NUMERICAL METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS

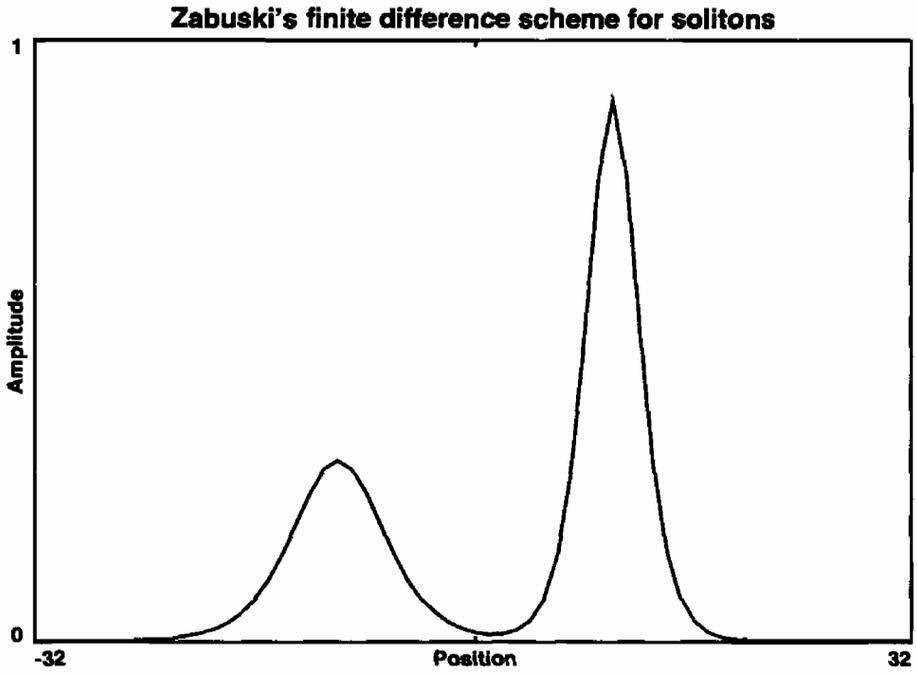
http://www.fusion.kth.se/courses/pde

André Jaun <jaun@fusion.kth.se>

assisted by

Johan Hedin <johanh@fusion.kth.se>
Thomas Johnson <thomasj@fusion.kth.se>

August 11, 1999



Graduate Student Course, Alfvén Laboratory
Royal Institute of Technology
Stockholm, June 1999

Cover

The cover is a collage of the students web pages presenting their solutions to the exercises and the lecture notes. The image processing is done with Gimp[1].

Production of this document

This document is typeset with LaTeX 2_ε[2] using the macros in the `latex2html`[3] package on Sun[4] and Linux[5] platforms.

TRITA-ALF-1999-05

Stockholm 1999 KTH Högskoletryckeriet

[1] http://www.gimp.org
[2] http://www.tug.org/
[3] http://cdc-server.cdc.informatik.tu-darmstadt.de/%7Elatex2html
[4] http://www.sun.com
[5] http://www.linux.org

Contents

PREFACE

This is the first web edition of a course taught 1997–1999 at the Royal Institute of Technology (KTH, Stockholm) to graduate students in physics, engineering and quantitative social sciences.

Realizing the need of an introductory text giving a comparative overview of a variety of methods commonly used to solve partial differential equations, and more than ever convinced that numerical techniques are best illustrated directly with a computer, new tools for the electronic publishing have been used to create a highly interactive document. Following hyper-links from the lecture notes directly into the relevant sections of the program and executing the JBONE applet in the web page with different parameters and initial conditions, it is now possible to illustrate every step from the formulation of a problem, the discretization, until the final properties of numerical schemes, with practical examples. Comparisons show what are the advantages and drawbacks of different approaches, which are generally treated separately in more advanced and specialized books. As a by-product of this new approach of teaching, students rapidly acquired the knowledge for the electronic publishing of their home assignments and learned how to interact and help each other with newsgroups — two very useful tools when the research is carried out in international collaborations. The complete source of the JBONE java program can be obtained free of charge for personal use by sending an e-mail request to <jaun@fusion.kth.se>.

For this web edition immediately accessible to everyone on the internet, I would like first to thank Kurt Appert (EPFL, Lausanne) for the inspiration and guidance he provided ever since I became a student of his own course *Expérimentation Numérique.* Johan Carlsson, Johan Hedin and Thomas Jonsson (KTH, Stockholm) have one after the other been responsible for the Monte-Carlo method, Johan Hedin providing everyone of us with rapid and very competent advice in a sophisticated, geographically distributed, multi-platform environment. Ambrogio Fasoli (MIT, Cambridge), with whom I have the pleasure to collaborate for research by comparing numerical solutions of PDEs with experiments, carried out the measurement of Alfvén instabilities illustrating how important aliasing can be for the digital data acquisition. I am finally greatful for the suggestions, criticisms and encouragements from the students, who will keep on forging the course as it evolves in the future.

André JAUN, Stockholm July 1999

1 INTRODUCTION

1.1 Objectives

1. **By focusing** on the numerical aspects, acquire a **working** knowledge of a **variety** of methods and equations.

   Numerical methods. Finite Differences (FD), finite elements (FEM), Fourier transform (FFT), Monte-Carlo method (MCM), characteristics (CHA), wavelets (WAW).

   Prototype equations. Advection, diffusion, Burger, Korteweg-DeVries (KdV), Black-Scholes.

2. **By context**, assimilate new tools for your research.

   Programming. Structure, language (java, fortran).

   E-publishing. WWW making international collaborations active.

   Sharing knowledge. Newsgroup to help each other in the class.

3. **Experiment** with a new form of "electronically-assisted" teaching.

   Lectures. Guide you through the course, provide a common ground for interaction, raise and answer questions.

   Exercises. The *core of the course, they are crucially important* ! To be delivered electronically to <jaun@fusion.kth.se> no later than 6am the day of the following lesson.

   Project. One week opportunity to acquire some depth. Four A4 pages to be delivered electronically to <jaun@fusion.kth.se> until August 15, 1999.

1.2 Differential Equations

Ordinary differential equations. ODEs involve sets of **first order** equations with one variable

$$\frac{d^2X}{dt^2} = \frac{F}{m} \quad \longrightarrow \quad \frac{d}{dt}\begin{pmatrix} X \\ V \end{pmatrix} = \begin{pmatrix} V \\ \frac{F}{m} \end{pmatrix} \tag{1}$$

Check [1.6] or try a **Runge-Kutta** integration, but careful if the problem is **stiff** (exercise 1.5), i.e. involves two very different scale-lengths which strongly limits the step size.

Partial differential equations. PDEs have at least 2 variables involving space (**boundary value**) or/and time (**initial value**) problems [1.6]:

$$A\frac{\partial^2 f}{\partial t^2} + 2B\frac{\partial^2 f}{\partial t \partial x} + C\frac{\partial^2 f}{\partial x^2} + D(t, x, \frac{\partial f}{\partial t}, \frac{\partial f}{\partial x}) = 0 \tag{2}$$

Boundary (BC) or/and initial conditions (IC) have to be imposed accordingly

$$BC : af + b\frac{\partial f}{\partial x} = c\,,\ \forall x \in \partial\Omega\ \forall t \qquad IC : f(x, t=0) = f_0(x)\,,\ \forall x \in \Omega \tag{3}$$

and are called *Dirichlet (b=0), Neumann (a=0), Robin (c=0), periodic* $f(x_L) = f(x_R)$ where $\{x_L, x_R\} \in \partial\Omega\ \forall t$, and sometimes *outgoing-wave* if the domain is open.

Characteristics. Are trajectories along which discontinuities and initial conditions propagate; using the chain rule, it is customary to classify second order equations

$$\frac{\partial^2 f}{\partial x^2}\left[A\left(\frac{\partial x}{\partial t}\right)^2 + 2B\frac{\partial x}{\partial t} + C\right] = 0 \implies \frac{\partial x}{\partial t} = A^{-1}\left(-B \pm \sqrt{B^2 - AC}\right) \tag{4}$$

as *elliptic* if $B^2 - AC < 0$ (no real characteristic, Laplace eq.), *parabolic* if $B^2 - AC = 0$ (one real characteristic, heat eq.), *hyperbolic* if $B^2 - AC > 0$ (two real characteristics, wave eq.).

Moments. Phase space integration $\mathcal{M}_K = \int_\Omega dV x^K [\cdots] = 0$ generally yields conserved quantities such as density (K=0) or momentum (K=1) that are useful for **self-consistency checks.**

1.3 Prototype problems

Differential calculus is at the heart of many areas of science and engineering because of the description it provides for interactions **locally**, relating infinitesimal changes at the microscopic scale to the macroscopic scale of a system. Using initial value problems $t \geq t_0$ in a 1D periodic slab $x \in [0; L[$ for illustration, listed below are some the most important processes with their prototype equations.

Advection. Also called *convection*, it models the streaming of elements in a fluid. Advection generally appears in an Eulerian representation through a convective derivative

$$\frac{d}{dt}f \equiv \frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = 0 \tag{5}$$

which, for a constant advection speed u, can be solved analytically as $f(x,t) = f_0(x-ut)$ $\forall f_0 \in \mathcal{C}(\Omega)$, showing explicitly the characteristics $x = ut$ [1.2]. Use the JBONE applet [6] to compute a numerical solution for a Gaussian pulse advected using the Lagrangian CIP/FEM method later discussed in sect.6.5.

Note that the wave equation can be written in flux-conservative form as

$$\frac{\partial^2 h}{\partial t^2} - v^2\frac{\partial^2 h}{\partial x^2} = 0 \quad \Longleftrightarrow \quad \frac{\partial}{\partial t}\begin{pmatrix} f \\ g \end{pmatrix} + \frac{\partial}{\partial x}\left[\begin{pmatrix} 0 & -v \\ -v & 0 \end{pmatrix}\cdot\begin{pmatrix} f \\ g \end{pmatrix}\right] = 0 \tag{6}$$

Diffusion. Can be related at the microscopic scale to a *random walk* process (exercise 1.5) with a prototype equation

$$\frac{\partial f}{\partial t} - D\frac{\partial^2 f}{\partial x^2} = 0 \tag{7}$$

where $D \geq 0$ is the diffusion coefficient. In a homogeneous medium, the equation combining advection and diffusion can be solved analytically in terms of a **Green's function** (exercise 1.5) and yields

$$f(x,t) = \int_{-\infty}^{+\infty} f_0(x_0)G(x-x_0-ut,t)dx_0 \tag{8}$$

$$G(x-x_0-ut,t) = \frac{1}{\sqrt{\pi 4Dt}}\exp\left(-\frac{(x-x_0-ut)^2}{4Dt}\right) \tag{9}$$

Check the document on-line to see an example of a numerical solution for the advection-diffusion of a box computed with the finite element method from sect.3.6.

Fourier analyzing in time and space $f \sim \exp i(kx-\omega t)$ yields the dispersion relation

$$\omega = -iDk^2 \tag{10}$$

showing that short wavelengths (large k) are more strongly damped. Change the initial conditions in the applet to `Sinus` and try to model this numerically.

Dispersion. Occurs when different wavelengths propagate with different phase velocities. Take for example the third order dispersion equation

$$\frac{\partial f}{\partial t} - \frac{\partial^3 f}{\partial x^3} = 0 \tag{11}$$

Fourier analyze in time and space $f \sim \exp i(kx-\omega t)$ and calculate the phase velocity

$$-if(\omega - k^3) = 0 \quad \Longrightarrow \quad \frac{\omega}{k} = k^2 \tag{12}$$

showing that short wavelengths (large k) propagate faster than longer wavelengths. In the example below (Eq.15), this explains why large amplitude *solitons* with shorter wavelengths propagate more rapidly than the low amplitude ones.

[6] Appears only in the electronic version of the lecture notes `http://www.fusion.kth.se/~jaun/Teach.html`

Wave-breaking. This non-linearity is best experienced when surfing on a see shore, where the shallow water forces the wave fronts to steepen and break ! In our context, the phenomenon is best understood from the advection equation), replacing the advection speed by a term proportional to the amplitude $u = f$

$$\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial x} = 0 \tag{13}$$

Sharp wave fronts develop and the function eventually becomes multi-valued, causing the wave (and our numerical schemes !) to break. Sometimes, however, the wave-breaking is balanced by a competing mechanism. This is the case in the **Burger** equation for **shock-waves**

$$\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial x} - D\frac{\partial^2 f}{\partial x^2} = 0 \tag{14}$$

where the creation of a sharp front (with short wavelengths) is limited by diffusion, affecting mainly the shorter wavelengths (Eq.10). Check the document on-line to see a shock formation computed using a 3-levels explicit finite difference scheme from sect.2.5.

Another example, resulting this time from the balance between wave-breaking and dispersion (Eq.12), is the famous **Korteweg-DeVries** equation for **solitons**

$$\frac{\partial f}{\partial t} + f\frac{\partial f}{\partial x} + b\frac{\partial^3 f}{\partial x^3} = 0 \tag{15}$$

The on-line document shows indeed that large amplitude (short wavelengths) solitons propagate faster than low amplitude (long wavelength) solitons.

1.4 Discretization

To solve differential equations numerically, a discrete set of values $\{f_j\}$, $j = 1, N$ is evolved by taking small steps in time Δt to **approximate** what should be a continuous function of space and time $f(x,t)$, $x \in [a;b]$, $t \geq t_0$. A variety of approaches are possible, the only requirement being that the approximation **converges** locally (exercise 1.5) to the exact solution as the sampling becomes accurate enough $N \to \infty$, $\Delta t \to 0$. Unfortunately, there is **no universal method !** Rather than adopting the favorite of your "local guru", a reasonable choice for a discretization really should depend on

- the structure of the solution (continuity, regularity, precision), the post-processing (filters) and the diagnostics (Fourier spectrum) that might be required anyway,
- the boundary conditions, which can be difficult to implement with some methods,
- the structure of the differential operator (formulation, memory×time consumption, numerical stability) and maybe the computer architecture (vectorization, parallelization).

This course is a lot about advantages and limitations of different methods, enabling you to make an optimal choice and consult advanced literature if necessary. Let's first examine the **spatial discretization** and the boundary conditions.

Sampling on a mesh (grid). The main advantage is certainly its **simplicity**: a finite number of values are sampled $f(x) \to \{(x_j; f_j)\}$ generally on a homogeneous mesh $x_j = j\Delta x$, $j = 1, N$. Boundary values appear explicitly as x_1 and x_N. As suggested by figure 1, the function is **unknown almost everywhere** except on the grid points, from which finite differences can be evaluated from a Taylor expansion to approximate derivatives — again for well defined locations:

$$f_{j+1} \equiv f(x_j + \Delta x) = f_j + \Delta x f'_j + \frac{\Delta x^2}{2} f''_j + \mathcal{O}(\Delta x^3)$$

$$f_{j-1} \equiv f(x_j - \Delta x) = f_j - \Delta x f'_j + \frac{\Delta x^2}{2} f''_j + \mathcal{O}(\Delta x^3)$$

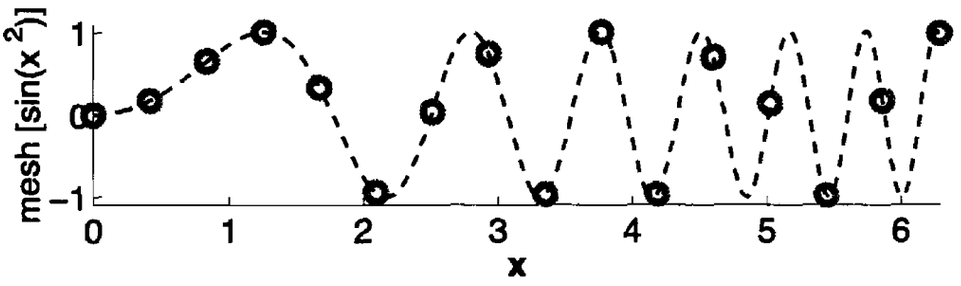


Figure 1: Approximation of $\sin(x^2)$ on a homogeneous mesh.

These are easily combined and give

$$f'_j = \frac{f_j - f_{j-1}}{\Delta x} + \mathcal{O}(\Delta x) \qquad f'_j = \frac{f_{j+1} - f_j}{\Delta x} + \mathcal{O}(\Delta x) \tag{16}$$

$$f'_j = \frac{f_{j+1} - f_{j-1}}{2\Delta x} + \mathcal{O}(\Delta x^2) \tag{17}$$

Because this approximation is not compact, the convergence depends on the approximation of the derivatives (e.g. Eq.17) and the interpolation between mesh points [1.6].

Finite element (FEM) functions. Following the spirit of Hilbert space methods, the function $f(x)$ decomposed on a *complete* set of *nearly orthogonal* basis functions $e_j \in \mathcal{B}$

$$f(x) = \sum_{j=1}^{N} f_j e_j(x) \tag{18}$$

spanning only as far as to the neighboring mesh points. Most common are the normalized "roof-top" FEM functions

$$e_j(x) = \begin{cases} (x - x_{j-1})/(x_j - x_{j-1}) & x \in [x_{j-1}; x_j] \\ (x_{j+1} - x)/(x_{j+1} - x_j) & x \in [x_j; x_{j+1}] \end{cases} \tag{19}$$

which yield a piecewise linear approximation for $f(x)$ and a piecewise constant derivative $f'(x)$ **defined almost everywhere** in the interval $[x_1; x_N]$. Boundary conditions are incor-

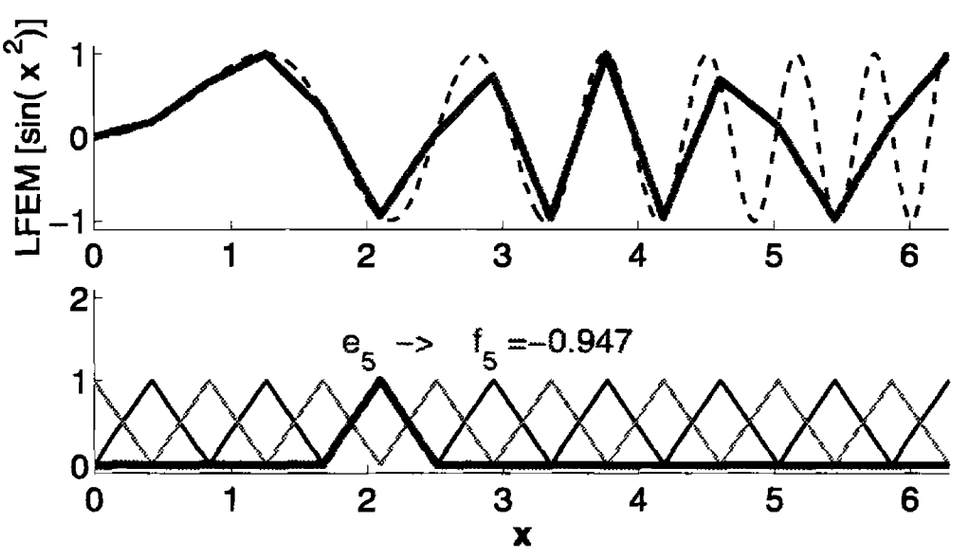


Figure 2: Approximation of $\sin(x^2)$ with roof-top linear finite elements.

porated by modifying the functional space $\mathcal{B}$ (e.g. taking "unilateral roofs" at the boundaries). Generalization to a "piecewise constant" or higher order "quadratic" and "cubic"

FEM is possible [1.6]. Important is the capability of **densifying the mesh** for a better resolution of short spatial scales. Fig.2 doesn't exploit that, but illustrates instead what happens when the numerical resolution becomes insufficient: around 20 linear (and 2 cubic) FEM per wavelength are usually required to achieve a precision below 1% but a minimum of 2 is of course necessary to resolve the oscillation.

Splines. Given an approximation on an inhomogeneous mesh, the idea of splines is to provide a global **interpolation** which is continuous up to a certain derivative. Using a cubic polynomial with tabulated values for the function and the second derivative, this is achieved by writing

$$f(x) = Af_j + Bf_{j+1} + Cf''_j + Df''_{j+1} \tag{20}$$

$$\begin{array}{lllllll} A(x) & = & (x_{j+1}-x)/(x_{j+1}-x_j) & & B(x) & = & 1-A \\ C(x) & = & (x_{j+1}-x_j)^2(A^3-A)/6 & & D(x) & = & (x_{j+1}-x_j)^2(B^3-B)/6 \end{array}$$

from which it is straight forward to derive

$$f'(x) = \frac{f_{j+1}-f_j}{x_{j+1}-x_j} - \frac{3A^2-1}{6}(x_{j+1}-x_j)f''_j + \frac{3B^2-1}{6}(x_{j+1}-x_j)f''_{j+1} \tag{21}$$

$$f''(x) = Af''_j + Bf''_{j+1} \tag{22}$$

Usually f''_i, $i = 1, N$ **is calculated**, requiring that $f'(x)$ be continuous from one interval to another: using (Eq.21) this yields a tridiagonal system for $j = 2, N-1$

$$\frac{x_j - x_{j-1}}{6}f''_{j-1} + \frac{x_{j+1}-x_{j-1}}{3}f''_j + \frac{x_{j+1}-x_j}{6}f''_{j+1} = \frac{f_{j+1}-f_j}{x_{j+1}-x_j} - \frac{f_j - f_{j-1}}{x_j - x_{j-1}} \tag{23}$$

leaving two conditions to be specified for the boundaries f'_1 and f'_N (Eq.21) [1.6].

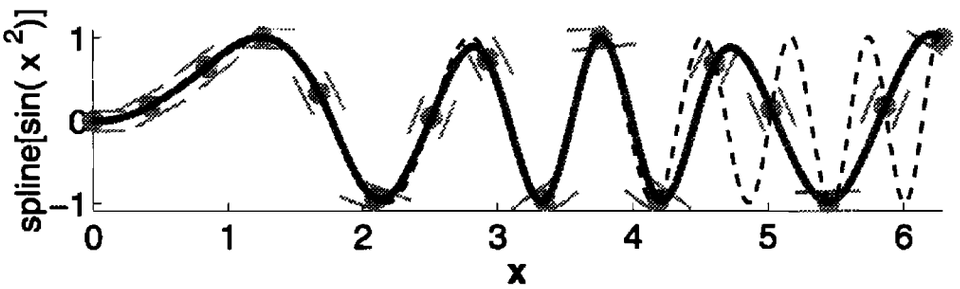


Figure 3: Approximation of $\sin(x^2)$ with cubic splines.

The figure 3 illustrates the procedure and shows the excellent quality of a cubic approximation until it breaks down at the limit of 2 mesh points per wavelength.

Harmonic functions. This approximation is based on the discrete Fourier transform (DFT) using a **regular mesh** and a **periodicity** L. With the notations $x_m = m\Delta x = mL/M$ and $k_m = 2\pi m/L$, the forward and backward transformations are given by:

$$\widehat{f}(k_m) = \frac{1}{M}\sum_{j=0}^{M-1} f(x_j)W^{-k_m x_j} \quad ; \qquad W = exp(2\pi i/M) \tag{24}$$

$$f(x_j) = \sum_{m=0}^{M-1} \widehat{f}(k_m)W^{+k_m x_m} \tag{25}$$

If M is a power of 2, the number of operation can be dramatically reduced from $8M^2$ to $M \log_2 M$ in the so-called **fast Fourier transform (FFT)** applying recursively the decomposition

$$\widehat{f}(k_m) = \frac{1}{M} \sum_{j=0}^{2^M-1} f(x_j) W^{-k_m x_j} = \sum_{j \text{ even}} + \sum_{j \text{ odd}} = \tag{26}$$

$$= \frac{1}{M} \sum_{j=0}^{2^{M-1}-1} f(x_{2j}) (W^2)^{-k_m x_j} + \frac{W^{-k_m}}{M} \sum_{j=0}^{2^{M-1}-1} f(x_{2j+1}) (W^2)^{-k_m x_j} \tag{27}$$

with $W^2 = \exp(2i\pi/2^{M-1})$ until a sum of DFT of length $M = 2$ is obtained. Figure 4 illustrates how the approximation of a periodic square wave converges with an increasing resolution; note the **Gibbs phenomenon** showing that we will never really get there...

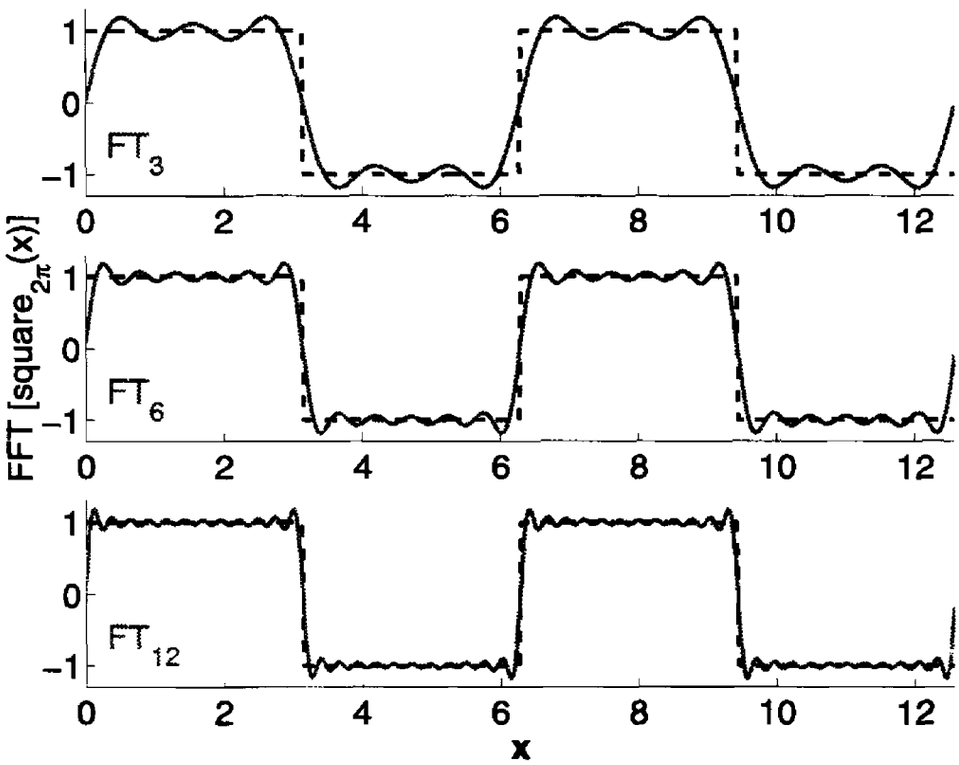


Figure 4: Square wave $\sum_{m=0}^{M} \frac{2}{\pi(2m+1)} \sin\left(\frac{(2m+1)\pi x}{L}\right)$ with $M = 3, 6, 12$.

It should be no surprise to anyone to hear that harmonic decompositions are well suited for smooth global functions with long wavelengths $\lambda \sim L$ which yield a rather narrow spectrum $|k| \leq 2\pi/\lambda \ll \pi/\Delta x$. Finally note that the convergence is not polynomial and that the implementation of non-periodic boundary conditions can be problematic.

Wavelets. Starting with a coarse (global) approximation, the first idea behind wavelets is to *successively refine* the representation and store the difference from one scale to the next

$$V_J = V_{J-1} \oplus W_{J-1} = \ldots = V_0 \oplus W_0 \oplus \ldots W_{J-1} \tag{28}$$

This is illustrated with *Haar wavelets* in Fig.5, showing that the piecewise constant approximation at the level V4 can be brought to the higher level V5 by adding a correction W4.

Appropriate for integral equations and best suited for the understanding, *Haar* wavelets are however not practical for the evaluation of derivatives in PDEs. In the spirit of the FFT,

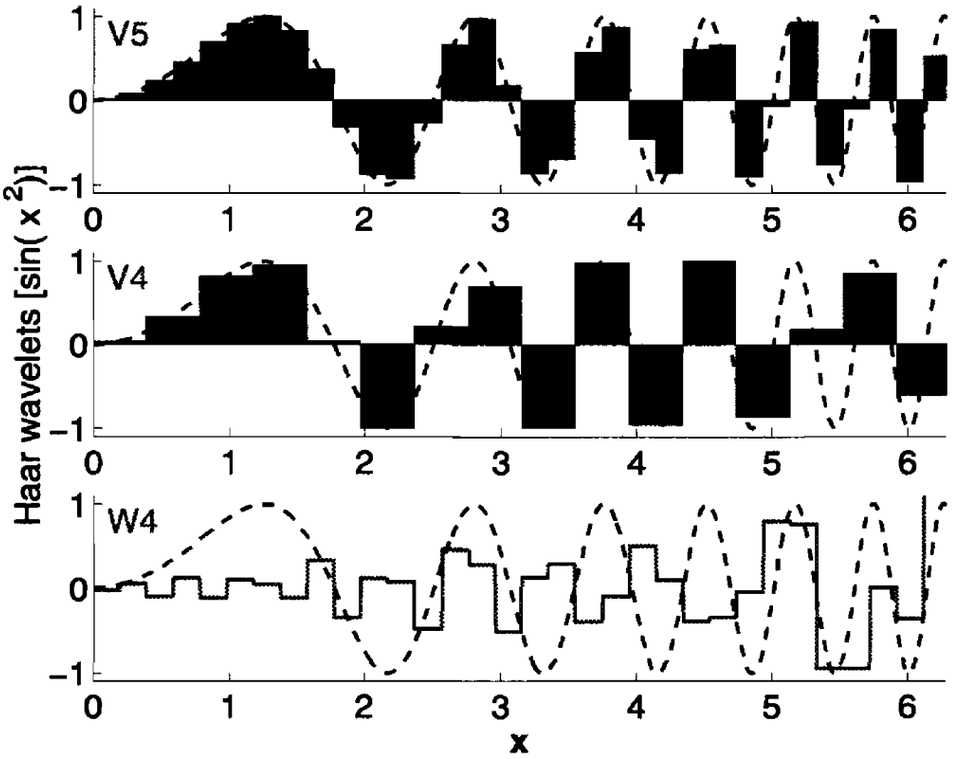


Figure 5: Successive approximation of $\sin(x^2)$ using *Haar* wavelets.

Daubechies proposed a fast $\mathcal{O}(N)$ linear discrete wavelet transformation (DWT)

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & & & \\ c_3 & -c_2 & c_1 & -c_0 & & & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & & & \\ & & c_3 & -c_2 & c_1 & -c_0 & & & & & & \\ \vdots & \vdots & & & & & \ddots & & & & & \\ & & & & & & c_0 & c_1 & c_2 & c_3 & & \\ & & & & & & c_3 & -c_2 & c_1 & -c_0 & & \\ c_2 & c_3 & & & & & & & c_0 & c_1 & & \\ c_1 & -c_0 & & & & & & & c_3 & -c_2 & & \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_5 \\ f_6 \\ f_7 \\ f_8 \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{g}_1 \\ \tilde{f}_2 \\ \tilde{g}_2 \\ \vdots \\ \tilde{f}_3 \\ \tilde{g}_3 \\ \tilde{f}_4 \\ \tilde{g}_4 \end{bmatrix} \rightarrow \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{g}_1 \\ \tilde{g}_2 \\ \tilde{g}_3 \\ \tilde{g}_4 \end{bmatrix} \tag{29}$$

$$c_0 = (1+\sqrt{3})/4\sqrt{4} \qquad c_1 = (3+\sqrt{3})/4\sqrt{4} \qquad c_2 = (3-\sqrt{3})/4\sqrt{4} \qquad c_3 = (1-\sqrt{3})/4\sqrt{4}$$

called DAUB4, which is applied successively with the permutation to the function $\tilde{f}$ until only the first two components remain. The inverse simply is obtained by reversing the procedure and using the inverse matrix:

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_5 \\ f_6 \\ f_7 \\ f_8 \end{bmatrix} = \begin{bmatrix} c_0 & c_3 & & & \cdots & & & & c_2 & c_1 \\ c_1 & -c_2 & & & \cdots & & & & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & & & & & & \\ c_3 & -c_0 & c_1 & -c_2 & & & & & & \\ & & & & \ddots & & & & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & \\ & & & & & & & c_2 & c_1 & c_0 & c_3 \\ & & & & & & & c_3 & -c_0 & c_1 & -c_2 \end{bmatrix} \begin{bmatrix} \tilde{f}_1 \\ \tilde{g}_1 \\ \tilde{f}_2 \\ \tilde{g}_2 \\ \vdots \\ \tilde{f}_3 \\ \tilde{g}_3 \\ \tilde{f}_4 \\ \tilde{g}_4 \end{bmatrix} \leftarrow \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \\ \tilde{f}_4 \\ \vdots \\ \tilde{g}_1 \\ \tilde{g}_2 \\ \tilde{g}_3 \\ \tilde{g}_4 \end{bmatrix} \tag{30}$$

Approximations with DAUB4 have the peculiar property that the derivative exists only **almost everywhere**. Even if wavelets are not smooth, they still can represent exactly piecewise polynomial functions of arbitrary slope — the cusps in the wavelets all cancel exactly out ! Figure 6 illustrates how the solution converges when the first 2^4 DWT components of DAUB4[$\sin(x^2)$] are taken to a higher level of refinement with 2^5 components.

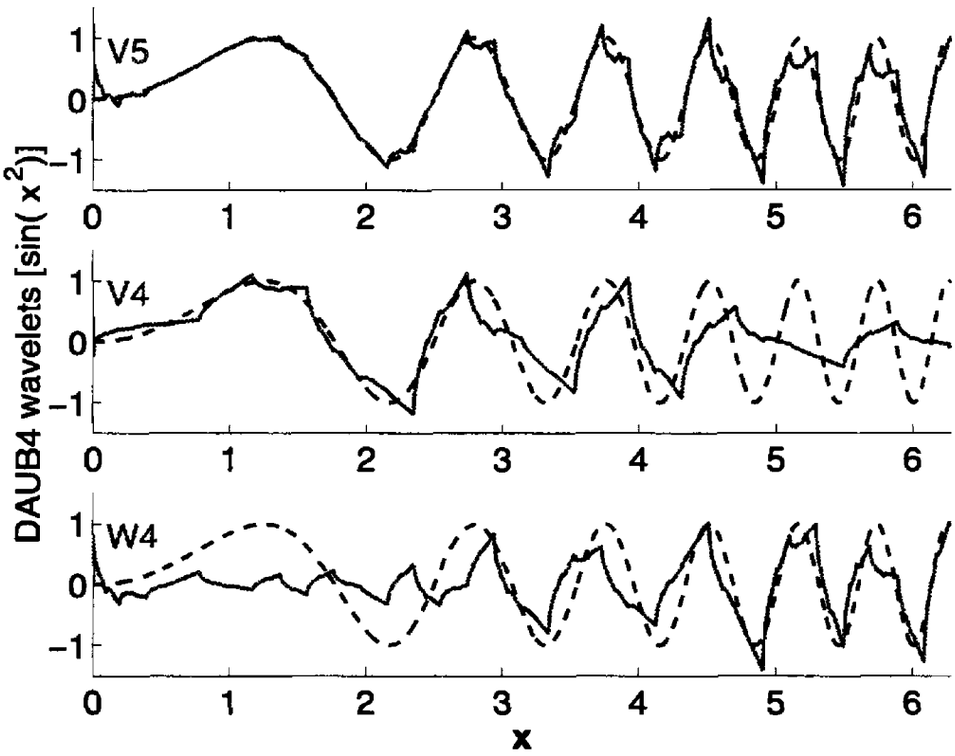


Figure 6: Successive approximation of $\sin(x^2)$ using DAUB4 wavelets.

Note that apart from *Haar* and *Daubechies* wavelets, a whole familly can be generated with a cascading process very nicely explained and illustrated with the Wavelet Cascade Applet[7]. Wavelets are still relatively new and remain a matter of active research [1]: they have so far been proven useful in approximation theory and are **currently being developed** both for PDE and integral equations [1.6]. Apparent in the figures 5 and 6 is the **problem with boundary conditions**; another issue which will become clear with the coming sections is the difficulty of calculating inexpensive inner products.

Quasi-particles. Yet another way of approximating a function is to use quasi-particles

$$f(x) = \sum_{i=1}^{N} w_i S_i(x - x_i) \tag{31}$$

where w_i is the weight of the particle and S_i is the shape. To simplify the equations, the weight will be assumed to be 1 through the rest of the text, yielding

$$f(x) = \sum_{i=1}^{N} S_i(x - x_i) \tag{32}$$

To calculate moments of the solution or to plot, the solution is often projected onto a basis set $\{\varphi_j\}_{j=1}^{N_\varphi}$

$$\begin{aligned} f_\varphi(x) &= \sum_{j=1}^{N_\varphi} \frac{\langle f(x) | \varphi_j(x) \rangle}{\|\varphi_j(x)\|} \varphi_j(x) = \sum_{j=1}^{N_\varphi} \frac{\left\langle \sum_{i=1}^{N} S_i(x - x_i) \,\middle|\, \varphi_j(x) \right\rangle}{\|\varphi_j(x)\|} \varphi_j(x) \\ &= \sum_{j=1}^{N_\varphi} \sum_{i=1}^{N} \frac{\int_{-\infty}^{\infty} S_i(\xi - x_i)\varphi_j(\xi)\, d\xi}{\sqrt{\int_{-\infty}^{\infty} \varphi_j(\xi)\varphi_j(\xi)\, d\xi}} \varphi_j(x) \end{aligned} \tag{33}$$

[7] http://cm.bell-labs.com/cm/ms/who/wim/cascade/

The plotting and the projection base $\{\varphi_j\}_{j=1}^{N_\varphi}$ must be equal, or the curve will be misleading. Figure 7 shows an example of the projection of Gaussian distributed random numbers with a box base and the root-top base (19). The dashed line is using the box base for projection and the roof-top base for plotting. The integral of all the three curves are equal.

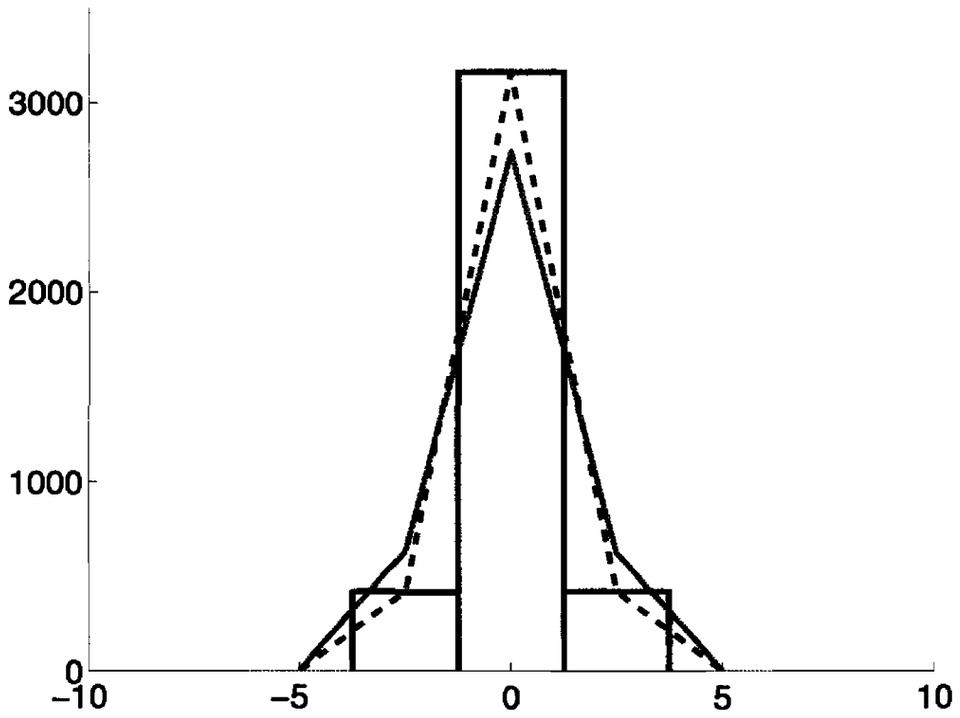


Figure 7: The solid lines are a projection of Gaussian generated random numbers with a box and the roof-top base. The dashed line is the plot using the box base for the projection and the roof-top base for the plotting.

A quasi-particle approximation, where

$$S_i \triangleq \delta \tag{34}$$

and φ is the roof-top function, is included in the JBONE applet. In this case, the projection is particularly easy

$$f_\varphi(x) = \sum_{j=1}^{N_\varphi} \frac{\varphi_j(x_i)}{\|\varphi_j(x)\|} \varphi_j(x) \tag{35}$$

In the document on-line, try to press `INITIALIZE` a few times to get a grasp about how good a quasi-particle approximation is for a box function using 64 grid points. The applet uses random numbers to generate a particle distribution from the initial condition $f_i(x)$ in the interval $[a, b]$ according to the scheme:

Let $i = 1$
while $i \leqslant N$

- **Let** x be a uniformly distributed random number on the interval $[a, b]$.
- **Let** y be a uniformly distributed random number on the interval $[f_{\min}, f_{\max}]$.
- **If** $y < f_i(x)$ **then let** $x_i = x$ and **advance** i by 1 **else do** nothing.

end while

With this preliminary knowledge on how to approximate functions, let's now turn to the discretization of spatial differential operators — and very much depending on how the function is discretized, how to formulate an **explicit** or **implicit** integration in time.

1.5 Exercises

1.1. Stiff ODE. Starting from the boundary conditions $u(0) = 0; v(0) = 0$, use the MATLAB commands `ode23` and `ode23s` to integrate

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v \end{aligned}$$

in the interval $[0; 1]$. Use the variable transformation $u = 2y - z; v = -y + z$ to compare with an analytical solution and show that the problem is stiff.

1.2. E-publishing. Use the newsgroup `alfvenlab.pde-course.test` to ask somebody to send you by e-mail the postscript picture `ex.ps`. Change format into `ex.gif` and publish the result under your personal course web page, e.g. `http://www.fusion.kth.se/~pde99-??/ex1.2.html`. Check the news once again and if possible forward a copy of `ex.ps` to two of your colleagues.

1.3. Fourier-Laplace transform. Solve the advection-diffusion analytically in an infinite 1D slab assuming both the advection speed u and the diffusion coefficient D constant. (Hint: use a Fourier-Laplace transform to first determine the evolution of the Green's function $G(x - x_0, t)$ starting from an initial condition $\delta(x - x_0)$. Superpose to describe an arbitrary initial function $f_0(x)$.

1.4. Random-walk. Determine the diffusion constant for a random-walk process with steps of a typical duration τ and mean free path $\lambda = \sqrt{\langle v^2 \rangle}\tau$. (Hint: calculate first the RMS displacement $\langle z^2 \rangle (t)$ of the position after a large number $M = t/\tau$ of *statistically independent* steps took place. Take the second moment of the diffusion equation and integrate by parts to calculate the average $\overline{z^2}(t)$. Conclude by relating each other using the *ergodicity theorem*).

1.5. Convergence. Calculate a discrete representation of $\sin(kx), kx \in [0; 2\pi]$ for each of the numerical approximations introduced in sect.1.4. Plot the relative local error and show how each converges to the analytical value when the numerical resolution is multiplied stepwise by a factor two. What about the first derivative ?

1.6. Laplacian in 2D. Use a Taylor expansion to calculate an approximation of the Laplacian operator on a rectangular grid $\Delta x = \Delta y = h$. Repeat the calculation for an evenly-spaced, equilateral triangular mesh. (Hint: determine the coordinate transformation for a rotation of $0, 60, 120^o$ and apply the chain rule for partial derivatives in all the three direction).

1.6 Further Reading

- **ODE and stiff equations.**
  Numerical Recipes [2] §16, 16.6, Dahlquist [3] §13
- **PDE properties.**
  Fletcher [4] §2
- **Interpolation and differentiation.**
  Abramowitz [5] §25.2–25.3, Dahlquist [3] §4.6, 4.7, Fletcher [4] §3.2, 3.3
- **FEM approximation.**
  Fletcher [4] §5.3, Johnson [6], Appert [7]
- **Splines.**
  Numerical Recipes [2] §12.0–12.2, Dahlquist [3] §4.8

- **FFT.**
  Numerical Recipes [2] §12.0–12.2

- **Wavelets.**
  Numerical Recipes [2] §13.10, `www.wavelet.org`[8] [1

- **Software.**
  Guide to Available Mathematical Software[9] [8], C
  brary[10] [9]

[8] http://www.wavelet.org/wavelet/index.html
[9] http://gams.nist.gov
[10] http://www.cpc.cs.qub.ac.uk/cpc/

2 FINITE DIFFERENCES (FD)

2.1 Intuitive — Explicit (2 and 3 levels)

Explicit – 2 levels. Approximating the advection (Eq.5) with a spatial difference to **the left and** the diffusion (Eq.7) spatially centered yields an explicit 2 time levels scheme

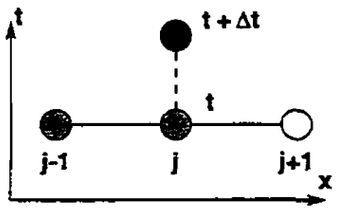


Figure 8: Explicit 2 levels.

$$\begin{aligned} \frac{f_j^{t+\Delta t} - f_j^t}{\Delta t} &+ u\frac{f_j^t - f_{j-1}^t}{\Delta x} - D\frac{f_{j+1}^t - 2f_j^t + f_{j+1}^t}{\Delta x^2} = 0 \\ f_j^{t+\Delta t} &= f_j^t - \beta\left[f_j^t - f_{j-1}^t\right] + \alpha\left[f_{j+1}^t - 2f_j^t + f_{j-1}^t\right] \end{aligned} \tag{36}$$

where the so-called **Courant-Friedrich-Lewy (CFL) number** $\beta = u\Delta t/\Delta x$ and the coefficient $\alpha = D\Delta t/\Delta x^2$ measure typical advection and diffusion velocities compared with the characteristic speed of the mesh $\Delta x/\Delta t$.

