Lab04: Finite difference schemes for hyperbolic equations

1 Semi-discretization

Semi-discretization of the advection equation

$$q_t + u q_x = 0$$

using a centered difference scheme leads to the ODE system

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{Q} = -\frac{u}{2h}\mathbf{B}\boldsymbol{Q}$$

with $\mathbf{B} = \operatorname{diag}([\,-1 \;\; 0 \;\; 1\,])$. We consider the solution of the above ODE system by:

1. Forward Euler (FTCS scheme)

   $$\boldsymbol{Q}^{n+1} = \left(\boldsymbol{I} - \frac{\nu}{2}\mathbf{B}\right)\boldsymbol{Q}^n$$

   with $\nu = uk/h$ known as the CFL number.

2. Midpoint (leap-frog scheme)

   $$\boldsymbol{Q}^{n+1} = \boldsymbol{Q}^{n-1} - \nu\mathbf{B}\boldsymbol{Q}^n$$

3. Lax-Friedrichs

   $$\boldsymbol{Q}^{n+1} = \left(\boldsymbol{\mu} - \frac{\nu}{2}\mathbf{B}\right)\boldsymbol{Q}^n$$

   with $\boldsymbol{\mu} = \frac{1}{2}(\operatorname{diag}([\,1 \;\; 0 \;\; 0\,]) + \operatorname{diag}([\,0 \;\; 0 \;\; 1\,]))$ the averaging operator

4. Lax-Wendroff

   $$\boldsymbol{Q}^{n+1} = \left(\boldsymbol{I} - \frac{\nu}{2}\mathbf{B} + \frac{\nu^2}{2}\mathbf{D}\right)\boldsymbol{Q}^n$$

   with $\mathbf{D} = \operatorname{diag}([\,1 \;\; -2 \;\; 1\,])$

This Lab introduces a number of useful techniques:

- linking efficient compiled implementations (in Fortran) to an interpreted language (Python), and thus develop an environment for interactive investigation of behavior of numerical schemes;
- literate programming within TeXmacs;
- development of a Makefile to automate common tasks

Literate programming is the simultaneous development of code implementing an algorithm with construction of full documentation of the underlying mathematics. Within this document tabular material such as the following examples contains code in the left column and comments on the code in the right column. The following block can be copy/pasted to generate additional blocks.

`! MATH761, Lab04: Finite difference methods for hyperbolic PDEs`	Fortran comments start with !

`// MATH761, Lab04: Finite difference methods for hyperbolic PDEs`	C++ comments start with //

`# MATH761, Lab04: Finite difference methods for hyperbolic PDEs`	Python comments start with #

The `./lab04/Makefile` extracts verbatim all code within the Fortran-code TeXmacs environment into a `lab04.f90` file that is compiled into a Python-loadable module by the `f2py` utility. The TeXmacs file is first converted to LaTeX, after which the `awk` utility is invoked to extract text within the `tmcode[fortran]` environment.

./lab04/Makefile:

```
# TeXmacs literate programming example
SOURCE = lab04.tm
BASEN  = $(basename $(SOURCE))
FCODE  = $(BASEN).f90
MODULE = $(BASEN).so
default: $(MODULE)

# Convert TeXmacs file to LaTex
%.tex : %.tm
        texmacs -c $< $@ -q

# Extract embedded Fortran code
%.f90 : %.tex
        awk ’/\\begin\{tmcode\}\[fortran\]/{flag=1; next} /\\end\{tmcode\}/{flag=0} flag’ $<  > $@

$(MODULE): $(FCODE)
        f2py -c -m $(BASEN) $(FCODE)

clean:
        rm --force *.f90 *.so
```

2 Implementation

2.1 Global definitions

A module is constructed with global definitions.

```
MODULE Global
  INTEGER, PUBLIC, PARAMETER :: sgl = SELECTED_REAL_KIND( 7,16)
  INTEGER, PUBLIC, PARAMETER :: dbl = SELECTED_REAL_KIND(14,32)
  INTEGER, PUBLIC, PARAMETER :: qPrec = dbl, xPrec = dbl
  INTEGER, PUBLIC, PARAMETER :: FTCS=1, Upwind=2, LaxFriedrichs=3, &
           LeapFrog=4, LaxWendroff=5, BeamWarming=6
END MODULE Global
```

SELECTED_REAL_KIND(P,R) is a Fortran intrinsic function that returns the available type closest to the requested precision of P decimal digits and an exponent range R. Two, possibly different, precisions are defined for the dependent variables (q) and the independent variables (space, time).

