, the nodes are not represented. `line` take the value one if we want to connect the nodes. `rotate` can be used to perform different rotation of the graphic. To adjust the size and color of the symbols, we can modify the two global variables `_pde_symsiz` and `_pde_symclr`.

2.5 Other procedures

2.5.1 The `derivePDE` procedure

We could employ the procedure `derivePDE` to compute the numerical first and second derivatives of the solution of a PDE problem

```
{d1,d2} = derivePDE(x,u);
```

2.5.2 The `FindIndex` procedure

`FindIndex` returns the indices of the elements of a vector x equal to the elements of a vector v. To understand how the procedure works, let's try an example:

```
new;
library pde;

xmin = -3;
xmax = 3;
Nx = 101;
hx = (xmax-xmin)/(Nx-1);

x = seqa(xmin,hx,Nx);

FindIndex(x,0|3);
       51.000000
       101.00000
indexcat(x,0);
          .
indexcat(x,3);
       101.00000
```

The `indexcat` procedure does not find the index of an element of x equal to 0 due to numerical truncation. In this case, you may use the `FindIndex` procedure.

2.5.3 The `solveTDG` procedure

`solveTDG` solves the tridiagonal system

$$[a; b; c]\, x = d$$

Its syntax is

```
x = solveTDG(a,b,c,d);
```

It is used by the procedures **solvePDE** and **solvePDE2** to solve the tridiagonal system. `solveTDG` requires on the dll file *pde.dll*, written in C. If you don't want to use it, you have to specify **#declare not_DLLs;**. It can be useful for Unix system. Nevertheless, we have included the C code in the *dlib* directory for Unix users. The .so library can then be created easily by changing the entry point.

Chapter 3

Command Reference

The following global variables and procedures are defined in **PDE**. They are the *reserved words* of **PDE**.

`derivePDE, FindIndex, generateGrid1, generateGrid2, generateGrid3, loadGrid, _pde_built, _pde_approx, _pde_aproc, _pde_bproc, _pde_computex, _pde_cproc, _pde_derivcond, _pde_dproc, _pde_dxmaxbound, _pde_dxminbound, _pde_Elliptic, _pde_eproc, _pde_eq, _pde_invsinh, _pde_ne, _pde_neumann, _pde_PrintIters, _pde_SaveLastIter, _pde_solvethesystem, _pde_spline, _pde_symclr, _pde_symsiz, _pde_tminbound, _pde_computeustar, _pde_writer, _pde_xmaxbound, _pde_xminbound, PDEset, plotGrid, readPDE, readPDE2, saveGrid, solvePDE, solvePDE2, solveTDG`

The default global control variables are

`_PDE_Elliptic`	0
`_PDE_approx`	1
`_PDE_PrintIters`	0
`_PDE_SaveLastIter`	0
`_PDE_Built`	0
`_PDE_symclr`	15
`_PDE_symsiz`	0

derivePDE

■ **Purpose**
Computes the numerical first and second derivatives of the solution of a PDE problem.

■ **Format**
{d1,d2} = derivePDE(x,u);

■ **Input**

x	$N \times E$ matrix, values of $x_i^{(m)}$
u	$N \times E$ matrix, values of u_i^m

■ **Output**

d1	$N \times E$ matrix, numerical first derivative
d2	$N \times E$ matrix, numerical second derivative

■ **Remark**
The second derivative is computed according to formula (2.6).

■ **Source**
src/pde.src

FindIndex

Purpose
Returns the indices of the elements of a vector x equal to the elements of a vector v.

Format
y = FindIndex(x,v);

Input

x	$N \times 1$ vector
v	$L \times 1$ vector

Output

y	$L \times 1$ vector, `y[i]` contains the indice of the first element of x which is equal to `v[i]`

Globals

_fcmptol	scalar (default = 1e-15) the procedure `FindIndex` uses _fcmptol to fuzz the comparison operations to allow for round off error

Remarks
The procedure `FindIndex` is similar to the GAUSS `indexcat` command. The main difference is that `FindIndex` returns only one index (or a missing value) for each value v_i. Note that the global variable `_fcmptol` is used to check the equality $x_{y_i} = v_i$.

Source
src/pde.src

generateGrid1

- **Purpose**

  Generates a uniform grid.

- **Format**

  x = generateGrid1(xmin,xmax,N);

- **Input**

xmin	scalar, value of x^-
xmax	scalar, value of x^+
N	scalar, number of discretization points

- **Output**

x	$N \times 1$ vector, values of the grid x_i

- **Globals**

- **Source**

  src/pde.src

generateGrid2

■ Purpose
Generates a non uniform grid with the inverse distribution method.