For every step in time, a new value is computed explicitly by linear combination of the current neighbors. In JBONE, this is implemented as

```
for (int i=1; i<n; i++) {
  fp[i]=f[i] -beta *(f[i]-f[i-1])+alpha*(f[i+1]-2.*f[i]+f[i-1]); }
fp[0]=f[0]   -beta *(f[0]-f[ n ])+alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
fp[n]=f[n]   -beta *(f[n]-f[n-1])+alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
```

where the last two statements take care of the **periodicity**.

Although it should **NEVER** be used for a physical calculation, it is instructive when studying numerical properties of a scheme to examine the evolution of a box function, showing at the same time how short and long wavelengths propagate. The on-line document illustrates an evolution for a constant advection $u = 1$, first with no physical diffusion $D = \alpha = 0$:

after 128 steps of duration Δt= 0.5, the pulse propagates exactely once accross the domain with a period $L = 64$ and discretized with 64 mesh points so that $\Delta x = 1$. The lowest order moment (density) is conserved to a very good accuracy and the function **remains positive** everywhere as it should; the shape, however, is strongly affected by **numerical diffusion** !

Numerical experiments:

1. Change the initial condition from `Box` to `Cosine`, and vary the wavelength λ = 2–64 to verify that it is indeed the short wavelengths associated with steep wavefronts that get damped: exactly what you expect from diffusion (Eq.10) except that with $D = 0$, it is here a numerical artifact ! Without special care, this can easily cover the physical process you want to model.
2. Looking for a quick fix, you reduce the time step and the CFL number from $\beta = 0.5$ down 0.1. What happens ? Adding further to the confusion, increase now the time step to exactly 1 and check what happens.

For a better understanding of numerical schemes, *von Neumann* calculated the **numerical dispersion** resulting from a local ansatz $f \sim F = \exp(i[kx - \omega t])$. Define the amplification factor as

$$G = \exp(-i\omega\Delta t) \tag{37}$$

and after simplification by F, cast the difference scheme (Eq.36) into

$$\begin{aligned} G &= 1-\beta\left[1-\exp(-ik\Delta x)\right]+\alpha\left[\exp(+ik\Delta x)-2+\exp(-ik\Delta x)\right] \\ &= 1-\beta\left[1-\exp(-ik\Delta x)\right]-4\alpha\sin^2\left(\frac{k\Delta x}{2}\right) \end{aligned} \tag{38}$$

This monitors how all the Fourier components represented locally by the mesh $|k\Delta x| < \pi$ grow and decay in time. In the presence of advection only $D = 0$, Fig.9 illustrates how the 3 first term add together in the complex plane, explaining why short wavelengths $k\Delta x \simeq \pi$ get

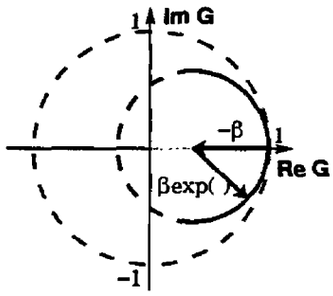


Figure 9: Numerical dispersion / stability (explicit 2 levels).

damped $|G| < 1$ as long as the CFL number β doesn't exceed unity. If it does, $|G| > 1$ and the shortest wavelength modes grow into a so-called **numerical instability**. Try and model this in the JBONE applet ! For a pure diffusive process $\beta = 0$, the dispersion relation (Eq.38) shows that the scheme is stable for all wavelengths provided that $G = 1 - 4\alpha\sin^2(\pi/2) < 1$ or $\alpha < 1/2$. Although the superposition of advection and diffusion is slightly more limiting, the **conditions for numerical stability** may here be conveniently summarized as

$$\alpha = \frac{D\Delta t}{\Delta x^2} < \frac{1}{2} \qquad\qquad \beta = \frac{u\Delta t}{\Delta x} < 1 \quad \text{(CFL condition)} \tag{39}$$

It is worth pointing out that the finite difference evaluated backwards for the advection is unstable for negative velocities $u < 0$. In an inhomogeneous medium, this defect of de-centered schemes can be cured with a so-called **upwind difference**, taking the finite difference forward or backward according to the local direction of propagation (exercise 2.4).

Explicit − 3 levels. A more accurate scheme in $\mathcal{O}(\Delta x^2\Delta t^2)$ can be obtained with differences centered both in time and space

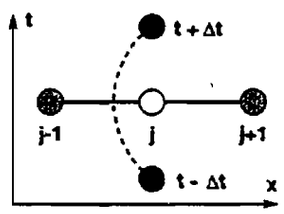


Figure 10: Explicit 3 levels.

$$\begin{aligned} \frac{f_j^{t+\Delta t}-f_j^{t-\Delta t}}{2\Delta t} + u\frac{f_{j+1}^t-f_{j-1}^t}{2\Delta x} - D\frac{f_{j+1}^t-2f_j^t+f_{j+1}^t}{\Delta x^2} &= 0 \\ f_j^{t+\Delta t} &= f_j^{t-\Delta t}-\beta\left[f_{j+1}^t-f_{j-1}^t\right]+2\alpha\left[f_{j+1}^t-2f_j^t+f_{j+1}^t\right] \end{aligned} \tag{40}$$

and has been implemented in JBONE as

```
for (int i=1; i<n; i++) {
  fp[i]=fm[i] -beta*(f[i+1]-f[i-1]) +2*alpha*(f[i+1]-2.*f[i]+f[i-1]);}
fp[0]=fm[0]   -beta*(f[ 1 ]-f[ n ]) +2*alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
fp[n]=fm[n]   -beta*(f[ 0 ]-f[n-1]) +2*alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
```

Note the special starting procedure which is required to define an approximation for a time $-\Delta t$ anterior to the initial condition at $t = 0$, for example by taking an explicit step (Eq.36) backwards in time. The on-line document shows an evolution for the same advection conditions chosen before. how well the scheme performs for the advection of harmonic functions – if dispersion is not an issue. Even though it uses strictly the same spatial differencing for diffusion as before (Eq.36), the 3 levels scheme is here unfortunately always unstable for $D \neq 0$ (exercise 2.4) .

2.2 Second thoughts — Explicit (Lax-Wendroff, Leapfrog)

Rather than counting on intuition for the right combination of terms, is it possible to formulate a **systematic approach** for solving an equation in **Eulerian coordinates** with a chosen **accuracy** ? Yes, using the so-called

Lax-Wendroff approach, which can easily be generalized to non-linear and vector equations.

1. Discretize the function on a regular grid $f(x) \rightarrow (x_j, f_j)$, $j = 1, N$,
2. Expand the differential operators in time using a Taylor series

$$f_{x_j}^{t+\Delta t} = f_{x_j}^t + \Delta t \frac{\partial f}{\partial t}\bigg|_{x_j} + \frac{\Delta t^2}{2}\frac{\partial^2 f}{\partial t^2}\bigg|_{x_j} + \mathcal{O}(\Delta t^3) \tag{41}$$

3. Substitute time derivatives from the master equation. Using advection (Eq.5) for illustration, this yields

$$f_{x_j}^{t+\Delta t} = f_{x_j}^t - u\Delta t \frac{\partial f}{\partial x}\bigg|_{x_j} + \frac{(u\Delta t)^2}{2}\frac{\partial^2 f}{\partial x^2}\bigg|_{x_j} + \mathcal{O}(\Delta t^3) \tag{42}$$

4. Use centered differences for spatial operators

$$f_j^{t+\Delta t} = f_j^t - \frac{\beta}{2}\left(f_{j+1}^t - f_{j-1}^t\right) + \frac{\beta^2}{2}\left(f_{j+1}^t - 2f_j^t + f_{j-1}^t\right) + \mathcal{O}(\Delta x^3 \Delta t^3) \tag{43}$$

This procedure results in a third order scheme which is explicit, centered and stable as long as the CFL number $\beta = u\Delta t / \Delta x$ remains below unity :

```
for (int i=1, i<n, i++) {
  fp[i]=f[i] -0.5*beta *(f[i+1]-f[i-1])
             +0.5*beta*beta*(f[i+1]-2.*f[i]+f[i-1]); }
fp[0]=f[0] -0.5*beta *(f[ 1 ]-f[ n ])
           +0.5*beta*beta*(f[ 1 ]-2.*f[0]+f[ n ]);
fp[n]=f[n] -0.5*beta *(f[ 0 ]-f[n-1])
           +0.5*beta*beta*(f[ 0 ]-2.*f[n]+f[n-1]);
```

Numerical diffusion damps again mainly the shorter wavelengths, distorting somewhat the trial box function as it propagates.

Staggered leapfrog, is an explicit remedy against this numerical damping, which also benefits from the higher accuracy $\mathcal{O}(\Delta x^2 \Delta t^2)$ allowing you to take larger time steps: it is perhaps the best finite-difference scheme for the evolution of hyperbolic wave problems. The idea is to use two so called **staggered grids**, where the mesh points are shifted with respect to each other by half an interval and evaluate the solution using two functions of the form

$$\begin{cases} \frac{1}{\Delta t}\left[f_{i+1/2}^{t+\Delta t} - f_{i+1/2}^t\right] = \frac{u}{\Delta x}\left[g_{i+1}^{t+\Delta t/2} - g_i^{t+\Delta t/2}\right] \\ \frac{1}{\Delta t}\left[g_i^{t+\Delta t/2} - g_i^{t-\Delta t/2}\right] = \frac{u}{\Delta x}\left[f_{i+1/2}^t - f_{i+1/2}^t\right] \end{cases} \tag{44}$$

By substitution, this is strictly equivalent to the 3 levels scheme

$$\frac{1}{(\Delta t)^2}\left(f_i^{t+\Delta t} - 2f_i^t + f_i^{t-\Delta t}\right) = \frac{u^2}{(\Delta x)^2}\left(f_{i+1}^t - 2f_i^t + f_{i-1}^t\right). \tag{45}$$

Using the first formulation to illustrate with scalar fields (f, g) how the Maxwell equations can be solved by **finite differences in the time domain (FDTD)** in terms of the electric and magnetic fields $(\vec{E}, \vec{B})$ [2.5], a leapfrog algorithm has been implemented in JBONE as

```
for (int i=1; i<=n; i++) {                    //time + time_step
  fp[i]=f[i] -beta*(g[i]-g[i-1]); }
fp[0]=f[0] -beta*(g[0]-g[n]);
for (int i=0; i<=n-1; i++) {                  //time + 1.5*time_step
  gp[i]=g[i] -beta*(fp[i+1]-fp[i]); }
gp[n]=g[n] -beta*(fp[0]-fp[n]);
```

Special care is required when starting the integration, here particularly since it is the initial condition which determines the direction of propagation !

Vary the wavelength of the harmonic oscillation in the applet and check how well it performs in terms of numerical diffusion / dispersion compared to the previous schemes. Try also to propagate a box function.

2.3 Implicit scheme (Crank-Nicholson)

All the schemes seen so far are called **explicit**, because every unknown can be obtained explicitly by linear combination from a scalar equation. **Implicit methods** allow more generally for a coupling between the unknowns and require therefore a matrix inversion. This makes the implementation considerably more complicated, so that a finite element approach (without *lumping the mass matrix*) is generally preferable – because it offers more flexibility for the same programming effort. Here we show only one scheme, which is still commonly used for solving in particular diffusive type of problems.

Implicit – Crank-Nicholson. The scheme combines centered differences in space, evaluated with equal weights from the current and the future discretization

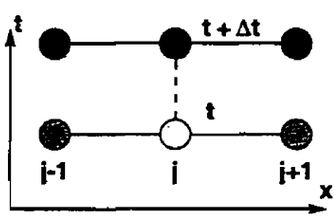


Figure 11: Implicit Crank-Nicholson.

$$\begin{aligned}\frac{f_j^{t+\Delta t} - f_j^t}{\Delta t} &+ \frac{u}{2}\left\{\frac{f_{j+1}^{t+\Delta t} - f_{j-1}^{t+\Delta t}}{2\Delta x} + \frac{f_{j+1}^t - f_{j-1}^t}{2\Delta x}\right\} \\ &- \frac{D}{2}\left\{\frac{f_{j+1}^{t+\Delta t} - 2f_j^{t+\Delta t} + f_{j-1}^{t+\Delta t}}{\Delta x^2} + \frac{f_{j+1}^t - 2f_j^t + f_{j-1}^t}{\Delta x^2}\right\} = 0\end{aligned} \tag{46}$$

which can conveniently be rewritten as a linear system

$$\begin{pmatrix} -\alpha/2 - \beta/4 \\ 1 + \alpha \\ -\alpha/2 + \beta/4 \end{pmatrix}^T \cdot \begin{pmatrix} f_{j-1}^{t+\Delta t} \\ f_j^{t+\Delta t} \\ f_{j+1}^{t+\Delta t} \end{pmatrix} = \begin{pmatrix} \alpha/2 + \beta/4 \\ 1 - \alpha \\ \alpha/2 - \beta/4 \end{pmatrix}^T \cdot \begin{pmatrix} f_{j-1}^t \\ f_j^t \\ f_{j+1}^t \end{pmatrix} \tag{47}$$

showing the tri-diagonal structure of the matrix.

In JBONE, the Crank-Nicholson scheme has been implemented as

```
BandMatrix a = new BandMatrix(3, f.length);
BandMatrix b = new BandMatrix(3, f.length);
double[]   c = new double[f.length];
for (int i=0; i<=n; i++) {
  a.setL(i,-0.25*beta -0.5*alpha);            //Matrix elements
  a.setD(i,            1. +alpha);
  a.setR(i, 0.25*beta -0.5*alpha);
  b.setL(i, 0.25*beta +0.5*alpha);            //Right hand side
  b.setD(i,            1. -alpha);
  b.setR(i,-0.25*beta +0.5*alpha);
}
c=b.dot(f);                                   //Right hand side
c[0]=c[0]+b.getL(0)*f[n];                     // with periodicity
c[n]=c[n]+b.getR(n)*f[0];

fp=a.solve3(c);                               //Solve linear problem
```

relying here on the simple solve3() method to solve the linear system efficiently with LU-factorisation in $\mathcal{O}(N)$ operations [2.5] and take care of the periodicity.

Repeating the *von Neumann* analysis from sect.2.1 to determine the numerical dispersion implied by (Eq.46) for a pure diffusive process ($u = \alpha = 0$) yields the amplification factor

$$G = \frac{1 - 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)}{1 + 2\alpha \sin^2\left(\frac{k\Delta x}{2}\right)} \tag{48}$$

proving that this scheme is stable for all Δx and Δt, with phase errors affecting short wavelength modes $k\Delta x \sim 1$. The analogue is true also for the advective part. This favorable stability property can actually be exploited when solving diffusion dominated problems, concerned mainly with the evolution of large scale features $\lambda \gg \Delta x$.

Starting from a relatively smooth Gaussian pulse subject both to advection and diffusion $u = D = 1$, the on-line document shows that a reasonably accurate solution (12 % for the valley to peak ratio as the time reaches 100) can be computed using extremely large time steps with $\alpha = \beta = 5$. Try and see what happens with an initial box function !

2.4 Exercises

2.1. Upwind differences, boundary conditions. Use upwind differences and modify the explicit 2 level scheme in JBONE to make it stable both for forward and backward propagation. Implement Dirichlet conditions to always maintain a constant value on the boundary up-the-wind (i.e. in the back of the pulse) and, depending on the sign of u, either an outgoing wave on the right or a Neumann condition $f'(x_L) = 0$ on the left.

2.2. Numerical dispersion. Determine how the 3 levels scheme (Eq.40) affects the advection of short wavelength components and calculate the growth rate of the numerical instability for $D \neq 0$. Use a harmonic initial condition to confirm your results with the JBONE applet.

2.3. Shock waves using Lax-Wendroff. Use the Lax-Wendroff approach to solve the Burger equation in the JBONE applet. Implement first and second order schemes, studying the numerical convergence for different values of the physical diffusion D. Hint: approximate derivatives of higher order $f^{(d)}$ using the formulas for finite differences from Abramowitz [5] §25.1.2

$$f_j^{(2n)} = \sum_{k=0}^{2n} (-1)^k \binom{2n}{k} f_{j+n-k} \qquad f_{j+1/2}^{(2n+1)} = \sum_{k=0}^{2n+1} (-1)^k \binom{2n+1}{k} f_{j+n+1-k} \tag{49}$$

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n(n-1)\cdots(n-k+1)}{k!}$$

2.4. Leapfrog resonator. Modify the leapfrog scheme in JBONE to incorporate perfectly reflecting boundary conditions; play with the initial conditions to excite an eigenmode of this very simple system.

2.5. Option pricing. Use one of the finite difference schemes to solve the Black-Scholes equation

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r - D_0) S \frac{\partial V}{\partial S} - rV = 0 \tag{50}$$

for the value of a *European put* option using an underlying asset $S \in [0; 16]$ and an exercise price $E = 10$, a fixed annual interest rate $r = 0.1$, a volatility $\sigma = 0.45$, no dividend $D_0 = 0$ and $T = 4$ months to expiry. Start with a change of variables $t = T - t'$ to form a backward-parabolic equation in time and propose an explicit scheme for the variables (S, t').

After giving some thoughts to the numerical stability of this solution, examine the transformation into normalized variables (x, τ) defined by $t = T - 2\tau/\sigma^2$, $S = E\exp(x)$ and use the ansatz

$$\begin{aligned} V(S,t) &= E\exp\left[-\frac{1}{2}(k_2-1)x - \left(\frac{1}{4}(k_2-1)^2 x + k_1\right)\tau\right] u(x,\tau) \\ k_1 &= 2r/\sigma^2, \qquad k_2 = 2(r-D_0)/\sigma^2 \end{aligned} \tag{51}$$

to reduce (Eq.50) to the standard diffusion problem

$$\frac{\partial u}{\partial \tau} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{52}$$

Derive initial and boundary conditions and implement the improved scheme in the JBONE applet. Compare with the Monte-Carlo solution previously developed in this course [10], now running very nicely as an applet[11] !

2.5 Further Reading

- **Eulerian schemes for advection-diffusion problems.**
  Numerical Recipes [2] §19.1, 19.2, Boris [11]
- **FDTD leapfrog for Maxwell's equation.**
  Jin [12], EMLIB homepage[12] [13]
- **Direct and iterative methods for solving linear systems.**
  Numerical Recipes [2] §2.4, Dahlquist [3] §6.3, §11.1–11.4, Saad [14]
- **Option pricing.**
  Wilmott [15], Björk [16], Duffie [17]

[11] http://fedu52.fed.ornl.gov/%7ecarlsson/MonteCarlo/BlaSES/
[12] http://emlib.jpl.nasa.gov/

3 FINITE ELEMENTS METHOD (FEM)

3.1 Mathematical background

To approximate a set of linear partial differential equations

$$\mathbf{L}\vec{v} = \vec{r} \quad \text{in} \quad \Omega \tag{53}$$

for an unknown $\vec{v} \in \mathcal{V} \subset \mathcal{C}^n(\Omega)$ continuously defined with n derivatives in the volume Ω and subject to the boundary conditions

$$\mathbf{B}\vec{v} = \vec{s} \quad \text{in} \quad \partial\Omega \tag{54}$$

a mathematician would probably first involve

Weighted residuals. Having defined a scalar product $(\cdot,\cdot)$ and a norm $||\cdot||$, the calculation essentially amounts to the minimization of a residual vector

$$||\vec{R}|| = ||\vec{r} - \mathbf{L}\vec{v}|| \tag{55}$$

which can conveniently be carried out using tools from the variational calculus.

Variational form. A quadratic form is constructed for that purpose by choosing a **test function** $\vec{w}$ in a sub-space $\mathcal{W}$ which is "sufficiently general" and satisfies the boundary conditions. The linear equation (Eq.53) can then be written as the equivalent variational problem

$$(\vec{w}, \vec{R}) = (\vec{w}, \mathbf{L}\vec{v} - \vec{r}) = 0 \qquad \vec{v} \in \mathcal{V}, \ \forall \vec{w} \in \mathcal{W} \tag{56}$$

Integration by parts. If $\vec{w}$ is differentiable at least once $\mathcal{W} \subset \mathcal{C}^1(\Omega)$, the regularity required by the forth-coming discretization can often be relaxed by partial integrations. Using $\mathcal{L} = \nabla^2$ for illustration, Leibniz' rule states that

$$\nabla \cdot (v\vec{w}) = (\nabla v) \cdot \vec{w} + v\nabla \cdot \vec{w} \quad \Longrightarrow \quad \nabla v \cdot \vec{w} = -v\nabla \cdot \vec{w} + \nabla \cdot (v\vec{w}) \tag{57}$$

Integrating over the volume Ω and using Gauss' divergence theorem yields a generalized formula for partial integration:

$$\int_\Omega \nabla \cdot \vec{F}\, dV = \int_{\partial\Omega} \vec{F} \cdot d\vec{S} \quad \Longrightarrow \quad \int_\Omega \nabla v \cdot \vec{w}\, dV = -\int_\Omega v\nabla \cdot \vec{w}\, dV + \int_{\partial\Omega} v\vec{w} \cdot d\vec{S} \tag{58}$$

For the special case where $\vec{w} = \nabla u$, this is known as Green's formula

$$\int_\Omega v\nabla^2 u\, dV = -\int_\Omega \nabla v \cdot \nabla u\, dV - \int_{\partial\Omega} v\nabla u \cdot d\vec{S} \tag{59}$$

Note that the last (surface-) term can sometimes be imposed to zero (or to a finite value) when applying so-called **natural boundary conditions**.

Numerical approximation. It turns out that the construction of the variational problem is general enough that the solution $\vec{v}$ of (Eq.56) remains a converging approximation of (Eq.53) even when the sub-spaces $\mathcal{V}, \mathcal{W}$ are **restricted** to finite, a priori non-orthogonal, but still complete sets $\overline{\mathcal{V}}, \overline{\mathcal{W}}$ of functions; the superposition coefficients of these functions can then be handled simply as linear algebra by the computer.

In general, the discretized solution $\vec{v}$ is expanded in **basis functions** $\vec{e_j} \in \overline{\mathcal{V}} \subset \mathcal{V}$ which either reflect a property of the solution (e.g. the operator Green's function in the **Method of Moments**), or which are simple and localized enough so that they yields cheap inner products $(\cdot, \vec{e_j})$ and sparse linear systems (e.g. the roof-top function for the linear **Finite Element Method**). Different discretizations are possible also for the **test functions** $\vec{w}$. Among the most popular choices is the **Galerkin** method where test and basis functions are both chosen from the same sub-space $\overline{\mathcal{V}} \equiv \overline{\mathcal{W}}$; the method of **collocation** consists in taking $\vec{w} \in \overline{\mathcal{W}} = \{\delta(\vec{x} - \vec{x_j})\}, j = 1, N$ which then yields a point-wise evaluation of the integrand on a discrete mesh $\{\vec{x_j}\}, j = 1, N$.

3.2 An engineer's formulation

After a section of what a physicist believes might be the mathematician's view of the subject, it is great time for an example. Using the format of a "recipe" applicable to a rather broad class of practical problems, this shows how the advection-diffusion problem (Eq.5 and 7) is formulated using Galerkin linear finite elements (FEMs) before it is implemented in the JBONE applet.

Derive a weak variational form. "Multiply" your equation by $\forall g \in \mathcal{C}^1(\Omega), \int_\Omega dx\ g^*(x)$ where the conjugation is necessary only if your equation(s) is (are) complex:

$$\forall g \in \mathcal{C}^1(\Omega), \qquad \int_{x_L}^{x_R} dx \quad g\left[\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} - D\frac{\partial^2 f}{\partial x^2}\right] = 0 \tag{60}$$

Integrate by parts the second derivative in the diffusion term, avoiding later having to use quadratic basis functions for the spatial discretization:

$$\int_{x_L}^{x_R} dx \quad \left[g\frac{\partial f}{\partial t} + ug\frac{\partial f}{\partial x} + D\frac{\partial g}{\partial x}\frac{\partial f}{\partial x}\right] = Dg\frac{\partial f}{\partial x}\Big|_{x_L}^{x_R} = 0 \quad \forall g \tag{61}$$

Assuming a periodic domain, the surface term is cancelled by imposing so-called **natural boundary conditions**.

Discretize time with a partially implicit scheme $f = (1-\theta)f^t + \theta f^{t+\Delta t}$, choosing $\theta \in [1/2; 1]$:

$$\int_{x_L}^{x_R} dx \left[g\left(\frac{f^{t+\Delta t} - f^t}{\Delta t}\right) + ug\frac{\partial}{\partial x}\left[(1-\theta)f^t + \theta f^{t+\Delta t}\right] + D\frac{\partial g}{\partial x}\frac{\partial}{\partial x}\left[(1-\theta)f^t + \theta f^{t+\Delta t}\right]\right] = 0$$

$$\int_{x_L}^{x_R} dx \left[\frac{g}{\Delta t} + \theta ug\frac{\partial}{\partial x} + \theta D\frac{\partial g}{\partial x}\frac{\partial}{\partial x}\right] f^{t+\Delta t} = \int_{x_L}^{x_R} dx \left[\frac{g}{\Delta t} - (1-\theta)ug\frac{\partial}{\partial x} - (1-\theta)D\frac{\partial g}{\partial x}\frac{\partial}{\partial x}\right] f^t \ (6$$

$\forall g$, where all the unknowns have been reassembled on the left. Re-scale by Δt, and

Discretize space using a linear FEMs expansion and a Galerkin choice for the test function:

$$f^t(x) = \sum_{j=1}^{N} f_j^t e_j(x), \qquad \forall g \in \{e_i(x)\},\ \ i = 1, N \tag{63}$$

$$\int_{x_L}^{x_R} dx \left[e_i \sum_{j=1}^{N} f_j^{t+\Delta t} e_j + \Delta t\theta e_i u \sum_{j=1}^{N} f_j^{t+\Delta t}\frac{\partial e_j}{\partial x} + \Delta t\theta D\frac{\partial e_i}{\partial x}\sum_{j=1}^{N} f_j^{t+\Delta t}\frac{\partial e_j}{\partial x}\right] =$$

$$\int_{x_L}^{x_R} dx \left[e_i \sum_{j=1}^{N} f_j^{t} e_j + \Delta t(\theta-1) e_i u \sum_{j=1}^{N} f_j^{t}\frac{\partial e_j}{\partial x} + \Delta t(\theta-1) D\frac{\partial e_i}{\partial x}\sum_{j=1}^{N} f_j^{t}\frac{\partial e_j}{\partial x}\right] \quad \forall i = 1, N \tag{64}$$

Note how the condition $\forall g \in \mathcal{C}^1([x_L; x_R])$ is used to create as many independent equations as there are unknowns $\{f_j^{t+\Delta t}\}, j = 1, N$. All the **essential boundary conditions** have implicitly been imposed by allowing $e_1(x)$ and $e_N(x)$ to overlap in the periodic domain. Since only the basis and test functions $e_i(x), e_j(x)$ and maybe the problem coefficients $u(x), D(x)$ remain space dependent, the discretized equations can all be written in terms of **inner products** for example of the form $(e_i, ue_j') = \int_{x_L}^{x_R} dx\ u(x)e_i(x)e_j'(x)$. Reassembling them in matrix notation,

Write a linear system through which the unknown values from the next time step $f_j^{t+\Delta t}$ can implicitly be calculated in terms of the current values f_j^t

$$\mathbf{A_{ij}} f_j^{t+\Delta t} = \mathbf{B_{ij}} f_j^t \tag{65}$$

To relate this Galerkin linear FEM scheme with the code which has been implemented in the JBONE applet, it is necessary first to evaluate the integrals from the inner product; this is usually performed with a numerical quadrature.

3.3 Numerical quadrature and solution

Except when the PDE coefficients become singular and require a special treatment, the precision required for the numerical integration really depends on the type of FEMs; in general, it should simply preserve the convergence rate which is already guaranteed by the discretization. Using a **numerical quadrature** of the form

$$\int_a^b f(y)dy = \frac{b-a}{2}\sum_{i=1}^{n} w_i f(y_i) + R_n \tag{66}$$

$$y_i = \left(\frac{b-a}{2}\right)x_i + \left(\frac{b+a}{2}\right) \tag{67}$$

with abscissas x_i, weights w_i and rests R_n given in the table 1 below it is possible to show [3.6]

scheme	n terms	$\pm x_i$	w_i	$R_n \sim \mathcal{O}(f^{(p)})$
mid-point	2	0	1	p=2
trapezoidal	2	1	1	p=2
Gaussian	2	$\sqrt{1/3}$	1	p=4
Gaussian	3	0	8/9	p=6
		$\sqrt{3/5}$	5/9	
Gaussian	4	$\sqrt{3/7+\sqrt{120}/35}$	$1/2-5/(3\sqrt{120})$	p=8
		$\sqrt{3/7-\sqrt{120}/35}$	$1/2+5/(3\sqrt{120})$	

Table 1: Quadrature abscissas and weights for the interval $x_i \in [-1;1]$

that polynomials with a degree up to $(p-1)$ can be integrated **exactly**, simply by superposing n terms for which the integrand is evaluated at the specified location $y_i(x_i) \in [a;b]$ and weighted by the factor w_i. Using the powerful **Gaussian quadrature**, an approximation based on cubic FEMs will therefore require no more than two evaluations of the integrand within each interval of a unidimensional mesh.

When dealing with linear FEMs, the mid-point and the trapezoidal rules are in fact both precise enough; although slightly more expensive due to one extra evaluation of the integrand, they can nicely be combined into a so-called **tunable integration** [18]

$$\int_a^b f(y)dy = (b-a)\left[\frac{p}{2}\left[f(a)+f(b)\right] + (1-p)f\left(\frac{a+b}{2}\right)\right] + R_2 \qquad p \in [0;1] \tag{68}$$

A piecewise linear FEM discretization (obtained for $p = 1/3$) can then be continuously changed into either an equivalent FD discretization (for $p = 1$) or a piecewise constant FEM approximation (for $p = 0$). Apart from an academic interest, this feature can sometimes be used to change the slope of the numerical convergence and even to stabilize an approximation which is marginally unstable because of the numerical discretization.

Armed with new tools to complete the linear FEM discretization from sect.3.2, the matrix elements in (Eq.64) are now evaluated using the tunable integration (Eq.68). Since FEMs reach only as far as to the nearest neighbors, all the matrix elements $\mathbf{A_{ij}}$ with $|i-j| > 1$ vanish, except those which are created by the periodicity on the domain boundaries. Using a sequential numbering of the unknowns $\{f_j^{t+\Delta t}\}, j = 1, N$, this results in a **tri-diagonal structure** of the matrix, plus two extra elements in the upper-right and lower left corner from the periodic boundary conditions.

To keep the FEM implementation in JBONE as simple as possible, homogeneity is assumed $u \neq u(x)$, $D \neq D(x)$ and the matrix coefficients are calculated directly in terms of the inner

products

$$\int_{x_{i-1}}^{x_i} e_i e_i dx = \int_{x_i}^{x_{i+1}} e_i e_i dx = (1+p)\frac{\Delta x}{4} \quad (69)$$

$$\int_{x_i}^{x_{i+1}} e_i e_{i+1} dx = \int_{x_i}^{x_{i+1}} e_{i+1} e_i dx = (1-p)\frac{\Delta x}{4} \quad (70)$$

$$\int_{x_i}^{x_{i+1}} e_i e_i' dx = -\int_{x_i}^{x_{i+1}} e_i e_{i+1}' dx = -\frac{1}{2} \quad (71)$$

$$\int_{x_i}^{x_{i+1}} e_i' e_i' dx = -\int_{x_i}^{x_{i+1}} e_i' e_{i+1}' dx = \frac{1}{\Delta x} \quad (72)$$

In a real code, this would of course be replaced by a call to the function returning a local value of the integrand, and combined with the summation from one of the quadratures described above. Substituting back into (Eq.64) finally yields the FEM scheme

$$\begin{pmatrix} (1-p)\frac{\Delta x}{4} + \theta\Delta t\frac{u}{2} - \theta\Delta t\frac{D}{\Delta x} \\ (1+p)\frac{2\Delta x}{4} + \theta\Delta t\frac{2D}{\Delta x} \\ (1-p)\frac{\Delta x}{4} - \theta\Delta t\frac{u}{2} - \theta\Delta t\frac{D}{\Delta x} \end{pmatrix}^T \cdot \begin{pmatrix} f_{i-1}^{t+\Delta t} \\ f_i^{t+\Delta t} \\ f_{i+1}^{t+\Delta t} \end{pmatrix} =$$
$$\begin{pmatrix} (1-p)\frac{\Delta x}{4} + (\theta-1)\Delta t\frac{u}{2} - (\theta-1)\Delta t\frac{D}{\Delta x} \\ (1+p)\frac{2\Delta x}{4} + (\theta-1)\Delta t\frac{2D}{\Delta x} \\ (1-p)\frac{\Delta x}{4} - (\theta-1)\Delta t\frac{u}{2} - (\theta-1)\Delta t\frac{D}{\Delta x} \end{pmatrix}^T \cdot \begin{pmatrix} f_{i-1}^{t} \\ f_i^{t} \\ f_{i+1}^{t} \end{pmatrix} \quad (73)$$

After an initialization where the initial condition is discretized with a projection on piecewise linear "roof-top" FEMs which is particularly simple, the scheme is implemented in JBONE using two tri-diagonal matrixes **a**, **b** and a vector **c**:

```
BandMatrix a = new BandMatrix(3, f.length);
BandMatrix b = new BandMatrix(3, f.length);
double[]   c = new double[f.length];

double h   = dx[0];
double htm = h*(1-tune)/4;
double htp = h*(1+tune)/4;

for (int i=0; i<=n; i++) {
  a.setL(i,   htm +h*(-0.5*beta -alpha)* theta    );
  a.setD(i,2*(htp +h*(           alpha)* theta   ));
  a.setR(i,   htm +h*( 0.5*beta -alpha)* theta    );
  b.setL(i,   htm +h*(-0.5*beta -alpha)*(theta-1) );
  b.setD(i,2*(htp +h*(           alpha)*(theta-1)));
  b.setR(i,   htm +h*( 0.5*beta -alpha)*(theta-1) );
}

c=b.dot(f);                                  //Right hand side
c[0]=c[0]+b.getL(0)*f[n];                    // with periodicity
c[n]=c[n]+b.getR(n)*f[0];

fp=a.solve3(c);                              //Solve linear problem
```

Here again, the simple solve3() method is used to compute a direct solution in $\mathcal{O}(N)$ operations with LU-factorization. The on-line document illustrates the advection of a box, calculated with a piecewise linear "roof-top" FEMs discretization ($p = 1/3$) slightly decentered in time ($\theta = 0.55$).

After a considerable effort spent in understanding this FEM discretization, isn't it frustrating to see how similar the code is with the implicit Crank-Nicholson FD scheme from sect.2.3 ? This should be the main argument for the community who still uses implicit finite difference schemes ! With some **thinking** but the **same computational cost**, a finite element approach offers considerably more flexibility: it is now for example easy to **densify the mesh**[13], vary the **partially**

[13]Although this has, for pedagogical reasons, here not been exploited

implicit time integration from centered to fully implicit $\theta \in [\frac{1}{2}; 1]$ and **tune the integration** $p \in [0; 1]$. Convince yourself that Crank-Nicholson (Eq.47) and this FEM scheme (Eq.73) are strictly equivalent for a homogeneous mesh $x_j = j\Delta x$, $j = 1, N$, a time centered integration $\theta = \frac{1}{2}$ and a trapezoidal quadrature $p = 1$. Also take a couple of minute to experiment how you can affect the numerical dispersion / diffusion of short wavelengths by varying both the parameters $\theta \in [\frac{1}{2}; 1]$ and $p \in [0; 1]$.

3.4 Linear solvers

Writing efficient solvers for linear systems is a complete chapter of numerical analysis — and involves much more than what can be introduced here with a couple of sentences ! As a user of software libraries such as Netlib [19] or PetSc [20], it is however sufficient to have a rough idea of what type of solvers exist and what they do.

Direct LU factorization. Remember that you should a priori never calculate a matrix inverse; you can solve a linear problem directly with only a third of the operations by decomposing it first into **lower and upper triangular** parts [3.6] with $\frac{2}{3}N^3$ operations $(*, +)$

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L}(\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{74}$$

and then solve a **forward-backward** substitution with $2N^2$ operations

$$y_1 = \frac{b_1}{L_{11}}; \qquad y_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} y_i \right), \quad i = 2, ..., N \tag{75}$$

$$x_N = \frac{y_N}{U_{11}}; \qquad x_i = \frac{1}{U_{ii}} \left(y_i - \sum_{j=i+1}^{N} U_{ij} x_j \right), \quad j = N-1, ..., 1 \tag{76}$$

A particularly simple version has been implemented in JBONE for tri-diagonal matrices, where the solve3() method computes the solution in $\mathcal{O}(N)$ operations; the first and the last equations are simply eliminated "by hand" to take care of the periodicity. Many different implementations of one and the same algorithm exist and adapt the LU-factorization to specific non-zero patterns of $\mathbf{A}$; it is likely that Netlib has a routine already tailored for your application.

For matrices with more than three diagonals, it is important to allow for a **pivoting** during the factorization process — interchanging rows and columns to avoid divisions by small values created on the diagonal. For particularly large problems, note the possibility of storing most of the matrix on disk with a **frontal implementation** of the same algorithm.

If memory consumption, calculation time or the parallelization of a solver becomes an issue for 2D, 3D or even higher dimensions, it might be useful to consider **iterative methods** as an alternative to direct solvers. The idea behind them is best understood from a **splitting** of the matrix into a sum[14] of a diagonal $\mathbf{D}$ and strict lower $-\mathbf{E}$ and upper $-\mathbf{F}$ triangular parts

$$\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F} \tag{77}$$

An iterative solution of the problem

$$(\mathbf{b} - \mathbf{A} \cdot \mathbf{x_{k+1}})_i = 0, \quad i = 1, ...N \tag{78}$$

is then sought where $\mathbf{x_{k+1}} = \left(\xi_i^{(k+1)}\right)$ is the $(k+1)$-th iterate approximating the solution, with components ranging from $i = 1, ...N$.

[14] Don't get confused here with the product previously used for the LU factorization !

Iterative Jacobi. The simplest form of iteration consists in inverting only the diagonal

$$a_{ii}\xi_i^{(k+1)} = \beta_i - \sum_{i \neq j} a_{ij}\xi_j^{(k)} \tag{79}$$

which is equivalent in matrix notation to

$$\mathbf{D} \cdot \mathbf{x_{k+1}} = (\mathbf{E} + \mathbf{F}) \cdot \mathbf{x_k} + \mathbf{b} \tag{80}$$

Starting from an arbitrary initial guess $\mathbf{x_0}$, simple elliptic problems will –albeit slowly– converge to a stationary point which is the solution of the linear problem. It is important to note that the matrix-vector multiplication in (Eqs.79 or 80) can be performed in **sparse format**, i.e. using only those matrix elements which are different from zero. This is why iterative methods are much more economical than direct solvers which fill the matrix during the LU factorization process.

Gauss-Seidel. As the equations $i = 1, ...N$ are updated one after the other, it is in fact possible to immediately use the updated information available in an explicit **forward** scheme

$$a_{ii}\xi_i^{(k+1)} + \sum_{j<i} a_{ij}\xi_j^{(k+1)} = \beta_i - \sum_{j>i} a_{ij}\xi_j^{(k)} \tag{81}$$

or

$$(\mathbf{D} - \mathbf{E}) \cdot \mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} + \mathbf{b} \tag{82}$$

where the calculation is carried out with an equation index increasing from $i = 1, ...N$. The reverse is also possible and is called **backward** Gauss-Seidel

$$(\mathbf{D} - \mathbf{F}) \cdot \mathbf{x_{k+1}} = \mathbf{E} \cdot \mathbf{x_k} + \mathbf{b} \tag{83}$$

Successive over-relaxation — SOR is obtained when either of the Gauss-Seidel iterations (Eqs.82 or 83) is linearly combined with the value of the previous step

$$\mathbf{x_{k+1}} = \omega \mathbf{x_k^{GS}} + (1 - \omega)\mathbf{x_k} \tag{84}$$

using the parameter $\omega \in [0; 1]$ for an *under-relaxation* or $\omega \in [1; 2]$ for an *over-relaxation*. A **symmetric-SOR (SSOR)** scheme can be obtained for the combination

$$\begin{cases} (\mathbf{D} - \omega\mathbf{E}) \cdot \mathbf{x_{k+1/2}} = [\omega\mathbf{F} + (1 - \omega)\mathbf{D}] \cdot \mathbf{x_k} + \omega\mathbf{b} \\ (\mathbf{D} - \omega\mathbf{F}) \cdot \mathbf{x_{k+1}} = [\omega\mathbf{E} + (1 - \omega)\mathbf{D}] \cdot \mathbf{x_{k+1/2}} + \omega\mathbf{b} \end{cases} \tag{85}$$

Preconditionning / approximate inverse. Realizing that all the methods above are of the form $\mathbf{x_{k+1}} = \mathbf{G} \cdot \mathbf{x_k} + \mathbf{f}$ where $\mathbf{G} = \mathbf{1} - \mathbf{M^{-1}} \cdot \mathbf{A}$, the linear problem approximatively be diagonalized with a preconditionning

$$\mathbf{M^{-1}} \cdot \mathbf{A} \cdot \mathbf{x} = \mathbf{M^{-1}} \cdot \mathbf{b} \tag{86}$$

$$\mathbf{M^{Jacobi}} = \mathbf{D} \qquad \mathbf{M^{SOR}} = \frac{1}{\omega}(\mathbf{D} - \omega\mathbf{E}) \tag{87}$$

$$\mathbf{M^{GS}} = \mathbf{D} - \mathbf{E} \qquad \mathbf{M^{SSOR}} = \frac{1}{\omega(2 - \omega)}(\mathbf{D} - \omega\mathbf{E}) \cdot \mathbf{D^{-1}} \cdot (\mathbf{D} - \omega\mathbf{F}) \tag{88}$$

Note that a product $\mathbf{s} = \mathbf{M^{-1}} \cdot \mathbf{A} \cdot \mathbf{x}$ for a given $\mathbf{x}$ can be calculated without ever forming the inverse, first with a product $\mathbf{r} = \mathbf{A} \cdot \mathbf{x}$ and then solving the linear system $\mathbf{M} \cdot \mathbf{s} = \mathbf{r}$ in sparse format.

All these variants of Gauss-Seidel are simple and work fairly well for reasonable sized (e.g. 20×20) elliptic problems, making the much more complicated **multigrid** approach attractive only for larger applications which have to be solved many times.

Solving hyperbolic (wave-) problems and non-Hermitian operators iteratively is however more complicated and remains a matter of research [12],[21],[22]. Methods rely in general both on a suitable **preconditionning** and the rapid convergence of Krylov-space and minimal residual methods such as **GMRES** or **TFQMR** [3.6].

3.5 Exercises

3.1. Quadrature. Calculate the integral $\int_0^\pi \sin(x)dx$ for a piecewise linear FEM approximation with two intervals, comparing the analytical result both with the tunable integration scheme (Eq.68) and a two point Gaussian quadrature (Eq.66, table 1 with $m = 2$).

3.2. Diffusion in a cylinder. Use a Galerkin linear FEM approximation to compute the diffusion of heat when a homogeneous cylinder of radius r_c, initial temperature T_c and conduction capacity κ is plunged into an infinite warm bath at a fixed temperature T_w. (Hint: solve the heat equation for the evolution of the temperature $T(r,t)$ in cylindrical geometry

$$\frac{\partial T}{\partial t} - \kappa \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) = 0 \tag{89}$$

Formulate a variational problem, integrate by parts and impose natural boundary conditions to guarantee a regular behavior of the solution in the limit $r \to 0$. Choose either a 2-points Gaussian or a trapezoidal quadrature and form a linear system of equations which you complete with essential boundary conditions on the cylinder surface $r = r_c$. Implement the scheme in the JBONE applet.

3.3. Mass lumping. Examine the possibility of "lumping the mass matrix", a procedure invented by structural engineers in the 1970's where the **mass matrix** ($\mathbf{A_{ij}}$) in Eq.65 is artificially set to unity to obtain an explicit scheme in terms only of the **stiffness matrix** ($\mathbf{B_{ij}}$). Study the advection problem separately from the diffusion.

3.4. Iterative solver. Implement an iterative–SOR solver in JBONE and study the number of iterations required to achieve a 10^{-6} precision for different values of the diffusion coefficient and advection velocity.

3.5. Obstacle problem. Extending your knowledge from exercise 2.4, use a Galerkin linear FEM formulation to solve the Black-Scholes equation for an *American put option*, which differs from the *European* in that it can be exercised at anytime until it expires:

$$\int_{x_-}^{x^+} \frac{\partial u}{\partial \tau}(\phi - u) + \frac{\partial u}{\partial x}\left(\frac{\partial \phi}{\partial x} - \frac{\partial u}{\partial x}\right) dx \geq 0, \qquad \forall \phi \in \mathcal{W}, \quad \forall \tau \in [0; \frac{1}{2}\sigma^2 T] \tag{90}$$

Restrict $\phi \in \mathcal{W}$ to a piecewise linear test function which remains larger than the transformed payoff functions $\phi(x,\tau) \geq g(x,\tau)$ for all x and τ

$$g_{\text{put}}(x,\tau) = \exp\left\{\left[\frac{1}{4}(k_2-1)^2 + 4k_1\right]\tau\right\} \max\left\{0, \exp\left[\frac{1}{2}(k_2-1)x - \frac{1}{2}(k_2+1)x\right]\right\} \tag{91}$$

$$g_{\text{call}}(x,\tau) = \exp\left\{\left[\frac{1}{4}(k_2-1)^2 + 4k_1\right]\tau\right\} \max\left\{0, \exp\left[\frac{1}{2}(k_2+1)x - \frac{1}{2}(k_2-1)x\right]\right\} \tag{92}$$

and satisfying the boundary conditions $\phi(x_+,\tau) = g(x_+,\tau)$, $\phi(x_-,\tau) = g(x_-,\tau)$, $\phi(x,0) = g(x,0)$. Implement the scheme in JBONE and compare with the solution previously obtained for the *European put* in exercise 2.4.