2.2 Time stepping

```
SUBROUTINE scheme(method,m,nSteps,cfl,Q0,Q1,Q,Qlft,Qrgt)
  USE Global
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: method,m,nSteps
  REAL(KIND=xPrec), INTENT(IN) :: cfl
  REAL(KIND=qPrec), DIMENSION(0:m+1), INTENT(INOUT) :: Q0,Q1
  REAL(KIND=qPrec), DIMENSION(0:m+1), INTENT(OUT) :: Q
  REAL(KIND=qPrec), DIMENSION(0:nSteps), INTENT(IN) :: Qlft,Qrgt
```

A common interface to all finite difference schemes:
method: scheme to apply
m: number of interior nodes
nSteps: number of time steps
cfl: CFL number
Q0,Q1: initial conditions
Q: final state after time stepping
Qlft: boundary values at left
Qrgt: boundary values at right

```
  INTEGER mm1,mp1,n
  REAL(KIND=xPrec) :: hcfl, hcfl2
```

Internal variable declarations

```
  mm1=m-1; mp1=m+1; hcfl=cfl/2; hcfl2=cfl**2/2
```

Precomputation of common expresions

```
  SELECT CASE (method)
```

Start of SELECT statement

```
    CASE (FTCS)
      DO n=1,nSteps
        IF (cfl>0) THEN
          Q0(0) = Qlft(n-1); Q0(mp1) = Q0(m)
        ELSE
          Q0(0) = Q0(1); Q0(mp1) = Qrgt(n-1)
        END IF
        Q(1:m) = Q0(1:m) - hcfl*(Q0(2:mp1) - Q0(0:mm1))
        Q0(1:m) = Q(1:m)
      END DO
```

Carry out time steps.
For $u > 0$, use specified left boundary value, extrapolate at right. For $u < 0$ use specified right boundary value, extrapolate at left.

$Q_i^{n+1} = Q_i^n - \frac{\nu}{2}(Q_{i+1}^n - Q_{i-1}^n)$

```
    CASE (Upwind)
      DO n=1,nSteps
        IF (cfl>0) THEN
          Q0(0) = Qlft(n-1); Q0(mp1) = Q0(m)
          Q(1:m) = Q0(1:m) - cfl*(Q0(1:m) - Q0(0:mm1))
        ELSE
          Q0(0) = Q0(1); Q0(mp1) = Qrgt(n-1)
          Q(1:m) = Q0(1:m) - cfl*(Q0(2:mp1) - Q0(1:m))
        END IF
        Q0(1:m) = Q(1:m)
      END DO
```

$Q_i^{n+1} = Q_i^n - \nu(Q_i^n - Q_{i-1}^n)$ for $u > 0$

$Q_i^{n+1} = Q_i^n - \nu(Q_{i+1}^n - Q_i^n)$ for $u < 0$

```
    CASE (LeapFrog)
      DO n=1,nSteps
        IF (cfl>0) THEN
          Q0(0) = Qlft(n-1); Q1(0) = Qlft(n)
          Q0(mp1) = Q1(m); Q1(mp1) = Q1(m)
        ELSE
          Q0(0) = Q0(1); Q1(1) = Q1(0)
          Q0(mp1) = Qrgt(n-1); Q0(mp1) = Qrgt(n)
        END IF
        Q(1:m) = Q0(1:m) - cfl*(Q1(2:mp1) - Q1(0:mm1))
        Q0(1:m) = Q1(1:m); Q1(1:m) = Q(1:m)
      END DO
```

$Q_i^{n+1} = Q_i^{n-1} - \nu(Q_{i+1}^n - Q_{i-1}^n)$

```
    CASE (LaxWendroff)
      DO n=1,nSteps
        IF (cfl>0) THEN
          Q0(0) = Qlft(n-1); Q0(mp1) = Q1(m)
        ELSE
          Q0(0) = Q0(1); Q0(mp1) = Qrgt(n-1); Q0(mp1) = Qrgt(n)
        END IF
        Q(1:m) = Q0(1:m) - hcfl*(Q0(2:mp1) - Q0(0:mm1)) + &
                 hcfl2*(Q0(2:mp1)-2*Q0(1:m)+Q0(0:mm1))
        Q0(1:m) = Q(1:m)
      END DO
```

$Q_i^{n+1} = Q_i^{n-1} - \frac{\nu}{2}(Q_{i+1}^n - Q_{i-1}^n) + \frac{\nu^2}{2}(Q_{i+1}^n - 2Q_i^n + Q_{i-1}^n)$

```
  END SELECT
END SUBROUTINE scheme
```