■ Format
x = generateGrid2(xmin,xmax,N,&invcdf);

■ Input

xmin	scalar, value of x^-
xmax	scalar, value of x^+
N	scalar, number of discretization points
&invcdf	pointer to a procedure which computes the inverse of the distribution $\mathbf{F}(x)$

■ Output

x	$N \times 1$ vector, values of the grid x_i

■ Globals

■ Source
src/pde.src

generateGrid3

■ **Purpose**

Generates a non uniform grid with the Tavella-Rendall method.

■ **Format**

x = generateGrid3(xmin,xmax,N,xstar,alpha);

■ **Input**

xmin	scalar, value of x^-
xmax	scalar, value of x^+
N	scalar, number of discretization points
xstar	scalar, value of the parameter $x^\star$
alpha	scalar, value of the parameter α

■ **Output**

x	$N \times 1$ vector, values of the grid x_i

■ **Globals**

■ **Source**

src/pde.src

loadGrid

■ **Purpose**

Loads the dataset `xFile`.

■ **Format**

{t,x} = loadGrid(xFile);

■ **Input**

xFile	string, name of the grid dataset file

■ **Output**

t	$M \times 1$ vector, values of t_m
x	$N \times M$ matrix, values of $x_i^{(m)}$

■ **Globals**

■ **Source**

src/pde.src

PDE

■ **Purpose**

Initializes the PDE problem.

■ **Format**

call PDE(aProc,bProc,cProc,dProc,eProc,tminBound,
    xminBound,xmaxBound,DxminBound,DxmaxBound);

■ **Input**

aProc	scalar, pointer to a procedure which computes $a(t,x)$
bProc	scalar, pointer to a procedure which computes $b(t,x)$
cProc	scalar, pointer to a procedure which computes $c(t,x)$
dProc	scalar, pointer to a procedure which computes $d(t,x)$
eProc	scalar, pointer to a procedure which computes $e(t,x)$ – or – scalar 0
tminBound	scalar, pointer to a procedure which computes $u_{(t^-)}(x)$
xminBound	scalar, pointer to a procedure which computes $u_{(x^-)}(t)$
xmaxBound	scalar, pointer to a procedure which computes $u_{(x^+)}(t)$
DxminBound	scalar, pointer to a procedure which computes $u'_{(x^-)}(t)$
DxmaxBound	scalar, pointer to a procedure which computes $u'_{(x^+)}(t)$

■ **Output**

■ **Globals**

__output	scalar 1 – print information about the PDE problem 0 – no printing

■ **Source**

src/pde.src

PDEset

■ Purpose

Resets the global control variables declared in *PDE.DEC*.

■ Format

PDEset;

■ Remarks

The default global control variables are

`_PDE_Elliptic`	0
`_PDE_approx`	1
`_PDE_PrintIters`	0
`_PDE_SaveLastIter`	0
`_PDE_Built`	0
`_PDE_symclr`	15
`_PDE_symsiz`	0

■ Source

src/pde.src

plotGrid

Purpose

Plots the grid.

Format

{psym,pline} = plotGrid(t,x,symbol,line,rotate);

Input

t	$M \times 1$ vector, values of t_m
x	$N \times M$ matrix, values of $x_i^{(m)}$
symbol	scalar, type of symbol
line	scalar, 1 to connect the nodes
rotate	scalar, controls the rotation

Output

psym	matrix of the symbols
pline	matrix of the lines

Globals

_pde_symclr	scalar, color of the symbol
_pde_symsiz	scalar, size of the symbol

Remarks

To draw the grid, we set

$$\begin{aligned} \texttt{\_psym} &= \texttt{psym;} \\ \texttt{\_pline} &= \texttt{pline;} \end{aligned}$$

Source

src/pde.src

readPDE

■ **Purpose**

Extracts solution of the database `uFile` computed by `solvePDE`.

■ **Format**

data = readPDE(uFile,cn);

■ **Input**

uFile	string, name of the solution dataset file
cn	scalar or vector 2×1

■ **Output**

data	$M \times 1$ vector, values t_m if cn is the string `"t"`
	$N \times 1$ vector, values x_i if cn is the string `"x"`
	– or –
	$M \times 1$ vector, values $u(t, x_i)$ if cn is equal to `"x"|xi`
	$N \times 1$ vector, values $u(t_m, x)$ if cn is equal to `"t"|tm`

■ **Source**

src/pde.src

readPDE2

■ **Purpose**

Extracts solution of the database `uFile` computed by `solvePDE2`.