3.6 Further Reading

- **Finite elements.**
  Johnson [6], Fletcher [4], Saad [14] §2.3

- **Quadrature.**
  Abramowitz [5] §25.4.29 with table 25.4, Numerical Recipes [2] §4.5

- **Linear solvers.**
  Dahlquist [3] §6, Saad [14], Numerical Recipes [2] §2.4.
  Software from `netlib`[15], and `PetSc`[16],

[15] http://www.netlib.org
[16] http://www.mcs.anl.gov/petsc

4 FOURIER TRANSFORM (FT)

4.1 Fast Fourier Transform (FFT) with the computer

As mentioned earlier in sect.1.4, it is thanks to the possibility of computing efficiently the Fourier transformation in $\mathcal{O}(N \log N)$ operations that FFT's can be considered as a viable alternative to solve partial differential equations. And here again, it is sufficient to have only a rough idea of the underlying process to efficiently use the routines from software libraries. Particularly for FFT's, you should **privilege vendors implementations** which have in general been optimized for the specific computer you are using. Since no library is available yet in JAVA, a couple of routines from Numerical Recipes [2] have here been translated for JBONE into the FFT.java class, making as little modification as possible to the original code to enable you reading further about the algorithm directly in the book. The Complex.java class has moreover been imported to illustrate how rewarding it is to work with Netlib [19] libraries.

You should remember that a complex function $f(x)$ of period $L = 2\pi/K$ can be represented with a **complex series**

$$f(x) = \sum_{m=-\infty}^{\infty} c_m \exp(imKx), \qquad c_m = \frac{1}{L}\int_a^{a+L} f(x)\exp(-imKx)dx \tag{93}$$

On the other hand, you can use the **cosine − sine** form

$$f(x) = \frac{a_0}{2} + \sum_{m=1}^{\infty} [a_m \cos(mKx) + b_m \sin(mKx)] \tag{94}$$

$$a_m = \frac{2}{L}\int_a^{a+L} f(x)\cos(mKx)dx \quad (m \geq 0), \qquad b_m = \frac{2}{L}\int_a^{a+L} f(x)\sin(mKx)dx \quad (m \geq 1) \tag{95}$$

with the following relations between the Fourier coefficients

$$\begin{array}{lll} c_0 = \frac{1}{2}a_0, & c_m = \frac{1}{2}(a_m - ib_m), & c_{-m} = c_m^*, \quad m \geq 1 \\ a_0 = 2c_0, & a_m = c_m + c_{-m}, & b_m = i(c_m - c_{-m}), \quad m \geq 1 \end{array} \tag{96}$$

and only if $f(x)$ is real

$$a_0 = 2c_0, \quad a_m = 2\mathrm{Re}(c_m), \quad b_m = -2\mathrm{Im}(c_m), \quad c_m = c_{-m}^*, \quad m \geq 1 \tag{97}$$

This is almost how the transformation is implemented in FFT.java, except that

1. the number of modes is **assumed to be an integer power of 2**. The creation of an FFT-object from data sampled on a homogeneous mesh relies on the command

```
double f = new double[64];
FFT myFTObject = new FFT(f, FFT.inXSpace);
```

   where `FFT.inXSpace` chooses the original location of the variable `myFTObject` of type `FFT`.

2. To **avoid negative indices** and relying on the periodicity $f(x+L) = f(x)$, negative harmonics $(-m)$ are stored **in wrap-around order** after the positive components and are indexed by $N/2 - m$. For a real function $f(x) = [7 + 8\cos(2\pi x/64) + 4\sin(2\pi x/32) + 3\cos(2\pi x/2)]$ sampled on a mesh $x_j = 0, 1, ...63$ with a period $L = 64$, the call to

```
Complex spectrum = new Complex[64];
double periodicity=64.;
spectrum = myFTObject.getFromKSpace(FFT.firstPart,periodicity);
for (int m=0;  m<N;  m++)
   System.out.println("spectrum["+m+"] = "+spectrum[m]);
```

will print the non-zero components

```
spectrum[0] = (7. +0.0i)
spectrum[1] = (4. +0.0i)          spectrum[63] = (4. +0.0i)
spectrum[2] = (0. +2.0i)          spectrum[62] = (0. -2.0i)
spectrum[32]= (3. +0.0i)
```

It is easy to see that the shortest wavelength component [32] will always be real if $f(x)$ was real, since $\sin(2\pi x/2)$ is then sampled exactly for the multiples of π.

3. Relying on the linearity of the FFT, **two real numbers** are packed into one complex number and transformed for the cost of a single complex transformation by initializing

```
double f = new double[64];
double g = new double[64];
FFT myPair = new FFT(f,g, FFT.inXSpace);
```

This is the reason for the argument `FFT.firstPart` used here above, asking for the spectrum of only the first (and here in fact only) function.

Apart from the array index which starts with [0] in JAVA, the implementation is equivalent to the routines in Numerical Recipes [2] and indeed very similar to many computer vendors.

4.2 Linear equations.

Albeit slower for a single time-step than all the numerical schemes seen so far, an FFT can now be used to compute the Fourier representation of the advection-diffusion problem (Eqs.5 and 7), and describe the evolution of each Fourier component f_m separately

$$\frac{d\widehat{f_m}}{dt} + ik_m u\widehat{f_m} + k_m^2 D\widehat{f_m} = 0, \qquad k_m = \frac{2\pi m}{L} \tag{98}$$

This can be integrated analytically **without any restriction on the size of the time-step**; starting directly from the initial condition yields for every Fourier component

$$\widehat{f_m}(t) = \widehat{f_m}(0)\exp[-(ik_m u + k_m^2 D)t)] \tag{99}$$

After an initialization, where the initial condition is discretized by sampling on a regular mesh and stored for subsequent transformation, the scheme implemented in JBONE reads:

```
int N         = mesh_.size();                    //A power of 2
double boxLen = mesh_.size()*mesh_.interval(0);  //Periodicity
double   k    = 2*Math.PI/boxLen;                //Notation
Complex ik1   = new Complex(  0.,  k  );
Complex ik2   = new Complex(-k*k,  0.  );

Complex advection = new Complex(ik1.scale(velocity));  //Variables
Complex diffusion = new Complex(ik2.scale(diffusCo));

Complex[] s0 = new Complex[f.length];              //FFT real to KSpace
s0=keepFFT.getFromKSpace(FFT.firstPart,boxLen);    //  only once

s[0] = s0[0];
for (int m=1;  m<N/2+1;  m++) {                    //Propagate directly
  total=          advection.scale((double)(m ));   //  from initial
  total=total.add(diffusion.scale((double)(m*m))); //  contition

  exp=(total.scale(timeStep*(double)(jbone.step))).exp();
```

```
    s[m  ] = s0[m].mul(exp);                    //  s0 contains the IC
    s[N-m] = s[m].conj();                       //  f is real
  }
  FFT ffts = new FFT(s,FFT.inKSpace);           //Initialize in Kspace
  f = ffts.getFromXSpacePart(FFT.firstPart,boxLen); //FFT real back for plot
```

If you were careful enough when reading, you should now wonder about the sign of the phase factor, which is exactly the opposite from (Eq.99); the reason is the phase of the spatial harmonics which is reversed when compared with choice in the FFT routine from Numerical Recipes [2]. This makes the scheme look like as if it evolves backwards in time, it doesn't ! Also note that once the initial condition has been transformed to K-space, subsequent transformations back to X-space are in fact only required for the plotting. The on-line document illustrates the advection of a box calculated with the same time step $\Delta t = 0.5$ as previously. The **Gibbs phenomenon** —an artifact from using harmonic functions for the discretization, see sect.1.4— is clearly visible except when $u\Delta t/\Delta x$ is an integer; don't get fooled however by the plotting, which (for simplicity, but still incorrectly) misses parts of the oscillations visible in figure 4 by linear interpolation between the grid points. The power of the method is evident when taking larger time-steps: edit the parameter to `TIMSTP=64`, add some diffusion `DIFFUS=0.3` and compare the solution obtained in one single step with the result computed using your favorite FD or FEM scheme from the previous sections[17]. This is very nice indeed, but remember that dealing with an inhomogeneous medium $u(x)$, $D(x)$ or complicated boundary conditions can be rather problematic in Fourier space.

4.3 Aliasing, filters and convolution.

One of the beauties when using Fourier transforms, is the ability to work with a spectrum of modes and act on each of the components individually with a filter. By **sampling** a function of period L with a finite number of values $N = L/\Delta$ where Δ is the size of the sampling interval, the spectrum gets truncated at the shortest wavelength $\lambda_c = 2\Delta = 2\pi/k_c$ called **Nyquist critical wavelength** corresponding to exactly 2 mesh points per wavelength. This does however not mean that shorter wavelengths $|k| > k_c$ do not contribute to the Fourier coefficients (Eq.93). Figure 12 illustrates how they get **aliased** back into the lower components of the spectrum. This can be important

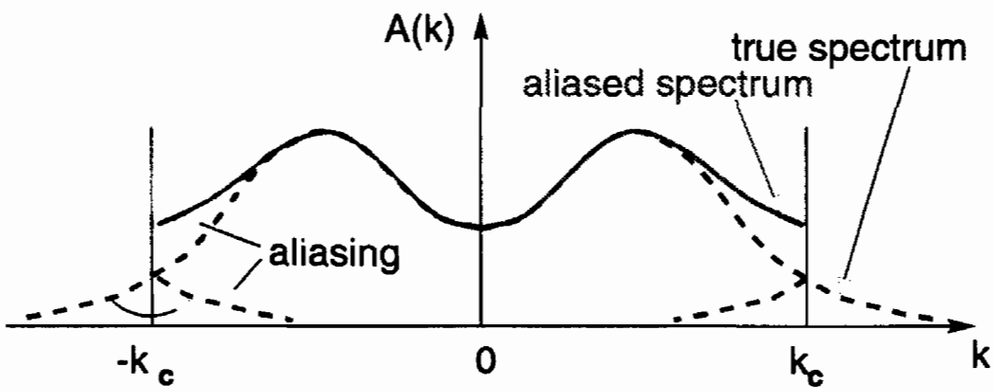


Figure 12: Aliasing from Fourier components shorter than the Nyquist critical wavelength $|k| > k_c$.

for the digital data acquisition of an experiment, where the signal has to be **low-pass filtered before the sampling**. Figure 13 shows that even with the greatest precautions, such an aliasing can sometimes not be avoided, an needs then to be correctly interpreted.

Note that it is extremely easy to design **a filter in Fourier space** simply by multiplying the spectrum by a suitable filter function $\mathcal{H}(k)$ (exercise 4.5). Simply remember that

- to keep the data real after transforming back to X-space, you must keep $\mathcal{H}(-k) = \mathcal{H}(k)^*$, for example by choosing $\mathcal{H}$ real and even in k,

[17] Don't forget to reduce again the time-step !

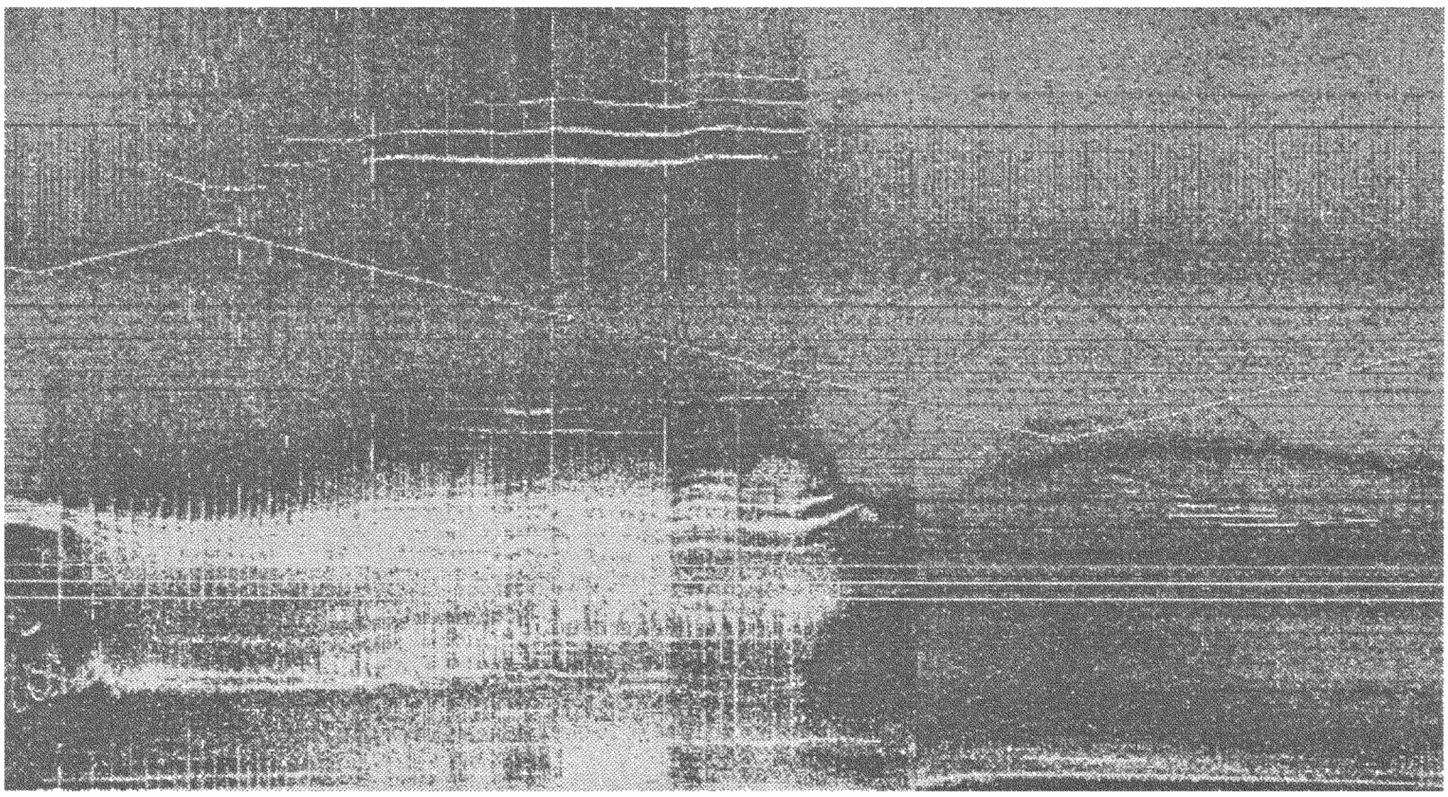


Figure 13: Experimental spectrum 0-500 kHz digitally recorded during 2 sec from the magnetic perturbations in a fusion plasma in the Joint European Torus. Apart from the Alfvén instabilities which are the subject of this research, you can see the sawteeth-like trace of an exciter antenna reaching a minimum of 200 kHz around 1.5 sec; despite heavy analogic low-pass filtering before the signal is sampled, the large dynamic range of the measurement above 80 dB is here sensitive enough to pick up (dark red line) a non-linearly induced high-frequency component which is **aliased** down into the frequency range of interest. Courtesy of Prof. A. Fasoli (MIT/USA).

- the filter has to be defined in the entire interval $k \in [-k_c; k_c]$ and should be smooth to avoid phase errors and dampings for wavelengths corresponding to the sharp edges.

Although they are present from the beginning when the initial condition is discretized (try to initialize and propagate an aliased cosine with a wavelength `IC1WID=1.05` using the JBONE applet above), aliases do not actually interfere with the resolution of linear equations. The story is however different for spatial non-linearities such as the quadratic term $\partial_x f^2(x)$ responsible for the wave-breaking Eq.13). This can be understood from the **convolution theorem**, telling that the Fourier transform of a convolution $\widehat{f * g}$ is just the product of the individual Fourier transforms $\hat{f}\hat{g}$. The converse is unfortunately also true: what can be viewed as a simple product in X-space becomes a convolution in K-space

$$\widehat{f(x)g(x)} = \hat{f} * \hat{g}, \qquad \hat{f} * \hat{g} \equiv \int \hat{f}(k')\hat{g}(k-k')dk' \tag{100}$$

or in discrete form

$$(\hat{f} * \hat{g})_m \equiv \sum_{k=-N/2+1}^{N/2} \hat{f}_k \hat{g}_{m-k} \tag{101}$$

For the quadratic "wave-breaking" non-linearity, this shows that a short wavelength component such as $\hat{f}_{+31}$ in a sampling with 64 points, will falsely **"pollute" a long wavelength channel through aliasing**: $(\hat{f}_{+31} * \hat{f}_{+31}) = \widehat{f^2}_{+62} \longrightarrow \widehat{f^2}_{-2}$. A simple cure for this, is to **expand the size of the arrays** by a factor two **before** the convolution takes place and **pad them with zeros**; changing representation to calculate the multiplication of arrays twice the orgiginal size, the upper

part of the spectrum is then simply discarded **after** the data has been transformed back. The entire procedure is illustrated in the coming section, where the non-linear Korteweg-DeVries (15) and Burger equations (14) are solved with a convolution in Fourier space.

4.4 Non-linear equations.

Combining the linear terms from advection (Eq.5), diffusion (Eq.7), dispersion (Eq.11) and the non-linear wave-breaking term (Eq.13) into a single non-linear equation, yields

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} - D\frac{\partial^2 f}{\partial x^2} + b\frac{\partial^3 f}{\partial x^3} + \frac{1}{2}\frac{\partial f^2}{\partial x} = 0 \tag{102}$$

where the last term has been written so as to explicitly show the quadratic non-linearity. After transformation to Fourier space, all the spatial operators become algebraic and an ordinary evolution equation is obtained for each individual Fourier component

$$\frac{d\widehat{f_m}}{dt} + (ik_m u + k_m^2 D - ik_m^3 b)\widehat{f_m} + \frac{1}{2}ik_m\widehat{f_m^2} = 0 \tag{103}$$

It would of course formally be possible to write the non-linear term as a convolution in K-space, but it is here much easier to write and efficient to compute the multiplication in X-space. Following the same lines as in sect.4.2, the linear terms are integrated analytically and yield a phase shift proportional to the time-step Δt. The convolution is calculated numerically by transforming back and forth from Fourier to configuration space, and after a simple Euler integration in time yields the formal solution

$$\widehat{f_m}^{t+\Delta t} = \widehat{f_m}^{t} \exp[-(ik_m u + k_m^2 D - ik_m^3 b)t] + \frac{\Delta t}{2} ik_m \widehat{f_m^2}^{t} \tag{104}$$

This has been implemented in JBONE using the variable keepFFT to store the current values of spectrum $\widehat{f_m}^t$ and the variable toolFFT for the transformation to X-space required by the convolution and the plotting. As mentionned earlier in sect.4.2, the sign of time in (Eq.104) has been changed to stick to the definition of the phase factor used in Numerical Recipes [2].

```
int N         = mesh_.size();                    //A power of 2
double boxLen = mesh_.size()*mesh_.interval(0);  //Periodicity
double   k    = 2*Math.PI/boxLen;                //Notation
Complex ik1   = new Complex(  0.,  k  );
Complex ik2   = new Complex(-k*k,  0.  );
Complex ik3   = new Complex(  0.,-k*k*k);

Complex advection = new Complex(ik1.scale(velocity));  //Variables
Complex diffusion = new Complex(ik2.scale(diffusCo));
Complex dispersion= new Complex(ik3.scale(disperCo));

//----- Non-linear term: convolution
s  = keepFFT.getFromKSpace(FFT.bothParts,boxLen);  //Current Spectrum
toolFFT = new FFT(s,s,FFT.inKSpace);               //  for convolution

if (scheme_.equals(Data.ALIASED))                  //With-/out aliasing,
  sp = toolFFT.aliasedConvolution(boxLen);         //  use an FFT to
else //scheme_.equals(Data.EXPAND)                 //  calculate product,
  sp = toolFFT.expandedConvolution(boxLen);        //  FFT back to KSpace


//----- Linear terms: complex terms in spectrum s
s = keepFFT.getFromKSpace(FFT.bothParts,boxLen);   //Current Spectrum
linear= s[0];
sp[0]=linear;
```

```
for (int m=1;  m<=N/2;  m++) {
  total=          advection.scale((double)(m ));
  total=total.add(diffusion.scale((double)(m*m)));
  total=total.add(dispersion.scale((double)(m*m*m)));
  exp=(total.scale(timeStep)).exp();
  linear    = s[m].mul(exp);
  nonlin    = sp[m].mul(ik1.scale(0.5*timeStep*(double)(m)));
  sp[m  ]   = linear.add(nonlin);
  if (m<N/2) sp[N-m] = sp[m].conj();              //For a real spectrum
}

keepFFT = new FFT(sp,FFT.inKSpace);               //Spectrum is complete
toolFFT = new FFT(sp,FFT.inKSpace);               //inv FFT for plotting
f=toolFFT.getFromXSpacePart(FFT.firstPart,boxLen);
```

Note that the convolution can here, depending on the choice of the scheme, be calculated either without precaution and is then subject to aliasing, or by temporarily expanding the spectrum padding the upper part with zeros.

The on-line document shows the evolution obtained for the Korteweg-DeVries equation, when two solitons collide as the non-linear wave-breaking is balanced by dispersion. Change the switch to `Aliased Convolution` and verify how the aliasing pollutes the spectrum with short wavelengths which then rapidly develop into an instability. Replacing the dispersion with a small amount of diffusion `DIFFUS=0.1`, evolve a Gaussian into a shock front and verify how much less aliasing seems to be an issue when a diffusive process damps the short wavelengths. Remember however that the cascade of energy from one wavelength to another is still affected by the aliasing and is now much more delicate to diagnose !

4.5 Exercises

4.1. Advection-diffusion. Propose an alternative scheme solving the linear advection-diffusion problem in Fourier space, evolving the solution with small steps in time Δt.

4.2. Equivalent filter for Zabusky's FD scheme. Having studied and tested Zabusky's finite difference scheme for the Korteweg-DeVries equation [23]

$$\frac{f_j^{t+\Delta t} - f_j^{t-\Delta t}}{2\Delta t} + \frac{1}{3}\left[f_{j+1}^t + f_j^t + f_{j-1}^t\right]\frac{f_{j+1}^t - f_{j-1}^t}{2\Delta x} + b\frac{f_{j+2}^t - 2f_{j+1}^t + 2f_{j-1}^t - f_{j-2}^t}{2\Delta x^3} = 0 \tag{105}$$

using JBONE with the common choice $b = 1/2$, calculate the equivalent Fourier space filter which is in fact implied when the calculation is performed in configuration space using (Eq.105). Add the filter to the non-linear Fourier scheme available in JBONE and check that after filtering, both the FD and FT methods indeed become equivalent.

4.3. Prototype problems. Modify the parameters included in the scheme of (Eq.104) to develop a better qualitative understanding of what are advection, diffusion, dispersion and wave-breaking. Choose a regime where you trust the numerical description and propose a combination of parameters you believe are particularly interesting.

4.6 Further Reading

- **FFT algorithm.**
  Numerical Recipes [2] §12

5 MONTE-CARLO METHOD (MCM)

5.1 Monte Carlo integration

A Monte Carlo method is a numerical method involving random numbers. The most common use of Monte Carlo methods is for calculating multi dimensional integrals. Consider the two formulations of the mean value of the function f.

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(\xi)\, d\xi \tag{106a}$$

$$\langle f \rangle = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{N-1} f(x_i) \qquad x_i = a + \frac{b-a}{N-1} i \tag{106b}$$

If instead $\{x_i\}$ are uniformly distributed random number on the interval $[a, b]$, the integral can be

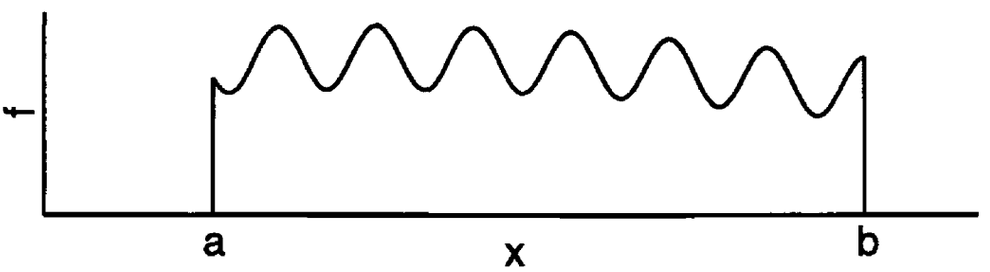


Figure 14: The most common use of Monte Carlo methods is for calculating integrals.

approximated by

$$\frac{1}{b-a} \int_a^b f(\xi)\, d\xi = \frac{1}{N} \sum_{i=1}^{N} f(x_i) + \mathcal{O}\left(\frac{1}{\sqrt{N}}\right) \tag{107}$$

The real advantage of the MCM is for integrals of dimensionality d over a hyper volume V

$$\langle f \rangle = \frac{1}{V} \int_V f(\boldsymbol{\xi})\, d^d\xi = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_i) + \mathcal{O}\left(\frac{1}{\sqrt{N}}\right) \tag{108}$$

where the error scales like $N^{-1/2}$, whereas (106b) would give an error of the order $\mathcal{O}\left(N^{-1/d}\right)$.

5.2 Stochastic theory

This section includes a brief review of the stochastic theory behind Monte Carlo methods. For further reading see Kloeden [24].

Definition *$\mathcal{N}(\mu, \sigma)$ is the set of Gaussian or normal distributed stochastic variable X with density distribution function*

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \tag{109}$$

where μ is the mean and σ^2 is the variance.

Definition *$\mathcal{U}(a, b)$ is the set uniformly distributed random number on the interval $[a, b]$, where $a < b$.*

Definition *A process W_t is a Wiener process (or Brownian motion) in t if and only if*

1. $W_{t+\Delta t} - W_t \in \mathcal{N}(0, \sqrt{\Delta t})$, *where* $\mathcal{N}$ *is the set of normal distributed random numbers.*

2. $\mathcal{E}[W_t dW_t] = \mathcal{E}[W_t]\mathcal{E}[dW_t]$, *where* $\mathcal{E}$ *denotes expected value,* $dW_t = W_{t+\Delta t} - W_t$ *and* $\Delta t > 0$, *i.e. a Wiener increment* dW_t *is independent of the past.*

Note that the distribution function of a Wiener increment $W_{t+\Delta t} - W_t$ is essentially the same function as the Green's function of the diffusion equation (8).

Due to causality, the Wiener increment dW_t must always be evaluated in future time, i.e. with an explicit discretization

$$dW_t = W_{t+dt} - W_t \tag{110}$$

An implicit discretization would give a fundamentally different process.

Definition *The Itô differential is given by the limit of an explicit (forward Euler) discretization*

$$b(W_t, t) \circ dW_t = \lim_{\Delta t \to 0} b(W_t, t)(W_{t+\Delta t} - W_t) \tag{111}$$

where the circle, $\circ$ *stands for Itô differential and* $b(W_t, t)$ *is a function.*

This definition is the base of the so called Itô *calculus* which is fundamentally different from ordinary Riemann calculus. For example

$$\int_0^T W_t dW_t = \frac{1}{2}(W_T^2 + T) \tag{112}$$

5.3 Particle orbits

Solving PDEs with a MCM is fundamentally different from fluid methods such as FD and FEM. When solving PDEs with a MCM, the function is discretized using quasi particles, and therefore the equations needs to be formulated as acting on a particle. Later on methods for transforming the equation from a PDE to an equation for the particles will be presented.

A deterministic particle orbit can be described by the position $\mathbf{X}(t)$ at the time t, where

$$\frac{d\mathbf{X}(t)}{dt} = \mathbf{v}(\mathbf{X}(t), t) \tag{113}$$

or

$$d\mathbf{X}(t) = \mathbf{v}(\mathbf{X}(t), t)dt \tag{114}$$

The time discretization of this equation could be done both explicitly and implicitly

$$\mathbf{X}(t + \Delta t) = \mathbf{X}(t) + \mathbf{a}(\mathbf{X}(t), t)\Delta t \qquad \text{"explicit"} \tag{115a}$$

$$\mathbf{X}(t + \Delta t) = \mathbf{X}(t) + \mathbf{a}\left(\mathbf{X}(t + \Delta t), t + \Delta t\right)\Delta t \qquad \text{"implicit"} \tag{115b}$$

For an ensemble of N particles positioned at $\mathbf{x}_i(t)$, a particle density distribution function $f(\mathbf{x}, t)$ can be constructed according to section 1.4. Here, the Dirac distribution $\delta(\mathbf{x})$ will be used as the particle shape $S_i(\mathbf{x})$ in order to simplify the calculations of the MC increments

$$f(\mathbf{x}, t) = \sum_{i=0}^{N} \delta(\mathbf{x} - \mathbf{x}_i(t)) \tag{116}$$

For a **large** number of particles the density distribution function can be approximated by a smooth function, as is done in continuum physics. For a smooth density distribution function $f(\mathbf{x}, t)$, in which the individual particles move according to

$$d\mathbf{X}(t) = \mathbf{v}(\mathbf{X}(t), t)dt. \tag{117}$$

is described by the advection equation

$$\frac{\partial f}{\partial t} + \nabla \cdot (\mathbf{v}f) = 0 \tag{118}$$

This is called **Taylor's transport theorem**. This theorem gives us the possibility of studying a PDE instead of a N-particle system, but the reverse is of course also possible and used in particle methods for advection equation.

We now introduce the concept of **stochastic particle orbits**, having an ensemble of possible orbits of different probability. As an example picture a snowflake, slowly falling from the sky. The motion is irregular, and position at time t can be described by a stochastic process $\mathbf{X(t)}$. In general stochastic particle orbit is given by

$$d\mathbf{X}(t) = \mathbf{v}(\mathbf{X}(t), t)dt + b(\mathbf{X}(t), t) \circ d\mathbf{W}_t \tag{119}$$

where $\mathbf{W}_t$ is a vector Wiener processes (or Brownian motions). In the general case, b is a matrix, but here, we only consider it as a scalar.

To characterize a stochastic motion, the expected value $\mathcal{E}$ and the variance $\mathcal{V}$ is used

$$\mathcal{E}[X(t)](g(x)) \triangleq \int_{-\infty}^{\infty} g(x) f_X(x, t)\, dx \tag{120}$$

$$\mathcal{V}[X(t)](x) \triangleq \mathcal{E}\left[X(t)\right]\left((x - \mathcal{E}[X(t)](x))^2\right) \tag{121}$$

$$= \mathcal{E}\left[X(t)\right]\left(x^2\right) - \mathcal{E}^2\left[X(t)\right](x) \tag{122}$$

where $f_X(x, t)$ is the density distribution function of the stochastic variable X. In exercise 5.2 we will show that

$$\frac{\partial}{\partial t}\mathcal{E}[\mathbf{X(t)}](\mathbf{x}) = \mathbf{v}(\mathbf{x}, t) \tag{123}$$

$$\frac{\partial}{\partial t}\mathcal{V}[\mathbf{X}(t)](\mathbf{x}) = \left[b(\mathbf{x}, t)\right]^2. \tag{124}$$

where $\frac{\partial}{\partial t}\mathcal{E}[\mathbf{X(t)}](\mathbf{x})$ is the average particle velocity and $\frac{\partial}{\partial t}\mathcal{V}[\mathbf{X}(t)](\mathbf{x})$ is a measure of broadening of the distribution of possible orbits. An alternative way of expressing (119) is then

$$d\mathbf{X}_t = \frac{\partial}{\partial t}\mathcal{E}\left[\mathbf{X}_t\right](\mathbf{x})dt + \sqrt{\frac{\partial}{\partial t}\mathcal{V}\left[\mathbf{X}_t\right](\mathbf{x})} \circ d\mathbf{W}_t \tag{125}$$

Example For the advection diffusion

$$\frac{\partial f(x,t)}{\partial t} + \frac{\partial}{\partial x}\left[v(x)f(x,t)\right] = \frac{\partial}{\partial x}\left(D(x)\frac{\partial f}{\partial x}\right), \quad x \in (-\infty, \infty) \tag{126}$$

then

$$\frac{\partial}{\partial t}\mathcal{E}[X(t)](x) = v(X(t)) + \left(\frac{\partial}{\partial x}D(x)\right)_{x=X(t)} \tag{127a}$$

$$\frac{\partial}{\partial t}\mathcal{V}[X(t)](x) = 2D(X(t)) \tag{127b}$$

The derivation is done in exercise 5.1.

A differentiable density distribution function $f(\mathbf{x}, t)$, in which the individual particles move according to

$$d\mathbf{X}(t) = \mathbf{v}(\mathbf{X}(t), t)dt + b(\mathbf{X}(t), t) \circ d\mathbf{W}_t \tag{128}$$

is described by the Kolmogorov-Fokker-Planck equation

$$\frac{\partial f}{\partial t} = -\nabla \cdot (\mathbf{v}f) + \nabla^2 \left(\frac{b^2}{2} f\right) \tag{129}$$

which is an advection-diffusion equation. This is called the **Feynman-Kac theorem.**

5.4 A Monte Carlo method for advection diffusion equation

The time discretization of equation (119) must, due to causallity, be done using explicit time stepping

$$\mathbf{X}_{t+\Delta t} = \mathbf{X}_t + \mathbf{v}(\mathbf{X}_t, t)\Delta t + b(\mathbf{X}_t, t)(\mathbf{W}_{t+\Delta t} - \mathbf{W}_t) \tag{130}$$

where $\mathbf{X}_t = \mathbf{X}(t)$. Since $(W_{t+\Delta t} - W_t) \in \mathcal{N}\left(0, \sqrt{\Delta t}\right)$, the Wiener process can be written as

$$\mathbf{W}_{t+\Delta t} - \mathbf{W}_t = \zeta\sqrt{\Delta t} \tag{131}$$

where ζ is a vector of normal distributed random numbers in $\mathcal{N}(0, 1)$. According to the central limit theorem any sum of equally distributed random numbers with zero mean and unit variance converge to $\mathcal{N}(0, 1)$. Therefore any such number could be used in ζ if a large number of time steps is used. However, $\zeta \in \mathcal{N}(0, 1)$ will give fastest convergence.

Consider a N-particle ensemble. A Monte Carlo method is constructed by for each particle in the ensemble iterating forward in time

$$\mathbf{X}_{t+\Delta t} = \mathbf{X}_t + \mathbf{v}(\mathbf{X}_t, t)\Delta t + \zeta b(\mathbf{X}_t, t)\sqrt{\Delta t} \tag{132}$$

or

$$\mathbf{X}_{t+\Delta t} = \mathbf{X}_t + \frac{\partial}{\partial t}\mathcal{E}\left[\mathbf{X}_t\right]\Delta t + \zeta\sqrt{\left(\frac{\partial}{\partial t}\mathcal{V}\left[\mathbf{X}_t\right]\right)\Delta t} \tag{133}$$

This equation describes an ensemble of orbits since ζ is a stochastic variable. In the Monte Carlo method we choose one value of ζ giving one of the orbits. By using a large number of particles the entire ensemble of orbits could be approximated.

Example The advection diffusion (133) is implemented in JBONE

```
for(int j = 0; j < numberOfParticles; j++){
  particlePosition[j] += velocity * timeStep +
    random.nextGaussian() *
    Math.sqrt(2 * diffusCo * timeStep);
  // Periodic boundary conditions

  ...

} // for
```

JAVA is of the few programming languages that do have a pseudo random numbers $\mathcal{N}(0, 1)$. Most programming languages do not, but they usually have a pseudo random numbers in $\mathcal{U}(0, 1)$. Random numbers in $\mathcal{N}(0, 1)$, can then be obtained by using the **Box Müller method**

- Construct two uniformly distributed random numbers $U_1, U_2 \in \mathcal{U}(0,1)$.
- Then

$$N_1 = \sqrt{-2\ln(U_2)}\cos(2\pi U_1) \tag{134a}$$
$$N_2 = \sqrt{-2\ln(U_2)}\sin(2\pi U_1) \tag{134b}$$

  are two independent pseudo random numbers in $\mathcal{N}(0,1)$.

Example: 1D Diffusion equation As an example let us calculate a Monte Carlo approximation of the temperature in a 1D slab $u(x,t)$, when

$$\frac{\partial u(x,t)}{\partial t} = D\frac{\partial^2}{\partial x^2}u(x,t) \qquad 0 \leqslant t \tag{135}$$

with the initial condition

$$u(x,0) = \begin{cases} 1 & \text{if } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{136}$$

- Imagine N heat-particles, such that the total heat is given by the local density of particles. According to the Feynman-Kac theorem and equation (135) the position of the i'th particle is given by

$$dx_i(t) = \sqrt{2D} \circ dW_t, \quad i = 1, 2...N \tag{137}$$

  where W_t is a Wiener process.

- Discretize

$$x_i(t + \Delta t) = x_i(t) + \zeta\sqrt{2D\Delta t}. \tag{138}$$

- Randomize the initial positions $x_i(0)$ of the N particles, by using a good pseudo random number generator of $\mathcal{U}(0,1)$.
- Use the Box Müller method to calculate the pseudo random numbers in $\zeta \in \mathcal{N}(0,1)$.
- For each particle calculate step by step the evolution $x_i(0) \rightarrow x_i(\Delta t) \rightarrow ...$
- The solution could be visualized by projecting it on a base suitable for plotting according to (33).

5.5 When to use Monte Carlo methods

Monte Carlo methods are efficient for high dimensions and in certain complex geometries. For d-dimensional problem the accuracies are show in table 2. As seen in the table, for $d/n > 2$, Monte

Method	**Accuracy**
Monte Carlo	$\mathcal{O}\left(N^{-1/2}\right)$, N is the number of particles.
FEM/FD (n:th-order)	$\mathcal{O}\left(N^{-n/d}\right)$, N is number of grid points.

Table 2: Accuracies for a d-dimensional problem.

Carlo methods are more efficient then FD or FEM (in the limit of $N \rightarrow \infty$).

For problems with complicated boundary conditions the MCM might be preferable. For illustration picture a 3D cube with a ball bouncing inside. Let the larger cube contain a gas, which

bounces on the surfaces of the cube and the ball. For simplicity let the cube and the ball have infinite mass during the collisions with the gas particles. The gas distribution is fairly easy to solve with MCM, but untraceable with a fluid method. Another advantage is that the range in x is limited only by the range of the floating numbers of the computer, c.f. FEM and FD, where this problem needs to be addressed with renormalization.

Parallelization is easy and efficient in MCM. However, the MCM is not that efficient for non-linear problems. If a and b are functions of the density distribution, the continuous density distribution function needs to be approximated at each time step. This will dramaticly reduce the performance and parallelization efficiency.

For a problem with completely decoupled particles, parallelization is especially easy. Just run a copy of the simulation program on several machines simultaneously. The result can then be added and normalized afterwards since, for any linear operator $\mathcal{L}$ acting on f

$$\mathcal{L}[f] = \mathcal{L}\left[\sum_{i=1}^{N} f_i\right] = \sum_{i=1}^{N} \mathcal{L}[f_i] \tag{139}$$

where f_i is either the particle or the projection of the particle (33). One should however be careful to seed the random numbers differently on the different machines, or the simulations will be identical.

5.6 Exercises

5.1. Statistics of particle diffusion. Calculate $\frac{\partial}{\partial t}\mathcal{E}[X(t)](x)$ and $\frac{\partial}{\partial t}\mathcal{V}[X(t)](x)$, where $X(t)$ is the position of a particle with a density distribution function $f(x,t)$, given by

$$\frac{\partial f(x,t)}{\partial t} + \frac{\partial}{\partial x}\left[v(x)f(x,t)\right] = \frac{\partial}{\partial x}\left(D(x)\frac{\partial f}{\partial x}\right), \quad x \in (-\infty, \infty) \tag{140}$$

With initial condition $X(t) = x_i$ the distribution density is

$$f(x,t) = \delta(x - x_i(t)) \tag{141}$$

Hint: Use partial integration to remove the derivatives of the Dirac distribution. The time derivative of $\mathcal{E}$ is obtained by

$$\frac{\partial}{\partial t}\mathcal{E}[X(t)](x) = \frac{\partial}{\partial t}\int_{-\infty}^{\infty} f(x,t)x\,dx = \int_{-\infty}^{\infty} \frac{\partial f(x,t)}{\partial t}x\,dx \tag{142}$$

5.2. Statistics of particle diffusion. Verify equation (123) and (124) using your results from the previous exercise and the Feynman-Kac theorem.

5.3. Periodic boundary conditions. Add periodic boundary conditions to the Monte Carlo solver in the JBONE applet. *Hint:* Periodic boundary conditions are equivalent with that there are meshes with identical particle distributions to the left and right of each mesh, i.e. for a particle lost to the right an identical particle should enter from the left. Also note that a kick might be larger than the mesh. To get the size of the mesh use

```
double[] lim = {mesh_.point(0),
                mesh_.point(mesh_.size() - 1) + dx[0]};
```

where `lim[0]` will be the left boundary and `lim[1]` the right. `lim[1]` includes the extra non plotted mesh cell between the identical meshes.

5.4. Steady state with velocity gradient. Simulate the equation

$$\frac{\partial f}{\partial t} = -\frac{\partial}{\partial x}\left(\frac{(x_0 - x)}{s} f\right) + D\frac{\partial^2 f}{\partial x^2} \tag{143}$$

Modify x_0, D and s in order to get a steady state. Note that a steady state with random walkers can still be fairly noisy.

5.5. Diffusion coefficient gradient. Simulate the equation

$$\frac{\partial f}{\partial t} = -d\frac{\partial f}{\partial x} + D\frac{\partial}{\partial x}\left(\left[\frac{1}{4} - \left(\frac{x - (x_R + x_L)/2}{x_R - x_L}\right)^2\right]\frac{\partial f}{\partial x}\right) \tag{144}$$

Why is the motion of the pulse retarded at the right boundary? Play around with d and D, to see for which values the particles passes the boundary. *Hint:* Use the increments calculated in exercise 5.1.

5.7 Further readings

- **Numerical Solution of Stochastic Differential Equations**
  Kloeden [24].
- **The Black-Scholes equation with an Applet**
  Carlsson[18] [10]
- **Particle Methods**
  Birdsall[19] [25] and Hockney [26]
- **Example of Monte Carlo integration**
  Ising model[20]
- **Monte Carlo Methods**
  Monte Carlo Methods[21] in WWW virtual library[22], Taygeta Scientific Incorporated[23] including C++ classes

[18] http://fedu52.fed.ornl.gov/%7Ecarlsson/MonteCarlo
[19] http://ptsg.eecs.berkeley.edu/
[20] http://www2.truman.edu/%7Evelasco/ising.html
[21] http://random.mat.sbg.ac.at/others/
[22] http://vlib.org/Overview.html
[23] http://www.taygeta.com/stochastics.html

6 LAGRANGIAN METHODS

6.1 Introduction

Rather than solving the convective derivative $\frac{d}{dt} = \frac{\partial}{\partial t} + u\frac{\partial}{\partial x}$ in **Eulerian coordinates**, for example by using a Taylor expansion with the Lax-Wendroff method (sect.2.2), the idea behind **Lagrangian schemes** is first to **split the evolution** into a sequence of alternating advection and non-advection phases

$$\frac{df}{dt} = G(f) \quad \Longrightarrow \quad \begin{cases} \dfrac{df}{dt} = 0 & \text{(advection)} \\ \dfrac{\partial f}{\partial x} = G(f) & \text{(all the rest)} \end{cases} \tag{145}$$

The advection is then evolved independently by **propagating the solution** along the characteristics (sect.1.2) in a suitable and if possible explicit manner with no restriction on the time step, keeping a simple method such as explicit finite differences for the non-advection phase.

6.2 Cubic-Interpolated Propagation (CIP)

Introduced less than a decade ago by Yabe and Aoki [27], a whole family of schemes have been proposed along the same lines, relying on different **interpolatations** to propagate the solution along the **characteristics** (sect.1.2).

Using a cubic-Hermite polynomial, the discretized function and its derivatives $\{x_j, f_j, f'_j\}$ can be approximated in a continuous manner with

$$\begin{aligned} F_j(x) &= \left[(a_j X - b_j)X + \Delta x f'_j\right] X + f_j \; ; & a_j &= \Delta x(f'_j + f'_{j+1}) - 2(f_{j+1} - f_j) \\ X &= \frac{(x - x_j)}{\Delta x} \; ; \; \Delta x = x_j - x_{j-1} \; ; & b_j &= \Delta x(f'_j + 2f'_{j+1}) - 3(f_{j+1} - f_j) \end{aligned} \tag{146}$$

Both satisfy the master evolution equation and its derivative

$$\begin{cases} \dfrac{df}{dt} \equiv \dfrac{\partial f}{\partial t} + \dfrac{\partial}{\partial x}(uf) = g \\ \dfrac{df'}{dt} \equiv \dfrac{\partial f'}{\partial t} + \left(\dfrac{\partial u}{\partial x} f' + u\dfrac{\partial f'}{\partial x}\right) = \dfrac{\partial g}{\partial x} \end{cases} \tag{147}$$

which is split into an advection and non-advection phase

$$\begin{cases} \dfrac{df}{dt} = 0 \\ \dfrac{df'}{dt} = 0 \end{cases} \quad \longleftrightarrow \quad \begin{cases} \dfrac{\partial f}{\partial t} = g \\ \dfrac{\partial f'}{\partial t} = \dfrac{\partial g}{\partial x} - \dfrac{\partial u}{\partial x} f' \end{cases} \tag{148}$$

For the advection phase, the solution is integrated analytically simply by shifting the cubic polynomials $F_j(x,t) = F_j(x - u\Delta t, t - \Delta t)$ along the characteristics

$$\begin{cases} f_{j+1}^{t+\Delta t} = F_{j+1}(x_{j+1} - u\Delta t, t - \Delta t) = f_{j+1}^{t} - \beta\left[\Delta x f_{j+1}'^{t} - \beta(b_{j+1} - \beta a_{j+1})\right] \\ f_{j+1}'^{t+\Delta t} = \frac{d}{dx}F_{j+1}(x_{j+1} - u\Delta t, t - \Delta t) = f_{j+1}'^{t} - \frac{\beta}{\Delta x}(2b_{j+1} - 3\beta a_{j+1}) \end{cases} \tag{149}$$

where $\beta = u\Delta t/\Delta x$ is the Courant-Friedrich-Lewy (CFL) number.