3 Numerical experiments

Compilation of the above implementation leads to a Python-loadable module than can be used for numerical experiments.

```
Python] from pylab import *
Python] import os,sys
        os.chdir('/home/student/courses/MATH761/lab04')
        cwd=os.getcwd()
        sys.path.append(cwd)
Python] from lab04 import *
Python] print scheme.__doc__
   q = scheme(method,cfl,q0,q1,qlft,qrgt,[m,nsteps])

   Wrapper for ``scheme``.

   Parameters
   ----------
   method : input int
   cfl : input float
   q0 : in/output rank-1 array('d') with bounds (m + 2)
   q1 : in/output rank-1 array('d') with bounds (m + 2)
   qlft : input rank-1 array('d') with bounds (nsteps + 1)
   qrgt : input rank-1 array('d') with bounds (nsteps + 1)

   Other Parameters
   ----------------
   m : input int, optional
       Default: (len(q0)-2)
   nsteps : input int, optional
       Default: (len(qlft)-1)

   Returns
   -------
   q : rank-1 array('d') with bounds (m + 2)
Python]
```

3.1 Smooth boundary condition

The boundary value problem

$$\begin{cases} q_t + q_x = 0, & t>0, 0<x<1 \\ q(t=0,x)=0, & 0\leqslant x\leqslant 1 \\ q(t,x=0)=g(t)=\sin(2\pi\kappa t), & t>0. \end{cases} \quad (1)$$

is a simplified model of the penetration of a wave into a domain from the left.

```
Python] def f(x,kappa):
          return zeros(size(x))
Python] def g(t,kappa):
          return sin(2*kappa*pi*t)
Python] m=99; h=1./(m+1); x=arange(m+2)*h;
Python] u=1;
Python] FTCS=1; Upwind=2; LaxFriedrichs=3; LeapFrog=4; LaxWendroff=5; BeamWarming=6;
Python]
```

3.1.1 FTCS

The Fortran code can be invoked from within Python for interactive investigation of the behavior of the numerical schemes.

```
Python] method=FTCS; nSteps=100; cfl=1; k=h*cfl/u; t=arange(nSteps+1)*k; kappa=4;
```

```
Python] q0=f(x,kappa); q1=zeros(size(x)); Q0=f(x,kappa); Q1=zeros(size(x));
Python] Qlft=g(t,kappa); Qrgt=zeros(size(t));
Python] Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
Python] plot(x[1:m],q0[1:m],'b',x[1:m],Q1[1:m],'r');
        xlabel('x'); ylabel('q'); title('FTCS solution of advection equation');
        savefig("Lab04Fig01.pdf");
Python]
```