■ **Format**

{x,u}= readPDE2(xFile,uFile,t);

■ **Input**

xFile	string, name of the grid dataset file
uFile	string, name of the solution dataset file
t	vector $E \times 1$, values of t_m

■ **Output**

x	$N \times E$, values of $x_i^{(m)}$
u	$N \times E$, values of $u\left(t_m, x_i^{(m)}\right)$

■ **Source**

src/pde.src

saveGrid

■ **Purpose**

Saves the dataset `xFile`.

■ **Format**

call saveGrid(cn,&gridProc,xFile);

■ **Input**

cn	scalar or vector
&gridProc	pointer to a procedure which compute t_m and $x_i^{(m)}$
xFile	string, name of the grid dataset file

■ **Output**

■ **Globals**

■ **Source**

src/pde.src

solvePDE

■ **Purpose**
Solves the PDE problem with non uniform grids.

■ **Format**
call solvePDE(t,x,uFile,theta);

■ **Input**

t	vector $M \times 1$, values of t_m
x	vector $N \times 1$, values of x_i
uFile	string, name of the solution dataset file
theta	scalar, value of the parameter θ of the θ-scheme

■ **Output**

■ **Globals**

_PDE_approx	scalar, the approximation method of the second derivative 1 for the first method 2 for the second method
_PDE_Elliptic	scalar (default = 0) 0 if the PDE problem is not an elliptic problem 1 if the PDE problem is an elliptic problem
_PDE_PrintIters	scalar (default = 0) 0 – does not print iterations I – printing after each I iterations
_PDE_SaveLastIter	scalar (default = 0) 0 for saving the solution for all the iterations m 1 for saving only the last solution for $t_m = t^+$
__output	scalar (default = 0) 1 – print informations about the algorithm and the mesh ratios 0 – no printing

■ **Remarks**
To extract the solution, you may use the `readPDE` procedure.

■ **Source**
src/pde.src

solvePDE2

■ **Purpose**

Solves the PDE problem with temporal non uniform grids.

■ **Format**

call solvePDE2(xFile,uFile,theta);

■ **Input**

xFile	string, name of the grid dataset file
uFile	string, name of the solution dataset file
theta	scalar, value of the parameter θ of the θ-scheme

■ **Output**

■ **Globals**

_PDE_approx	scalar, the approximation method of the second derivative 1 for the first method 2 for the second method
_PDE_Elliptic	scalar (default = 0) 0 if the PDE problem is not an elliptic problem 1 if the PDE problem is an elliptic problem
_PDE_PrintIters	scalar (default = 0) 0 – does not print iterations I – printing after each I iterations
_PDE_SaveLastIter	scalar (default = 0) 0 for saving the solution for all the iterations m 1 for saving only the last solution for $t_m = t^+$
__output	scalar (default = 0) 1 – print informations about the algorithm and the mesh ratios 0 – no printing

■ **Remarks**

To extract the solution, you may use the `readPDE2` procedure.

■ **Source**

src/pde.src

Chapter 4

The tutorial

The programs used for the article "Non-uniform grids for PDE in finance" are included in the examples\pde directory. Moreover, we have added some tutorial files with a very simple example. These files cover all the procedures. Because the use of these procedures are very simple, we just do some remarks and do not provide a full description of them.

The *tutor1.prg–tutor11.prg* programs consider the linear parabolic PDE problem defined by

$$\begin{aligned} a(t,x) &= \frac{1}{2}x^2 \\ b(t,x) &= x \\ c(t,x) &= 1 \\ d(t,x) &= -\left(3x^2+x\right)e^{-t} \end{aligned}$$

$\mathfrak{X}$ is set to $[-1,1]$ and we have

$$\begin{aligned} u(t,-1) = -1 \quad &\bigvee \quad u_x(t,-1) = -e^{-t}+1 \\ u(t,1) = 2e^{-t}+1 \quad &\bigvee \quad u_x(t,1) = 3e^{-t}+1 \end{aligned}$$

The solution of the Cauchy problem with $u(0,x) = x^2+2x$ is

$$u(t,x) = \left(x^2+x\right)e^{-t}+x$$

- tutor1: initialisation of the PDE problem with two Dirichlet conditions.
- tutor2: initialisation of the PDE problem with two Neumann conditions.
- tutor3: mixing of Dirichlet and Neumann conditions.
- tutor4: incompatibility of Dirichlet and Neumann conditions.
- tutor5: generates and plots a uniform grids.
- tutor6: generates and plots a non uniform grids (based on the inversion method of distribution).
- tutor7: generates and plots a non uniform grids (based on the Tavella-Rendall method).
- tutor8: solves the PDE problem.
- tutor9: extracts solution computed with the program *tutor8*.
- tutor10: extracts solution computed with the program *tutor8*.
- tutor11: generates a temporal non uniform grids and solves the PDE problem.
- tutor12: solves an elliptic PDE problem.

Index