Although this is not at all mandatory (exercise 6.4), the scheme implemented in JBONE assumes for simplicity that $\beta \in [0;1]$ so that the quantities $(f_{j+1}^{t+\Delta t}, f_{j+1}'^{t+\Delta t})$ can be interpolated from only the polynomial F_{j+1} continuously defined in the interval $[x_j; x_{j+1}]$. After an initialization where the function is discretized with cubic-Hermite polynomials by sampling on a grid and the derivative calculated with centered finite differences, the CIP scheme reads

```
double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are only constant
double beta =timeStep*velocity/(dx[0]);        //  if the problem and the
int n=f.length-1;                              //  mesh are homogeneous

for (int i=0; i<n; i++) {
  a=dx[0]*(df[i]+  df[i+1])-2*(f[i+1]-f[i]);
  b=dx[0]*(df[i]+2*df[i+1])-3*(f[i+1]-f[i]);
  fp[i+1]=  f[i+1] -beta*(dx[0]*df[i+1]-beta*(b-beta*a));
 dfp[i+1]= df[i+1] -beta/dx[0]*(2*b-3*beta*a);
}
a=dx[0]*(df[n]+  df[0])-2*(f[0]-f[n]);
b=dx[0]*(df[n]+2*df[0])-3*(f[0]-f[n]);
fp[0]=  f[0] -beta*(dx[0]*df[0]-beta*(b-beta*a));
dfp[0]= df[0] -beta/dx[0]*(2*b-3*beta*a);
```

The on-line document illustrates the high quality of this approach with the advection of the box function previously tested with the other methods.

Change the initial condition to `Cosine` and vary the wavelength with the parameter `IC1WID` down to 4 and 2 mesh points per wavelength to verify how small the numerical diffusion and dispersion are in comparison with other schemes.

6.3 Non-Linear equations with CIP

The same approach is applicable more generally for non-linear and vector equations

$$\frac{\partial \vec{f}}{\partial t} + \frac{\partial}{\partial t}(u\vec{f}) = \vec{g} \tag{150}$$

where $u = u(\vec{f})$ and $\vec{g} = \vec{g}(\vec{f})$. The problem is again decomposed in alternating phases without / with advection describing the evolution of the function

$$\begin{cases} \dfrac{\partial \vec{f}}{\partial t} = \vec{g} - \vec{f}\dfrac{\partial u}{\partial x} = \vec{G} & \text{(non-advection with compression term)} \\ \dfrac{\partial \vec{f}}{\partial t} + u\dfrac{\partial \vec{f}}{\partial x} = 0 & \text{(advection)} \end{cases} \tag{151}$$

and by differentiation of (eq.150), the evolution of the derivative

$$\begin{cases} \dfrac{\partial \vec{f'}}{\partial t} = \vec{g'} - u\dfrac{\partial \vec{f'}}{\partial x} = \vec{G'} - \vec{f'}\dfrac{\partial u}{\partial x} & \text{(non-advection)} \\ \dfrac{\partial \vec{f'}}{\partial t} + u\dfrac{\partial \vec{f'}}{\partial x} = 0 & \text{(advection)} \end{cases} \tag{152}$$

Starting with the non-advection phase, the discretized function is first evolved according to

$$\vec{f}_j^{\,*} = \vec{f}_j^{\,t} + \vec{G}_j \Delta t \tag{153}$$

where the super-script (*) refers to the intermediate step between the non- and advection phases. To avoid having to calculate $\vec{G'_j}$, the equation for the derivative is computed with

$$\frac{\vec{f'_j}^{\,*} - \vec{f'_j}^{\,t}}{\Delta t} \left[= \frac{\vec{G}_{j+1} - G_{j-1}}{2\Delta x} - \vec{f'_j}^{\,t}\frac{u_{j+1} - u_{j-1}}{2\Delta x} \right] = \frac{\vec{f'}^{\,*}_{j+1} - \vec{f'}^{\,*}_{j-1} - \vec{f'}^{\,t}_{j+1} - \vec{f'}^{\,t}_{j-1}}{2\Delta x \Delta t} - \vec{f'_j}^{\,t}\frac{u_{j+1} - u_{j-1}}{2\Delta x} \tag{154}$$

The advection phase is evolved in the same manner as before (eq.149), by shifting the cubic-Hermite polynomials along the characteristics (exercise 6.4).

6.4 Exercises

6.1. Arbitrary CFL number. Modify the CIP scheme in the JBONE applet to allow for arbitrarily large time-steps and negative advection velocities.

6.2. Diffusion in CIP. Use your favourite method to implement the diffusion phase with CIP in the JBONE applet. Discuss the merits of this combined solution.

6.3. Lagrangian method with splines. Analyze and discuss the Lagrangian scheme proposed by Guillaume Jost (EPFL, Lausanne) using a cubic-spline interpolation of the function.

6.4. Non-linear equation. Use the formalism in sect.6.3 to solve the general non-linear problem (eq.102). Study each of the wave-breaking, diffusion and dispersion terms separatly and compare with the solutions obtained previously with another method.

6.5 Further Reading

- **Cubic-Interpolated Propagation (CIP).**
  Yabe and Aoki [27], [28], [29]

7 WAVELETS

7.1 Remain a matter of research

There is currently a growing interest in using wavelets not only for the discretization of functions (sect.1.4), but also to approximate differential and integral operators. Motivations for that are the potential gain of solving global problems with the same $\mathcal{O}(N)$ operations as there are unknowns, relying on recent advances in iterative methods (sect.3.4) to solve linear systems in sparse format. Having not had the possibility so far to implement wavelets into the JBONE applet and extract the essence of research papers in a pedagogical manner, this section is limited to a number of links to recent papers maintained on a web site from MathSoft[24]:

1. D. M. Bond and S. A. Vavasis, Fast Wavelet Transforms for Matrices Arising From Boundary Element Methods.[25]

2. T. Chan, W. Tang and W. Wan, Wavelet sparse approximate inverse preconditioners[26]

3. P. Charton and V. Perrier, Factorisation sur Bases d'Ondelettes du Noyeau de la Chaleur et Algorithmes Matriciels Rapides Associes[27]

4. P. Charton and V. Perrier, Towards a Wavelet Based Numerical Scheme for the Two-Dimensional Navier-Stokes Equations.[28]

5. P. Charton and V. Perrier, A Pseudo-Wavelet Scheme for the Two-Dimensional Navier-Stokes Equations.[29]

6. S. Dahlke and A. Kunoth, Biorthogonal Wavelets and Multigrid.[30]

7. S. Dahlke and I. Weinreich, Wavelet-Galerkin Methods: An Adapted Biorthogonal Wavelet Basis.[31]

8. S. Dahlke and I. Weinreich, Wavelet Bases Adapted to Pseudo-Differential Operators.[32]

9. W. Dahmen and A. Kunoth, Multilevel Preconditioning.[33]

10. R. Glowinski, T. Pan , R. O. Wells, Jr. and X. Zhou, Wavelet and Finite Element Solutions for the Neumann Problem Using Fictitious Domains[34]

11. R. Glowinski, A. Rieder, R. O. Wells, Jr. and X. Zhou, A Wavelet Multigrid Preconditioner for Dirichlet Boundary Value Problems in General Domains.[35]

12. R. Glowinski, A. Rieder, R. O. Wells, Jr. and X. Zhou, A Preconditioned CG-Method for Wavelet-Galerkin Discretizations of Elliptic Problems[36]

13. F. Heurtaux, F. Planchon and M. V. Wickerhauser, Scale Decomposition in Burgers' Equation[37]

---

[24] http://www.mathsoft.com/wavelets.html
[25] ftp://ftp.tc.cornell.edu/pub/tech.reports/tr174.ps
[26] ftp://ftp.math.ucla.edu/pub/camreport/cam96-33.ps.gz
[27] ftp://ftp.lmd.ens.fr/MFGA/pub/wavelets/produits2d.ps.Z
[28] ftp://ftp.lmd.ens.fr/MFGA/pub/wavelets/iciam95.ps.Z
[29] ftp://ftp.lmd.ens.fr/MFGA/pub/wavelets/ns.ps.Z
[30] ftp://ftp.igpm.rwth-aachen.de/pub/dahlke/dksh.ps.Z
[31] ftp://ftp.igpm.rwth-aachen.de/pub/ilona/wega.ps.Z
[32] ftp://ftp.igpm.rwth-aachen.de/pub/ilona/wega2.ps.Z
[33] ftp://ftp.igpm.rwth-aachen.de/pub/dahmen/mulpre.ps.gz
[34] ftp://cml.rice.edu/pub/reports/9201.ps.Z
[35] ftp://cml.rice.edu/pub/reports/9306.ps.Z
[36] ftp://cml.rice.edu/pub/reports/9414.ps.Z
[37] http://wuarchive.wustl.edu/doc/techreports/wustl.edu/math/papers/burgers.ps.Z

14. A. Jiang, Fast wavelet based methods for certain time dependent problems[38]

15. A. Kunoth, Multilevel Preconditioning – Appending Boundary Conditions by Lagrange Multipliers.[39]

16. J. Lewalle, Wavelet Transforms of some Equations of Fluid Mechanics[40]

17. J. Lewalle, Energy Dissipation in the Wavelet-Transformed Navier-Stokes Equations[41]

18. J. Lewalle, On the effect of boundary conditions on the multifractal statistics of incompressible turbulence[42]

19. J. Lewalle, Diffusion is Hamiltonian[43]

20. D. Lu, T. Ohyoshi and L. Zhu, Treatment of Boundary Conditions in the Application of Wavelet-Galerkin Method to a SH Wave Problem[44]

21. A. Rieder and X. Zhou, On the Robustness of the Damped V-Cycle of the Wavelet Frequency Decompositions Multigrid Method[45]

22. A. Rieder, R. O. Wells, Jr. and X. Zhou, A Wavelet Approach to Robust Multilevel Solvers for Anisotropic Elliptic Problems.[46]

23. A. Rieder, R. O. Wells, Jr. and X. Zhou, On the Wavelet Frequency Decomposition Method[47]

24. O. V. Vasilyev and S. Paolucci, A Dynamically Adaptive Multilevel Wavelet Collocation Method for Solving Partial Differential Equations in a Finite Domain.[48]

25. O. V. Vasilyev, S. Paolucci and M. Sen, A Multilevel Wavelet Collocation Method for Solving Partial Differential Equations in a Finite Domain.[49]

26. R. O. Wells, Jr. and X. Zhou, Wavelet Solutions for the Dirichlet Problem[50]

27. R. O. Wells, Jr. and X. Zhou, Wavelet Interpolation and Approximate Solution of Elliptic Partial Differential Equations[51]

28. R. O. Wells, Jr. and X. Zhou, Representing the Geometry of Domains by Wavelets with Applications to Partial Differential Equations[52]

29. R. O. Wells, Jr., Multiscale Applications of Wavelets to Solutions f Partial Differential Equations[53]

---

[38] ftp://ftp.math.ucla.edu/pub/camreport/cam96-20.ps.gz
[39] ftp://ftp.igpm.rwth-aachen.de/pub/kunoth/cosh.ps.Z
[40] http://www.mame.syr.edu/faculty/lewalle/acta-94.html
[41] http://www.mame.syr.edu/faculty/lewalle/dissip-93.html
[42] http://www.mame.syr.edu/faculty/lewalle/camb-93.html
[43] http://www.mame.syr.edu/faculty/lewalle/hamdiff.html
[44] ftp://ftp.mathsoft.com/pub/wavelets/bc.ps.gz
[45] ftp://cml.rice.edu/pub/reports/9310.ps.Z
[46] ftp://cml.rice.edu/pub/reports/9307.ps.Z
[47] ftp://cml.rice.edu/pub/reports/9413.ps.Z
[48] http://landau.mae.missouri.edu/%7evasilyev/Publications/adaptive.ps.gz
[49] http://landau.mae.missouri.edu/%7evasilyev/Publications/WML.ps.gz
[50] ftp://cml.rice.edu/pub/reports/9202.ps.Z
[51] ftp://cml.rice.edu/pub/reports/9203.ps.Z
[52] ftp://cml.rice.edu/pub/reports/9214.ps.Z
[53] ftp://cml.rice.edu/pub/reports/9409.ps.Z

8 THE JBONE APPLET

8.1 User Manual

This section is really meant for the HTML version of the lecture notes, providing the necessary links to all the documents.
The *Java Bed for ONE* dimensional evolution equations JBONE provides a flexible environement to easily test numerical schemes.

Source code. Written in JAVA[54], the core of the program listing can be viewed[55] and the source files obtained by sending an e-mail request to <jaun@fusion.kth.se> under the conditions specified in the program header. If you consult this document on-line, the previous link shows you how *markers* have been used to target specific section in the code.

Installation. To run the examples and change the parameters **you don't have to install anything** except maybe update to a sufficiently recent version of your web-browser (Netscape 4.04 or higher[56]). To carry out the exercises and modify the code on a UNIX platform with a pre-installed java compiler, go through the installation procedure

```
mkdir ~/public_html              # Create a public directory

mv jbone.tar.Z ~                 # You DON'T want the source in your
cd                               # public_html but in your main directory
uncompress jbone.tar.Z
tar xvf jbone.tar
rm jbone.tar

cd ~/jbone*
make all                         # Compile, make documentation, listing
java jbone                       # Execute a compiled java program

                                     # On the fusion.kth.se workstations,
~andrej/bin/publish *.class *.html   #   publish your applet and docs
                                     #   on the web.
```

Please note that all files in the `pde99-??` accounts are world readible. If this is not acceptable for you, change the `umask` value in your `.cshrc`.

Documentation. Is generated automatically from the source by typing

```
make docs
```

and can be consulted on-line for the class tree and variable index.

[54] http://java.sun.com/docs/books/tutorial/index.html
[55] ../jbone/listing.html
[56] http://home.netscape.com

Execute. On the UNIX platform, you may execute the code simply by typing

```
java jbone
```

To insert the applet in a web page, create a file `test.html` containing

```
<html>
  <head>
    <! -- @(#)jbone.html
    Andre JAUN (jaun@fusion.kth.se) and Johan HEDIN (johanh@fusion.kth.se)
    Alfven Laboratory, Royal Institute of Technology, SE-100 44 Stockholm.
    (C) July 1999, all rights reserved.
    This shareware can be obtained without fee by e-mail from the authors.
    Permission to use, copy, and modify the source and its documentation
    for your own personal use is hereby granted provided that this copyright
    notice appears in all copies.
    -->
    <title>JBONE</title>
  </head>
  <body>
    <h1>Here is my own example</h1>
    Illustrates the advection of a sinusoidal perturbation etc.
    <applet codebase="./" code=jbone.class
            align=center width=900 height=280>
       <param name=PDE     value=0>    <param name=IC      value=2>
       <param name=METH    value=0>    <param name=SCHEM   value=6>
       <param name=TIMSTP  value=0.5>  <param name=VELCTY  value=1.>
       <param name=DIFFUS  value=0.>   <param name=DISPER  value=0.5>
       <param name=THETA   value=0.7>  <param name=TUNINT  value=0.333>
       <param name=BOXLEN  value=64.>  <param name=RUNTIM  value=128.>
       <param name=NPTS    value=64>
    </applet>
  </body>
</html>
```

After saving it in the directory `~/public_html/jbone` you can view the result in your web browser from the address

```
http://www.fusion.kth.se/~pde99-??/jbone/test.html
```

with a result that should be similar to what you get from this link. Because of an annoying bug, you may have to **click in the white area, press shift and select reload** the page to start the execution properly.

Modify. After every modification of the source `*.java`, the code has to be recompiled with the command `make jbone`. You must again press `Shift` and click `reload` to force Netscape to load the newly compiled code.

8.2 Object-oriented programming: Monte Carlo in JBONE

The Monte Carlo, MC, solver in JBONE is different from the FD and FEM solver in the sense that the solution is represented by the set of particles and the function f is just used for diagnostics. As found in the class tree, the solvers could be divided into particle methods and fluid methods. The discretization part is contained in the class `ParticleSolution` and the Monte Carlo time stepping and boundary conditions in the class `MCMSolution`. The class `ParticleSolution` contains a vector of the particles. As `f[]` is only defined as a projection onto the roof-top base, it now becomes clear why e.g. the method `limits()` is declared `abstract` in `Solution`. In `FluidSolution` `limits()` could be given directly from the solution `f[]`, whereas in `ParticleSolution` the method `generateDistribution()` needs to be called prior to finding min and max of `f[]`.

The first order moment will be perfectly conserved, since no particles are lost. Also note that pure advection will be exactly solved within the Applet.

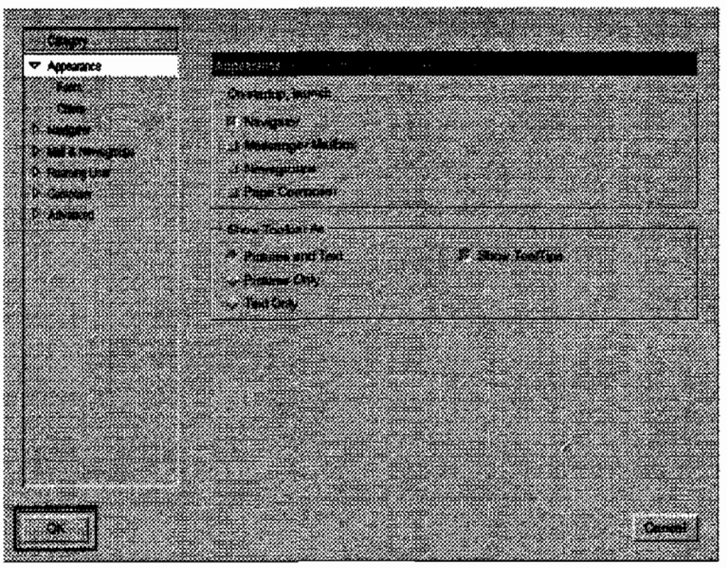


Figure 15: The preferences window in Netscape.

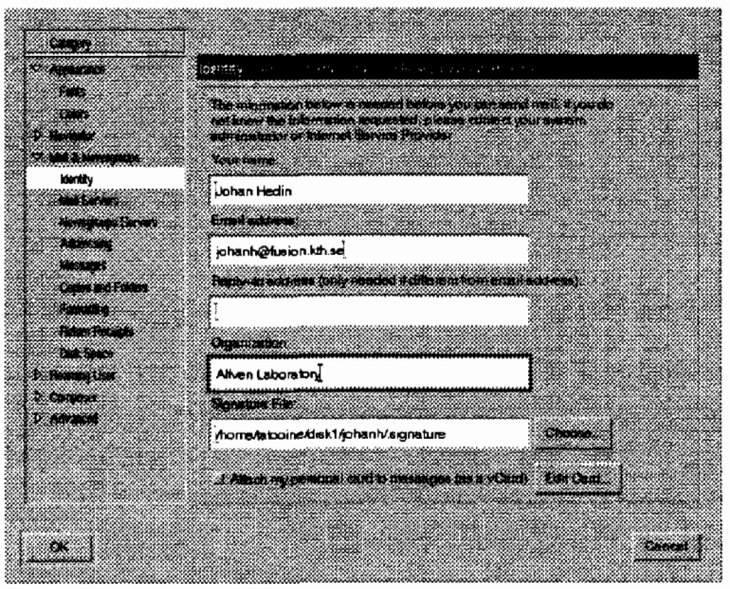


Figure 16: The identity sheet of the preferences window in Netscape.

9 THE WORLD OF INTERNET

9.1 Newsgroups

The following describes how to install the news groups and mail on Netscape 4.5 under Solaris. For other configurations, refer to vendor supplied information. Of course, you are free to use a mail and news program of your choice.

- Open the preferences window by clicking on the menu `Edit->Preferences...` (see figure 15).
- Open the `Mail & Newsgroups` folder by clicking on the arrow in front of the name.
- Open the `Identity` sheet by clicking it (figure 16). Fill in your name and email address. Every `pde99-??` account can be used with an e-mail address `pde99-??@fusion.kth.se` in case students are not able to access the usual mail account.
- Press the `Mail Server` sheet. Look at the `Incoming Mail Servers` window. If you are a local user, e.g. are an user on the Sun system (`pde99-??` accounts included), press `pop` and then `Delete`. Then press `Add...` and set the server type to `MoveMail` (figure 17). For

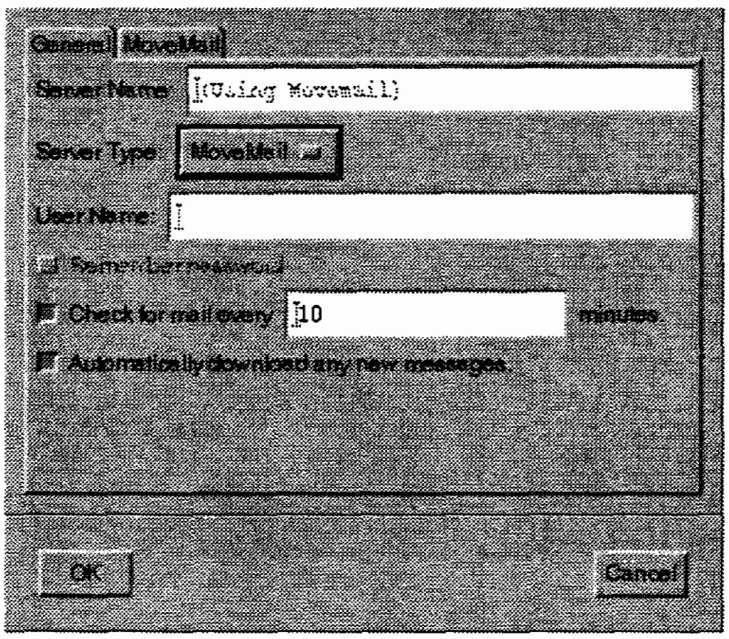


Figure 17: For local users, set the server type to `MoveMail`.

non local users set the server type to pop and the `Server Name` to pop server of your mail provider. For accounts on the local Sun system, that machine is `mail.fusion.kth.se`.

- Close the preferences window.
- Open the news reader by selecting the menu `Communicator->Messenger`.
- Select the menu `File->Subscribe...`. Click the `Add server...` button and enter the server `news.fusion.kth.se`. Expand all groups on this server by clicking on the small + signs. For each group under `alfvenlab.pde-course`, select the group by clicking it and then the `Subscribe` button. You should now have blue check marks after the three groups listed in sections 9.1.1–9.1.3 (figure 18).

9.1.1 `alfvenlab.pde-course.announce`

`alfvenlab.pde-course.announce`[57] is used for announcement during the course. Postings to this group is restricted.

9.1.2 `alfvenlab.pde-course.help`

Use `alfvenlab.pde-course.help`[58] to ask for help. Also help each other through this group.

9.1.3 `alfvenlab.pde-course.test`

Use `alfvenlab.pde-course.test`[59] to learn the news system. Messages in this group are deleted after one day.

9.2 E-publishing

In section 8.2, it is explained how to create a directory for the home page. For information about how to write HTML, refer to the world wide web consortium at `http://www.w3.org`.

There is a small script on the system for creating Post Script files to GIFs.

```
~andrej/bin/togif foo.ps bar.ps
```

[57] news://news.fusion.kth.se/alfvenlab.pde-course.announce
[58] news://news.fusion.kth.se/alfvenlab.pde-course.help
[59] news://news.fusion.kth.se/alfvenlab.pde-course.test

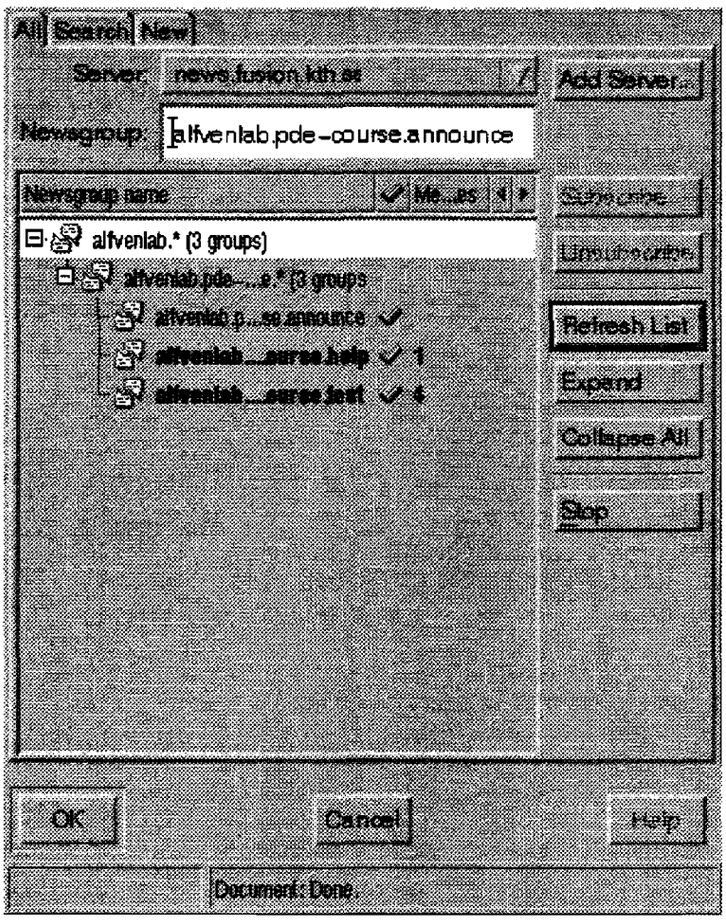


Figure 18: Subscribe to the three groups in `alfvenlab.pde-course`.

will convert the Post Script files `foo.ps` and `bar.ps` to the GIF files `foo.gif` and `bar.gif`. The students mastering LaTeXwill find the utility `latex2html` very useful.

9.3 Software and hardware used in developing this course

The course material is typeset in LaTeX 2_ε, using the extensions available in `latex2html`[60]. The Java applet is developed with `jdk-1.1.x` for Solaris[61] and Linux[62]. The plots are made in Matlab[63]. For every project involving several authors, CVS[64] is an invaluable tool. The entire course is developed on Solaris[65] and Linux[66].

[60] http://cdc-server.cdc.informatik.tu-darmstadt.de/%7Elatex2html
[61] http://java.sun.com
[62] http://www.blackdown.org/java-linux.html
[63] http://www.mathworks.com
[64] http://www.cyclic.com
[65] http://www.sun.com
[66] http://www.linux.org

10 COURSE EVALUATION AND PROJECTS

10.1 Interactive evaluation form

Your anonymous opinion is precious! Please fill-in the anonymous form available in the web edition.

10.2 Suggestions for one-week projects

The best ideas for a small one-week project stems directly from your own field ! For those however who want some suggestions, here is a list of projects more or less in rising order of difficulty.

Inhomogeneous mesh with FEMs. Add the capability of refining the mesh in JBONE's finite elements method. Integrate analytically a sum of Lorentzian functions $L_{[x_j,w_j]}(x) = w_j \left[w_j^2 + (x-x_j)^2\right]^{-2}$ defining the locations x_j and the weights w_j of the regions you want to refine and derive the expression for an inhomogeneous distribution of mesh points

$$X_{packed}(x) = px + (1-p)\frac{\sum_{j=1}^{N} \arctan\left(\frac{x-x_j}{w_j}\right) + \arctan\left(\frac{x_j}{w_j}\right)}{\sum_{j=1}^{N} \arctan\left(\frac{1-x_j}{w_j}\right) + \arctan\left(\frac{x_j}{w_j}\right)} \tag{155}$$

Letting the advection $u(x)$ and diffusion coefficients $D(x)$ to vary in space, calculate the matrix elements corresponding to (Eq.73) using a 2-points Gaussian quadrature and study the numerical convergence of an advection-diffusion problem in a strongly inhomogeneous medium.

Iterative solvers for diffusion. Add a modular solver in JBONE including Jacobi, Gauss-Seidel and SOR iterations. Start by defining the matrix-vector product in sparse format and implement the iterative schemes proposed in sect.3.4. Compare the efficiency of iterative methods with a direct solver for a diffusion dominated problem using an increasing amount of advection.

2D Monte Carlo. Add a second dimension to the Monte Carlo solver. Initialize all the particles to $v_i = v_0$ and solve the equation

$$\frac{\partial f(x,v,t)}{\partial t} + v\frac{\partial f(x,v,t)}{\partial x} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial f(x,v,t)}{\partial x}\right) \tag{156}$$

Change the boundary conditions from periodic to bouncing. A possible extension of this project is to include a moving wall.

Wave equation as a driven problem. Use two different methods to solve the equation describing forced oscillation in a weakly absorbing bounded medium

$$\frac{\partial^2 f}{\partial t^2} - c^2\frac{\partial^2 f}{\partial x^2} = S_{\omega_0}(x,t) - 2\nu\frac{\partial f}{\partial t} \tag{157}$$

Choose a source $S_{\omega_0}(x,t) = S_0 \exp(x^2/\Delta^2)\sin(\omega_0 t)$ smoothly distributed inside the cavity $x \in [-L/2; L/2]$ and a sink $\nu \ll \omega_0 \simeq 2\pi c/L$ which is sufficiently small to allow the perturbations to propagate and reflect. Vary the driving frequency ω_0 and study the possibility of exciting resonances inside the cavity.

Particle weight. Apply individual weights w_i to the particles in the Monte Carlo solver. Try to reduce the amount of noise at low levels in a steady state solution, e.g. exercise 5.4, by splitting particles with $w_i > w_{\text{limit}}(x)$. Try to find a good maximum weight function $w_{\text{limit}}(x)$. After a while the particles will be to many. Solve this by re-discretize the distribution function by

```
discretize(new ShapeNumerical(this));
```

The particle weight should be used to increase the accuracy in discretization. This project involves changing the projection, the discretization and learning the `Vector` class in Java.

Option pricing. Combine exercise 2.4 and 3.5 dealing with finite difference and finite element schemes for the pricing of European and American options. Compare with the solution obtained using the Monte-Carlo method under this[67] link.

Bose-Einstein condensation. Start with the linear Schrödinger equation

$$i\frac{\partial\psi}{\partial t} = \left[-\frac{\partial^2}{\partial x^2} + V(x)\right]\psi \tag{158}$$

normalized with Plank's constant $\hbar = 1$ and a particle mass $m = 1/2$ and use two different schemes to calculate the scattering of a wavepacket by a simple one-dimensional potential $V(x)$. Having validated your schemes with analytical results, solve the non-linear equation describing the Bose-Einstein condensation in a parabolic trapping potential $V(r) = \frac{\alpha}{2}r^2$ with cylindrical symmetry

$$i\frac{\partial\psi}{\partial t} = \left[-\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) + V(r) + 2\pi a \mid \psi \mid^2\right]\psi \tag{159}$$

where the parameter a defines the scattering length. Project in collaboration with Federica Cattani (Nonlinear Electrodynamics, CTH, Gothenburg).

Slowing down of beams. The slowing down of a beam in a collisional media is given by

$$\frac{\partial f(x,v,t)}{\partial t} + v\frac{\partial f(x,v,t)}{\partial x} + \frac{\partial}{\partial v}[-v_p v f(x,v,t)] = \frac{\partial}{\partial v}\left(D_p\frac{\partial f(x,v,t)}{\partial v}\right) \tag{160}$$

Use Monte Carlo for solving the beam distribution function. Identify the drift and diffusion coefficients in x and v. Start the pulse to the left with $v = v_0$. Modify v_p and D_p in order to get a nice slowing down of the pulse.

Iterative BiCGSTAB solver for wave-equations.

KdV with wavelets.

[67] http://fedu52.fed.ornl.gov/%7ecarlsson/MonteCarlo/cookbook/notes.html

References

[1] W. Sweldens. The Wavelet Home Page[68].

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipies*[69]. Cambridge University Press, second edition, 1992.

[3] G. Dahlquist and Å Bjorck. *Numerical Methods.* Prentice-Hall, 1974.

[4] C. A. Fletcher. *Computational Techniques for Fluid Dynamics.* Springer, second edition, 1991.

[5] M. Abramowitz and A. Stegun. *Handbook of Mathematical Functions.* Dover Publications, New York, tenth edition, 1972.

[6] C. Johnson. *Numerical Solution of PDEs by the FEM Method.* Studentlitteratur – Lund, Sweden, 1987.

[7] K. Appert. Experimentation numérique. unpublished lecture notes, CRPP-EPFL, CH-1015 Lausanne, Switzerland, 1988.

[8] National Institute of Standards and Technology (NIST). Guide to Available Mathematical Software[70].

[9] J. Eastwood. Computer Physics Communications Library[71].

[10] J. Carlsson. The Monte-Carlo method – a cookbook approach[72]. Alfvén Laboratory, KTH, SE-100 44 Stockholm, Sweden, 1997.

[11] J. Boris. *Methods in Computational Physics, Vol.16.* Academic, 1976.

[12] J. M. Jin. *The Finite Element Method in Electromagnetics.* John Wiley, 1993.

[13] D. Katz and T. Cwik. EMLIB homepage[73].

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1996.

[15] P. Wilmott, J. Dewynne, and S. Howison. *Option Pricing.* Oxford Financial Press, 1993.

[16] T. Bjork. Examples in optimization theory. unpublished lecture notes, Inst. for Optimization and System Theory, KTH, SE-100 44 Stockholm, Sweden, 1995.

[17] D. Duffie. *Dynamic Asset Pricing Theory.* Princeton University Press, 1992.

[18] A. Bondeson and G. Y. Fu. Tunable integration scheme. *Computer Physics Communications*, 66:167, 1991.

[19] S. Betts, S. Browne, J. Dongarra, E. Grosse, P McMahan, and T. Rowan. Netlib Software Repository[74].

[20] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc home page. The PETSc home page[75], 1999.

[21] K. Blomqvist. Iterative solution of maxwell's equations. Master's thesis, Luleå University of Technology, 1999.

[68] http://www.wavelet.org/wavelet/index.html
[69] http://www.ulib.org/webRoot/Books/Numerical_Recipes/bookcpdf.html
[70] http://gams.nist.gov
[71] http://www.cpc.cs.qub.ac.uk/cpc/
[72] http://fedu52.fed.ornl.gov/%7ecarlsson/MonteCarlo/cookbook/notes.html
[73] http://http://emlib.jpl.nasa.gov/
[74] http://www.netlib.org
[75] http://www.mcs.anl.gov/petsc

[22] A. Jaun, K. Blomqvist, A. Bondeson, and T. Rylander. Iterative solution of maxwell's equations with finite elements. *Computer Physics Communications*, 1999. submitted.

[23] N. J. Zabusky and M. D. Kruskal. Interaction of solitons in a collisionless plasma. *Physical Review Letters*, 15:240, 1965.

[24] P. E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations.* Springer-Verlag, second corrected printing edition, 1995.

[25] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation.* McGraw-Hill, 1985. The Plasma Theory and Simulation Group Homepage[76].

[26] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles.* Adam Hilger, 1988.

[27] T. Yabe and T. Aoki. A universal solver for hyberbolic equations by cip. *Computer Physics Communications*, 66:219, 1991.

[28] H. Takewaki and T. Yabe. The cubic-interpolated pseudo particle (cip) method. *Computer Physics Communications*, 70:355, 1987.

[29] T. Utsumi, T. Kunugi, and T. Aoki. Stability and accuracy of the cip scheme. *Computer Physics Communications*, 101:9, 1997.

[76] http://ptsg.eecs.berkeley.edu/

Appendix: the JBONE source program

A copy of the complete source program can be obtained free of charge for personnal use by sending an e-mail request to <jaun@fusion.kth.se>.

Index with variable names

See *http://www.fusion.kth.se/~jaun/Teach/Num/jbone/AllNames.html*

Class Hierarchy

From *http://www.fusion.kth.se/~jaun/Teach/Num/jbone/tree.html*

- class java.lang.Object
  - class BandMatrix
  - class Complex (implements java.lang.Cloneable, java.io.Serializable)
  - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - class java.awt.Container
      - class java.awt.Panel
        - class java.applet.Applet
          - class jbone (implements java.lang.Runnable)
      - class java.awt.Window
        - class java.awt.Frame (implements java.awt.MenuContainer)
          - class MainFrame
  - class Data
  - class FFT
  - class Mesh
  - class ShapeFunction
    - class ShapeBox
    - class ShapeCosinus
    - class ShapeGaussian
    - class ShapeNumerical
    - class ShapeSoliton
  - class Solution
    - class FluidSolution
      - class CHASolution
      - class FDSolution
      - class FEMSolution
      - class FFTSolution
    - class ParticleSolution
      - class MCMSolution

Program listing

See *http://www.fusion.kth.se/~jaun/Teach/Num/jbone/listing.html*

It is important to note that the object oriented character of the JAVA language makes it **unnessessary to read** the program listing (**browse the class hierarchy** instead). To carry out the exercises, modifications are strictly limeted to the **extensions of the class Solution.** The following pages begin with these extensions and are followed for completeness only by the rest of the listing.

```
/* $Id: Solution.java,v 1.14.4.3 1999/07/31 14:20:52 andrej Exp $ */

/*****************************************************************************
* Solution -- Is an abstract class containing all the solutions and solvers.
*   @see FluidSolution
*   @see ParticleSolution
*****************************************************************************/

abstract public class Solution{

  /** Equation name @see #setEquation */          protected String equation_;
  /** Numerical scheme name @see #setMethod */    protected String method_;
  /** Numerical scheme name @see #setScheme */    protected String scheme_;
  /** Absolute time @see #setTime*/               protected double time_;

  /** Function time level n-1 */                  protected double[]  fm;
  /** Function time level n   */                  protected double[]  f;
  /** Function time level n+1 */                  protected double[]  fp;
  /** Derivative time level n-1 @see #fm */       protected double[]  dfm;
  /** Derivative time level n   @see #f */        protected double[]  df;
  /** Derivative time level n+1 @see #fp */       protected double[]  dfp;

  /** Extra Function time level n-1 */            protected double[]  gm;
  /** Extra Function time level n   */            protected double[]  g;
  /** Extra Function time level n+1 */            protected double[]  gp;
  /** Extra Derivative time level n-1 @see #gm */ protected double[]  dgm;
  /** Extra Derivative time level n   @see #g */  protected double[]  dg;
  /** Extra Derivative time level n+1 @see #gp */ protected double[]  dgp;

  /** Complex Function time level n-1 */          protected Complex[] hm;
  /** Complex Function time level n   */          protected Complex[] h;
  /** Complex Function time level n+1 */          protected Complex[] hp;
  /** Derivative time level n-1 @see #hm */       protected Complex[] dhm;
  /** Derivative time level n   @see #h */        protected Complex[] dh;
  /** Derivative time level n+1 @see #hp */       protected Complex[] dhp;

  /** Fourier spectrum time level n-1 */          protected Complex[] sm;
  /** Fourier spectrum time level n */            protected Complex[] s;
  /** Fourier spectrum time level n+1 */          protected Complex[] sp;

  /** The underlaying mesh of the solution */     protected Mesh mesh_;
  /** The mesh points @see #mesh_ */              protected double[] x;
  /** The mesh intervals @see #mesh_ */           protected double[] dx;
  /** Store initial moments */                    protected double[]
                                                    initialMoments = {0.,0.,0.};


  /** Creates an instance of a Solution object with all the attributes.
      @param mesh The underlying mesh of the solution
  *****************************************************************************/
  public Solution(Mesh mesh) {
    int n = mesh.size();
    x  = mesh.points();  dx = mesh.intervals();
    fm = new double[n];  f  = new double[n];  fp  = new double[n];
    dfm= new double[n];  df = new double[n];  dfp = new double[n];
    gm = new double[n];  g  = new double[n];  gp  = new double[n];
    dgm= new double[n];  dg = new double[n];  dgp = new double[n];
    hm = new Complex[n]; h  = new Complex[n]; hp  = new Complex[n];
    dhm= new Complex[n]; dh = new Complex[n]; dhp = new Complex[n];
    sm = new Complex[n]; s  = new Complex[n]; sp  = new Complex[n];
    mesh_ = mesh;
  } // Solution

                                                                //TAG_moments_TAG//
  /** Internal function calculating the initial moments of f[] or their
      deviation from the beginning of the evolution.
      @param m The order of the moment
      @return The value or deviation of the m:th moment of f[]
      @see #momentsDeviation
  *****************************************************************************/
  protected double calculateMoments(int m){
    int n=f.length-1;
    double moment = 0.0;
    if (m==0) {
      for (int i=0; i<n; i++) {
        moment= moment+0.5*dx[i]*(f[i+1]+f[i
      }
      moment = moment +0.5*dx[n]*(f[0]+f[n
    } else { moment=0.0; }
    if (initialMoments[m] != 0.) {                              //Deviation
      return (moment-initialMoments[m])/initialMoments[m];
    } else {
      return moment;                                            //Initial value
    }
  } // calculateMoments

  /** Internal function calculating the limits of the solution. This
      function assumes that the distribution function is in f[].
      @return A vector consisting of {min(solution), max(solution)}
      @see #limits
  *****************************************************************************/
  protected double[] calculateLimits() {
    double[] lim = {f[0], f[0]};
    for (int i=1;i<f.length;i++) {
      if (f[i]<lim[0]) {lim[0]=f[i];};
      if (f[i]>lim[1]) {lim[1]=f[i];};
    } // for
    return lim;
  } // calculateLimits


  /** Number of values in the discretization
      @return The size of the underlaying mesh
  *****************************************************************************/
  public int size(){  return mesh_.size(); }


  /** Set the equation as a string.
      @param scheme The name of the equation
      @return True
      @see Data#ADVECTION
  *****************************************************************************/
  public boolean setEquation(String equation){
    equation_ = new String(equation);
    return true;
  } // setEquation


  /** Set the numerical method as a string.
      @param method The name of the numerical method
      @return True
  *****************************************************************************/
  public boolean setMethod(String method){
    method_ = new String(method);
    return true;
  } // setMethod


  /** Set the numerical scheme as a string.
      @param scheme The name of the numerical scheme
      @return True
  *****************************************************************************/
  public boolean setScheme(String scheme){
    scheme_ = new String(scheme);
    return true;
  } // setScheme


  /** Set the current physical time
      @param current time
      @return True
  *****************************************************************************/
  public boolean setTime(double time){
    time_ = time;
    return true;
  } // setTime


  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  */
  abstract public void discretize(ShapeFunction function);


  /** Calculates the deviation from the m:th initial moment.
      @param m The order of the moment
      @return The deviation of the m:th moment from the beginning
      @see discretize
  */
  abstract public double momentsDeviation(int m);


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if a scheme is implemented for that equation
  */
  abstract public boolean next(Data runData);


  /** Take the solution backward one step to initialize schemes
      with 3 time levels.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if an initialisation scheme is implemented for that e
  */
  abstract public boolean previous(Data runData);


  /** Calculates the solution boundaries.
      @return A vector with {min(solution), max(solution)}
  */
  abstract public double[] limits();

  /** Gives the value of the distribution function for an index
      @param index The index for which to get the value
      @return The value of the distribution function at a given index
  */
  abstract public double getValue(int index);

} // class Solution
```

```
/* $Id: FluidSolution.java,v 1.7 1999/06/13 22:11:35 andrej Exp $ */

/**************************************************************************
* FluidSolution -- Is an abstract class at the top of all the solvers that
*   that operate directly on the fluid function.
*   @see Solution
**************************************************************************/
abstract public class FluidSolution extends Solution{

  /** True if needs to shiftLevels */      protected boolean isForward = true;

  /** Creates a FluidSolution object.
      @param mesh The underlying mesh of the solution
  **************************************************************************/
  public FluidSolution(Mesh mesh){
    super(mesh);
  } // FluidSolution


  /** Calculates the deviation from the m:th initial moment.
      @param m The order of the moment
      @return The deviation of the m:th moment from the beginning
      @see discretize
  **************************************************************************/
  public double momentsDeviation(int m){
    return calculateMoments(m);
  } // moments


  /** Internal function for copying new -> old
  **************************************************************************/
  protected void shiftLevels() {
    for (int i=0;i<f.length;i++) {
      fm[i]=f[i]; dfm[i]=df[i]; f[i]=fp[i]; df[i]=dfp[i]; fp[i]=0.; dfp[i]=0.;
      gm[i]=g[i]; dgm[i]=dg[i]; g[i]=gp[i]; dg[i]=dgp[i]; gp[i]=0.; dgp[i]=0.;
      sm[i]=s[i];               s[i]=sp[i];               sp[i]=new Complex();
    }
  }


  /** Calculates the limits of the solution.
      @return A vector consisting of {min(solution), max(solution)}
  **************************************************************************/
  public double[] limits(){
    return calculateLimits();
  } // limits


  /** Gives the value of the fluid function for an index
      @param index The index for which to get the value
      @return The value of the fluid function at a given index
  **************************************************************************/
  public double getValue(int index){
    return f[index];
  } // getValue


  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  */
  abstract public void discretize(ShapeFunction function);

} // class FluidSolution
```

```
/* $Id: CHASolution.java,v 1.13.4.2 1999/08/05 11:48:12 andrej Exp $ */

/** The class CHASolution contains the Characteristics solver.
    @version $Revision: 1.13.4.2 $
*/
public class CHASolution extends FluidSolution{
  /** Creates a CHASolution object.
      @param mesh The mesh of the solution
   */
  public CHASolution(Mesh mesh){
    super(mesh);
  } // FDSolution

                                                          //TAG_CHAInit_TAG//
  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  ************************************************************************/
  public void discretize(ShapeFunction function){
    for (int j = 0; j < mesh_.size(); j++)       //Discretizing is very simple
      f[j] = function.getValue(mesh_, j);        //  with FEM and splines
    initialMoments[0]=calculateMoments(0);       //Set conserved quantities
    initialMoments[1]=calculateMoments(1);
  } // discretize


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if a scheme is implemented for that equation
  ************************************************************************/
  public boolean next(Data runData) {
    double timeStep = runData.getDouble(Data.TIMSTP);         //Parameters
    double velocity = runData.getDouble(Data.VELCTY);
    double diffusCo = runData.getDouble(Data.DIFFUS);
    double perLength= (double)mesh_.size()
                      *mesh_.interval(0);

double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are only constant if
double beta =timeStep*velocity/(dx[0]      );  //  the problem and mesh are
                                               //  homogeneous
int n=f.length-1;
boolean isDefined=false;

if(equation_.equals(Data.ADVECTION) &&
   scheme_.equals(Data.CUBFEM)){                        //TAG_CHAAdvCFEM_TAG//
  //=========================================================================
  // CHA - Advection/diffusion - CubicFEM - Yabe+Aoki CPC 66(1991)219
  //=========================================================================
  double a,b;
  for (int i=0; i<n; i++) {
    a=dx[0]*(df[i]+  df[i+1])-2*(f[i+1]-f[i]);
    b=dx[0]*(df[i]+2*df[i+1])-3*(f[i+1]-f[i]);
    fp[i+1]=  f[i+1] -beta*(dx[0]*df[i+1]-beta*(b-beta*a));
   dfp[i+1]= df[i+1] -beta/dx[0]*(2*b-3*beta*a);
  }
  a=dx[0]*(df[n]+  df[0])-2*(f[0]-f[n]);
  b=dx[0]*(df[n]+2*df[0])-3*(f[0]-f[n]);
  fp[0]=  f[0] -beta*(dx[0]*df[0]-beta*(b-beta*a));
  dfp[0]= df[0] -beta/dx[0]*(2*b-3*beta*a);
  isDefined = true;

} else if(equation_.equals(Data.ADVECTION) &&
          scheme_.equals(Data.CUBSPL)){              //TAG_CHAAdvCSPL_TAG//
  //=========================================================================
  //  CHA - Advection/diffusion - Cubic splines
  //=========================================================================
  BandMatrix a= new BandMatrix(3, f.length);
  double[]   c= new double[f.length];
  int        i, i1, i2;
  double     z, z1, z2, dm, dp;
  double     h= dx[0];

  for (i=0; i<=n; i++) {           // Matrix
    a.setL(i,   h/6 );
    a.setD(i, 4*h/6 );
    a.setR(i,   h/6 );
  }

  dp= (f[0]-f[n])/h;               //Right hand side 2nd derivative
  for (i=0; i<n; i++) {
    dm= dp; dp= (f[i+1]-f[i])/h;
    c[i]= dp-dm;
  }
  dm= dp; dp= (f[0]-f[n])/h;
  c[n]= dp-dm;

  dfp=a.solve3(c);                 //Solve linear problem

  for (i=0; i<=n; i++) {           //Propagate along characteristics
    z= x[i]-timeStep*velocity;     //Allow arbitrary time step
    if (z> perLength) { z= z-perLength*   (int)( z/perLength); }
    if (z< 0)         { z= z+perLength*(1+(int)(-z/perLength)); }
    i1= (int)(z/h); i2=i1+1;
    if (i1 == n)  { i2= 0; }
    z1 = (z-x[i1])/h;  z2 = 1-z1;  //Cubic spline interpolation
    fp[i] = z2*f[i1] +z1*f[i2] +(h*h)/6*((z2*z2*z2 -z2)*dfp[i1] +
                                         (z1*z1*z1 -z1)*dfp[i2] );
  }
  isDefined = true;

} else if(equation_.equals(Data.ADVECTION) &&
          scheme_.equals(Data.CHAMINE)){             //TAG_CHAAdvMine_TAG//
  //=========================================================================
  //  CHA - Implement your own scheme here
  //=========================================================================
  isDefined = true;

} else if(scheme_.equals(Data.CHAEX61)){
  //=========================================================================
  //  FEM - Exercise 6.1
  //=========================================================================
  isDefined = false;

} else if(scheme_.equals(Data.CHAEX62)){
  //=========================================================================
  //  FEM - Exercise 6.2
  //=========================================================================
  isDefined = false;

} else if(scheme_.equals(Data.CHAEX64)){
  //=========================================================================
  //  FEM - Exercise 6.4
  //=========================================================================
    isDefined = false;

  } // if
  if(isDefined)
    shiftLevels();
  return isDefined;
} // next

/** Take the solution backward one step to initialize schemes
    with 3 time levels.
    The equation is set by the internal variable equation_ and
    the numerical scheme by the internal variable scheme_
    @param runData List of run parameters
    @see Data#TIMSTP
    @see Data#VELCTY
    @see Data#DIFFUS
    @see Data#THETA
    @see Data#TUNINT
    @return True if an initialisation scheme is implemented for that equat
************************************************************************/
public boolean previous(Data runData){
  int n=f.length-1;
  boolean isDefined=false;

  if(equation_.equals(Data.ADVECTION) &&
     scheme_.equals(Data.CUBFEM)){                  //TAG_CHAAdvCFEMInit_TA
    //=========================================================================
    // CHA - Advection/diffusion - CubicFEM - Yabe+Aoki CPC 66(1991)219
    //=========================================================================
    for (int i=1; i<n; i++) {
      df[i] =(f[i+1]-f[i-1])/(2.*dx[0]);
    }
    df[0]=(f[1]-f[n  ])/(2.*dx[0]);
    df[n]=(f[0]-f[n-1])/(2.*dx[0]);
    isDefined=true;
  }
  return isDefined;
} // previous

} // class CHASolution
```

```
/* $Id: FDSolution.java,v 1.18.4.3 1999/08/05 11:48:14 andrej Exp $ */

/*****************************************************************************
*  FDSolution -- contains the finite differance solver.
*    @see FluidSolution
*    @see Solution
*****************************************************************************/
public class FDSolution extends FluidSolution{
  /** Creates a FDSolution object.
      @param mesh The mesh of the solution
  */
  public FDSolution(Mesh mesh){
    super(mesh);
  } // FDSolution


  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  ****************************************************************************/
  public void discretize(ShapeFunction function){
    for (int j = 0; j < mesh_.size(); j++)       //Discretizing is very simple
      f[j] = function.getValue(mesh_, j);        //  with finite differences
    initialMoments[0]=calculateMoments(0);       //Set conserved quantities
    initialMoments[1]=calculateMoments(1);
  } // discretize


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if a scheme is implemented for that equation
  ****************************************************************************/
  public boolean next(Data runData) {
    double timeStep = runData.getDouble(Data.TIMSTP);          //Parameters
    double velocity = runData.getDouble(Data.VELCTY);
    double diffusCo = runData.getDouble(Data.DIFFUS);
    double disperCo = runData.getDouble(Data.DISPER);

    double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are only constant if
    double beta =timeStep*velocity/(dx[0]      );  //  the problem and mesh are
                                                   //  homogeneous
    int n=f.length-1;
    boolean isDefined=false;

    if(equation_.equals(Data.ADVECTION)
       && scheme_.equals(Data.EXP2LVL)){                    //TAG_FDAdvExp2_TAG//
      //=======================================================================
      //  FD  - Advection/diffusion - explicit 2 levels (Godunov advection)
      //=======================================================================
      for (int i=1; i<n; i++) {
        fp[i]=f[i] -beta *(f[i]-f[i-1])+alpha*(f[i+1]-2.*f[i]+f[i-1]); }
      fp[0]=f[0]   -beta *(f[0]-f[ n ])+alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
      fp[n]=f[n]   -beta *(f[n]-f[n-1])+alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.EXP3LVL)){             //TAG_FDAdvExp3_TAG//
      //=======================================================================
      //  FD  - Advection/diffusion - explicit 3 levels
      //=======================================================================
      for (int i=1; i<n; i++) {
        fp[i]=fm[i] -beta*(f[i+1]-f[i-1]) +2*alpha*(f[i+1]-2.*f[i]+f[i-1]);}
      fp[0]=fm[0]   -beta*(f[ 1 ]-f[ n ]) +2*alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
      fp[n]=fm[n]   -beta*(f[ 0 ]-f[n-1]) +2*alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.IMP2LVL)){             //TAG_FDAdvImp2_TAG//
      //=======================================================================
      //  FD  - Advection/diffusion - implicit (Crank-Nicholson diffusion)
      //=======================================================================
      BandMatrix a = new BandMatrix(3, f.length);
      BandMatrix b = new BandMatrix(3, f.length);
      double[]   c = new double[f.length];

      for (int i=0; i<=n; i++) {
        a.setL(i,-0.25*beta -0.5*alpha);             //Matrix elements
        a.setD(i,            1. +alpha);
        a.setR(i, 0.25*beta -0.5*alpha);
        b.setL(i, 0.25*beta +0.5*alpha);             //Right hand side
        b.setD(i,            1. -alpha);
        b.setR(i,-0.25*beta +0.5*alpha);
      }

      c=b.dot(f);                                    //Right hand side
      c[0]=c[0]+b.getL(0)*f[n];                      // with periodicity
      c[n]=c[n]+b.getR(n)*f[0];

      fp=a.solve3(c);                                //Solve linear problem
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.EXPLAX)){          //TAG_FDAdvExpLax_TAG//
      //=======================================================================
      //  FD  - Advection - explicit Lax-Wendroff - precision in O(delta t^3)
      //=======================================================================
      for (int i=1; i<n; i++) {
        fp[i]=f[i] -0.5*beta *(f[i+1]-f[i-1])
                   +0.5*beta*beta*(f[i+1]-2.*f[i]+f[i-1]); }
      fp[0]=f[0] -0.5*beta *(f[ 1 ]-f[ n ])
                 +0.5*beta*beta*(f[ 1 ]-2.*f[0]+f[ n ]);
      fp[n]=f[n] -0.5*beta *(f[ 0 ]-f[n-1])
                 +0.5*beta*beta*(f[ 0 ]-2.*f[n]+f[n-1]);
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.IMPLAX)){          //TAG_FDAdvImpLax_TAG//
      //=======================================================================
      //  FD  - Advection - implicit Lax-Wendroff
      //=======================================================================
      BandMatrix a = new BandMatrix(3, f.length);
      BandMatrix b = new BandMatrix(3, f.length);
      double[]   c = new double[f.length];

      for (int i=0; i<=n; i++) {
        a.setL(i, (1.-beta)/(1.+beta));              //Matrix elements
        a.setD(i, 1.);
        a.setR(i, 0.);
        b.setL(i, 1.);                               //Right hand side
        b.setD(i, (1.-beta)/(1.+beta));
        b.setR(i, 0.);
      }

      c=b.dot(f);                                    //Right hand side
      c[0]=c[0]+b.getL(0)*f[n];                      // with periodicity
      c[n]=c[n]+b.getR(n)*f[0];

      fp=a.solve3(c);                                //Solve linear problem
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.LPFROG)){          //TAG_FDLeapFrog_TAG//
      //=======================================================================
      //  FD  - Advection/diffusion - Leap-frog
      //=======================================================================
      for (int i=1; i<=n; i++) {                     //time + time_step
        fp[i]=f[i] -beta*(g[i]-g[i-1]); }
      fp[0]=f[0] -beta*(g[0]-g[n]);

      for (int i=0; i<=n-1; i++) {                   //time + 1.5*time_step
        gp[i]=g[i] -beta*(fp[i+1]-fp[i]); }
      gp[n]=g[n] -beta*(fp[0]-fp[n]);
      isDefined = true;

    } else if(equation_.equals(Data.BURGER)
              && scheme_.equals(Data.EXP3LVL)){         //TAG_FDBrgExp3_TAG//
      //=======================================================================
      //  FD  - Burger's shock wave equation - explicit 3 levels
      //=======================================================================

      double fm1 = f[ n ];           //Extra mesh points for periodicity
      double f0  = f[ 0 ];
      double fp1 = f[ 1 ];

      for (int i=0; i<=n; ++i) {
        fp[i]= f[i]
          -timeStep/dx[0]* f0*(fp1-fm1)          //Wave breaking term
          +alpha*(fp1-2*f0+fm1);                 //Diffusive term
        fm1=f0;  f0 =fp1;
        fp1=f[(i+2) % (n+1)];
      }
      isDefined = true;

    } else if(equation_.equals(Data.KDV)
              && scheme_.equals(Data.EXP3LVL)){         //TAG_FDKdvExp3_TAG//
      //=======================================================================
      // FD  - KdV - Scheme Zabusky & Kruskal, Phys.Rev.Lett. 15(1965)240
      //=======================================================================
      double nolin=timeStep/(dx[0]);
      double gamma=timeStep/(dx[0]*dx[0]*dx[0])*disperCo;

      double fm2 = f[n-1];  double fm1 = f[ n ];
      double f0  = f[ 0 ];  double fp1 = f[ 1 ];  double fp2 = f[ 2 ];

      for (int i=0; i<=n; ++i) {
        fp[i]= fm[i]
          -nolin*(fp1+f0+fm1)/3.*(fp1-fm1)       //Wave breaking term
          -gamma*(fp2 -2.*(fp1-fm1) -fm2);       //Dispersive term

        fm2=fm1;  fm1=f0;  f0 =fp1;  fp1=fp2;    //Periodicity
        fp2=f[(i+3) % (n+1)];
      }
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.FDMINE)){          //TAG_FDAdvMine1_TAG//
      //=======================================================================
      //  FD  - Implement your own scheme here
      //=======================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FDEX21)){
      //=======================================================================
      //  FD  - Exercise 2.1
      //=======================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FDEX23)){
      //=======================================================================
      //  FD  - Exercise 2.3
      //=======================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FDEX24)){
      //=======================================================================
      //  FD  - Exercise 2.4
      //=======================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FDEX25)){
      //=======================================================================
      //  FD  - Exercise 2.5
      //=======================================================================
      isDefined = false;

    } // if

    if(isDefined && isForward)
      shiftLevels();
    return isDefined;
  } // next


  /** Take the solution backward one step to initialize schemes
      with 3 time levels.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if an initialisation scheme is implemented for that equation
  ****************************************************************************/
  public boolean previous(Data runData){
    int n=f.length-1;
    boolean isDefined=false;

    if(equation_.equals(Data.ADVECTION) && scheme_.equals(Data.EXP3LVL) ||
       equation_.equals(Data.KDV)       && scheme_.equals(Data.EXP3LVL)    ){
                                                       //TAG_FD31vlInit_TAG//
      //=======================================================================
      // FD init - 3 levels - Common to all such schemes
      //=======================================================================
      for (int i=0; i<=n; i++)          //Note how backward is implemented:
        { fm[i]=0.;}                    //  first reset the oldest level,
      isForward = false;                //  then calculate a pseudo-forward
      this.next(runData);               //  solution without shifting
      isForward = true;                 //  levels.

      for (int i=0; i<=n; i++) {        //  Divide by (-2) to gives a scheme
```

```
        fm[i]=f[i]-0.5*fp[i];         //  equivalent to two levels backward
      }

    } else if (equation_.equals(Data.ADVECTION)&&scheme_.equals(Data.LPFROG)){
                                                          //TAG_FDLpFrgInit_TAG//
      //=====================================================================
      // FD init - Leap-Frog
      //=====================================================================
      double timeStep = runData.getDouble(Data.TIMSTP);
      double velocity = runData.getDouble(Data.VELCTY);
      double diffusCo = runData.getDouble(Data.DIFFUS);
      double alpha=timeStep*diffusCo/(dx[0]*dx[0]);
      double beta =timeStep*velocity/(dx[0]      );

      for (int i=0; i<=n-1; i++) {               //time=0, initialize pulse
          gm[i] = 0.5*(f[i+1]+f[i]);             //  running to the right
        //gm[i] =-0.5*(f[i+1]+f[i]);             //              the left
        //gm[i] = 0.;                            //      in both directions
      }
      gm[n]=0.5*(f[0]+f[n]);

      for (int i=0; i<=n-1; i++) {               //time=0.5*time_step
          g[i] = gm[i]-0.5*beta*(f[i+1]-f[i]); }
      g[n]=gm[n]-0.5*beta*(f[0]-f[n]);;

    }
    return isDefined;
  } // previous

} // class FDSolution
```

FEMSolution.java Thu Aug 05 13:48:1

```
/* $Id: FEMSolution.java,v 1.12.4.2 1999/08/05 11:48:15 andrej Exp $ */

/*************************************************************************
*  FEMSolution -- contains the finite elements solver.
*    @see FluidSolution
*    @see Solution
*************************************************************************/
public class FEMSolution extends FluidSolution{
  /** Creates a FEMSolution object.
      @param mesh The mesh of the solution
  */
  public FEMSolution(Mesh mesh){
    super(mesh);
  } // FEMSolution

                                                        //TAG_FEMInit_TAG//
  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  *************************************************************************/
  public void discretize(ShapeFunction function){
    for (int j = 0; j < mesh_.size(); j++)       //Discretizing is very simple
      f[j] = function.getValue(mesh_, j);        //  with normalized FEM
    initialMoments[0]=calculateMoments(0);       //Set conserved quantities
    initialMoments[1]=calculateMoments(1);
  } // discretize


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if a scheme is implemented for that equation
  *************************************************************************/
  public boolean next(Data runData) {
    double timeStep = runData.getDouble(Data.TIMSTP);         //Parameters
    double velocity = runData.getDouble(Data.VELCTY);
    double diffusCo = runData.getDouble(Data.DIFFUS);
    double theta    = runData.getDouble(Data.THETA);
    double tune     = runData.getDouble(Data.TUNINT);

    double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are only constant if
    double beta =timeStep*velocity/(dx[0]      );  //  the problem and mesh are
                                                   //  homogeneous
    int n=f.length-1;
    boolean isDefined=false;

    if(equation_.equals(Data.ADVECTION)
       && scheme_.equals(Data.TUNFEM)){                    //TAG_FEMAdv_TAG//
      //=====================================================================
      //  FEM - Advection/diffusion, tunable integration    t=1.  FD
      //                                                    t=1/3 linear FEM
      //                                                    t=0.  constant FEM
      //=====================================================================
      BandMatrix a = new BandMatrix(3, f.length);
      BandMatrix b = new BandMatrix(3, f.length);
      double[]   c = new double[f.length];

      double h   = dx[0];
      double htm = h*(1-tune)/4;
      double htp = h*(1+tune)/4;

      for (int i=0; i<=n; i++) {
        a.setL(i,   htm +h*(-0.5*beta -alpha)* theta    );
        a.setD(i,2*(htp +h*(          alpha)* theta   ));
        a.setR(i,   htm +h*( 0.5*beta -alpha)* theta    );
        b.setL(i,   htm +h*(-0.5*beta -alpha)*(theta-1) );
        b.setD(i,2*(htp +h*(          alpha)*(theta-1)));
        b.setR(i,   htm +h*( 0.5*beta -alpha)*(theta-1) );
      }

      c=b.dot(f);                                  //Right hand side
      c[0]=c[0]+b.getL(0)*f[n];                    // with periodicity
      c[n]=c[n]+b.getR(n)*f[0];

      fp=a.solve3(c);                              //Solve linear problem
      isDefined = true;

    } else if(equation_.equals(Data.ADVECTION)
              && scheme_.equals(Data.FEMMINE)){          //TAG_FEMAdvMine_TAG//
      //=====================================================================
      //  FEM  - Implement your own scheme here
      //=====================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FEMEX32)){
      //=====================================================================
      //  FEM - Exercise 3.2
      //=====================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FEMEX33)){
      //=====================================================================
      //  FEM - Exercise 3.3
      //=====================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FEMEX34)){
      //=====================================================================
      //  FEM - Exercise 3.4
      //=====================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FEMEX35)){
      //=====================================================================
      //  FEM - Exercise 3.5
      //=====================================================================
      isDefined = false;

    } // if

    if(isDefined)
      shiftLevels();
    return isDefined;
  } // next

  /** Take the solution backward one step to initialize schemes
      with 3 time levels; not really appropriate in this context.
      @param runData List of run parameters
      @return False since it is not used here
  *************************************************************************/
  public boolean previous(Data runData){ return false; }

} // class FEMSolution
```

```
/* $Id: FFTSolution.java,v 1.16.4.5 1999/08/05 11:48:16 andrej Exp $ */

/*****************************************************************************
* FFTsolution - Fast Fourier Transform solver
*    @see FluidSolution
*    @see Solution
*****************************************************************************/
public class FFTSolution extends FluidSolution{
  public FFTSolution(Mesh mesh){
    super(mesh);
  } // FFTSolution

  /** FFT previous (or initial) step   */  protected  FFT keepFFT;
  /** FFT tool                         */  protected  FFT toolFFT;

                                                          //TAG_FFTInit_TAG//
  /** Discretize the initial Shape function and initialize the moments
      @param The initial shape to be approximated
  *****************************************************************************/
  public void discretize(ShapeFunction function){
    double boxLen = mesh_.size()*mesh_.interval(0);

    for (int j=0; j<mesh_.size(); j++)          //Homogeneous mesh approx
      f[j] = function.getValue(mesh_, j);

    keepFFT  = new FFT(f,  FFT.inXSpace);       //Load into FFT transformer

    initialMoments[0]=calculateMoments(0);      //Reset conserved quantities
    initialMoments[1]=calculateMoments(1);
  } // discretize


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @return True if a scheme is implemented for that equation
  *****************************************************************************/
  public boolean next(Data runData) {
    double timeStep = runData.getDouble(Data.TIMSTP);      //Parameters
    double velocity = runData.getDouble(Data.VELCTY);
    double diffusCo = runData.getDouble(Data.DIFFUS);
    double disperCo = runData.getDouble(Data.DISPER);

    int N           = mesh_.size();                        //A power of 2
    double boxLen   = mesh_.size()*mesh_.interval(0);      //Periodicity

    double   k    = 2*Math.PI/boxLen;                      //Notation
    Complex ik1   = new Complex(  0.,   k  );
    Complex ik2   = new Complex(-k*k,  0.  );
    Complex ik3   = new Complex(  0.,-k*k*k);
    Complex exp   = new Complex();

    Complex advection = new Complex(ik1.scale(velocity));  //Variables
    Complex diffusion = new Complex(ik2.scale(diffusCo));
    Complex dispersion= new Complex(ik3.scale(disperCo));
    Complex total     = new Complex(0.);
    Complex nonlin    = new Complex(0.);
    Complex linear    = new Complex(0.);

    boolean isDefined = false;

    if(equation_.equals(Data.ADVECTION)
       && (scheme_.equals(Data.ALIASED)||
           scheme_.equals(Data.EXPAND )  )){                //TAG_FFTAdv_TAG//
      //========================================================================
      //  FFT - Advection/diffusion - No problem with aliasing if linear
      //========================================================================
      Complex[] s0 = new Complex[f.length];

      s0=keepFFT.getFromKSpace(FFT.firstPart,boxLen);    //FFT real to KSpace
                                                         //  only once
      s[0] = s0[0];
      for (int m=1;  m<N/2+1;  m++) {                    //Propagate directly
        total=          advection.scale((double)(m ));   //  from initial
        total=total.add(diffusion.scale((double)(m*m))); //  contition
        exp=(total.scale(time_)).exp();

        s[m  ] = s0[m].mul(exp);                         //  s0 contains the IC
        s[N-m] = s[m].conj();                            //  f is real
      }
      FFT ffts = new FFT(s,FFT.inKSpace);                //Initialize Kspace
      f = ffts.getFromXSpacePart(FFT.firstPart,boxLen);  //FFT back for plot

      isForward = false;
      isDefined = true;

    } else  if((equation_.equals(Data.KDV)||
                equation_.equals(Data.BURGER))
                && (scheme_.equals(Data.ALIASED)||
                    scheme_.equals(Data.EXPAND )  )){        //TAG_FFTKdV_TAG//
      //========================================================================
      //  FFT - KdV solitons and Burger's shocks - Aliasing here is an issue
      //========================================================================

      //----- Non-linear term: convolution                 //TAG_FFTnonLin_TAG//
      s  = keepFFT.getFromKSpace(FFT.bothParts,boxLen);    //Current Spectrum
      toolFFT = new FFT(s,s,FFT.inKSpace);                 //  for convolution

      if (scheme_.equals(Data.ALIASED))                    //With-/out aliasing,
        sp = toolFFT.aliasedConvolution(boxLen);           //  use an FFT to
      else //scheme_.equals(Data.EXPAND)                   //  calculate product,
        sp = toolFFT.expandedConvolution(boxLen);          //  FFT back to KSpace


      //----- Linear terms: complex terms in spectrum s    //TAG_FFTlinear_TAG//
      s = keepFFT.getFromKSpace(FFT.bothParts,boxLen);     //Current Spectrum
      linear= s[0];
      sp[0]=linear;
      for (int m=1;  m<=N/2;  m++) {
        total=          advection.scale((double)(m ));
        total=total.add(diffusion.scale((double)(m*m)));
        total=total.add(dispersion.scale((double)(m*m*m)));
        exp=(total.scale(timeStep)).exp();
        linear    = s[m].mul(exp);
        nonlin    = sp[m].mul(ik1.scale(0.5*timeStep*(double)(m)));
        sp[m  ]   = linear.add(nonlin);
        if (m<N/2) sp[N-m] = sp[m].conj();                 //For a real spectrum
      }

      keepFFT = new FFT(sp,FFT.inKSpace);                  //Spectrum is complete
      toolFFT = new FFT(sp,FFT.inKSpace);                  //inv FFT for plotting
      f=toolFFT.getFromXSpacePart(FFT.firstPart,boxLen);

      isForward = false;
      isDefined = true;

    } else  if(equation_.equals(Data.ADVECTION) ||
               equation_.equals(Data.FFTMINE)  ){          //TAG_FFTAdvMine_TAG/
      //========================================================================
      //  FFT - Implement your own scheme here
      //========================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FFTEX41)){
      //========================================================================
      //  FEM - Exercise 4.1
      //========================================================================
      isDefined = false;

    } else if(scheme_.equals(Data.FFTEX42)){
      //========================================================================
      //  FEM - Exercise 4.2
      //========================================================================
      isDefined = false;

    }

    if(isDefined && isForward)
      shiftLevels();
    return isDefined;
  } // next


  /** Starting procedure for FFT schemes.
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @see Data#BOXLEN
      @return True if an initialisation scheme is implemented for that equation
  *****************************************************************************/
  public boolean previous(Data runData){
    int n=f.length-1;
    boolean isDefined=false;
    return isDefined;
  } // previous

} // class FFTSolution
```

```
/* $Id: ParticleSolution.java,v 1.8.4.4 1999/06/28 10:04:36 johanh Exp $ */

import java.util.Random;

//TAG_ParticleSolution_TAG//

/** The ParticleSolution solves the equation with particle methods
    @version $Revision: 1.8.4.4 $
    @see Solution
*/
abstract class ParticleSolution extends Solution{

  /** The number of test particles in the simulation */
  protected int numberOfParticles;
  /** A vector with the position of the particles */
  protected double[] particlePosition;
  /** The amount of mass each test particle represents */
  protected double particleDensity;
  /** Whether the distribution function is up to date */
  protected boolean distributionUpToDate;
  /** Random number generator */
  protected Random random;

  /** Creates a ParticleSolution object. discretize must be called
      before next is called for the first time.
      @param mesh The mesh of the solution
      @see Solution#next
      @see Solution#discretize
  */
  public ParticleSolution(Mesh mesh){
    super(mesh);
    random = new Random();
  } // ParticleSolution

  /** Take the solution backward one step to initialize schemes
      with 3 time levels; not really appropriate in this context.
      @param runData List of run parameters
      @return False since it is not used here
  */
  public boolean previous(Data runData){ return false; }

  /** Discretize the initial Shape function and initialize the moments
      Initializes a new set of particles, the number being determined by
      the numerical scheme parameter.
      @param The initial shape to be approximated
      @see Solution#setScheme
      @see Data#NPARTICLE1
  */
  public void discretize(ShapeFunction function){
    //TAG_ParticleDiscretize_TAG//
    // First set the distribution function and calculate the moments
    int j;
    for (j = 0; j < mesh_.size(); j++)
      f[j] = function.getValue(mesh_, j);
    initialMoments[0]=calculateMoments(0); //Conserved quantities
    initialMoments[1]=calculateMoments(1);
    initialMoments[2]=calculateMoments(2);
    if(scheme_.equals(Data.NPARTICLE1))
      numberOfParticles = 1;
    else if(scheme_.equals(Data.NPARTICLE100))
      numberOfParticles = 100;
    else if(scheme_.equals(Data.NPARTICLE1000))
      numberOfParticles = 1000;
    else if(scheme_.equals(Data.NPARTICLE10000))
      numberOfParticles = 10000;
    else
      numberOfParticles = 100;
    particlePosition = new double[numberOfParticles];
    particleDensity = initialMoments[0] / numberOfParticles;
    // Normalize the distribution function to (0,1)
    double[] lim = calculateLimits();
    for(j = 0; j < mesh_.size(); j++)
      f[j] = (f[j] - lim[0]) / (lim[1] - lim[0]);
    // Randomize the particle positions
    double position;
    lim = mesh_.limits();
    for(j = 0; j < numberOfParticles; j++){
      while(random.nextDouble() >
            getValue(position = lim[0] +
                     (lim[1] - lim[0]) * random.nextDouble()));
      particlePosition[j] = position;
    } // for
    distributionUpToDate = false;
  } // discretize

  /** Calculates the deviation from the m:th initial moment.
      @param m The order of the moment
      @return The deviation of the m:th moment from the beginning
      @see discretize
  */
  public double momentsDeviation(int m){
    if(!distributionUpToDate)
      generateDistribution();
    return calculateMoments(m);
  } // moments

  /** Calculates the limits of the solution.
      @return A vector consisting of {min(solution), max(solution)}
  */
  public double[] limits(){
    if(!distributionUpToDate)
      generateDistribution();
    return calculateLimits();
  } // limits

  /** Gives the value of the distribution function for an index
      @param index The index for which to get the value
      @return The value of the distribution function at a given index
  */
  public double getValue(int index){
    if(!distributionUpToDate)
      generateDistribution();
    return f[index];
  } // getValue

  /** Generates the distribution function from the set of
      test particles. Assumes a uniform mesh.
  */
  private void generateDistribution(){
    //TAG_ParticleProjection_TAG//
    int j;
    for(j = 0; j < mesh_.size(); j++)
      f[j] = 0;
    for(j = 0; j < numberOfParticles; j++){
      double[] lim = mesh_.limits();
      // Calculate the lower index of mesh cell
      int index = (int)Math.floor((particlePosition[j] - lim[0])
                                  / (lim[1] - lim[0]) *
                                  (mesh_.size() - 1));
      // Check whether the particle is inside the plot region
      if(index >= 0 && index <= mesh_.size()){
        if(index == mesh_.size())
          index--;
        double x_low = mesh_.point(index);
        double x_high = mesh_.point(index) + dx[0];
        // Find the interpolation coefficient
        double a = (particlePosition[j] - x_low) / ( x_high - x_low);
        // Add the particle contribution to the distribution function
        double cell_area_left = 1 / dx[0];  // Assume uniform mesh
        double cell_area_right = 1 / dx[0]; // Assume uniform mesh
        f[index] += (1 - a) * particleDensity / cell_area_left;
        f[(index + 1) % mesh_.size()] += a * particleDensity / cell_area
      } // if
    } // for
    distributionUpToDate = true;
  } // generateDistribution

  /** Gives a linear interpolation of the function. Assumes a uniform me
      @param position The x value
      @return A linear interpolation of the distribution function at a g
      position
  */
  private double getValue(double position){
    double[] lim = mesh_.limits();
    // Calculate the lower index of mesh cell
    int index = (int)Math.floor((position - lim[0]) / (lim[1] - lim[0])
                                (mesh_.size() - 1));
    if(index == mesh_.size() - 1)
      index--;
    double x_low = mesh_.point(index);
    double x_high = mesh_.point(index + 1);
    // Find the interpolation coefficient
    double a = (position - x_low) / ( x_high - x_low);
    return f[index] * (1 - a) + f[index + 1] * a;
  } // getValue


} // ParticleSolution
```

MCMSolution.java Sat Jul 31 16:06

```
/* $Id: MCMSolution.java,v 1.20.4.5 1999/06/28 10:04:35 johanh Exp $ */

import java.util.Random;

//TAG_MCMSolution_TAG//

/** The MCMSolution class solves the equation with a Monte Carlo method.
    @version $Revision: 1.20.4.5 $
    @see Solution
*/
public class MCMSolution extends ParticleSolution{

  /** Creates a MCMSolution object. discretize must be called
      before next is called for the first time.
      @param mesh The mesh of the solution
      @see Solution#next
      @see Solution#discretize
  */
  public MCMSolution(Mesh mesh){
    super(mesh);
  } // MCMSolution


  /** Advance the solution forward one step in time.
      The equation is set by the internal variable equation_ and
      the numerical scheme by the internal variable scheme_
      @param runData List of run parameters
      @see Data#TIMSTP
      @see Data#VELCTY
      @see Data#DIFFUS
      @see Data#THETA
      @see Data#TUNINT
      @return True if a scheme is implemented for that equation
  */
  public boolean next(Data runData) {
    double timeStep = runData.getDouble(Data.TIMSTP);         //Parameters
    double velocity = runData.getDouble(Data.VELCTY);
    double diffusCo = runData.getDouble(Data.DIFFUS);

    distributionUpToDate = false;
    boolean isDefined = false;
    double[] lim = {mesh_.point(0),
                    mesh_.point(mesh_.size() - 1) + dx[0]};
    if(equation_.equals(Data.ADVECTION)){               //TAG_MCMNext_TAG//
      for(int j = 0; j < numberOfParticles; j++){
        particlePosition[j] += velocity * timeStep +
          random.nextGaussian() *
          Math.sqrt(2 * diffusCo * timeStep);
        // Periodic boundary conditions
        //
        //  Exercise: add your boundary conditions here    //TAG_MCMExoBC_TAG//
        //
      } // for
      isDefined = true;
    } // if
    return isDefined;
  } // next


} // class MCMSolution
```

```
jbone: /* $Id: jbone.java,v 1.29.4.5 1999/08/05 11:48:17 andrej Exp $ */
jbone:
jbone: import java.util.*;
jbone: import java.text.*;
jbone: import java.awt.*;
jbone: import java.lang.Math;
jbone: import java.applet.Applet;
jbone:
jbone:
jbone:
jbone: /*****************************************************************************
jbone: * Java Bed for ONE dimensional partial differential equations.
jbone: *
jbone: *  Andre JAUN (jaun@fusion.kth.se) and Johan HEDIN (johanh@fusion.kth.se)
jbone: *  Alfven Laboratory, Royal Institute of Technology, SE-100 44 Stockholm.
jbone: *  Copyright 16-December-1998. All Rights Reserved.
jbone: *  @version Version: 1.0  $Revision: 1.29.4.5 $
jbone: *
jbone: *  This shareware can be obtained without fee by e-mail from the authors.
jbone: *  Permission to use, copy, and modify the source and its documentation
jbone: *  for your own personal use is hereby granted provided that this copyright
jbone: *  notice appears in all copies. This educational software is adapted from
jbone: *  an idea originally from Kurt.Appert@epfl.ch. The authors makes no
jbone: *  representations or warranties about the suitability of the software,
jbone: *  either express or implied, including but not limited to the implied
jbone: *  warranties of merchantability, fitness for a particular purpose, or
jbone: *  non-infringement. The authors shall not be liable for any damages
jbone: *  suffered by licensee as a result of using or modifying this software
jbone: *  or its derivatives.
jbone: *****************************************************************************/
jbone: public class jbone extends Applet implements Runnable {
jbone:
jbone:   //====== Control Variables
jbone:   /** Potentially reset by main*/          boolean isAnApplet   = true;
jbone:   /** Current step number*/                int     step         = 0;
jbone:   /** Operate nsteps before stopping*/     int     nstep        = 0;
jbone:   /** Milliseconds between plots*/         int     delay        = 84;
jbone:   /** Start-stop*/                         boolean frozen       = false;
jbone:   /** Start from a new initial cond*/      boolean initialize   = true;
jbone:   /** Rescale plot window*/                boolean setScale     = true;
jbone:   /** Display step+time+moments or not*/   boolean header       = true;
jbone:   /** Keep the same function*/             boolean keepFunction = false;
jbone:
jbone:   //===== Applet objects
jbone:   /** Thread label*/                       Thread    runThread;
jbone:   /** Set of run parameters*/              Data      runData;
jbone:   /** The whole window*/                   GUIWindow window;
jbone:   /** Only the plot area*/                 MyCanvas  plot;
jbone:   /** Pop-up to change values*/            MyEditor  editor;
jbone:   /** Panel of run options*/               Panel     shCards;
jbone:
jbone:   //===== Computational objects
jbone:   /** Discretization mesh*/                Mesh      mesh;
jbone:   /** Solution of the equation*/           Solution  solution;
jbone:
jbone:
jbone:   /** Master initialization and layout
jbone:   *****************************************************************************/
jbone:   public void init() {
jbone:     runData = new Data();                                  //Preset values
jbone:     if (isAnApplet) tagModify();                           //Modify with HTML tags
jbone:     window = new GUIWindow();                              //Create a window layout
jbone:     validate();                                            //Activate in applet
jbone:   }
jbone:
jbone:
jbone:   /** Applet start a new thread
jbone:   *****************************************************************************/
jbone:   public void start() {
jbone:     if (runThread == null) {                 //Start a new thread
jbone:        runThread = new Thread(this);
jbone:     }
jbone:     if (frozen) {                                  ... user
jbone:     } else { runThread.start(); }                  ... ation
jbone:   }
jbone:
jbone:
jbone:   /** Contains the main loop for th... ...
jbone:      @see ShapeFunction
jbone:      @see Mesh
jbone:      @see Solution
jbone:   *****************************************************************************/
jbone:   public void run() {
jbone:     Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
jbone:     long startTime = System.currentTimeMillis();
jbone:     Thread currentThread = Thread.currentThread();
jbone:
jbone:     if (initialize) {
jbone:
jbone:       step = 0;                                            //Reset time step counter
jbone:
jbone:       ShapeFunction    oldSolution = null;                 //Hot switch schemes
jbone:       if(keepFunction) oldSolution = new ShapeNumerical(solution);
jbone:
jbone:       mesh     = new Mesh(runData);                        //New Grid
jbone:       solution = createSolution();                         //New solution & solver
jbone:
jbone:       solution.setEquation(runData.pdeName(runData.pdeType));
jbone:       setInitialCondition(solution, oldSolution);
jbone:       solution.previous(runData);                          //3 time levels initialize
jbone:
jbone:       plot.repaint();                                      //Graphics
jbone:       initialize=false;
jbone:       setScale = true;
jbone:     }
jbone:
jbone:     while (currentThread == runThread &&
jbone:            (double)step*runData.getDouble(Data.TIMSTP) <
jbone:                     runData.getDouble(Data.RUNTIM)    &&
jbone:            nstep-- != 0 && !frozen) {
jbone:
jbone:       step++;                                              //One step forward in time
jbone:       solution.setTime(
jbone:         (double)step*runData.getDouble(Data.TIMSTP));
jbone:       solution.next(runData);
jbone:
jbone:       plot.repaint();                                      //Update graphics
jbone:       try { startTime += delay;                            //Delay according to lag
jbone:       Thread.sleep(Math.max(0, startTime-System.currentTimeMillis()));
jbone:       } catch (InterruptedException e) { break; }
jbone:     } // while
jbone:     frozen=true;
jbone:   } // run
jbone:
jbone:
jbone:   /** Instanciate a solution and select the method and scheme for computations
jbone:      @see Solution
jbone:      @return A new Solution
jbone:   *****************************************************************************/
jbone:   private Solution createSolution(){
jbone:     Solution newSol;
jbone:     if       ( runData.numMethIs(Data.FD) ) {
jbone:       newSol = new FDSolution(mesh);
jbone:       newSol.setScheme(runData.FDSchemName(runData.numSchem));
jbone:     } else if( runData.numMethIs(Data.FEM) ){
jbone:       newSol = new FEMSolution(mesh);
jbone:       newSol.setScheme(runData.FEMSchemName(runData.numSchem));
jbone:     } else if( runData.numMethIs(Data.FFT) ){
jbone:       newSol = new FFTSolution(mesh);
jbone:       newSol.setScheme(runData.FFTSchemName(runData.numSchem));
jbone:     } else if( runData.numMethIs(Data.MCM) ){
jbone:       newSol = new MCMSolution(mesh);
jbone:       newSol.setScheme(runData.MCMSchemName(runData.numSchem));
jbone:     } else { //runData.numMethIs(Data.CHA) ){
jbone:       newSol = new CHASolution(mesh);
jbone:       newSol.setScheme(runData.CHASchemName(runData.numSchem));
jbone:     } // if
jbone:     newSol.setMethod(runData.numMethName(runData.numMeth));
jbone:     return newSol;
jbone:   } // createSolution
jbone:
jbone:
jbone:   /** Set the initial condition accordind to the runData parameters.
jbone:       @param oldSolution obtained previously with another method, scheme
jbone:       @see Data#IC1AMP
jbone:       @see Data#IC1POS
jbone:       @see Data#IC1WID
jbone:       @see Solution
jbone:   *****************************************************************************/
jbone:   private void setInitialCondition(Solution inSolution,
jbone:                                    ShapeFunction oldSolution){
jbone:     if(keepFunction){
jbone:       inSolution.discretize(oldSolution);
jbone:       keepFunction = false;
jbone:
jbone:     } else if(runData.icIs(Data.BOX)){
jbone:       inSolution.discretize(new ShapeBox(     runData.getDouble(Data.IC1AMP),
jbone:                                               runData.getDouble(Data.IC1POS),
jbone:                                               runData.getDouble(Data.IC1WID)));
jbone:
jbone:     } else if(runData.icIs(Data.GAUSSIAN)){
jbone:       inSolution.discretize(new ShapeGaussian(runData.getDouble(Data.IC1AMP),
jbone:                                               runData.getDouble(Data.IC1POS),
jbone:                                               runData.getDouble(Data.IC1WID)));
jbone:
jbone:     } else if(runData.icIs(Data.COSINUS)){
jbone:       inSolution.discretize(new ShapeCosinus( runData.getDouble(Data.IC1AMP),
jbone:                                               runData.getDouble(Data.IC1POS),
jbone:                                               runData.getDouble(Data.IC1WID)));
jbone:
jbone:     } else if(runData.icIs(Data.SOLITON)){
jbone:       inSolution.discretize(new ShapeSoliton( runData.getDouble(Data.IC1AMP),
jbone:                                               runData.getDouble(Data.IC1POS),
jbone:                                               runData.getDouble(Data.IC1WID)));
jbone:     } // if
jbone:
jbone:   } // setInitialCondition
jbone:
jbone:   /** Breathing space between panel and contents (Java1.1) */
jbone:   public Insets getinsets() { return new Insets(5,5,5,5); }
jbone:
jbone:   /** Breathing space between panel and contents (Java1.0) @deprecated */
jbone:   public Insets insets() { return new Insets(5,5,5,5); }
jbone:
jbone:   /** Applet stop
jbone:   *****************************************************************************/
jbone:   public void stop() {
jbone:     runThread = null;
jbone:     frozen    = true;
jbone:    }
jbone:
jbone:
jbone:   /** Modify defaults parameters the HTML tags from the web page
jbone:   *****************************************************************************/
jbone:   public void tagModify() {
jbone:     String s0 = getParameter("NSTEP");
jbone:     String s1 = getParameter("PDE");
jbone:     String s2 = getParameter("IC");
jbone:     String s3 = getParameter("METH");
jbone:     String s4 = getParameter("SCHEM");
jbone:     if (s0 != null) nstep   =Integer.valueOf(s0).intValue();
jbone:     if (s1 != null) runData.pdeType =Integer.valueOf(s1).intValue();
jbone:     if (s2 != null) runData.icType  =Integer.valueOf(s2).intValue();
jbone:     if (s3 != null) runData.numMeth =Integer.valueOf(s3).intValue();
jbone:     if (s4 != null) runData.numSchem=Integer.valueOf(s4).intValue();
jbone:     for (Enumeration e = runData.getKeys(); e.hasMoreElements(); ) {
jbone:       String key = e.nextElement().toString();
jbone:       String name = key.substring(0,6);
jbone:       String value= getParameter(name);
jbone:       if (value != null) runData.set(key,value);
jbone:     }
jbone:   }
jbone:
jbone:
jbone:   /** Responds to the user actions through the mouse and buttons (Java1.0).
jbone:       @deprecated
jbone:   *****************************************************************************/
jbone:   public boolean action(Event e, Object arg) {
jbone:     boolean ret=true;
jbone:
jbone:     if (e.target instanceof Button) {
jbone:      if (((String)arg).equals(Data.START))          {frozen=!frozen;
jbone:       } else if (((String)arg).equals(Data.STEP))   {frozen=false;nstep=1;
jbone:       } else if (((String)arg).equals(Data.STEPA)) {frozen=false;nstep=10;
jbone:       } else if (((String)arg).equals(Data.DISPL)){setScale=true;header=!header;
jbone:       } else if (((String)arg).equals(Data.INIT))   {initialize=true;
jbone:       } else { ret=false;
jbone:       }
jbone:      if ((!frozen || initialize) && runThread != null)
jbone:        { runThread = new Thread(this); runThread.start(); }
jbone:
jbone:     } else if (e.target instanceof Choice) {
jbone:       int oldIcType = runData.icType;
jbone:       if (((String)arg).equals(Data.ADVECTION))        { runData.pdeType=0;
jbone:       } else if (((String)arg).equals(Data.BURGER))    { runData.pdeType=1;
jbone:       } else if (((String)arg).equals(Data.KDV))       { runData.pdeType=2;
jbone:       } else if (((String)arg).equals(Data.BSCHOLES)) { runData.pdeType=3;
jbone:       } else if (((String)arg).equals(Data.BOX))       { runData.icType=0;
jbone:       } else if (((String)arg).equals(Data.GAUSSIAN)) { runData.icType=1;
jbone:       } else if (((String)arg).equals(Data.COSINUS))   { runData.icType=2;
jbone:       } else if (((String)arg).equals(Data.SOLITON))   { runData.icType=3;
jbone:       } else if (((String)arg).equals(Data.CALL))      { runData.icType=4;
jbone:       } else if (((String)arg).equals(Data.PUT))       { runData.icType=5;
jbone:       } else if (((String)arg).equals(Data.FD))        { runData.numMeth=0;
jbone:           runData.numSchem=0;
jbone:           ((CardLayout)shCards.getLayout()).show(shCards,Data.FD);
jbone:           window.SetNumericalFDScheme(runData.numSchem);
jbone:       } else if (((String)arg).equals(Data.FEM))       { runData.numMeth=1;
```

```
jbone:            runData.numSchem=0;
jbone:            ((CardLayout)shCards.getLayout()).show(shCards,Data.FEM);
jbone:            window.SetNumericalFEMScheme(runData.numSchem);
jbone:        } else if (((String)arg).equals(Data.FFT))       { runData.numMeth=2;
jbone:            runData.numSchem=0;
jbone:            ((CardLayout)shCards.getLayout()).show(shCards,Data.FFT);
jbone:            window.SetNumericalFFTScheme(runData.numSchem);
jbone:        } else if (((String)arg).equals(Data.MCM))       { runData.numMeth=3;
jbone:            runData.numSchem=1;
jbone:            ((CardLayout)shCards.getLayout()).show(shCards,Data.MCM);
jbone:            window.SetNumericalMCMScheme(runData.numSchem);
jbone:        } else if (((String)arg).equals(Data.CHA))       { runData.numMeth=4;
jbone:            runData.numSchem=0;
jbone:            ((CardLayout)shCards.getLayout()).show(shCards,Data.CHA);
jbone:            window.SetNumericalCHAScheme(runData.numSchem);
jbone:        } else if (((String)arg).equals(Data.DEFAULT))   { runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.EXP3LVL))   { runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.EXP2LVL))   { runData.numSchem=1;
jbone:        } else if (((String)arg).equals(Data.IMP2LVL))   { runData.numSchem=2;
jbone:        } else if (((String)arg).equals(Data.EXPLAX))    { runData.numSchem=3;
jbone:        } else if (((String)arg).equals(Data.IMPLAX))    { runData.numSchem=4;
jbone:        } else if (((String)arg).equals(Data.LPFROG))    { runData.numSchem=5;
jbone:        } else if (((String)arg).equals(Data.FDMINE))    { runData.numSchem=6;
jbone:        } else if (((String)arg).equals(Data.FDEX21))    { runData.numSchem=7;
jbone:        } else if (((String)arg).equals(Data.FDEX23))    { runData.numSchem=8;
jbone:        } else if (((String)arg).equals(Data.FDEX24))    { runData.numSchem=9;
jbone:        } else if (((String)arg).equals(Data.FDEX25))    { runData.numSchem=10;
jbone:        } else if (((String)arg).equals(Data.TUNFEM))    { runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.FEMMINE))   { runData.numSchem=1;
jbone:        } else if (((String)arg).equals(Data.FEMEX32))   { runData.numSchem=2;
jbone:        } else if (((String)arg).equals(Data.FEMEX33))   { runData.numSchem=3;
jbone:        } else if (((String)arg).equals(Data.FEMEX34))   { runData.numSchem=4;
jbone:        } else if (((String)arg).equals(Data.FEMEX35))   { runData.numSchem=5;
jbone:        } else if (((String)arg).equals(Data.EXPAND))    { runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.ALIASED))   { runData.numSchem=1;
jbone:        } else if (((String)arg).equals(Data.FFTMINE))   { runData.numSchem=2;
jbone:        } else if (((String)arg).equals(Data.FFTEX41))   { runData.numSchem=3;
jbone:        } else if (((String)arg).equals(Data.FFTEX42))   { runData.numSchem=4;
jbone:        } else if (((String)arg).equals(Data.CUBFEM))    { runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.CUBSPL))    { runData.numSchem=1;
jbone:        } else if (((String)arg).equals(Data.CHAMINE))   { runData.numSchem=2;
jbone:        } else if (((String)arg).equals(Data.CHAEX61))   { runData.numSchem=3;
jbone:        } else if (((String)arg).equals(Data.CHAEX62))   { runData.numSchem=4;
jbone:        } else if (((String)arg).equals(Data.CHAEX64))   { runData.numSchem=5;
jbone:        } else if (((String)arg).equals(Data.NPARTICLE1)){ runData.numSchem=0;
jbone:        } else if (((String)arg).equals(Data.NPARTICLE100)){ runData.numSchem=1;
jbone:        } else if (((String)arg).equals(Data.NPARTICLE1000)){ runData.numSchem=2;
jbone:        } else if (((String)arg).equals(Data.NPARTICLE10000)){runData.numSchem=3;
jbone:        } else if (((String)arg).equals(Data.MCMMINE))  { runData.numSchem=4;
jbone:        } else { ret=false;
jbone:        }
jbone:        if(ret){
jbone:            initialize=true;
jbone:            if(oldIcType == runData.icType) keepFunction = true;
jbone:        } // if(ret)
jbone:
jbone:       if ((!frozen || initialize) && runThread != null)
jbone:         { runThread = new Thread(this); runThread.start(); }
jbone:
jbone:      } else if (e.target instanceof List) {       //Set up an editor for dialog
jbone:          if (editor == null) {
jbone:              editor = new MyEditor((String)arg, window);
jbone:          }
jbone:          editor.show();
jbone:
jbone:      } else { ret=false;
jbone:      }
jbone:      if (ret == true) { }                         //Update GUIWindow
jbone:      return ret;
jbone:    }
jbone:
jbone:    /** Information
jbone:    ********************************************************************************/
jbone:    public String getAppletInfo() {
jbone:      return "JBONE - an educational software by A. Jaun and J. Hedin";
jbone:    }
jbone:
jbone:
jbone:    /** Executed only if the program runs as a stand alone application
jbone:    ********************************************************************************/
jbone:    public static void main(String[] args) {
jbone:
jbone:      //Create a jbone instance and initialize it.
jbone:      jbone jbo = new jbone();
jbone:      jbo.isAnApplet = false;
jbone:      jbo.init();
jbone:
jbone:      //Create a new window to draw the applet.
jbone:      MainFrame f = new MainFrame("JBONE application");
jbone:      f.add("Center", jbo); f.pack(); f.show();
jbone:
jbone:      //Start the calculation
jbone:      jbo.start();
jbone:    }
jbone:
jbone:    /*******************************************************************************
jbone:    * GUI Window - All what you can finally see on the screen
jbone:    ********************************************************************************/
jbone:    public class GUIWindow {
jbone:
jbone:      Choice sha, shb, shc, shd, she;
jbone:      List params;
jbone:
jbone:       /** Default constructor */
jbone:       public GUIWindow() {
jbone:        GridBagLayout gb = new GridBagLayout();
jbone:        GridBagConstraints c = new GridBagConstraints();
jbone:
jbone:        setFont(new Font("Fixed", Font.PLAIN, 13));
jbone:        setLayout(gb);
jbone:
jbone:        c.gridwidth = 1; c.gridheight = 1;
jbone:        c.weightx= 0.0;  c.weighty= 0.0;  c.fill= GridBagConstraints.HORIZONTAL;
jbone:        Choice pde=new Choice();pde=runData.button(Data.BUT_PDE);gbAddC(gb,c,pde);
jbone:        Choice ic =new Choice();ic =runData.button(Data.BUT_IC); gbAddC(gb,c,ic);
jbone:        Choice mth=new Choice();mth=runData.button(Data.BUT_METH);gbAddC(gb,c,mth)
jbone:        pde.select(runData.pdeType);ic.select(runData.icType);
jbone:        mth.select(runData.numMeth);
jbone:
jbone:        shCards = new Panel(); shCards.setLayout(new CardLayout());
jbone:        sha = new Choice();sha=runData.button(Data.BUT_FD);
jbone:        shb = new Choice();shb=runData.button(Data.BUT_FEM);
jbone:        shc = new Choice();shc=runData.button(Data.BUT_FFT);
jbone:        shd = new Choice();shd=runData.button(Data.BUT_MCM);
jbone:        she = new Choice();she=runData.button(Data.BUT_CHA);
jbone:
jbone:        //Need somehow to add relevant scheme first independently of sh.select()
jbone:        if        (runData.numMethIs(Data.FD))
jbone:          { shCards.add(Data.FD ,sha); shCards.add(Data.FEM,shb);
jbone:            shCards.add(Data.FFT,shc); shCards.add(Data.MCM,shd);
jbone:            shCards.add(Data.CHA,she); sha.select(runData.numSchem);
jbone:        } else if (runData.numMethIs(Data.FEM))
jbone:          { shCards.add(Data.FEM,shb); shCards.add(Data.FD ,sha);
jbone:            shCards.add(Data.FFT,shc); shCards.add(Data.MCM,shd);
jbone:            shCards.add(Data.CHA,she); shb.select(runData.numSchem);
jbone:        } else if (runData.numMethIs(Data.FFT))
jbone:          { shCards.add(Data.FFT,shc); shCards.add(Data.FD ,sha);
jbone:            shCards.add(Data.FEM,shb); shCards.add(Data.MCM,shd);
jbone:            shCards.add(Data.CHA,she); shc.select(runData.numSchem);
jbone:        } else if (runData.numMethIs(Data.MCM))
jbone:          { shCards.add(Data.MCM,shd); shCards.add(Data.FD ,sha);
jbone:            shCards.add(Data.FEM,shb); shCards.add(Data.FFT,shc);
jbone:            shCards.add(Data.CHA,she); shd.select(runData.numSchem);
jbone:        } else if (runData.numMethIs(Data.CHA))
jbone:          { shCards.add(Data.CHA,she); shCards.add(Data.FD ,sha);
jbone:            shCards.add(Data.FEM,shb); shCards.add(Data.FFT,shc);
jbone:            shCards.add(Data.MCM,shd); she.select(runData.numSchem);
jbone:        }
jbone:        gb.setConstraints(shCards,c);add(shCards);
jbone:
jbone:        c.weightx = 1.0; c.gridwidth = GridBagConstraints.REMAINDER; //end of row
jbone:        Choice sel=new Choice(); sel=runData.button(Data.BUT_PARSET);
jbone:        gbAddC(gb,c,sel);
jbone:
jbone:        c.gridwidth = 4; c.gridheight = 1; c.weightx= 0.0;  c.weighty= 1.0;
jbone:        c.fill = GridBagConstraints.BOTH;
jbone:        plot = new MyCanvas(); gbAddP(gb,c,plot);
jbone:
jbone:        c.gridwidth = GridBagConstraints.REMAINDER; //end of row
jbone:        params = new List(runData.nbrParams(),false);
jbone:        runData.List(params);gbAddL(gb,c,params);
jbone:
jbone:        c.gridwidth = 1; c.gridheight = 1; c.weightx= 0.0;  c.weighty= 0.0;
jbone:        c.fill = GridBagConstraints.HORIZONTAL;
jbone:        Button b_start = new Button(Data.START); gbAddB(gb,c,b_start);
jbone:        Button b_step  = new Button(Data.STEP);  gbAddB(gb,c,b_step);
jbone:        Button b_stepa = new Button(Data.STEPA); gbAddB(gb,c,b_stepa);
jbone:        Button b_displ = new Button(Data.DISPL); gbAddB(gb,c,b_displ);
jbone:        c.weightx = 1.0; c.gridwidth = GridBagConstraints.REMAINDER; //end of row
jbone:        Button b_init  = new Button(Data.INIT);  gbAddB(gb,c,b_init);
jbone:       }
jbone:
jbone:      //Supposed to make it easier to read
jbone:      protected void gbAddB(GridBagLayout gb, GridBagConstraints c, Button cmp)
jbone:        { gb.setConstraints(cmp,c); add(cmp); }
jbone:      protected void gbAddC(GridBagLayout gb, GridBagConstraints c, Choice cmp)
jbone:        { gb.setConstraints(cmp,c); add(cmp); }
jbone:      protected void gbAddL(GridBagLayout gb, GridBagConstraints c, List cmp)
jbone:        { gb.setConstraints(cmp,c); add(cmp); }
jbone:      protected void gbAddP(GridBagLayout gb, GridBagConstraints c, MyCanvas cmp)
jbone:        { gb.setConstraints(cmp,c); add(cmp); }
jbone:      public Dimension minimumSize() { return new Dimension(400,200); }
jbone:      public Dimension preferredSize() { return minimumSize(); }
jbone:
jbone:      public void SetNumericalFDScheme(int scheme){
jbone:        sha.select(scheme);
jbone:      } // SetNumericalFDScheme
jbone:      public void SetNumericalFEMScheme(int scheme){
jbone:        shb.select(scheme);
jbone:      } // SetNumericalFEMScheme
jbone:      public void SetNumericalFFTScheme(int scheme){
jbone:        shc.select(scheme);
jbone:      } // SetNumericalFFTScheme
jbone:      public void SetNumericalMCMScheme(int scheme){
jbone:        shd.select(scheme);
jbone:      } // SetNumericalMCMScheme
jbone:      public void SetNumericalCHAScheme(int scheme){
jbone:        she.select(scheme);
jbone:      } // SetNumericalCHAScheme
jbone:
jbone:      public void UpdateParameters(Data runData){
jbone:        params.clear();
jbone:        runData.List(params);
jbone:      } // UpdateParameters
jbone:
jbone:    }//End of GUIWindow
jbone:
jbone:    /** The plot area
jbone:    ********************************************************************************/
jbone:    public class MyCanvas extends Canvas {
jbone:      StringBuffer lbl = new StringBuffer(100);
jbone:      int      n;              //Number of mesh points
jbone:      double[] cx,   cy;       //Current mesh+function coordinates
jbone:      double[] limx,  limy;    //Remember interval limits for normalization
jbone:      double[]        climy;   //Current interval limits for labeling
jbone:      double   mom;            //Current value of momentsDeviation[0]
jbone:      int[]    x,    y;        //Remeber (plot area) coodinates for erasing
jbone:      int      xoffs, yoffs;
jbone:      double   xsca = 0.9;     //Horizontal and vertical scaling ratio
jbone:      double   ysca = 0.7;
jbone:      double   xscal, yscal;   //Current normalization
jbone:
jbone:      public void paint(Graphics g1) {          //Total repaint only when frozen
jbone:        if (frozen) { plot.repaint(); }
jbone:      }
jbone:
jbone:      public void print(Graphics g1){
jbone:        n = solution.size();
jbone:        cx = mesh.points();
jbone:        x = new int[n];
jbone:        y= new int[n];
jbone:        for (int i=0; i<n; i++) {
jbone:          x[i]=(int)((cx[i]-limx[0])*xscal) +xoffs;
jbone:          y[i]=(int)((solution.getValue(i)-limy[0])*yscal) +yoffs;
jbone:          g1.fillRect(x[i]-1,y[i]-1,3,3);
jbone:        } /* for */
jbone:        try                          {g1.drawPolyline(x, y, n);}        //Java1.1
jbone:        catch (NoSuchMethodError e) {g1.drawPolygon(x, y, n);}          //Java1.0
jbone:      } /* print */
jbone:
jbone:      public void update(Graphics g1) {        //Partial update avoids flickering
jbone:
jbone:        if(solution == null){
jbone:          return;
jbone:        } // if
jbone:        //Get plot dimensions
jbone:        Dimension d = new Dimension();
jbone:        try                          { d=getSize();}                    //Java1.1
jbone:        catch (NoSuchMethodError e) { d =  size();}                     //Java1.0
jbone:        int w=d.width;                    int h=d.height;
jbone:        xoffs=(int)(0.5*(1.-xsca)*w);     yoffs=(int)(h-0.33*(1.-ysca)*h);
jbone:
jbone:        if (initialize || x == null || y == null) {       //Instanciate plot area
jbone:           x= null;  y= null;  limx= null;  limy= null;
jbone:          cx= null; cy= null;               climy= null;
jbone:          n=solution.size();
jbone:          limx= new double[2]; limy= new double[2]; climy=new double[2];
jbone:          cx  = new double[n];   cy= new double[n];
```

```
jbone:          x  = new int[n];       y= new int[n];
jbone:        g1.setColor(getBackground()); g1.fillRect(2,2,w-2,h-2);
jbone:        g1.setColor(Color.black);
jbone:      }
jbone:
jbone:      else {      //Avoid flickering by redrawing plot in getBackground() color
jbone:        g1.setColor(getBackground()); g1.fillRect(1,1,w-2,18);
jbone:        for (int i=0; i<n; i++) { g1.fillRect(x[i]-1,y[i]-1,3,3); }
jbone:        try                          {g1.drawPolyline(x, y, n);}    //Java1.1
jbone:        catch (NoSuchMethodError e) {g1.drawPolygon(x, y, n);}     //Java1.0
jbone:        g1.setColor(Color.black);
jbone:        n=solution.size();
jbone:      }
jbone:      g1.drawRect(0, 0, w - 1, h - 1);
jbone:
jbone:      if (initialize || setScale) {
jbone:        limx= mesh.limits(); limy=solution.limits();
jbone:                                   xscal=xsca*(w-0)/(limx[1]-limx[0]);
jbone:        if (limy[1]-limy[0] != 0.) {yscal=ysca*(0-h)/(limy[1]-limy[0]);};
jbone:        setScale=false;
jbone:      }
jbone:
jbone:      if (header) {
jbone:        DecimalFormat myDoubleFormatter = new DecimalFormat(".000");
jbone:        DecimalFormat myIntFormatter = new DecimalFormat("###");
jbone:        String lbl1b= myIntFormatter.format(step);
jbone:        String lbl1c= myDoubleFormatter.format(
jbone:                                step*runData.getDouble(Data.TIMSTP));
jbone:        String lbl1 = "Step="+lbl1b+"   Time="+lbl1c;
jbone:
jbone:        mom   = 100.*solution.momentsDeviation(0);
jbone:        climy = solution.limits();
jbone:        String lbl2a= myDoubleFormatter.format(climy[0]);
jbone:        String lbl2b= myDoubleFormatter.format(climy[1]);
jbone:        String lbl2c;
jbone:        if (mom<0) { lbl2c= myDoubleFormatter.format(mom);
jbone:        } else { lbl2c= " "+myDoubleFormatter.format(mom); }
jbone:        String lbl2="   Min="+lbl2a+"   Max="+lbl2b+"   Mom[%]="+lbl2c;
jbone:        g1.drawString(lbl1+lbl2,5,14);
jbone:      } else {
jbone:      }
jbone:
jbone:        cx= mesh.points();
jbone:        for (int i=0; i<n; i++) {
jbone:          x[i]=(int)((cx[i]-limx[0])*xscal) +xoffs;
jbone:          y[i]=(int)((solution.getValue(i)-limy[0])*yscal) +yoffs;
jbone:          g1.fillRect(x[i]-1,y[i]-1,3,3);
jbone:        }
jbone:        try                          {g1.drawPolyline(x, y, n);}          //Java1.1
jbone:        catch (NoSuchMethodError e) {g1.drawPolygon(x, y, n);}          //Java1.0
jbone:      }
jbone:
jbone:    } //End of MyCanvas
jbone:
jbone:    /*************************************************************************
jbone:    * Dialog allowing the user to change the run parameters
jbone:    *************************************************************************/
jbone:    class MyEditor extends Frame {   //Dialog s do not work properly with Java1.0
jbone:      TextField field;
jbone:      Button setButton;
jbone:      GUIWindow window;
jbone:
jbone:      /** Opens a dialog to modify a run parameter
jbone:      *************************************************************************/
jbone:      MyEditor(String param, GUIWindow inWindow) {
jbone:        window = inWindow;
jbone:        int    index = param.indexOf("=");
jbone:        String key   = param.substring(0,index);
jbone:        String val   = runData.getString(key);
jbone:        String edit  = key+"="+val;
jbone:
jbone:        Panel p1 = new Panel();
jbone:        Label label = new Label("Enter new parameter:");  p1.add(label);
jbone:        field = new TextField(25); field.setText(edit);
jbone:        p1.add(field); add("Center", p1);
jbone:
jbone:        Panel p2 = new Panel();
jbone:        p2.setLayout(new FlowLayout(FlowLayout.RIGHT));
jbone:        Button cancelButton = new Button("Cancel");
jbone:        setButton = new Button("Set");
jbone:        p2.add(cancelButton); p2.add(setButton); add("South", p2);
jbone:
jbone:        pack();  //Initialize this dialog to its preferred size.
jbone:      }
jbone:
jbone:        /** Double-click on parameter requires editing
jbone:            @deprecated
jbone:        *************************************************************************/
jbone:        public boolean action(Event event, Object arg) {
jbone:          if ( (event.target == setButton)
jbone:              | (event.target == field)) {
jbone:            runData.dialogModify(field.getText());
jbone:            window.UpdateParameters(runData);
jbone:            field.selectAll();
jbone:          }
jbone:          hide(); editor = null;
jbone:          return true;
jbone:        }
jbone:
jbone:    } //End of MyEditor
jbone:
jbone: } //End of Applet jbone
jbone:
jbone:
jbone: /*************************************************************************
jbone: *  Provides a window if this program is run as an application.
jbone: *************************************************************************/
jbone: class MainFrame extends Frame {
jbone:   MainFrame(String title) {
jbone:     super(title);
jbone:   }
jbone:
jbone:   /** Destroy application
jbone:       @deprecated
jbone:   *************************************************************************/
jbone:   public boolean handleEvent(Event e) {                          //Java1.0
jbone:     if (e.id == Event.WINDOW_DESTROY) { System.exit(0); }
jbone:     return super.handleEvent(e);
jbone:   }
jbone: }
Data:
Data:
Data: /* $Id: Data.java,v 1.10.4.5 1999/08/05 11:48:13 andrej Exp $ */
Data:
Data: import java.util.*;          //Hashtable
Data: import java.awt.*;           //Choice, List
Data: import java.applet.Applet;   //getParameter()
Data:
Data: /*************************************************************************
```

```
Data: * Initializes and manage the run parameters
Data: *************************************************************************/
Data: public class Data {
Data:
Data:   //====== Control Variables
Data:   /** Type of equation*/              public int pdeType      = 0;
Data:   /** Type of initial condition*/     public int icType       = 0;
Data:   /** Type of method (FD,FEM,FFT)*/   public int numMeth     = 0;
Data:   /** Type of scheme (ex-/implicit)*/ public int numSchem     = 0;
Data:
Data:   //====== Definitions
Data:   /** Label*/ final static String ADVECTION = "Advection";
Data:   /** Label*/ final static String BURGER    = "Burger (shock)";
Data:   /** Label*/ final static String KDV       = "KdV (solitons)";
Data:   /** Label*/ final static String BSCHOLES  = "Black-Scholes";
Data:   /** List*/  static String[] pdeNames =
Data:                                {ADVECTION,BURGER,KDV,BSCHOLES};
Data:   /** Label*/ final static String BOX       = "Box";
Data:   /** Label*/ final static String GAUSSIAN  = "Gaussian";
Data:   /** Label*/ final static String COSINUS   = "Cosine";
Data:   /** Label*/ final static String SOLITON   = "Soliton";
Data:   /** Label*/ final static String CALL      = "Call";
Data:   /** Label*/ final static String PUT       = "Put";
Data:   /** List*/  static String[] icNames =
Data:                                {BOX,GAUSSIAN,COSINUS,SOLITON,CALL,PUT};
Data:   /** Label*/ final static String FD        = "Finite differences";
Data:   /** Label*/ final static String FEM       = "Finite elements";
Data:   /** Label*/ final static String FFT       = "Fourier transform";
Data:   /** Label*/ final static String MCM       = "Monte-Carlo";
Data:   /** Label*/ final static String CHA       = "Lagrangian";
Data:   /** List*/  static String[] numMethNames  = {FD,FEM,FFT,MCM,CHA};
Data:
Data:   /** Label*/ final static String DEFAULT   = "Standard  scheme";
Data:   /** Label*/ final static String EXP2LVL   = "Explicit 2-level";
Data:   /** Label*/ final static String EXP3LVL   = "Explicit 3-level";
Data:   /** Label*/ final static String IMP2LVL   = "Implicit 2-level";
Data:   /** Label*/ final static String EXPLAX    = "Expl-LaxWendroff";
Data:   /** Label*/ final static String IMPLAX    = "Impl-LaxWendroff";
Data:   /** Label*/ final static String LPFROG    = "Leap-frog (FDTD)";
Data:   /** Label*/ final static String FDMINE    = "My FD scheme    ";
Data:   /** Label*/ final static String FDEX21    = "FD Exercise 2.1";
Data:   /** Label*/ final static String FDEX23    = "FD Exercise 2.3";
Data:   /** Label*/ final static String FDEX24    = "FD Exercise 2.4";
Data:   /** Label*/ final static String FDEX25    = "FD Exercise 2.5";
Data:   /** Label*/ final static String TUNFEM    = "Tunable Integration";
Data:   /** Label*/ final static String FEMMINE   = "My FEM scheme   ";
Data:   /** Label*/ final static String FEMEX32   = "FEM Exercise 3.2";
Data:   /** Label*/ final static String FEMEX33   = "FEM Exercise 3.3";
Data:   /** Label*/ final static String FEMEX34   = "FEM Exercise 3.4";
Data:   /** Label*/ final static String FEMEX35   = "FEM Exercise 3.5";
Data:   /** Label*/ final static String EXPAND    = "Expanded  convolution";
Data:   /** Label*/ final static String ALIASED   = "Aliased convolution";
Data:   /** Label*/ final static String FFTMINE   = "My FFT scheme   ";
Data:   /** Label*/ final static String FFTEX41   = "FFT Exercise 4.1";
Data:   /** Label*/ final static String FFTEX42   = "FFT Exercise 4.2";
Data:   /** Label*/ final static String NPARTICLE1    = "1 particle";
Data:   /** Label*/ final static String NPARTICLE100  = "100 particles";
Data:   /** Label*/ final static String NPARTICLE1000 = "1000 particles";
Data:   /** Label*/ final static String NPARTICLE10000= "10000 particles";
Data:   /** Label*/ final static String MCMMINE   = "My MCM scheme    ";
Data:   /** Label*/ final static String MCMEX53   = "MCM Exercise 5.3";
Data:   /** Label*/ final static String MCMEX54   = "MCM Exercise 5.4";
Data:   /** Label*/ final static String MCMEX55   = "MCM Exercise 5.5";
Data:   /** Label*/ final static String CUBFEM    = "CubicHermite FEM";
Data:   /** Label*/ final static String CUBSPL    = "Cubic -- Splines";
Data:   /** Label*/ final static String CHAMINE   = "My CHA scheme   ";
Data:   /** Label*/ final static String CHAEX61   = "CHA Exercise 6.1";
Data:   /** Label*/ final static String CHAEX62   = "CHA Exercise 6.2";
Data:   /** Label*/ final static String CHAEX64   = "CHA Exercise 6.4";
Data:   /** List*/  static String[] numSchemFD  = {EXP3LVL,EXP2LVL,IMP2LVL,
Data:                                              EXPLAX,IMPLAX,LPFROG,FDMINE,
Data:                                              FDEX21,FDEX23,FDEX24,FDEX25};
Data:   /** List*/  static String[] numSchemFEM = {TUNFEM,FEMMINE,
Data:                                              FEMEX32,FEMEX33,FEMEX34,FEMEX35};
Data:   /** List*/  static String[] numSchemFFT = {EXPAND,ALIASED,FFTMINE,
Data:                                              FFTEX41,FFTEX42};
Data:   /** List*/  static String[] numSchemMCM = {NPARTICLE1, NPARTICLE100,
Data:                                         NPARTICLE1000,NPARTICLE10000,MCMMINE,
Data:                                              MCMEX53,MCMEX54,MCMEX55};
Data:   /** List*/  static String[] numSchemCHA = {CUBFEM,CUBSPL,CHAMINE,
Data:                                              CHAEX61,CHAEX62,CHAEX64};
Data:   //====== Buttons for events
Data:   /** Button aspect*/ final static String START = "  START/STOP  ";
Data:   /** Button aspect*/ final static String STEP  = "    STEP 1    ";
Data:   /** Button aspect*/ final static String STEPA = "      STEP  10      ";
Data:   /** Button aspect*/ final static String DISPL = "  TOGGLE DISPLAY  ";
Data:   /** Button aspect*/ final static String INIT  = "INITIALIZE";
Data:
Data:   //====== Run parameter names
Data:   /** Label dataset nbr 1*/ final static String SET1 = "Preset #1";
Data:   /** Preset data nbr 1  */ static String[] setParamNames = {SET1};
Data:   /** Final time to reach*/ final static String RUNTIM = "RUNTIM Run time    ";
Data:   /** Advection velocity */ final static String VELCTY = "VELCTY Velocity    ";
Data:   /** Diffusion constant */ final static String DIFFUS = "DIFFUS Diffusion   ";
Data:   /** Dispersion constant*/ final static String DISPER = "DISPER Dispersion  ";
Data:   /** Spatial resolution */ final static String NPTS   = "NPTS   Mesh points ";
Data:   /** Interval duration  */ final static String TIMSTP = "TIMSTP Time step   ";
Data:   /** Implicit time param*/ final static String THETA  = "THETA  TimeImplicit";
Data:   /** FEM tune integration*/final static String TUNINT = "TUNINT Tune integr ";
Data:   /** IC amplitude pulse1*/ final static String IC1AMP = "IC1AMP Amplitude   ";
Data:   /** IC position pulse1 */ final static String IC1POS = "IC1POS Position    ";
Data:   /** IC width pulse1    */ final static String IC1WID = "IC1WID Width       ";
Data:   /** Left coordinate    */ final static String BOX0   = "BOX0   Box left    ";
Data:   /** Length of domain   */ final static String BOXLEN = "BOXLEN Box length  ";
Data:   /** Inhomogeneous mesh */ final static String PCKDEN = "PCKDEN PackDensity ";
Data:   /** Inhomogeneous mesh */ final static String PCKPOS = "PCKPOS PackPosition";
Data:   /** Inhomogeneous mesh */ final static String PCKWID = "PCKWID PackWidth   ";
Data:   /** Param list*/  static StringBuffer[] runParamNames = new StringBuffer[30];
Data:
Data:   //====== The collections of all the choices
Data:   Selection pdeTypes  = new Selection(pdeNames.length);
Data:   Selection icTypes   = new Selection(icNames.length);
Data:   Selection numMeths  = new Selection(numMethNames.length);
Data:   Selection numSchemsFD = new Selection(numSchemFD.length);
Data:   Selection numSchemsFEM= new Selection(numSchemFEM.length);
Data:   Selection numSchemsFFT= new Selection(numSchemFFT.length);
Data:   Selection numSchemsMCM= new Selection(numSchemMCM.length);
Data:   Selection numSchemsCHA= new Selection(numSchemCHA.length);
Data:   Selection setParams = new Selection(setParamNames.length);
Data:   Hashtable runParams = new Hashtable(30,0.1f);
Data:
Data:
Data:   /** Instanciate an entire parameter set and preset to default values
Data:     @see RUNTIM
Data:     @see VELCTY
Data:     @see DIFFUS
Data:     @see DISPER
Data:     @see NPTS
Data:     @see TIMSTP
```

```
Data:     @see THETA
Data:     @see TUNINT
Data:     @see IC1AMP
Data:     @see IC1POS
Data:     @see IC1WID
Data:     @see BOX0
Data:     @see BOXLEN
Data:     @see PCKDEN
Data:     @see PCKPOS
Data:     @see PCKWID
Data:   ************************************************************************/
Data:   public Data() {
Data:     int i;
Data:                                                          //Control variables
Data:
Data:                                                          //Default values
Data:     for (i=0; i<30; i++) {runParamNames[i]= new StringBuffer(20);};i=0;
Data:     runParamNames[i++].append(TIMSTP); runParams.put(TIMSTP,"    0.5");
Data:     runParamNames[i++].append(VELCTY); runParams.put(VELCTY,"     1.");
Data:     runParamNames[i++].append(DIFFUS); runParams.put(DIFFUS,"     0."); //0.8
Data:     runParamNames[i++].append(DISPER); runParams.put(DISPER,"    0.5"); //0.5
Data:     runParamNames[i++].append(RUNTIM); runParams.put(RUNTIM,"   256.");
Data:     runParamNames[i++].append(THETA ); runParams.put(THETA ,"    0.7");
Data:     runParamNames[i++].append(TUNINT); runParams.put(TUNINT," 0.3333");
Data:     runParamNames[i++].append(IC1AMP); runParams.put(IC1AMP,"     1.");
Data:     runParamNames[i++].append(IC1POS); runParams.put(IC1POS,"    18.");
Data:     runParamNames[i++].append(IC1WID); runParams.put(IC1WID,"     8.");
Data:     runParamNames[i++].append(NPTS  ); runParams.put(NPTS  ,"    64 ");
Data:     runParamNames[i++].append(BOXLEN); runParams.put(BOXLEN,"    64.");
Data:     runParamNames[i++].append(BOX0  ); runParams.put(BOX0  ,"     0.");
Data:
Data:                                                          //Default selectors
Data:     for (i=0; i<pdeNames.length; i++)
Data:       { pdeTypes.put(pdeNames[i],pdeNames[i].equals(ADVECTION)); }
Data:     for (i=0; i<icNames.length; i++)
Data:       { icTypes.put(icNames[i],icNames[i].equals(GAUSSIAN)); }
Data:     for (i=0; i<numMethNames.length; i++)
Data:       { numMeths.put(numMethNames[i],numMethNames[i].equals(FD)); }
Data:
Data:                                                          //Default schemes
Data:     for (i=0; i<numSchemFD.length; i++)
Data:       { numSchemsFD.put(numSchemFD[i],i==0); }
Data:     for (i=0; i<numSchemFEM.length; i++)
Data:       { numSchemsFEM.put(numSchemFEM[i],i==0); }
Data:     for (i=0; i<numSchemFFT.length; i++)
Data:       { numSchemsFFT.put(numSchemFFT[i],i==0); }
Data:     for (i=0; i<numSchemMCM.length; i++)
Data:       { numSchemsMCM.put(numSchemMCM[i],i==0); }
Data:     for (i=0; i<numSchemCHA.length; i++)
Data:       { numSchemsCHA.put(numSchemCHA[i],i==0); }
Data:
Data:     for (i=0; i<setParamNames.length; i++)
Data:       { setParams.put(setParamNames[i],i==0); }
Data:   }
Data:
Data:
Data:   /** Modify a parameter in an editor dialog if required by an action
Data:   ************************************************************************/
Data:   public void dialogModify(Object arg) {
Data:     for (Enumeration e = runParams.keys(); e.hasMoreElements(); ) {
Data:       String key   = e.nextElement().toString();
Data:       String table = key.substring(0,6);
Data:       String edit  = ((String)arg);
Data:       String var   = edit.substring(0,6);
Data:       if (var.equals(table)) {
Data:         int    index = edit.indexOf("=");
Data:         String val   = edit.substring(index+1);
Data:         runParams.put(key,val);
Data:         break;
Data:       }
Data:     }
Data:   }
Data:
Data:
Data:   /** GUI List of run parameters with th... ve ...
Data:   ************************************************************************/
Data:   public void List(List list) {
Data:     for (int i=0; i<nbrParams(); i+...
Data:       String name = runParamNames[i]...
Data:       String value= runParams.get(nam... ...
Data:       list.addItem(name +"="+ value);
Data:     }
Data:   }
Data:
Data:
Data:   public String pdeName(int type)       { return pdeNames[type]; }
Data:   public String icName(int type)        { return icNames[type]; }
Data:   public String numMethName(int type)   { return numMethNames[type]; }
Data:   public String FDSchemName(int type)   { return numSchemFD[type]; }
Data:   public String FEMSchemName(int type)  { return numSchemFEM[type]; }
Data:   public String FFTSchemName(int type)  { return numSchemFFT[type]; }
Data:   public String CHASchemName(int type)  { return numSchemCHA[type]; }
Data:   public String MCMSchemName(int type)  { return numSchemMCM[type]; }
Data:
Data:   public boolean pdeIs(String type)
Data:     { return pdeNames[pdeType].equals(type); }
Data:
Data:   public boolean icIs(String type)
Data:     { return icNames[icType].equals(type); }
Data:
Data:   public boolean numMethIs(String type)
Data:     { return numMethNames[numMeth].equals(type); }
Data:
Data:
Data:   /** Number of parameters currently stored in the hash table*/
Data:   public int nbrParams() { return runParams.size(); }
Data:
Data:   /** Set a parameters in the hash table to a given value*/
Data:   public void set(String name, String value) { runParams.put(name,value); }
Data:
Data:   /** Name of a parameters in the hash table*/
Data:   public String getString(String name)
Data:     { return runParams.get(name).toString().trim(); }
Data:
Data:   /** Enumerate all the keys contained the hash table*/
Data:   public Enumeration getKeys()
Data:     { return runParams.keys(); }
Data:
Data:   /** Get an integer value from the hash table*/
Data:   public int getInt(String name)
Data:     { return Integer.valueOf(runParams.get(name)
Data:                                .toString().trim()).intValue(); }
Data:
Data:   /** Get a double value from the hash table*/
Data:   public double getDouble(String name)
Data:     { return Double.valueOf(runParams.get(name)
Data:                                .toString().trim()).doubleValue(); }
Data:
Data:
```

```
Data:
Data:   /** Button ID*/          final static int BUT_PDE   = 0;
Data:   /** Button ID*/          final static int BUT_IC    = 1;
Data:   /** Button ID*/          final static int BUT_METH  = 2;
Data:   /** Button ID*/          final static int BUT_PARSET= 3;
Data:   /** Button ID*/          final static int BUT_FD    = 4;
Data:   /** Button ID*/          final static int BUT_FEM   = 5;
Data:   /** Button ID*/          final static int BUT_FFT   = 6;
Data:   /** Button ID*/          final static int BUT_MCM   = 7;
Data:   /** Button ID*/          final static int BUT_CHA   = 8;
Data:
Data:
Data:   /** GUI choice button creator
Data:       @param A button ID
Data:       @return A choice button
Data:   ************************************************************************/
Data:   public Choice button(int type) {
Data:     switch(type) {
Data:     case BUT_PDE:    return pdeTypes.makeButton();
Data:     case BUT_IC:     return icTypes.makeButton();
Data:     case BUT_METH:   return numMeths.makeButton();
Data:     case BUT_PARSET: return setParams.makeButton();
Data:     case BUT_FD:     return numSchemsFD.makeButton();
Data:     case BUT_FEM:    return numSchemsFEM.makeButton();
Data:     case BUT_FFT:    return numSchemsFFT.makeButton();
Data:     case BUT_MCM:    return numSchemsMCM.makeButton();
Data:     case BUT_CHA:    return numSchemsCHA.makeButton();
Data:     default : return null; // results in an error
Data:     }
Data:   }
Data:
Data:
Data:   /************************************************************************
Data:   * Simple Helper object that associates booleans with choices
Data:   ************************************************************************/
Data:   public class Selection {
Data:     boolean[] active;
Data:     String[] name;
Data:     int maxSize = 0;
Data:     int curSize = 0;
Data:     int index = -1;
Data:
Data:     public Selection(int size) {
Data:       maxSize = size;
Data:       name = new String[size];
Data:       active = new boolean[size];
Data:     }
Data:
Data:     public void put(String nam, boolean val) {
Data:       index++;
Data:       curSize++;
Data:       name[index] = new String(nam);
Data:       active[index] = val;
Data:     }
Data:
Data:     public Choice makeButton() {
Data:       Choice b = new Choice();
Data:       for (int i=0; i<size(); i++)
Data:        b.addItem(name(i));
Data:       return b;
Data:     }
Data:
Data:     public int nextActive() {
Data:       int next = index;
Data:       int i    = next+1;
Data:       while (i != next) {
Data:         if (active[i]) { next=i; }
Data:         else if (i<curSize) { i++; } else { i=0; }
Data:       }
Data:       index = next;
Data:       return next;
Data:     }
Data:
Data:     public int size() { return curSize; };
Data:     public String name(int i) { index=i; return name[i]; };
Data:     public boolean isActive(int i) { index= i; return active[i]; };
Data:   } // Selection
Data:
Data: } // Data
Mesh:
Mesh:
Mesh: /* $Id: Mesh.java,v 1.10 1999/06/14 06:36:57 andrej Exp $ */
Mesh:
Mesh: /************************************************************************
Mesh: * Mesh - Discretization in one dimension
Mesh: ************************************************************************/
Mesh: class Mesh {
Mesh:
Mesh:   /** Coordinates */    double[] x;
Mesh:   /** Interval sizes */ double[] dx;
Mesh:
Mesh:   public Mesh(Data runData) {
Mesh:     int    nbrPts_ = runData.getInt(Data.NPTS);            //Parameters
Mesh:     double start_  = runData.getDouble(Data.BOX0);
Mesh:     double end_    = runData.getDouble(Data.BOXLEN);
Mesh:
Mesh:     x  = new double[nbrPts_];                              //Targets
Mesh:     dx = new double[nbrPts_];
Mesh:
Mesh:     double dx0=(end_-start_)/((double)(nbrPts_));          //Rules
Mesh:     for (int j=0; j<x.length; j++) {
Mesh:       x[j]=dx0*(double)j;
Mesh:       dx[j]=dx0;
Mesh:     }
Mesh:   }
Mesh:
Mesh:   /** Grid Size
Mesh:       @return Number of mesh points */
Mesh:   public int size() { return x.length; }
Mesh:
Mesh:   /** Coordinate values
Mesh:       @return An Array with coordinates */
Mesh:   public double[] points() { return x; }
Mesh:
Mesh:   /** Coordinate value
Mesh:       @return The value of a coordinate */
Mesh:   public double   point(int i) { return x[i]; }
Mesh:
Mesh:   /** Spacing between mesh points
Mesh:       @return An Array with the interval sizes */
Mesh:     public double[] intervals() { return dx; }
Mesh:
Mesh:   /** Space to next grid point
Mesh:       @return The interval size */
Mesh:     public double   interval(int i) { return dx[i]; }
Mesh:
Mesh:   /** Coordinate boundaries
Mesh:       @return {min(x), max(x) */
Mesh:   public double[] limits() {
```

```
Mesh:     double[] lim = {x[0], x[0]};
Mesh:     for (int i=1;i<x.length;i++) {
Mesh:       if (x[i]<lim[0]) {lim[0]=x[i];};
Mesh:       if (x[i]>lim[1]) {lim[1]=x[i];};
Mesh:     }
Mesh:     return lim;
Mesh:   }
Mesh:
Mesh:   /** For debugging */
Mesh:   public String toString() {
Mesh:     StringBuffer str = new StringBuffer();
Mesh:     str.append("RegularMesh[").append(Double.toString(x[0]))
Mesh:       .append(" , ").append(Double.toString(x[2]))
Mesh:       .append(" .. ").append(Double.toString(x[x.length-1]))
Mesh:       .append("]");
Mesh:     return str.toString();
Mesh:   }
Mesh:
Mesh: }//End of Mesh
Solution:
Solution:
Solution: /* $Id: Solution.java,v 1.14.4.3 1999/07/31 14:20:52 andrej Exp $ */
Solution:
Solution: /*************************************************************************
Solution: * Solution -- Is an abstract class containing all the solutions and solvers.
Solution: *   @see FluidSolution
Solution: *   @see ParticleSolution
Solution: *************************************************************************/
Solution:
Solution: abstract public class Solution{
Solution:
Solution:   /** Equation name @see #setEquation */          protected String equation_;
Solution:   /** Numerical scheme name @see #setMethod */   protected String method_;
Solution:   /** Numerical scheme name @see #setScheme */   protected String scheme_;
Solution:   /** Absolute time @see #setTime*/              protected double time_;
Solution:
Solution:   /** Function time level n-1 */                 protected double[]  fm;
Solution:   /** Function time level n   */                 protected double[]  f;
Solution:   /** Function time level n+1 */                 protected double[]  fp;
Solution:   /** Derivative time level n-1 @see #fm */      protected double[]  dfm;
Solution:   /** Derivative time level n   @see #f */       protected double[]  df;
Solution:   /** Derivative time level n+1 @see #fp */      protected double[]  dfp;
Solution:
Solution:   /** Extra Function time level n-1 */          protected double[]  gm;
Solution:   /** Extra Function time level n   */          protected double[]  g;
Solution:   /** Extra Function time level n+1 */          protected double[]  gp;
Solution:   /** Extra Derivative time level n-1 @see #gm */protected double[]  dgm;
Solution:   /** Extra Derivative time level n   @see #g */ protected double[]  dg;
Solution:   /** Extra Derivative time level n+1 @see #gp */protected double[]  dgp;
Solution:
Solution:   /** Complex Function time level n-1 */        protected Complex[] hm;
Solution:   /** Complex Function time level n   */        protected Complex[] h;
Solution:   /** Complex Function time level n+1 */        protected Complex[] hp;
Solution:   /** Derivative time level n-1 @see #hm */     protected Complex[] dhm;
Solution:   /** Derivative time level n   @see #h */      protected Complex[] dh;
Solution:   /** Derivative time level n+1 @see #hp */     protected Complex[] dhp;
Solution:
Solution:   /** Fourier spectrum time level n-1 */        protected Complex[] sm;
Solution:   /** Fourier spectrum time level n */          protected Complex[] s;
Solution:   /** Fourier spectrum time level n+1 */        protected Complex[] sp;
Solution:
Solution:   /** The underlaying mesh of the solution */   protected Mesh mesh_;
Solution:   /** The mesh points @see #mesh_ */            protected double[] x;
Solution:   /** The mesh intervals @see #mesh_ */         protected double[] dx;
Solution:   /** Store initial moments */                  protected double[]
Solution:                                                 initialMoments = {0.,0.,0.};
Solution:
Solution:
Solution:   /** Creates an instance of a Solution object with all the attributes.
Solution:       @param mesh The underlying mesh of the solution
Solution:   *************************************************************************/
Solution:   public Solution(Mesh mesh) {
Solution:     int n = mesh.size();
Solution:     x  = mesh.points();  dx = mesh.intervals();
Solution:     fm = new double[n];  f  = new double[n];  fp  = new double[n];
Solution:     dfm= new double[n];  df = new double[n];  dfp = new double[n];
Solution:     gm = new double[n];  g  = new double[n];  gp  = new double[n];
Solution:     dgm= new double[n];  dg = new double[n];  dgp = new double[n];
Solution:     hm = new Complex[n]; h  = new Complex[n]; hp  = new Complex[n];
Solution:     dhm= new Complex[n]; dh = new Complex[n]; dhp = new Complex[n];
Solution:     sm = new Complex[n]; s  = new Complex[n]; sp  = new Complex[n];
Solution:     mesh_ = mesh;
Solution:   } // Solution
Solution:
Solution:
Solution:   /** Internal function calculating the initial moments of f[] or their
Solution:       deviation from the beginning of the evolution.
Solution:       @param m The order of the moment
Solution:       @return The value or deviation of the m:th moment of f[]
Solution:       @see #momentsDeviation
Solution:   *************************************************************************/
Solution:   protected double calculateMoments(int m){
Solution:     int n=f.length-1;
Solution:     double moment = 0.0;
Solution:     if (m==0) {
Solution:       for (int i=0; i<n; i++) {
Solution:         moment= moment+0.5*dx[i]*(f[i+1]+f[i]);
Solution:       }
Solution:       moment = moment +0.5*dx[n]*(f[0]+f[n]);
Solution:     } else { moment=0.0; }
Solution:     if (initialMoments[m] != 0.) {                    //Deviation
Solution:       return (moment-initialMoments[m])/initialMoments[m];
Solution:     } else {
Solution:       return moment;                                  //Initial value
Solution:     }
Solution:   } // calculateMoments
Solution:
Solution:
Solution:   /** Internal function calculating the limits of the solution. This
Solution:       function assumes that the distribution function is in f[].
Solution:       @return A vector consisting of {min(solution), max(solution)}
Solution:       @see #limits
Solution:   *************************************************************************/
Solution:   protected double[] calculateLimits() {
Solution:     double[] lim = {f[0], f[0]};
Solution:     for (int i=1;i<f.length;i++) {
Solution:       if (f[i]<lim[0]) {lim[0]=f[i];};
Solution:       if (f[i]>lim[1]) {lim[1]=f[i];};
Solution:     } // for
Solution:     return lim;
Solution:   } // calculateLimits
Solution:
Solution:
Solution:   /** Number of values in the discretization
Solution:       @return The size of the underlaying mesh
Solution:   *************************************************************************/
Solution:   public int size(){  return mesh_.size(); }
Solution:
Solution:
```

```
Solution:   /** Set the equation as a string.
Solution:       @param scheme The name of the equation
Solution:       @return True
Solution:       @see Data#ADVECTION
Solution:   *************************************************************************/
Solution:   public boolean setEquation(String equation){
Solution:     equation_ = new String(equation);
Solution:     return true;
Solution:   } // setEquation
Solution:
Solution:
Solution:   /** Set the numerical method as a string.
Solution:       @param method The name of the numerical method
Solution:       @return True
Solution:   *************************************************************************/
Solution:   public boolean setMethod(String method){
Solution:     method_ = new String(method);
Solution:     return true;
Solution:   } // setMethod
Solution:
Solution:
Solution:   /** Set the numerical scheme as a string.
Solution:       @param scheme The name of the numerical scheme
Solution:       @return True
Solution:   *************************************************************************/
Solution:   public boolean setScheme(String scheme){
Solution:     scheme_ = new String(scheme);
Solution:     return true;
Solution:   } // setScheme
Solution:
Solution:
Solution:   /** Set the current physical time
Solution:       @param current time
Solution:       @return True
Solution:   *************************************************************************/
Solution:   public boolean setTime(double time){
Solution:     time_ = time;
Solution:     return true;
Solution:   } // setTime
Solution:
Solution:
Solution:   /** Discretize the initial Shape function and initialize the moments
Solution:       @param The initial shape to be approximated
Solution:   */
Solution:   abstract public void discretize(ShapeFunction function);
Solution:
Solution:
Solution:   /** Calculates the deviation from the m:th initial moment.
Solution:       @param m The order of the moment
Solution:       @return The deviation of the m:th moment from the beginning
Solution:       @see discretize
Solution:   */
Solution:   abstract public double momentsDeviation(int m);
Solution:
Solution:
Solution:   /** Advance the solution forward one step in time.
Solution:       The equation is set by the internal variable equation_ and
Solution:       the numerical scheme by the internal variable scheme_
Solution:       @param runData List of run parameters
Solution:       @see Data#TIMSTP
Solution:       @see Data#VELCTY
Solution:       @see Data#DIFFUS
Solution:       @see Data#THETA
Solution:       @see Data#TUNINT
Solution:       @return True if a scheme is implemented for that equation
Solution:   */
Solution:   abstract public boolean next(Data runData);
Solution:
Solution:
Solution:   /** Take the solution backward one step to initialize schemes
Solution:       with 3 time levels.
Solution:       The equation is set by the internal variable equation_ and
Solution:       the numerical scheme by the internal variable scheme_
Solution:       @param runData List of run parameters
Solution:       @see Data#TIMSTP
Solution:       @see Data#VELCTY
Solution:       @see Data#DIFFUS
Solution:       @see Data#THETA
Solution:       @see Data#TUNINT
Solution:       @return True if an initialisation is implemented for that equation
Solution:   */
Solution:   abstract public boolean previous(Data runData);
Solution:
Solution:
Solution:   /** Calculates the solution boundaries.
Solution:       @return A vector with {min(solution), max(solution)}
Solution:   */
Solution:   abstract public double[] limits();
Solution:
Solution:   /** Gives the value of the distribution function for an index
Solution:       @param index The index for which to get the value
Solution:       @return The value of the distribution function at a given index
Solution:   */
Solution:   abstract public double getValue(int index);
Solution:
Solution: } // class Solution
FluidSolution:
FluidSolution:
FluidSolution: /* $Id: FluidSolution.java,v 1.7 1999/06/13 22:11:35 andrej Exp $ */
FluidSolution:
FluidSolution: /*************************************************************************
FluidSolution: * FluidSolution -- Abstract class at the top of all the solvers that
FluidSolution: *   that operate directly on the fluid function.
FluidSolution: *   @see Solution
FluidSolution: *************************************************************************/
FluidSolution: abstract public class FluidSolution extends Solution{
FluidSolution:
FluidSolution:   /** True if needs to shiftLevels */ protected boolean isForward=true;
FluidSolution:
FluidSolution:   /** Creates a FluidSolution object.
FluidSolution:       @param mesh The underlying mesh of the solution
FluidSolution:   *************************************************************************/
FluidSolution:   public FluidSolution(Mesh mesh){
FluidSolution:     super(mesh);
FluidSolution:   } // FluidSolution
FluidSolution:
FluidSolution:
FluidSolution:   /** Calculates the deviation from the m:th initial moment.
FluidSolution:       @param m The order of the moment
FluidSolution:       @return The deviation of the m:th moment from the beginning
FluidSolution:       @see discretize
FluidSolution:   *************************************************************************/
FluidSolution:   public double momentsDeviation(int m){
FluidSolution:     return calculateMoments(m);
FluidSolution:   } // moments
FluidSolution:
FluidSolution:
FluidSolution:   /** Internal function for copying new -> old
FluidSolution:   *************************************************************************/
```

```
FluidSolution:   protected void shiftLevels() {
FluidSolution:     for (int i=0;i<f.length;i++) {
FluidSolution:      fm[i]=f[i];dfm[i]=df[i];f[i]=fp[i];df[i]=dfp[i];fp[i]=0.;dfp[i]=0.;
FluidSolution:      gm[i]=g[i];dgm[i]=dg[i];g[i]=gp[i];dg[i]=dgp[i];gp[i]=0.;dgp[i]=0.;
FluidSolution:      sm[i]=s[i];              s[i]=sp[i];              sp[i]=new Complex()
FluidSolution:     }
FluidSolution:   }
FluidSolution:
FluidSolution:
FluidSolution:   /** Calculates the limits of the solution.
FluidSolution:       @return A vector consisting of {min(solution), max(solution)}
FluidSolution:   **********************************************************************/
FluidSolution:   public double[] limits(){
FluidSolution:     return calculateLimits();
FluidSolution:   } // limits
FluidSolution:
FluidSolution:
FluidSolution:   /** Gives the value of the fluid function for an index
FluidSolution:       @param index The index for which to get the value
FluidSolution:       @return The value of the fluid function at a given index
FluidSolution:   **********************************************************************/
FluidSolution:   public double getValue(int index){
FluidSolution:     return f[index];
FluidSolution:   } // getValue
FluidSolution:
FluidSolution:
FluidSolution:   /** Discretize the initial Shape function and initialize the moments
FluidSolution:       @param The initial shape to be approximated
FluidSolution:   */
FluidSolution:   abstract public void discretize(ShapeFunction function);
FluidSolution:
FluidSolution: } // class FluidSolution
ParticleSolution:
ParticleSolution:
ParticleSolution: /*$Id: ParticleSolution.java,v 1.8.4.4 1999/06/28 10:04:36 johanh $*/
ParticleSolution:
ParticleSolution: import java.util.Random;
ParticleSolution:
ParticleSolution:
ParticleSolution:
ParticleSolution: /** The ParticleSolution solves the equation with particle methods
ParticleSolution:     @version $Revision: 1.8.4.4 $
ParticleSolution:     @see Solution
ParticleSolution: */
ParticleSolution: abstract class ParticleSolution extends Solution{
ParticleSolution:
ParticleSolution:   /** The number of test particles in the simulation */
ParticleSolution:   protected int numberOfParticles;
ParticleSolution:   /** A vector with the position of the particles */
ParticleSolution:   protected double[] particlePosition;
ParticleSolution:   /** The amount of mass each test particle represents */
ParticleSolution:   protected double particleDensity;
ParticleSolution:   /** Whether the distribution function is up to date */
ParticleSolution:   protected boolean distributionUpToDate;
ParticleSolution:   /** Random number generator */
ParticleSolution:   protected Random random;
ParticleSolution:
ParticleSolution:   /** Creates a ParticleSolution object. discretize must be called
ParticleSolution:       before next is called for the first time.
ParticleSolution:       @param mesh The mesh of the solution
ParticleSolution:       @see Solution#next
ParticleSolution:       @see Solution#discretize
ParticleSolution:   */
ParticleSolution:   public ParticleSolution(Mesh mesh){
ParticleSolution:     super(mesh);
ParticleSolution:     random = new Random();
ParticleSolution:   } // ParticleSolution
ParticleSolution:
ParticleSolution:   /** Take the solution backward one step to initialize schemes
ParticleSolution:       with 3 time levels; not really appropriate in this context.
ParticleSolution:       @param runData List of run parameters
ParticleSolution:       @return False since it is not used here
ParticleSolution:   */
ParticleSolution:   public boolean previous(Data runData){ return false; }
ParticleSolution:
ParticleSolution:   /** Discretize the initial Shape function and initialize moments
ParticleSolution:       Initializes a new set of particles, the number determined by
ParticleSolution:       the numerical scheme parameter.
ParticleSolution:       @param The initial shape to be approximated
ParticleSolution:       @see Solution#setScheme
ParticleSolution:       @see Data#NPARTICLE1
ParticleSolution:   */
ParticleSolution:   public void discretize(ShapeFunction function){
ParticleSolution:
ParticleSolution:     // First set the distribution function and calculate the moments
ParticleSolution:     int j;
ParticleSolution:     for (j = 0; j < mesh_.size(); j++)
ParticleSolution:       f[j] = function.getValue(mesh_, j);
ParticleSolution:     initialMoments[0]=calculateMoments(0); //Conserved quantities
ParticleSolution:     initialMoments[1]=calculateMoments(1);
ParticleSolution:     initialMoments[2]=calculateMoments(2);
ParticleSolution:     if(scheme_.equals(Data.NPARTICLE1))
ParticleSolution:       numberOfParticles = 1;
ParticleSolution:     else if(scheme_.equals(Data.NPARTICLE100))
ParticleSolution:       numberOfParticles = 100;
ParticleSolution:     else if(scheme_.equals(Data.NPARTICLE1000))
ParticleSolution:       numberOfParticles = 1000;
ParticleSolution:     else if(scheme_.equals(Data.NPARTICLE10000))
ParticleSolution:       numberOfParticles = 10000;
ParticleSolution:     else
ParticleSolution:       numberOfParticles = 100;
ParticleSolution:     particlePosition = new double[numberOfParticles];
ParticleSolution:     particleDensity = initialMoments[0] / numberOfParticles;
ParticleSolution:     // Normalize the distribution function to (0,1)
ParticleSolution:     double[] lim = calculateLimits();
ParticleSolution:     for(j = 0; j < mesh_.size(); j++)
ParticleSolution:       f[j] = (f[j] - lim[0]) / (lim[1] - lim[0]);
ParticleSolution:     // Randomize the particle positions
ParticleSolution:     double position;
ParticleSolution:     lim = mesh_.limits();
ParticleSolution:     for(j = 0; j < numberOfParticles; j++){
ParticleSolution:       while(random.nextDouble() >
ParticleSolution:             getValue(position = lim[0] +
ParticleSolution:               (lim[1] - lim[0]) * random.nextDouble()));
ParticleSolution:       particlePosition[j] = position;
ParticleSolution:     } // for
ParticleSolution:     distributionUpToDate = false;
ParticleSolution:   } // discretize
ParticleSolution:
ParticleSolution:   /** Calculates the deviation from the m:th initial moment.
ParticleSolution:       @param m The order of the moment
ParticleSolution:       @return The deviation of the m:th moment from the beginning
ParticleSolution:       @see discretize
ParticleSolution:   */
ParticleSolution:   public double momentsDeviation(int m){
ParticleSolution:     if(!distributionUpToDate)
ParticleSolution:       generateDistribution();
ParticleSolution:     return calculateMoments(m);
ParticleSolution:   } // moments
ParticleSolution:
ParticleSolution:   /** Calculates the limits of the solution.
ParticleSolution:       @return A vector consisting of {min(solution), max(solution)}
ParticleSolution:   */
ParticleSolution:   public double[] limits(){
ParticleSolution:     if(!distributionUpToDate)
ParticleSolution:       generateDistribution();
ParticleSolution:     return calculateLimits();
ParticleSolution:   } // limits
ParticleSolution:
ParticleSolution:   /** Gives the value of the distribution function for an index
ParticleSolution:       @param index The index for which to get the value
ParticleSolution:       @return The value of the distribution function at a given index
ParticleSolution:   */
ParticleSolution:   public double getValue(int index){
ParticleSolution:     if(!distributionUpToDate)
ParticleSolution:       generateDistribution();
ParticleSolution:     return f[index];
ParticleSolution:   } // getValue
ParticleSolution:
ParticleSolution:   /** Generates the distribution function from the set of
ParticleSolution:       test particles. Assumes a uniform mesh.
ParticleSolution:   */
ParticleSolution:   private void generateDistribution(){
ParticleSolution:
ParticleSolution:     int j;
ParticleSolution:     for(j = 0; j < mesh_.size(); j++)
ParticleSolution:       f[j] = 0;
ParticleSolution:     for(j = 0; j < numberOfParticles; j++){
ParticleSolution:       double[] lim = mesh_.limits();
ParticleSolution:       // Calculate the lower index of mesh cell
ParticleSolution:       int index = (int)Math.floor((particlePosition[j] - lim[0])
ParticleSolution:           / (lim[1] - lim[0]) *
ParticleSolution:           (mesh_.size() - 1));
ParticleSolution:       // Check whether the particle is inside the plot region
ParticleSolution:       if(index >= 0 && index <= mesh_.size()){
ParticleSolution:         if(index == mesh_.size())
ParticleSolution:           index--;
ParticleSolution:         double x_low = mesh_.point(index);
ParticleSolution:         double x_high = mesh_.point(index) + dx[0];
ParticleSolution:         // Find the interpolation coefficient
ParticleSolution:         double a = (particlePosition[j] - x_low) / ( x_high - x_low);
ParticleSolution:         // Add the particle contribution to the distribution function
ParticleSolution:         double cell_area_left = 1 / dx[0];  // Assume uniform mesh
ParticleSolution:         double cell_area_right = 1 / dx[0]; // Assume uniform mesh
ParticleSolution:         f[index] += (1 - a) * particleDensity / cell_area_left;
ParticleSolution:         f[(index + 1) % mesh_.size()] +=
ParticleSolution:                             a * particleDensity / cell_area_right;
ParticleSolution:       } // if
ParticleSolution:     } // for
ParticleSolution:     distributionUpToDate = true;
ParticleSolution:   } // generateDistribution
ParticleSolution:
ParticleSolution:   /** Linear interpolation of the function. Assumes a uniform mesh.
ParticleSolution:       @param position The x value
ParticleSolution:       @return A linear interpolation of the distribution function
ParticleSolution:       at a given position
ParticleSolution:   */
ParticleSolution:   private double getValue(double position){
ParticleSolution:     double[] lim = mesh_.limits();
ParticleSolution:     // Calculate the lower index of mesh cell
ParticleSolution:     int index = (int)Math.floor((position-lim[0])/(lim[1]-lim[0]) *
ParticleSolution:   (mesh_.size() - 1));
ParticleSolution:     if(index == mesh_.size() - 1)
ParticleSolution:       index--;
ParticleSolution:     double x_low = mesh_.point(index);
ParticleSolution:     double x_high = mesh_.point(index + 1);
ParticleSolution:     // Find the interpolation coefficient
ParticleSolution:     double a = (position - x_low) / ( x_high - x_low);
ParticleSolution:     return f[index] * (1 - a) + f[index + 1] * a;
ParticleSolution:   } // getValue
ParticleSolution:
ParticleSolution:
ParticleSolution: } // ParticleSolution
FDSolution:
FDSolution:
FDSolution: /* $Id: FDSolution.java,v 1.18.4.3 1999/08/05 11:48:14 andrej Exp $ */
FDSolution:
FDSolution: /*****************************************************************************
FDSolution: *  FDSolution -- contains the finite differance solver.
FDSolution: *    @see FluidSolution
FDSolution: *    @see Solution
FDSolution: *****************************************************************************/
FDSolution: public class FDSolution extends FluidSolution{
FDSolution:   /** Creates a FDSolution object.
FDSolution:       @param mesh The mesh of the solution
FDSolution:   */
FDSolution:   public FDSolution(Mesh mesh){
FDSolution:     super(mesh);
FDSolution:   } // FDSolution
FDSolution:
FDSolution:
FDSolution:   /** Discretize the initial Shape function and initialize the moments
FDSolution:       @param The initial shape to be approximated
FDSolution:   **********************************************************************/
FDSolution:   public void discretize(ShapeFunction function){
FDSolution:     for (int j = 0; j < mesh_.size(); j++)       //Discretizing is simple
FDSolution:       f[j] = function.getValue(mesh_, j);        // with finite differences
FDSolution:     initialMoments[0]=calculateMoments(0);       //Set conserved quantities
FDSolution:     initialMoments[1]=calculateMoments(1);
FDSolution:   } // discretize
FDSolution:
FDSolution:
FDSolution:   /** Advance the solution forward one step in time.
FDSolution:       The equation is set by the internal variable equation_ and
FDSolution:       the numerical scheme by the internal variable scheme_
FDSolution:       @param runData List of run parameters
FDSolution:       @see Data#TIMSTP
FDSolution:       @see Data#VELCTY
FDSolution:       @see Data#DIFFUS
FDSolution:       @see Data#THETA
FDSolution:       @see Data#TUNINT
FDSolution:       @return True if a scheme is implemented for that equation
FDSolution:   **********************************************************************/
FDSolution:   public boolean next(Data runData) {
FDSolution:     double timeStep = runData.getDouble(Data.TIMSTP);         //Parameters
FDSolution:     double velocity = runData.getDouble(Data.VELCTY);
FDSolution:     double diffusCo = runData.getDouble(Data.DIFFUS);
FDSolution:     double disperCo = runData.getDouble(Data.DISPER);
FDSolution:
FDSolution:     double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are constant if
FDSolution:     double beta =timeStep*velocity/(dx[0]      );  // the problem and mesh
FDSolution:                                                    // arehomogeneous
FDSolution:     int n=f.length-1;
FDSolution:     boolean isDefined=false;
FDSolution:
FDSolution:     if(equation_.equals(Data.ADVECTION)
FDSolution:        && scheme_.equals(Data.EXP2LVL)){
FDSolution:       //=============================================================
```

```
FDSolution:       //  FD  - Advection/diffusion - explicit 2 levels (Godunov advection)
FDSolution:       //=========================================================================
FDSolution:       for (int i=1; i<n; i++) {
FDSolution:         fp[i]=f[i] -beta *(f[i]-f[i-1])+alpha*(f[i+1]-2.*f[i]+f[i-1]); }
FDSolution:       fp[0]=f[0]   -beta *(f[0]-f[ n ])+alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
FDSolution:       fp[n]=f[n]   -beta *(f[n]-f[n-1])+alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.EXP3LVL)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Advection/diffusion - explicit 3 levels
FDSolution:       //=========================================================================
FDSolution:       for (int i=1; i<n; i++) {
FDSolution:         fp[i]=fm[i] -beta*(f[i+1]-f[i-1])+2*alpha*(f[i+1]-2.*f[i]+f[i-1]);}
FDSolution:       fp[0]=fm[0]   -beta*(f[ 1 ]-f[ n ])+2*alpha*(f[ 1 ]-2.*f[0]+f[ n ]);
FDSolution:       fp[n]=fm[n]   -beta*(f[ 0 ]-f[n-1])+2*alpha*(f[ 0 ]-2.*f[n]+f[n-1]);
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.IMP2LVL)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Advection/diffusion - implicit (Crank-Nicholson diffusion)
FDSolution:       //=========================================================================
FDSolution:       BandMatrix a = new BandMatrix(3, f.length);
FDSolution:       BandMatrix b = new BandMatrix(3, f.length);
FDSolution:       double[]   c = new double[f.length];
FDSolution:
FDSolution:       for (int i=0; i<=n; i++) {
FDSolution:         a.setL(i,-0.25*beta -0.5*alpha);            //Matrix elements
FDSolution:         a.setD(i,            1. +alpha);
FDSolution:         a.setR(i, 0.25*beta -0.5*alpha);
FDSolution:         b.setL(i, 0.25*beta +0.5*alpha);            //Right hand side
FDSolution:         b.setD(i,            1. -alpha);
FDSolution:         b.setR(i,-0.25*beta +0.5*alpha);
FDSolution:       }
FDSolution:
FDSolution:       c=b.dot(f);                                   //Right hand side
FDSolution:       c[0]=c[0]+b.getL(0)*f[n];                     // with periodicity
FDSolution:       c[n]=c[n]+b.getR(n)*f[0];
FDSolution:
FDSolution:       fp=a.solve3(c);                               //Solve linear problem
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.EXPLAX)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Advection - explicit Lax-Wendroff - precision O(delta t^3)
FDSolution:       //=========================================================================
FDSolution:       for (int i=1; i<n; i++) {
FDSolution:         fp[i]=f[i] -0.5*beta *(f[i+1]-f[i-1])
FDSolution:                    +0.5*beta*beta*(f[i+1]-2.*f[i]+f[i-1]); }
FDSolution:       fp[0]=f[0] -0.5*beta *(f[ 1 ]-f[ n ])
FDSolution:                  +0.5*beta*beta*(f[ 1 ]-2.*f[0]+f[ n ]);
FDSolution:       fp[n]=f[n] -0.5*beta *(f[ 0 ]-f[n-1])
FDSolution:                  +0.5*beta*beta*(f[ 0 ]-2.*f[n]+f[n-1]);
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.IMPLAX)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Advection - implicit Lax-Wendroff
FDSolution:       //=========================================================================
FDSolution:       BandMatrix a = new BandMatrix(3, f.length);
FDSolution:       BandMatrix b = new BandMatrix(3, f.length);
FDSolution:       double[]   c = new double[f.length];
FDSolution:
FDSolution:       for (int i=0; i<=n; i++) {
FDSolution:         a.setL(i, (1.-beta)/(1.+beta));           //Matrix elements
FDSolution:         a.setD(i, 1.);
FDSolution:         a.setR(i, 0.);
FDSolution:         b.setL(i, 1.);                            //Right hand side
FDSolution:         b.setD(i, (1.-beta)/(1.+beta));
FDSolution:         b.setR(i, 0.);
FDSolution:       }
FDSolution:
FDSolution:       c=b.dot(f);                                   //Right hand side
FDSolution:       c[0]=c[0]+b.getL(0)*f[n];                     // with periodicity
FDSolution:       c[n]=c[n]+b.getR(n)*f[0];
FDSolution:
FDSolution:       fp=a.solve3(c);                               //Solve linear problem
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.LPFROG)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Advection/diffusion - Leap-frog
FDSolution:       //=========================================================================
FDSolution:       for (int i=1; i<=n; i++) {                    //time + time_step
FDSolution:         fp[i]=f[i] -beta*(g[i]-g[i-1]); }
FDSolution:       fp[0]=f[0] -beta*(g[0]-g[n]);
FDSolution:
FDSolution:       for (int i=0; i<=n-1; i++) {                  //time + 1.5*time_step
FDSolution:         gp[i]=g[i] -beta*(fp[i+1]-fp[i]); }
FDSolution:       gp[n]=g[n] -beta*(fp[0]-fp[n]);
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.BURGER)
FDSolution:               && scheme_.equals(Data.EXP3LVL)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Burger's shock wave equation - explicit 3 levels
FDSolution:       //=========================================================================
FDSolution:
FDSolution:       double fm1 = f[ n ];            //Extra mesh points for periodicity
FDSolution:       double f0  = f[ 0 ];
FDSolution:       double fp1 = f[ 1 ];
FDSolution:
FDSolution:       for (int i=0; i<=n; ++i) {
FDSolution:         fp[i]= f[i]
FDSolution:           -timeStep/dx[0]* f0*(fp1-fm1)           //Wave breaking term
FDSolution:           +alpha*(fp1-2*f0+fm1);                  //Diffusive term
FDSolution:         fm1=f0;  f0 =fp1;
FDSolution:         fp1=f[(i+2) % (n+1)];
FDSolution:       }
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.KDV)
FDSolution:               && scheme_.equals(Data.EXP3LVL)){
FDSolution:       //=========================================================================
FDSolution:       // FD  - KdV - Scheme Zabusky & Kruskal, Phys.Rev.Lett. 15(1965)240
FDSolution:       //=========================================================================
FDSolution:       double nolin=timeStep/(dx[0]);
FDSolution:       double gamma=timeStep/(dx[0]*dx[0]*dx[0])*disperCo;
FDSolution:
FDSolution:       double fm2 = f[n-1];  double fm1 = f[ n ];
FDSolution:       double f0  = f[ 0 ];  double fp1 = f[ 1 ];  double fp2 = f[ 2 ];
FDSolution:
FDSolution:       for (int i=0; i<=n; ++i) {
FDSolution:         fp[i]= fm[i]
FDSolution:           -nolin*(fp1+f0+fm1)/3.*(fp1-fm1)        //Wave breaking term
FDSolution:           -gamma*(fp2 -2.*(fp1-fm1) -fm2);        //Dispersive term
FDSolution:
FDSolution:         fm2=fm1;  fm1=f0;  f0 =fp1;  fp1=fp2;     //Periodicity
FDSolution:         fp2=f[(i+3) % (n+1)];
FDSolution:       }
FDSolution:       isDefined = true;
FDSolution:
FDSolution:     } else if(equation_.equals(Data.ADVECTION)
FDSolution:               && scheme_.equals(Data.FDMINE)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Implement your own scheme here
FDSolution:       //=========================================================================
FDSolution:       isDefined = false;
FDSolution:
FDSolution:     } else if(scheme_.equals(Data.FDEX21)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Exercise 2.1
FDSolution:       //=========================================================================
FDSolution:       isDefined = false;
FDSolution:
FDSolution:     } else if(scheme_.equals(Data.FDEX23)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Exercise 2.3
FDSolution:       //=========================================================================
FDSolution:       isDefined = false;
FDSolution:
FDSolution:     } else if(scheme_.equals(Data.FDEX24)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Exercise 2.4
FDSolution:       //=========================================================================
FDSolution:       isDefined = false;
FDSolution:
FDSolution:     } else if(scheme_.equals(Data.FDEX25)){
FDSolution:       //=========================================================================
FDSolution:       //  FD  - Exercise 2.5
FDSolution:       //=========================================================================
FDSolution:       isDefined = false;
FDSolution:
FDSolution:     } // if
FDSolution:
FDSolution:     if(isDefined && isForward)
FDSolution:       shiftLevels();
FDSolution:     return isDefined;
FDSolution:   } // next
FDSolution:
FDSolution:   /** Take the solution backward one step to initialize schemes
FDSolution:       with 3 time levels.
FDSolution:       The equation is set by the internal variable equation_ and
FDSolution:       the numerical scheme by the internal variable scheme_
FDSolution:       @param runData List of run parameters
FDSolution:       @see Data#TIMSTP
FDSolution:       @see Data#VELCTY
FDSolution:       @see Data#DIFFUS
FDSolution:       @see Data#THETA
FDSolution:       @see Data#TUNINT
FDSolution:       @return True if an initialisation is implemented for that equation
FDSolution:   ************************************************************************/
FDSolution:   public boolean previous(Data runData){
FDSolution:     int n=f.length-1;
FDSolution:     boolean isDefined=false;
FDSolution:
FDSolution:     if(equation_.equals(Data.ADVECTION) && scheme_.equals(Data.EXP3LVL) ||
FDSolution:        equation_.equals(Data.KDV)       && scheme_.equals(Data.EXP3LVL)  ){
FDSolution:
FDSolution:       //=========================================================================
FDSolution:       // FD init - 3 levels - Common to all such schemes
FDSolution:       //=========================================================================
FDSolution:       for (int i=0; i<=n; i++)          //Note how backward is implemented:
FDSolution:         { fm[i]=0.;}                    //  first reset the oldest level,
FDSolution:       isForward = false;                //  then calculate a pseudo-forward
FDSolution:       this.next(runData);               //  solution without shifting
FDSolution:       isForward = true;                 //  levels.
FDSolution:
FDSolution:       for (int i=0; i<=n; i++) {        //  Divide by -2 to gives a scheme
FDSolution:         fm[i]=f[i]-0.5*fp[i];           //  equivalent to 2 levels backward
FDSolution:       }
FDSolution:
FDSolution:     } else if (equation_.equals(Data.ADVECTION) &&
FDSolution:                scheme_.equals(Data.LPFROG)){
FDSolution:       //=========================================================================
FDSolution:       // FD init - Leap-Frog
FDSolution:       //=========================================================================
FDSolution:       double timeStep = runData.getDouble(Data.TIMSTP);
FDSolution:       double velocity = runData.getDouble(Data.VELCTY);
FDSolution:       double diffusCo = runData.getDouble(Data.DIFFUS);
FDSolution:       double alpha=timeStep*diffusCo/(dx[0]*dx[0]);
FDSolution:       double beta =timeStep*velocity/(dx[0]       );
FDSolution:
FDSolution:       for (int i=0; i<=n-1; i++) {              //time=0, initialize pulse
FDSolution:           gm[i] = 0.5*(f[i+1]+f[i]);            //  running to the right
FDSolution:         //gm[i] =-0.5*(f[i+1]+f[i]);            //              the left
FDSolution:         //gm[i] = 0.;                           //      in both directions
FDSolution:       }
FDSolution:       gm[n]=0.5*(f[0]+f[n]);
FDSolution:
FDSolution:       for (int i=0; i<=n-1; i++) {              //time=0.5*time_step
FDSolution:           g[i] = gm[i]-0.5*beta*(f[i+1]-f[i]); }
FDSolution:       g[n]=gm[n]-0.5*beta*(f[0]-f[n]);;
FDSolution:
FDSolution:     }
FDSolution:     return isDefined;
FDSolution:   } // previous
FDSolution:
FDSolution: } // class FDSolution
FEMSolution:
FEMSolution:
FEMSolution: /* $Id: FEMSolution.java,v 1.12.4.2 1999/08/05 11:48:15 andrej Exp $ */
FEMSolution:
FEMSolution: /*************************************************************************
FEMSolution: *  FEMSolution -- contains the finite elements solver.
FEMSolution: *    @see FluidSolution
FEMSolution: *    @see Solution
FEMSolution: *************************************************************************/
FEMSolution: public class FEMSolution extends FluidSolution{
FEMSolution:   /** Creates a FEMSolution object.
FEMSolution:       @param mesh The mesh of the solution
FEMSolution:   */
FEMSolution:   public FEMSolution(Mesh mesh){
FEMSolution:     super(mesh);
FEMSolution:   } // FEMSolution
FEMSolution:
FEMSolution:
FEMSolution:   /** Discretize the initial Shape function and initialize the moments
FEMSolution:       @param The initial shape to be approximated
FEMSolution:   *************************************************************************/
FEMSolution:   public void discretize(ShapeFunction function){
FEMSolution:     for (int j = 0; j < mesh_.size(); j++)       //Discretizing is simple
```

```
FEMSolution:        f[j] = function.getValue(mesh_, j);        //  with normalized FEM
FEMSolution:      initialMoments[0]=calculateMoments(0);       //Conserved quantities
FEMSolution:      initialMoments[1]=calculateMoments(1);
FEMSolution:    } // discretize
FEMSolution:
FEMSolution:
FEMSolution:    /** Advance the solution forward one step in time.
FEMSolution:        The equation is set by the internal variable equation_ and
FEMSolution:        the numerical scheme by the internal variable scheme_
FEMSolution:        @param runData List of run parameters
FEMSolution:        @see Data#TIMSTP
FEMSolution:        @see Data#VELCTY
FEMSolution:        @see Data#DIFFUS
FEMSolution:        @see Data#THETA
FEMSolution:        @see Data#TUNINT
FEMSolution:        @return True if a scheme is implemented for that equation
FEMSolution:    ************************************************************************/
FEMSolution:    public boolean next(Data runData) {
FEMSolution:      double timeStep = runData.getDouble(Data.TIMSTP);          //Parameters
FEMSolution:      double velocity = runData.getDouble(Data.VELCTY);
FEMSolution:      double diffusCo = runData.getDouble(Data.DIFFUS);
FEMSolution:      double theta    = runData.getDouble(Data.THETA);
FEMSolution:      double tune     = runData.getDouble(Data.TUNINT);
FEMSolution:
FEMSolution:      double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are constant if
FEMSolution:      double beta =timeStep*velocity/(dx[0]      );  // the problem and mesh
FEMSolution:                                                     // are homogeneous
FEMSolution:      int n=f.length-1;
FEMSolution:      boolean isDefined=false;
FEMSolution:
FEMSolution:      if(equation_.equals(Data.ADVECTION)
FEMSolution:         && scheme_.equals(Data.TUNFEM)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM - Advection/diffusion, tunable integration  t=1.  FD
FEMSolution:        //                                                  t=1/3 linear FEM
FEMSolution:        //                                                  t=0.  cst FEM
FEMSolution:        //=================================================================
FEMSolution:        BandMatrix a = new BandMatrix(3, f.length);
FEMSolution:        BandMatrix b = new BandMatrix(3, f.length);
FEMSolution:        double[]   c = new double[f.length];
FEMSolution:
FEMSolution:        double h   = dx[0];
FEMSolution:        double htm = h*(1-tune)/4;
FEMSolution:        double htp = h*(1+tune)/4;
FEMSolution:
FEMSolution:        for (int i=0; i<=n; i++) {
FEMSolution:          a.setL(i,   htm +h*(-0.5*beta -alpha)* theta     );
FEMSolution:          a.setD(i,2*(htp +h*(           alpha)* theta    ));
FEMSolution:          a.setR(i,   htm +h*( 0.5*beta -alpha)* theta     );
FEMSolution:          b.setL(i,   htm +h*(-0.5*beta -alpha)*(theta-1) );
FEMSolution:          b.setD(i,2*(htp +h*(           alpha)*(theta-1)));
FEMSolution:          b.setR(i,   htm +h*( 0.5*beta -alpha)*(theta-1) );
FEMSolution:        }
FEMSolution:
FEMSolution:        c=b.dot(f);                                  //Right hand side
FEMSolution:        c[0]=c[0]+b.getL(0)*f[n];                    // with periodicity
FEMSolution:        c[n]=c[n]+b.getR(n)*f[0];
FEMSolution:
FEMSolution:        fp=a.solve3(c);                              //Solve linear problem
FEMSolution:        isDefined = true;
FEMSolution:
FEMSolution:      } else if(equation_.equals(Data.ADVECTION)
FEMSolution:             && scheme_.equals(Data.FEMMINE)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM  - Implement your own scheme here
FEMSolution:        //=================================================================
FEMSolution:        isDefined = false;
FEMSolution:
FEMSolution:      } else if(scheme_.equals(Data.FEMEX32)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM - Exercise 3.2
FEMSolution:        //=================================================================
FEMSolution:        isDefined = false;
FEMSolution:
FEMSolution:      } else if(scheme_.equals(Data.FEMEX33)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM - Exercise 3.3
FEMSolution:        //=================================================================
FEMSolution:        isDefined = false;
FEMSolution:
FEMSolution:      } else if(scheme_.equals(Data.FEMEX34)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM - Exercise 3.4
FEMSolution:        //=================================================================
FEMSolution:        isDefined = false;
FEMSolution:
FEMSolution:      } else if(scheme_.equals(Data.FEMEX35)){
FEMSolution:        //=================================================================
FEMSolution:        //  FEM - Exercise 3.5
FEMSolution:        //=================================================================
FEMSolution:        isDefined = false;
FEMSolution:
FEMSolution:      } // if
FEMSolution:
FEMSolution:      if(isDefined)
FEMSolution:        shiftLevels();
FEMSolution:      return isDefined;
FEMSolution:    } // next
FEMSolution:
FEMSolution:    /** Take the solution backward one step to initialize schemes
FEMSolution:        with 3 time levels; not really appropriate in this context.
FEMSolution:        @param runData List of run parameters
FEMSolution:        @return False since it is not used here
FEMSolution:    ************************************************************************/
FEMSolution:    public boolean previous(Data runData){ return false; }
FEMSolution:
FEMSolution:  } // class FEMSolution
FFTSolution:
FFTSolution:
FFTSolution:  /* $Id: FFTSolution.java,v 1.16.4.5 1999/08/05 11:48:16 andrej Exp $ */
FFTSolution:
FFTSolution:  /************************************************************************
FFTSolution:  * FFTsolution - Fast Fourier Transform solver
FFTSolution:  *     @see FluidSolution
FFTSolution:  *     @see Solution
FFTSolution:  ************************************************************************/
FFTSolution:  public class FFTSolution extends FluidSolution{
FFTSolution:    public FFTSolution(Mesh mesh){
FFTSolution:      super(mesh);
FFTSolution:    } // FFTSolution
FFTSolution:
FFTSolution:    /** FFT previous (or initial) step   */  protected  FFT keepFFT;
FFTSolution:    /** FFT tool                         */  protected  FFT toolFFT;
FFTSolution:
FFTSolution:
FFTSolution:    /** Discretize the initial Shape function and initialize the moments
FFTSolution:        @param The initial shape to be approximated
FFTSolution:    ************************************************************************/
FFTSolution:    public void discretize(ShapeFunction function){
FFTSolution:      double boxLen = mesh_.size()*mesh_.interval(0);
FFTSolution:
FFTSolution:      for (int j=0; j<mesh_.size(); j++)           //Homogeneous mesh approx
FFTSolution:        f[j] = function.getValue(mesh_, j);
FFTSolution:
FFTSolution:      keepFFT  = new FFT(f,  FFT.inXSpace);        //Load in FFT transformer
FFTSolution:
FFTSolution:      initialMoments[0]=calculateMoments(0);       //Conserved quantities
FFTSolution:      initialMoments[1]=calculateMoments(1);
FFTSolution:    } // discretize
FFTSolution:
FFTSolution:
FFTSolution:    /** Advance the solution forward one step in time.
FFTSolution:        The equation is set by the internal variable equation_ and
FFTSolution:        the numerical scheme by the internal variable scheme_
FFTSolution:        @param runData List of run parameters
FFTSolution:        @see Data#TIMSTP
FFTSolution:        @see Data#VELCTY
FFTSolution:        @see Data#DIFFUS
FFTSolution:        @return True if a scheme is implemented for that equation
FFTSolution:    ************************************************************************/
FFTSolution:    public boolean next(Data runData) {
FFTSolution:      double timeStep = runData.getDouble(Data.TIMSTP);      //Parameters
FFTSolution:      double velocity = runData.getDouble(Data.VELCTY);
FFTSolution:      double diffusCo = runData.getDouble(Data.DIFFUS);
FFTSolution:      double disperCo = runData.getDouble(Data.DISPER);
FFTSolution:
FFTSolution:      int N           = mesh_.size();                        //A power of 2
FFTSolution:      double boxLen   = mesh_.size()*mesh_.interval(0);      //Periodicity
FFTSolution:
FFTSolution:      double   k    = 2*Math.PI/boxLen;                     //Notation
FFTSolution:      Complex ik1   = new Complex(  0.,   k  );
FFTSolution:      Complex ik2   = new Complex(-k*k,  0.  );
FFTSolution:      Complex ik3   = new Complex(  0.,-k*k*k);
FFTSolution:      Complex exp   = new Complex();
FFTSolution:
FFTSolution:      Complex advection = new Complex(ik1.scale(velocity));  //Variables
FFTSolution:      Complex diffusion = new Complex(ik2.scale(diffusCo));
FFTSolution:      Complex dispersion= new Complex(ik3.scale(disperCo));
FFTSolution:      Complex total     = new Complex(0.);
FFTSolution:      Complex nonlin    = new Complex(0.);
FFTSolution:      Complex linear    = new Complex(0.);
FFTSolution:
FFTSolution:      boolean isDefined = false;
FFTSolution:
FFTSolution:      if(equation_.equals(Data.ADVECTION)
FFTSolution:         && (scheme_.equals(Data.ALIASED)||
FFTSolution:             scheme_.equals(Data.EXPAND )  )){
FFTSolution:        //=================================================================
FFTSolution:        //  FFT - Advection/diffusion - No problem with aliasing if linear
FFTSolution:        //=================================================================
FFTSolution:        Complex[] s0 = new Complex[f.length];
FFTSolution:
FFTSolution:        s0=keepFFT.getFromKSpace(FFT.firstPart,boxLen);    //FFT to KSpace
FFTSolution:                                                           //  only once
FFTSolution:        s[0] = s0[0];
FFTSolution:        for (int m=1;  m<N/2+1;  m++) {                    //Propagate
FFTSolution:          total=          advection.scale((double)(m ));   //  from initial
FFTSolution:          total=total.add(diffusion.scale((double)(m*m))); //  contition
FFTSolution:          exp=(total.scale(time_)).exp();
FFTSolution:
FFTSolution:          s[m  ] = s0[m].mul(exp);                         //  s0 has the IC
FFTSolution:          s[N-m] = s[m].conj();                            //  f is real
FFTSolution:        }
FFTSolution:        FFT ffts = new FFT(s,FFT.inKSpace);                //Initialize
FFTSolution:
FFTSolution:        f = ffts.getFromXSpacePart(FFT.firstPart,boxLen);  //invFFT for plot
FFTSolution:
FFTSolution:        isForward = false;
FFTSolution:        isDefined = true;
FFTSolution:
FFTSolution:      } else  if((equation_.equals(Data.KDV)||
FFTSolution:                  equation_.equals(Data.BURGER))
FFTSolution:                  && (scheme_.equals(Data.ALIASED)||
FFTSolution:                      scheme_.equals(Data.EXPAND )  )){
FFTSolution:        //=================================================================
FFTSolution:        //  FFT - KdV solitons and Burger's shocks - Aliasing is an issue
FFTSolution:        //=================================================================
FFTSolution:
FFTSolution:        //----- Non-linear term: convolution
FFTSolution:        s  = keepFFT.getFromKSpace(FFT.bothParts,boxLen);//Current Spectrum
FFTSolution:        toolFFT = new FFT(s,s,FFT.inKSpace);             //  for convolution
FFTSolution:
FFTSolution:        if (scheme_.equals(Data.ALIASED))                //Aliasing or not,
FFTSolution:          sp = toolFFT.aliasedConvolution(boxLen);       //  use an FFT for
FFTSolution:        else //scheme_.equals(Data.EXPAND)               //  product, FFT
FFTSolution:          sp = toolFFT.expandedConvolution(boxLen);      //  back to KSpace
FFTSolution:
FFTSolution:        //----- Linear terms: complex terms in spectrum s
FFTSolution:        s = keepFFT.getFromKSpace(FFT.bothParts,boxLen); //Current Spectrum
FFTSolution:        linear= s[0];
FFTSolution:        sp[0]=linear;
FFTSolution:        for (int m=1;  m<=N/2;  m++) {
FFTSolution:          total=          advection.scale((double)(m ));
FFTSolution:          total=total.add(diffusion.scale((double)(m*m)));
FFTSolution:          total=total.add(dispersion.scale((double)(m*m*m)));
FFTSolution:          exp=(total.scale(timeStep)).exp();
FFTSolution:          linear    = s[m].mul(exp);
FFTSolution:          nonlin    = sp[m].mul(ik1.scale(0.5*timeStep*(double)(m)));
FFTSolution:          sp[m  ]   = linear.add(nonlin);
FFTSolution:          if (m<N/2) sp[N-m] = sp[m].conj();             //Real spectrum
FFTSolution:        }
FFTSolution:
FFTSolution:        keepFFT = new FFT(sp,FFT.inKSpace);
FFTSolution:        toolFFT = new FFT(sp,FFT.inKSpace);              //invFFT for plot
FFTSolution:        f=toolFFT.getFromXSpacePart(FFT.firstPart,boxLen);
FFTSolution:
FFTSolution:        isForward = false;
FFTSolution:        isDefined = true;
FFTSolution:
FFTSolution:      } else  if(equation_.equals(Data.ADVECTION) ||
FFTSolution:                 equation_.equals(Data.FFTMINE)  ){
FFTSolution:        //=================================================================
FFTSolution:        //  FFT - Implement your own scheme here
FFTSolution:        //=================================================================
FFTSolution:        isDefined = false;
FFTSolution:
FFTSolution:      } else if(scheme_.equals(Data.FFTEX41)){
FFTSolution:        //=================================================================
FFTSolution:        //  FEM - Exercise 4.1
FFTSolution:        //=================================================================
FFTSolution:        isDefined = false;
FFTSolution:
FFTSolution:      } else if(scheme_.equals(Data.FFTEX42)){
FFTSolution:        //=================================================================
FFTSolution:        //  FEM - Exercise 4.2
FFTSolution:        //=================================================================
FFTSolution:        isDefined = false;
```

```
FFTSolution:
FFTSolution:     }
FFTSolution:
FFTSolution:     if(isDefined && isForward)
FFTSolution:       shiftLevels();
FFTSolution:     return isDefined;
FFTSolution:   } // next
FFTSolution:
FFTSolution:
FFTSolution:   /** Starting procedure for FFT schemes.
FFTSolution:       @param runData List of run parameters
FFTSolution:       @see Data#TIMSTP
FFTSolution:       @see Data#VELCTY
FFTSolution:       @see Data#DIFFUS
FFTSolution:       @see Data#THETA
FFTSolution:       @see Data#TUNINT
FFTSolution:       @see Data#BOXLEN
FFTSolution:       @return True if an initialisation is implemented for that equation
FFTSolution:    *************************************************************************/
FFTSolution:   public boolean previous(Data runData){
FFTSolution:     int n=f.length-1;
FFTSolution:     boolean isDefined=false;
FFTSolution:     return isDefined;
FFTSolution:   } // previous
FFTSolution:
FFTSolution: } // class FFTSolution
MCMSolution:
MCMSolution:
MCMSolution: /* $Id: MCMSolution.java,v 1.20.4.5 1999/06/28 10:04:35 johanh Exp $ */
MCMSolution:
MCMSolution: import java.util.Random;
MCMSolution:
MCMSolution:
MCMSolution:
MCMSolution: /** The MCMSolution class solves the equation with a Monte Carlo method.
MCMSolution:     @version $Revision: 1.20.4.5 $
MCMSolution:     @see Solution
MCMSolution: */
MCMSolution: public class MCMSolution extends ParticleSolution{
MCMSolution:
MCMSolution:   /** Creates a MCMSolution object. discretize must be called
MCMSolution:       before next is called for the first time.
MCMSolution:       @param mesh The mesh of the solution
MCMSolution:       @see Solution#next
MCMSolution:       @see Solution#discretize
MCMSolution:   */
MCMSolution:   public MCMSolution(Mesh mesh){
MCMSolution:     super(mesh);
MCMSolution:   } // MCMSolution
MCMSolution:
MCMSolution:
MCMSolution:   /** Advance the solution forward one step in time.
MCMSolution:       The equation is set by the internal variable equation_ and
MCMSolution:       the numerical scheme by the internal variable scheme_
MCMSolution:       @param runData List of run parameters
MCMSolution:       @see Data#TIMSTP
MCMSolution:       @see Data#VELCTY
MCMSolution:       @see Data#DIFFUS
MCMSolution:       @see Data#THETA
MCMSolution:       @see Data#TUNINT
MCMSolution:       @return True if a scheme is implemented for that equation
MCMSolution:   */
MCMSolution:   public boolean next(Data runData) {
MCMSolution:     double timeStep = runData.getDouble(Data.TIMSTP);          //Parameters
MCMSolution:     double velocity = runData.getDouble(Data.VELCTY);
MCMSolution:     double diffusCo = runData.getDouble(Data.DIFFUS);
MCMSolution:
MCMSolution:     distributionUpToDate = false;
MCMSolution:     boolean isDefined = false;
MCMSolution:     double[] lim = {mesh_.point(0),
MCMSolution:                     mesh_.point(mesh_.size() - 1) + dx[0]};
MCMSolution:     if(equation_.equals(Data.ADVECTION)){
MCMSolution:       for(int j = 0; j < numberOfParticles; j++){
MCMSolution:         particlePosition[j] += velocity * timeStep +
MCMSolution:          random.nextGaussian() *
MCMSolution:          Math.sqrt(2 * diffusCo * timeStep);
MCMSolution:         // Periodic boundary conditions
MCMSolution:         //
MCMSolution:         //  Exercise: add your boundary conditions here
MCMSolution:         //
MCMSolution:       } // for
MCMSolution:       isDefined = true;
MCMSolution:     } // if
MCMSolution:     return isDefined;
MCMSolution:   } // next
MCMSolution:
MCMSolution: } // class MCMSolution
CHASolution:
CHASolution:
CHASolution: /* $Id: CHASolution.java,v 1.13.4.2 1999/08/05 11:48:12 andrej Exp $ */
CHASolution:
CHASolution: /** The class CHASolution contains the Characteristics solver.
CHASolution:     @version $Revision: 1.13.4.2 $
CHASolution: */
CHASolution: public class CHASolution extends FluidSolution{
CHASolution:   /** Creates a CHASolution object.
CHASolution:       @param mesh The mesh of the solution
CHASolution:   */
CHASolution:   public CHASolution(Mesh mesh){
CHASolution:     super(mesh);
CHASolution:   } // FDSolution
CHASolution:
CHASolution:
CHASolution:   /** Discretize the initial Shape function and initialize the moments
CHASolution:       @param The initial shape to be approximated
CHASolution:    *************************************************************************/
CHASolution:   public void discretize(ShapeFunction function){
CHASolution:     for (int j = 0; j < mesh_.size(); j++)       //Discretizing is simple
CHASolution:       f[j] = function.getValue(mesh_, j);        //  with FEM and splines
CHASolution:     initialMoments[0]=calculateMoments(0);       //Conserved quantities
CHASolution:     initialMoments[1]=calculateMoments(1);
CHASolution:   } // discretize
CHASolution:
CHASolution:
CHASolution:   /** Advance the solution forward one step in time.
CHASolution:       The equation is set by the internal variable equation_ and
CHASolution:       the numerical scheme by the internal variable scheme_
CHASolution:       @param runData List of run parameters
CHASolution:       @see Data#TIMSTP
CHASolution:       @see Data#VELCTY
CHASolution:       @see Data#DIFFUS
CHASolution:       @see Data#THETA
CHASolution:       @see Data#TUNINT
CHASolution:       @return True if a scheme is implemented for that equation
CHASolution:    *************************************************************************/
CHASolution:   public boolean next(Data runData) {
CHASolution:     double timeStep = runData.getDouble(Data.TIMSTP);          //Parameters
CHASolution:     double velocity = runData.getDouble(Data.VELCTY);
CHASolution:     double diffusCo = runData.getDouble(Data.DIFFUS);
CHASolution:     double perLength= (double)mesh_.size()
CHASolution:                               *mesh_.interval(0);
CHASolution:
CHASolution:     double alpha=timeStep*diffusCo/(dx[0]*dx[0]);  //These are constant if
CHASolution:     double beta =timeStep*velocity/(dx[0]      );  // the problem and mesh
CHASolution:                                                    // are homogeneous
CHASolution:     int n=f.length-1;
CHASolution:     boolean isDefined=false;
CHASolution:
CHASolution:     if(equation_.equals(Data.ADVECTION) &&
CHASolution:        scheme_.equals(Data.CUBFEM)){
CHASolution:       //=====================================================================
CHASolution:       // CHA - Advection/diffusion - CubicFEM - Yabe+Aoki CPC 66(1991)219
CHASolution:       //=====================================================================
CHASolution:       double a,b;
CHASolution:       for (int i=0; i<n; i++) {
CHASolution:         a=dx[0]*(df[i]+  df[i+1])-2*(f[i+1]-f[i]);
CHASolution:         b=dx[0]*(df[i]+2*df[i+1])-3*(f[i+1]-f[i]);
CHASolution:         fp[i+1]=  f[i+1] -beta*(dx[0]*df[i+1]-beta*(b-beta*a));
CHASolution:        dfp[i+1]= df[i+1] -beta/dx[0]*(2*b-3*beta*a);
CHASolution:       }
CHASolution:       a=dx[0]*(df[n]+  df[0])-2*(f[0]-f[n]);
CHASolution:       b=dx[0]*(df[n]+2*df[0])-3*(f[0]-f[n]);
CHASolution:       fp[0]=  f[0] -beta*(dx[0]*df[0]-beta*(b-beta*a));
CHASolution:       dfp[0]= df[0] -beta/dx[0]*(2*b-3*beta*a);
CHASolution:       isDefined = true;
CHASolution:
CHASolution:     } else if(equation_.equals(Data.ADVECTION) &&
CHASolution:               scheme_.equals(Data.CUBSPL)){
CHASolution:       //=====================================================================
CHASolution:       //  CHA - Advection/diffusion - Cubic splines
CHASolution:       //=====================================================================
CHASolution:       BandMatrix a= new BandMatrix(3, f.length);
CHASolution:       double[]   c= new double[f.length];
CHASolution:       int        i, i1, i2;
CHASolution:       double     z, z1, z2, dm, dp;
CHASolution:       double     h= dx[0];
CHASolution:
CHASolution:       for (i=0; i<=n; i++) {           // Matrix
CHASolution:         a.setL(i,   h/6 );
CHASolution:         a.setD(i, 4*h/6 );
CHASolution:         a.setR(i,   h/6 );
CHASolution:       }
CHASolution:
CHASolution:       dp= (f[0]-f[n])/h;                //Right hand side 2nd derivative
CHASolution:       for (i=0; i<n; i++) {
CHASolution:         dm= dp; dp= (f[i+1]-f[i])/h;
CHASolution:         c[i]= dp-dm;
CHASolution:       }
CHASolution:       dm= dp; dp= (f[0]-f[n])/h;
CHASolution:       c[n]= dp-dm;
CHASolution:
CHASolution:       dfp=a.solve3(c);                  //Solve linear problem
CHASolution:
CHASolution:       for (i=0; i<=n; i++) {           //Propagate along characteristics
CHASolution:         z= x[i]-timeStep*velocity;     //Allow arbitrary time step
CHASolution:         if (z> perLength) { z= z-perLength*   (int)( z/perLength); }
CHASolution:         if (z< 0)         { z= z+perLength*(1+(int)(-z/perLength)); }
CHASolution:         i1= (int)(z/h); i2=i1+1;
CHASolution:         if (i1 == n)  { i2= 0; }
CHASolution:         z1 = (z-x[i1])/h;  z2 = 1-z1;  //Cubic spline interpolation
CHASolution:         fp[i] = z2*f[i1] +z1*f[i2] +(h*h)/6*((z2*z2*z2 -z2)*dfp[i1] +
CHASolution:                                              (z1*z1*z1 -z1)*dfp[i2] );
CHASolution:       }
CHASolution:       isDefined = true;
CHASolution:
CHASolution:     } else if(equation_.equals(Data.ADVECTION) &&
CHASolution:               scheme_.equals(Data.CHAMINE)){
CHASolution:       //=====================================================================
CHASolution:       //  CHA - Implement your own scheme here
CHASolution:       //=====================================================================
CHASolution:       isDefined = true;
CHASolution:
CHASolution:     } else if(scheme_.equals(Data.CHAEX61)){
CHASolution:       //=====================================================================
CHASolution:       //  FEM - Exercise 6.1
CHASolution:       //=====================================================================
CHASolution:       isDefined = false;
CHASolution:
CHASolution:     } else if(scheme_.equals(Data.CHAEX62)){
CHASolution:       //=====================================================================
CHASolution:       //  FEM - Exercise 6.2
CHASolution:       //=====================================================================
CHASolution:       isDefined = false;
CHASolution:
CHASolution:     } else if(scheme_.equals(Data.CHAEX64)){
CHASolution:       //=====================================================================
CHASolution:       //  FEM - Exercise 6.4
CHASolution:       //=====================================================================
CHASolution:       isDefined = false;
CHASolution:
CHASolution:     } // if
CHASolution:     if(isDefined)
CHASolution:       shiftLevels();
CHASolution:     return isDefined;
CHASolution:   } // next
CHASolution:
CHASolution:   /** Take the solution backward one step to initialize schemes
CHASolution:       with 3 time levels.
CHASolution:       The equation is set by the internal variable equation_ and
CHASolution:       the numerical scheme by the internal variable scheme_
CHASolution:       @param runData List of run parameters
CHASolution:       @see Data#TIMSTP
CHASolution:       @see Data#VELCTY
CHASolution:       @see Data#DIFFUS
CHASolution:       @see Data#THETA
CHASolution:       @see Data#TUNINT
CHASolution:       @return True if an initialisation is implemented for that equation
CHASolution:    *************************************************************************/
CHASolution:   public boolean previous(Data runData){
CHASolution:     int n=f.length-1;
CHASolution:     boolean isDefined=false;
CHASolution:
CHASolution:     if(equation_.equals(Data.ADVECTION) &&
CHASolution:        scheme_.equals(Data.CUBFEM)){
CHASolution:       //=====================================================================
CHASolution:       // CHA - Advection/diffusion - CubicFEM - Yabe+Aoki CPC 66(1991)219
CHASolution:       //=====================================================================
CHASolution:       for (int i=1; i<n; i++) {
CHASolution:         df[i] =(f[i+1]-f[i-1])/(2.*dx[0]);
CHASolution:       }
CHASolution:       df[0]=(f[1]-f[n  ])/(2.*dx[0]);
CHASolution:       df[n]=(f[0]-f[n-1])/(2.*dx[0]);
CHASolution:       isDefined=true;
CHASolution:     }
CHASolution:     return isDefined;
CHASolution:   } // previous
CHASolution:
```

```
CHASolution: } // class CHASolution
ShapeFunction:
ShapeFunction:
ShapeFunction: /* $Id: ShapeFunction.java,v 1.1 1999/06/10 10:15:40 andrej Exp $ */
ShapeFunction:
ShapeFunction: /** Abstract class for a describing initial shapes
ShapeFunction:     @see Solution#discretize
ShapeFunction: */
ShapeFunction: abstract public class ShapeFunction{
ShapeFunction:   /** The shape's vertical scale */    protected double amplitude_;
ShapeFunction:   /** The shape's horizontal scale */  protected double width_;
ShapeFunction:   /** The shape's abscissa */          protected double position_;
ShapeFunction:
ShapeFunction:   /** Evaluates the shape on a discrete mesh point position.
ShapeFunction:       @param mesh The mesh coordinates
ShapeFunction:       @param index The mesh index
ShapeFunction:       @return The value of the function at the specified mesh location
ShapeFunction:   */
ShapeFunction:   abstract public double getValue(Mesh mesh, int index);
ShapeFunction:
ShapeFunction: } // class ShapeFunction
ShapeBox:
ShapeBox:
ShapeBox: /* $Id: ShapeBox.java,v 1.2 1999/06/18 08:41:46 andrej Exp $ */
ShapeBox:
ShapeBox:
ShapeBox: /*****************************************************************************
ShapeBox:  ShapeBox -- A box function
ShapeBox:     @see Solution#discretize
ShapeBox: *****************************************************************************/
ShapeBox: public class ShapeBox extends ShapeFunction{
ShapeBox:
ShapeBox:   /** Creates an instance of the class
ShapeBox:       @param amplitude The box height
ShapeBox:       @param position The box center abscisa
ShapeBox:       @param width The box width
ShapeBox:   */
ShapeBox:   public ShapeBox(double amplitude, double position, double width){
ShapeBox:     amplitude_ = amplitude;
ShapeBox:     position_ = position;
ShapeBox:     width_ = width;
ShapeBox:   } // ShapeBox
ShapeBox:
ShapeBox:   /** Evaluates the shape on a discrete mesh point position.
ShapeBox:       @param mesh The mesh coordinates
ShapeBox:       @param index The mesh index
ShapeBox:       @return The value of the function at the specified mesh location
ShapeBox:   */
ShapeBox:   public double getValue(Mesh mesh, int index){
ShapeBox:     if(mesh.point(index) - position_ > -width_ &&
ShapeBox:        mesh.point(index) - position_ < width_)
ShapeBox:       return amplitude_;
ShapeBox:     else
ShapeBox:       return 0;
ShapeBox:   } // getValue
ShapeBox:
ShapeBox: } // class ShapeBox
ShapeGaussian:
ShapeGaussian:
ShapeGaussian: /* $Id: ShapeGaussian.java,v 1.1 1999/06/10 10:15:41 andrej Exp $ */
ShapeGaussian:
ShapeGaussian: /*****************************************************************************
ShapeGaussian:  ShapeGaussian -- A Gaussian function
ShapeGaussian:     @see Solution#discretize
ShapeGaussian: *****************************************************************************/
ShapeGaussian: public class ShapeGaussian extends ShapeFunction{
ShapeGaussian:
ShapeGaussian:   /** Creates an instance of the class
ShapeGaussian:       @param amplitude The Gaussian maximum amplitude
ShapeGaussian:       @param position The Gaussian abscissa
ShapeGaussian:       @param width The Gaussian width
ShapeGaussian:   */
ShapeGaussian:   public ShapeGaussian(double amplitude, double position, double width){
ShapeGaussian:     amplitude_ = amplitude;
ShapeGaussian:     position_ = position;
ShapeGaussian:     width_ = width;
ShapeGaussian:   } // ShapeGaussian
ShapeGaussian:
ShapeGaussian:   /** Evaluates the shape on a discrete mesh point position.
ShapeGaussian:       @param mesh The mesh coordinates
ShapeGaussian:       @param index The mesh index
ShapeGaussian:       @return The value of the function at the specified mesh location
ShapeGaussian:   */
ShapeGaussian:   public double getValue(Mesh mesh, int index){
ShapeGaussian:     double a = (mesh.point(index) - position_) / width_;
ShapeGaussian:     return amplitude_ * Math.exp(-a * a);
ShapeGaussian:   } // getValue
ShapeGaussian:
ShapeGaussian: } // class ShapeGaussian
ShapeCosinus:
ShapeCosinus:
ShapeCosinus: /* $Id: ShapeCosinus.java,v 1.2 1999/06/18 08:41:46 andrej Exp $ */
ShapeCosinus:
ShapeCosinus: /*****************************************************************************
ShapeCosinus:  ShapeCosinus -- A cosinusoidal function
ShapeCosinus:     @see Solution#discretize
ShapeCosinus: *****************************************************************************/
ShapeCosinus: public class ShapeCosinus extends ShapeFunction{
ShapeCosinus:
ShapeCosinus:   /** Creates an instance of the class
ShapeCosinus:       @param amplitude The wave amplitude
ShapeCosinus:       @param position The wave phase
ShapeCosinus:       @param width The wavelength
ShapeCosinus:   */
ShapeCosinus:   public ShapeCosinus(double amplitude, double position, double width){
ShapeCosinus:     amplitude_ = amplitude;
ShapeCosinus:     position_ = position;
ShapeCosinus:     width_ = width;
ShapeCosinus:   } // ShapeCosinus
ShapeCosinus:
ShapeCosinus:   /** Evaluates the shape on a discrete mesh point position.
ShapeCosinus:       @param mesh The mesh coordinates
ShapeCosinus:       @param index The mesh index
ShapeCosinus:       @return The value of the function at the specified mesh location
ShapeCosinus:   */
ShapeCosinus:   public double getValue(Mesh mesh, int index){
ShapeCosinus:     int    n  = mesh.size();
ShapeCosinus:     double dx = mesh.interval(0);
ShapeCosinus:     double a = 2*Math.PI * (double)(index)*dx / width_;
ShapeCosinus:     return amplitude_ * (1 + Math.cos(a)) / 1.0;
ShapeCosinus:   } // getValue
ShapeCosinus:
ShapeCosinus: } // class ShapeCosinus
ShapeSoliton:
ShapeSoliton:
ShapeSoliton: /* $Id: ShapeSoliton.java,v 1.3 1999/06/14 06:36:59 andrej Exp $ */
ShapeSoliton:
ShapeSoliton: /*****************************************************************************
ShapeSoliton:  ShapeSoliton -- A soliton function
ShapeSoliton:     @see Solution#discretize
ShapeSoliton: *****************************************************************************/
ShapeSoliton: public class ShapeSoliton extends ShapeFunction{
ShapeSoliton:
ShapeSoliton:   /** Creates an instance of the class
ShapeSoliton:       @param amplitude The soliton amplitude
ShapeSoliton:       @param position The soliton position
ShapeSoliton:       @param width Is not appropriate for this shape
ShapeSoliton:   */
ShapeSoliton:   public ShapeSoliton(double amplitude, double position, double width){
ShapeSoliton:     amplitude_ = amplitude;
ShapeSoliton:     position_ = position;
ShapeSoliton:     width_ = width;
ShapeSoliton:   } // ShapeSoliton
ShapeSoliton:
ShapeSoliton:   /** Evaluates the shape on a discrete mesh point position.
ShapeSoliton:       @param mesh The mesh coordinates
ShapeSoliton:       @param index The mesh index
ShapeSoliton:       @return The value of the function at the specified mesh location
ShapeSoliton:   */
ShapeSoliton:   public double getValue(Mesh mesh, int index){
ShapeSoliton:     int    n  = mesh.size();
ShapeSoliton:     double dx = mesh.interval(0);
ShapeSoliton:     double len = (double)n*dx;
ShapeSoliton:     double sca = 2./3.*amplitude_;
ShapeSoliton:     double a = (mesh.point(index) -
ShapeSoliton:   (len-position_))*Math.sqrt(sca/2);
ShapeSoliton:     double result = 3*sca/Math.pow( 0.5*(Math.exp(a)+Math.exp(-a)), 2);
ShapeSoliton:     sca=amplitude_ / 3.0;
ShapeSoliton:     a=(mesh.point(index)-position_) *
ShapeSoliton:       Math.sqrt(sca/2);
ShapeSoliton:     result += 3*sca/Math.pow( 0.5*(Math.exp(a)+Math.exp(-a)), 2);
ShapeSoliton:     return result;
ShapeSoliton:   } // getValue
ShapeSoliton:
ShapeSoliton: } // class ShapeSoliton
ShapeNumerical:
ShapeNumerical:
ShapeNumerical: /* $Id: ShapeNumerical.java,v 1.1 1999/06/10 10:15:41 andrej Exp $ */
ShapeNumerical:
ShapeNumerical: /*****************************************************************************
ShapeNumerical:  ShapeNumerical -- A function (interpolated) from numerical data
ShapeNumerical:     @see Solution#discretize
ShapeNumerical: *****************************************************************************/
ShapeNumerical: public class ShapeNumerical extends ShapeFunction{
ShapeNumerical:
ShapeNumerical:   /** Numerical function abscissa */       private Mesh     myMesh;
ShapeNumerical:   /** Numerical function values */         private Solution mySolution;
ShapeNumerical:
ShapeNumerical:   /** Creates an instance of the class
ShapeNumerical:       @param mesh Coordinate locations where the shape is known
ShapeNumerical:       @param solution Values defining the shape numerically
ShapeNumerical:   */
ShapeNumerical:   public ShapeNumerical(Solution solution){
ShapeNumerical:     mySolution = solution;
ShapeNumerical:   } // ShapeNumerical
ShapeNumerical:
ShapeNumerical:   /** Returns the shape on a discrete mesh point position.
ShapeNumerical:       In a future version, this should include a real interpolation,
ShapeNumerical:       since the mesh coordinates may not coincide with the data !
ShapeNumerical:       @param mesh The mesh coordinates. Now Ignored.
ShapeNumerical:       @param index The mesh index
ShapeNumerical:       @return The value of the function at the specified mesh location
ShapeNumerical:   */
ShapeNumerical:   public double getValue(Mesh mesh, int index){
ShapeNumerical:     return mySolution.getValue(index);
ShapeNumerical:   } // getValue
ShapeNumerical: } // class ShapeNumerical
BandMatrix:
BandMatrix:
BandMatrix: /* $Id: BandMatrix.java,v 1.7 1999/06/18 08:41:42 andrej Exp $ */
BandMatrix:
BandMatrix: /*****************************************************************************
BandMatrix:  * BandMatrix - Linear algebra package for band matrices
BandMatrix:  *****************************************************************************/
BandMatrix: public class BandMatrix {
BandMatrix:   double[][] m;                    //Matrix
BandMatrix:   double[]   d;                    //Temporary storage for diagonal
BandMatrix:   double     det;
BandMatrix:   int        diagos, lines, n;
BandMatrix:   boolean    isDecomposed = false;
BandMatrix:   int l=0;                         //Band indices (left, center, right)
BandMatrix:   int c=1;
BandMatrix:   int r=2;
BandMatrix:
BandMatrix:   /** Constructor */
BandMatrix:   public BandMatrix(int diagos, int lines) {
BandMatrix:     m = new double[diagos][lines];
BandMatrix:     this.diagos  =diagos;
BandMatrix:     this.lines   =lines;
BandMatrix:     this.n       =lines-1;
BandMatrix:   }
BandMatrix:
BandMatrix:   /** Store matrix elements */
BandMatrix:   public void set(int i, int j, double v) { m[j][i]=v; }
BandMatrix:   public void setL(int i, double v)       { m[l][i]=v; }
BandMatrix:   public void setD(int i, double v)       { m[c][i]=v; }
BandMatrix:   public void setR(int i, double v)       { m[r][i]=v; }
BandMatrix:
BandMatrix:   /** Retrieve value of matrix elements */
BandMatrix:   public double get(int j, int i) { return m[j][i]; }
BandMatrix:   public double getL(int i)       { return m[l][i]; }
BandMatrix:   public double getD(int i)       { return m[c][i]; }
BandMatrix:   public double getR(int i)       { return m[r][i]; }
BandMatrix:
BandMatrix:   /** Matrix times vector */
BandMatrix:   public double[] dot(double[] v) {
BandMatrix:     double[] res = new double[lines];
BandMatrix:     int hw=(diagos-1)/2;
BandMatrix:     for (int i=0; i<=n; i++) {
BandMatrix:       for (int j=Math.max(0,hw-i); j<=Math.min(2,hw+n-i); j++) {
BandMatrix:         res[i]=res[i]+m[j][i]*v[i+j-1];
BandMatrix:       }
BandMatrix:     }
BandMatrix:     return res;
BandMatrix:   }
BandMatrix:
BandMatrix:   /** Gaussian elimination for 3-banded matrix with multiple RHS and BC */
BandMatrix:   public double[] solve3(double[] rhs) {
BandMatrix:     double[] sol = new double[lines];
BandMatrix:     int i;
BandMatrix:
BandMatrix:     if (!isDecomposed) {                    //Forward substitution of matrix
BandMatrix:       d = new double[lines];
BandMatrix:       d[0]= m[c][0];
BandMatrix:       m[r][0]= m[r][0]/d[0];
BandMatrix:       m[l][0]=-m[l][0]/d[0];
BandMatrix:       for (i=1; i<n; i++) {
```

```
BandMatrix:          d[i]   = m[c][i]-m[l][i]*m[r][i-1];
BandMatrix:          m[r][i]= m[r][i]/d[i];
BandMatrix:          m[c][i]=-m[l][i]/d[i];
BandMatrix:          m[l][i]=-m[l][i]/d[i]*m[l][i-1];
BandMatrix:        }
BandMatrix:        d[n]=m[c][n]+m[l][n]*(m[l][n-1]-m[r][n-1]);
BandMatrix:        m[r][n]= m[r][n]/d[n];
BandMatrix:        m[c][n]=-m[l][n]/d[n];
BandMatrix:
BandMatrix:        m[l][n-1]=m[l][n-1]-m[r][n-1];      //Backward substitution of matrix
BandMatrix:        for (i=n-2; i>=0; i--) {
BandMatrix:          m[l][i]= m[l][i]-m[r][i]*m[l][i+1];
BandMatrix:        }
BandMatrix:        det = 1.0 + m[r][n]*m[l][0];
BandMatrix:        isDecomposed=true;
BandMatrix:      }
BandMatrix:
BandMatrix:      rhs[0]=rhs[0]/d[0];
BandMatrix:      for (i=1; i<=n; i++) {                    //Forward substitution of rhs
BandMatrix:        rhs[i]=rhs[i]/d[i] +m[c][i]*rhs[i-1];
BandMatrix:      }
BandMatrix:      for (i=n-2; i>=0; i--) {                 //Backward substitution of rhs
BandMatrix:        rhs[i]=rhs[i] -m[r][i]*rhs[i+1];
BandMatrix:      }
BandMatrix:
BandMatrix:      sol[0]=(rhs[0]+m[l][0]*rhs[n])/det;              //Built-in periodicity
BandMatrix:      sol[n]=(rhs[n]-m[r][n]*rhs[0])/det;
BandMatrix:      for (i=1;i<n;i++) {
BandMatrix:        sol[i]=rhs[i]+m[l][i]*sol[n];
BandMatrix:      }
BandMatrix:      return sol;
BandMatrix:    }
BandMatrix:
BandMatrix: } //End of BandMatrix
FFT:
FFT:
FFT: /* $Id: FFT.java,v 1.5.4.4 1999/07/31 14:20:49 andrej Exp $ */
FFT:
FFT: /**************************************************************************
FFT:  * FFT - Object oriented Fast Fourier Transfrom package.
FFT:  *       Adapted with minimal changes to routines from "Numerical Recipes",
FFT:  *       W.H.Press et al., Cambridge University Press (1986)
FFT:  *************************************************************************/
FFT: public class FFT {
FFT:   /* Array dimension                  */   public int size;
FFT:   /* Label configuration space        */   final static int inXSpace  =-1;
FFT:   /* Label Fourier space              */   final static int inKSpace  = 1;
FFT:   /* Label complex number             */   final static int bothParts = 0;
FFT:   /* Label real part or 1st transform */   final static int firstPart = 1;
FFT:   /* Label imag. part or 2nd transform */  final static int secondPart= 2;
FFT:
FFT:   /* Current space location           */   protected int location;
FFT:   /* Dataset stored internally        */   protected Complex[] data;
FFT:   /* True if two real datasets        */   protected boolean isTwoReals;
FFT:   /* NR definition                    */   protected final int toFourier= +1;
FFT:   /* NR definition                    */   protected final int toConfig = -1;
FFT:
FFT:
FFT:   /** FFT object constructed from ONE complex array of dimension power(n,2).
FFT:       @param arg Complex array
FFT:       @param location Either inXSpace or inKSpace
FFT:   *************************************************************************/
FFT:   public FFT(Complex arg[], int location) {
FFT:     this.location = location; isTwoReals = false;
FFT:     size = arg.length;
FFT:     data     = new Complex[size];
FFT:     for (int k=0; k<size; k++) {
FFT:       data[k] = arg[k];
FFT:     }
FFT:   }
FFT:
FFT:
FFT:   /** FFT object constructed from ONE real array of dimension power(n,2).
FFT:       @param arg Real array
FFT:       @param location Either inXSpace or inKSpace
FFT:   *************************************************************************/
FFT:   public FFT(double arg[], int location) {
FFT:     this.location = location; isTwoReals = true;
FFT:     size = arg.length;
FFT:     data     = new Complex[size];
FFT:     for (int k=0; k<size; k++) {
FFT:       data[k] = new Complex(arg[k], 0.);
FFT:     }
FFT:   }
FFT:
FFT:
FFT:   /** FFT object constructed out of TWO complex arrays of dimension power(n,2).
FFT:       @param arg1 First dataset
FFT:       @param arg2 Second dataset
FFT:       @param location Either inXSpace or inKSpace
FFT:   *************************************************************************/
FFT:   public FFT(Complex arg1[], Complex arg2[], int location) {
FFT:     this.location = location; isTwoReals = true;
FFT:     size = arg1.length;
FFT:     data = new Complex[size];
FFT:     for (int k=0; k<size; k++) {
FFT:       data[k] = arg1[k].add((Complex.i).mul(arg2[k]));
FFT:     }
FFT:   }
FFT:
FFT:
FFT:   /** FFT object constructed out of TWO real arrays of dimension power(n,2).
FFT:       @param arg1 First dataset
FFT:       @param arg2 Second dataset
FFT:       @param location Either inXSpace or inKSpace
FFT:   *************************************************************************/
FFT:   public FFT(double arg1[], double arg2[], int location) {
FFT:     this.location = location; isTwoReals = true;
FFT:     size = arg1.length;
FFT:     data = new Complex[size];
FFT:     for (int k=0; k<size; k++) {
FFT:       data[k] = new Complex(arg1[k],arg2[k]);
FFT:     }
FFT:   }
FFT:
FFT:
FFT:   /** FFT object constructed from another FFT object
FFT:       @param fft An FFT object to be duplicated
FFT:   *************************************************************************/
FFT:   public FFT(FFT fft) {
FFT:     this.size = fft.size;
FFT:     this.location = fft.location;
FFT:     this.isTwoReals = fft.isTwoReals;
FFT:     for (int k=0; k<size; k++)
FFT:       this.data[k]=fft.data[k];
FFT:   }
FFT:
FFT:
FFT:   /** Transform array to Fourier space.
FFT:       @param index bothParts = complex number,
FFT:                    firstPart = real part or first transform if there is two,
FFT:                    secondPart = imaginary part or second transform.
FFT:       @param units to scale with for physical results
FFT:       @return Complex array in Fourier space.
FFT:               If isTwoReals, the real and imaginary parts contain the result
FFT:               of transforms of two real numbers.
FFT:       @see FFT
FFT:   *************************************************************************/
FFT:   public Complex[] getFromKSpace(int index, double units) {
FFT:     Complex[] sol = new Complex[size];
FFT:     if (location == inXSpace) {
FFT:       if (isTwoReals) {
FFT:         if (index == firstPart) {
FFT:           Complex[] trash = new Complex[size];
FFT:           twofft(data, sol, trash, size); }
FFT:         else if (index == secondPart) {
FFT:           Complex[] trash = new Complex[size];
FFT:           twofft(data, trash, sol, size); }
FFT:         else { //index == bothParts
FFT:           for (int k=0; k<size; k++) sol[k]=data[k];
FFT:           four1(sol, size, toFourier);
FFT:         }
FFT:         location = inXSpace;       //Don't want to store two spectra inKSpace
FFT:         for (int k=0; k<size; k++)
FFT:           sol[k]=sol[k].scale(1./units);
FFT:       } else { //
FFT:         four1(data, size, toFourier);
FFT:         location = inKSpace;
FFT:         for (int k=0; k<size; k++)
FFT:           sol[k]=data[k].scale(1./units);
FFT:       }
FFT:     } else {
FFT:       for (int k=0; k<size; k++) sol[k]=data[k];
FFT:     }
FFT:     return sol;
FFT:   }
FFT:
FFT:
FFT:   /** Transform array to configuration space.
FFT:       @param index firstPart = real part or first transform if there is two,
FFT:                    secondPart = imaginary part or second transform.
FFT:       @param units to scale with for physical results
FFT:       @return Real array in Fourier space.
FFT:       @see FFT
FFT:   *************************************************************************/
FFT:   public double[] getFromKSpacePart(int index, double units) {
FFT:     Complex[] in = new Complex[size];
FFT:     double[] out = new double[size];
FFT:     in = this.getFromKSpace(index, units);
FFT:     for (int k=0; k<size; k++) {
FFT:       if (index==firstPart) out[k] = in[k].re();
FFT:       else                  out[k] = in[k].im();
FFT:     }
FFT:     return out;
FFT:   }
FFT:
FFT:
FFT:   /** Transform array to configuration space.
FFT:       @param units to scale with for physical results
FFT:       @return Complex array in X space.
FFT:               If isTwoReals, the real and imaginary parts contain the
FFT:               result of two real transforms.
FFT:       @see FFT
FFT:   *************************************************************************/
FFT:   public Complex[] getFromXSpace(double units) {
FFT:     if (location == inKSpace) {
FFT:       four1(data, size, toConfig);                //Inverse transfrom
FFT:       for (int k=0; k<size; k++)                  //  and normalization
FFT:         data[k]=data[k].scale(units/(double)(size));
FFT:       location = inXSpace;
FFT:     }
FFT:     return data;
FFT:   }
FFT:
FFT:
FFT:   /** Transform array to configuration space.
FFT:       @param index firstPart = real part or first transform,
FFT:                    secondPart = iminary part or second transform.
FFT:       @param units to scale with for physical results
FFT:       @return Real or imaginary part of a complex array in Xspace.
FFT:               If isTwoReals, the real and imaginary parts contain the result
FFT:               of two real transforms.
FFT:       @see FFT
FFT:   *************************************************************************/
FFT:   public double[] getFromXSpacePart(int index, double units) {
FFT:     Complex[] in = new Complex[size];
FFT:     double[] out = new double[size];
FFT:     in = this.getFromXSpace(units);
FFT:     for (int k=0; k<size; k++) {
FFT:       if (index==1) out[k] = in[k].re();
FFT:       else          out[k] = in[k].im();
FFT:     }
FFT:     return out;
FFT:   }
FFT:
FFT:
FFT:   /** Computes the convolution of two REAL numbers stored in the real and
FFT:       imaginary part of the data[] array which has previously been initialized.
FFT:       This is implement here WITHOUT any spectral expansion to show the
FFT:       effect of ALIASING in nonlinear problems.
FFT:       @param units to scale with for physical results
FFT:       @return Result of the convolution (carefull ALIASED !)
FFT:       @see expandedConvolution
FFT:       @see FFT
FFT:   *************************************************************************/
FFT:   public Complex[] aliasedConvolution(double units) {
FFT:     Complex[] out = new Complex[size];
FFT:     Complex[] re = new Complex[size];
FFT:     Complex[] im = new Complex[size];
FFT:
FFT:     four1(data, size, toFourier);                          //Transform 2 fields
FFT:
FFT:     for (int k=0; k<size; k++)                              //Convolution =
FFT:      data[k]=new Complex(data[k].re()*data[k].im()/units,0.);// Fourier product
FFT:
FFT:     four1(data, size, toConfig);                           //Inverse transform
FFT:
FFT:     for (int k=0; k<size; k++)
FFT:       data[k]=data[k].scale(units/(double)(size));         //Rescale
FFT:
FFT:     return data;
FFT:   }
FFT:
FFT:
FFT:
FFT:   /** Computes the convolution of two REAL numbers stored in the real and
FFT:       imaginary part of the data[] array which has previously been initialized.
```

```
FFT:      Here, the spectrum is temporarily extended to twice its original size
FFT:      by padding the transforms with zeros.
FFT:      @param units to scale with for physical results
FFT:      @return Result of a correct convolution (with no aliasing).
FFT:      @see aliasedConvolution
FFT:      @see FFT
FFT:   ************************************************************************/
FFT:   public Complex[] expandedConvolution(double units) {
FFT:     Complex[] dExp= new Complex[2*size];
FFT:     Complex[] re= new Complex[2*size];
FFT:     Complex[] im= new Complex[2*size];
FFT:
FFT:     for (int k=0; k<2*size-1; k++)                          //Initialize
FFT:       dExp[k]=new Complex(0.);
FFT:
FFT:     dExp[0]=data[0];
FFT:     for (int k=1; k<=size/2; k++) {                         //Expand array
FFT:       dExp[k       ]=data[k];
FFT:       dExp[2*size-k]=data[size-k];
FFT:     }
FFT:
FFT:     four1(dExp, 2*size, toFourier);                         //Transform 2 fields
FFT:
FFT:     for (int k=0; k<2*size; k++)                              //Convolution =
FFT:      dExp[k]=new Complex(dExp[k].re()*dExp[k].im()/units,0.);// Fourier product
FFT:
FFT:     four1(dExp, 2*size, toConfig);                          //Inverse transform
FFT:
FFT:     data[0]=dExp[0];
FFT:     for (int k=1; k<=size/2; k++) {                         //Rescale
FFT:       data[k     ]=dExp[k].scale(units/(double)(2*size));
FFT:       data[size-k]=dExp[2*size-k].scale(1./(double)(2*size));
FFT:     }
FFT:     return data;
FFT:   }
FFT:
FFT:
FFT:   /** Fast Fourier transforms of two real arrays of dimension power(n,2).
FFT:      Given a complex input array formed by packing two real arrays into
FFT:      real and imaginary parts, this routine calls four1() and returns
FFT:      two complex output arrays fft1[0:n-1] and fft2[0:n-1], each of complex
FFT:      length n, which contain the discrete transforms of the respective
FFT:      data arrays.
FFT:      Adapted with minimal changes from "Numerical Recipes", W.H.Press et al.,
FFT:      Cambridge University Press (1986)
FFT:   ************************************************************************/
FFT:   protected void twofft(Complex datac[],
FFT:                         Complex fft1[], Complex fft2[], int n){
FFT:     Complex h1 = new Complex();
FFT:     Complex h2 = new Complex();
FFT:     Complex c1 = new Complex(0.5, 0.0);
FFT:     Complex c2 = new Complex(0.0,-0.5);
FFT:
FFT:     for (int j=0;j<n;j++)
FFT:       fft1[j] = datac[j];                                   //datac[] remains preserved
FFT:
FFT:     four1(fft1,n,1);                                        //Transform
FFT:
FFT:     fft2[0]=new Complex(fft1[0].im(), 0.0);
FFT:     fft1[0]=new Complex(fft1[0].re(), 0.0);
FFT:     for (int j=1,k=n-j;j<=n/2;j++,k--) {
FFT:       h1=fft1[j]; h1=h1.add(fft1[k].conj()); h1=h1.mul(c1);
FFT:       h2=fft1[j]; h2=h2.sub(fft1[k].conj()); h2=h2.mul(c2);
FFT:       fft1[j]=h1; fft1[k]=h1.conj();
FFT:       fft2[j]=h2; fft2[k]=h2.conj();
FFT:     }
FFT:   }
FFT:
FFT:
FFT:   /** Fast Fourier transform of one complex array of dimension power(n,2).
FFT:      Replaces datac[0:n-1] by its discrete Fourier transform, if isign is 1,
FFT:      or replaces datac[0:n-1] by n times its inverse discrete Fourier
FFT:      transform, if isign is input as -1.
FFT:      Adapted with minimal changes from "Numerical Recipes", W.H.Press et al.,
FFT:      Cambridge University Press (1986)
FFT:   ************************************************************************/
FFT:   protected void four1(Complex datac[], int nh, int isign){
FFT:     int n,mmax,m,j,i,istep = 0;
FFT:     double wtemp,wr,wpr,wpi,wi,theta;
FFT:     double tempr,tempi;
FFT:
FFT:     n = nh << 1;                                            //Interface Complex to real
FFT:     double[] data = new double[n+2];
FFT:     for (int k=1;k<=nh;k++) {
FFT:       data[2*k-1]=datac[k-1].re();
FFT:       data[2*k  ]=datac[k-1].im();
FFT:     }
FFT:
FFT:     j=1;                                                    //Bit reversal section
FFT:     for (i=1;i<n;i+=2) {
FFT:       if (j > i) {                                          //Swap 2 complex numbers
FFT:         tempr=data[j  ];
FFT:         tempi=data[j+1];
FFT:         data[j  ]=data[i  ];
FFT:         data[j+1]=data[i+1];
FFT:         data[i  ]=tempr;
FFT:         data[i+1]=tempi;
FFT:       }
FFT:       m=n >> 1;
FFT:       while (m >= 2 && j > m) {
FFT:         j -= m;
FFT:         m >>= 1;
FFT:       }
FFT:       j += m;
FFT:     }
FFT:     mmax=2;                                                 //Danielson-Lanczos section
FFT:     while (n > mmax) {                                      //  executed log_2(n) times
FFT:       istep=2*mmax;
FFT:       theta=2*Math.PI/(isign*mmax);                         //Initialize trigonometric
FFT:       wtemp=Math.sin(0.5*theta);                            //  recurence
FFT:       wpr = -2.0*wtemp*wtemp;
FFT:       wpi=Math.sin(theta);
FFT:       wr=1.0;
FFT:       wi=0.0;
FFT:       for (m=1;m<mmax;m+=2) {                               //Two nested inner loops
FFT:         for (i=m;i<=n;i+=istep) {
FFT:             j=i+mmax;                                       //Danielson-Lanczos formula
FFT:             tempr=wr*data[j  ]-wi*data[j+1];
FFT:             tempi=wr*data[j+1]+wi*data[j];
FFT:             data[j  ]=data[i  ]-tempr;
FFT:             data[j+1]=data[i+1]-tempi;
FFT:             data[i  ] += tempr;
FFT:             data[i+1] += tempi;
FFT:         }
FFT:         wtemp=wr;                                           //Trigonometric recurrence
FFT:         wr=wtemp*wpr-wi*wpi+wr;
FFT:         wi=wi*wpr+wtemp*wpi+wi;
FFT:       }
FFT:       mmax=istep;
FFT:     }
FFT:     for (int k=1;k<=nh;k++)                                 //Back to Complex
FFT:       datac[k-1] = new Complex(data[2*k-1], data[2*k]);
FFT:   }
FFT:
FFT: }
```

TRITA-ALF-1999-05

Numerical Methods for Partial Differential Equations

A. Jaun, J. Hedin, T. Johnson

Department of Fusion Plasma Physics, Alfvn Laboratory

Royal Institute of Technology, SE-100 44 STOCKHOLM, Sweden

jaun@fusion.kth.se

81 pages

Abstract

A course has been held with 16 graduate students from different institutions geographically dispersed around Stockholm, using tools which have recently become available with the development of the internet.

Focusing common lectures on the introduction of numerical methods such as finite differences / elements, Fourier, Monte-Carlo and Lagrangian methods to solve the advection, diffusion, Black-Scholes, Burger and Korteweg-DeVries equations, students rapidly acquired by context the knowledge required for the electronic publishing of the assignments, helping each other with discussions in the newsgroups.

Each of the scheme discussed in the lecture notes is illustrated with a small section of code in the JBONE applet and can be executed on-line with java powered web-browsers from the link `http://www.fusion.kth.se/courses/pde`

Key words: Course, finite differences, finite elements, Fourier, Monte-Carlo, advection, diffusion, Black-Scholes, Burger, Korteweg-DeVries.