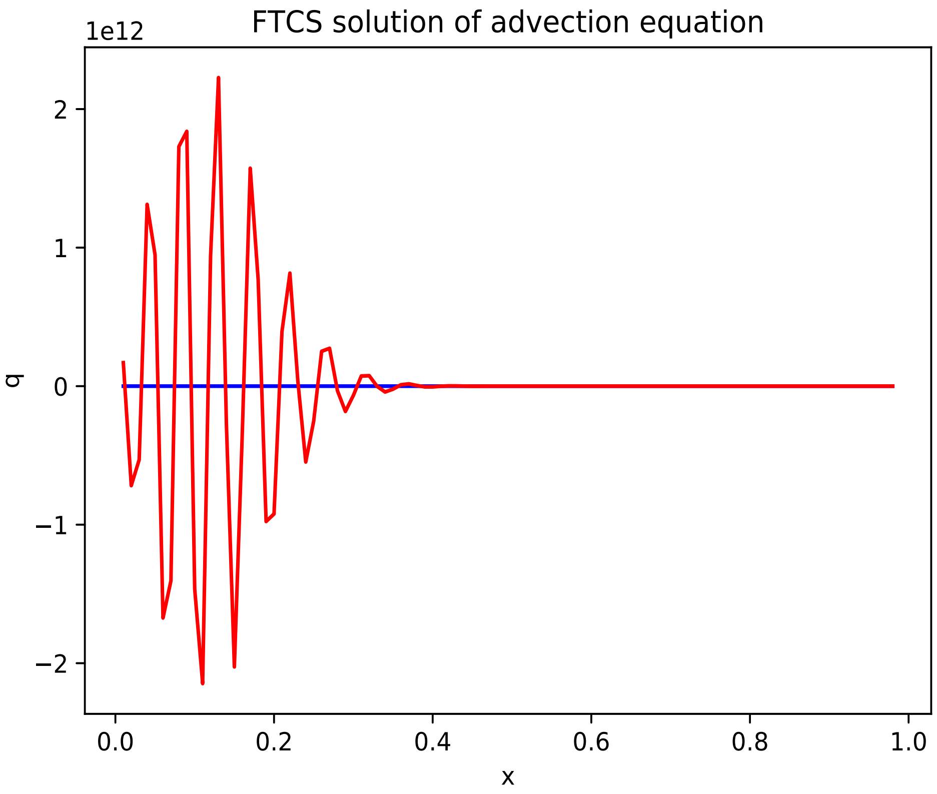

Figure 1. Instability of FTCS scheme

3.1.2 Upwind

More importantly, the Fortran code can be included in Python loops to study the influence of discretization parameters. For example, the predictions of a scheme at different CFL numbers can be investigated. Note that all Python commands are now grouped in a single input field to be executed together, thus allowing efficient modification of initial, boundary conditions, numerical parameters.

```
Python] method=Upwind;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.1; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x,kappa); Q1=zeros(size(x));
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Effect of CFL upon Upwind solution of advection equation');
        savefig("Lab04UpwindSmoothInitCond.pdf");
```

```
Python]
```

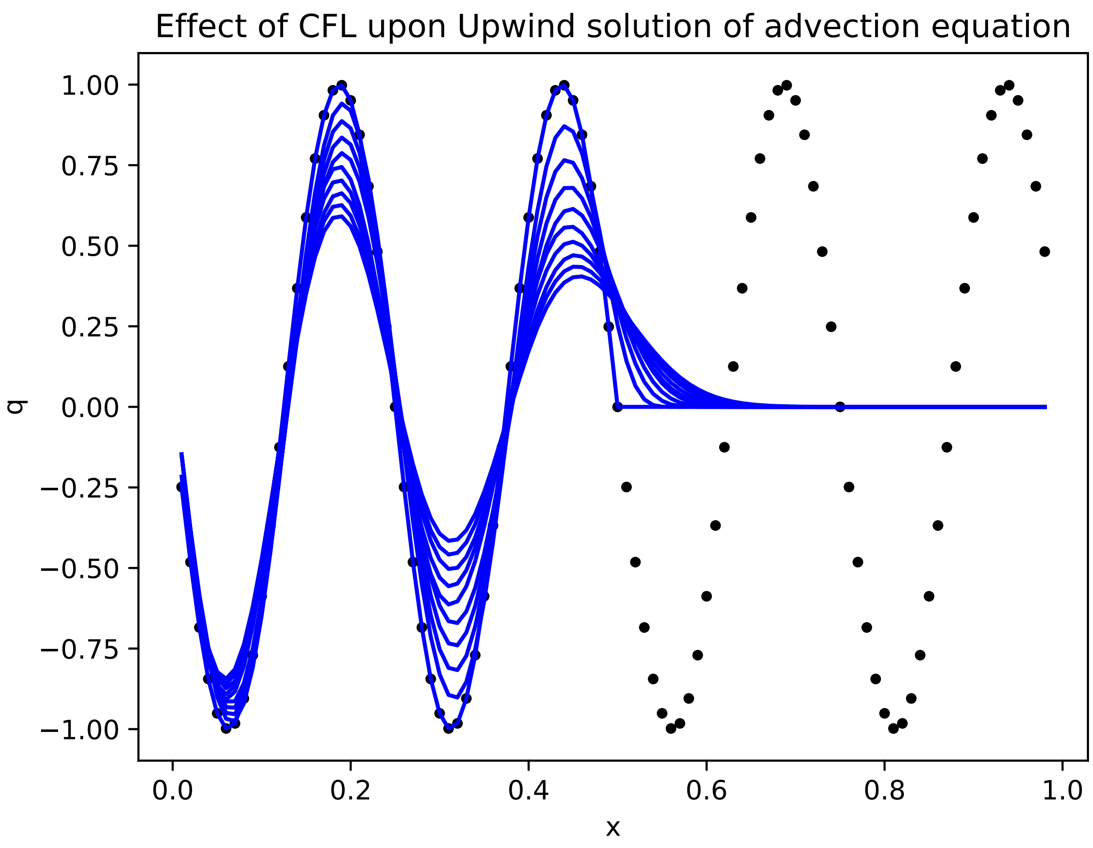

Figure 2. Upwind scheme exhibits large artificial diffusion for CFL < 1.

3.1.3 Leapfrog

```
Python] method=LeapFrog;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.1; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x-u*k,kappa); Q1=f(x,kappa);
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Effect of CFL upon leap-frog solution of advection equation');
        savefig("Lab04LeapFrogSmoothInitCond.pdf");
```

```
Python]
```

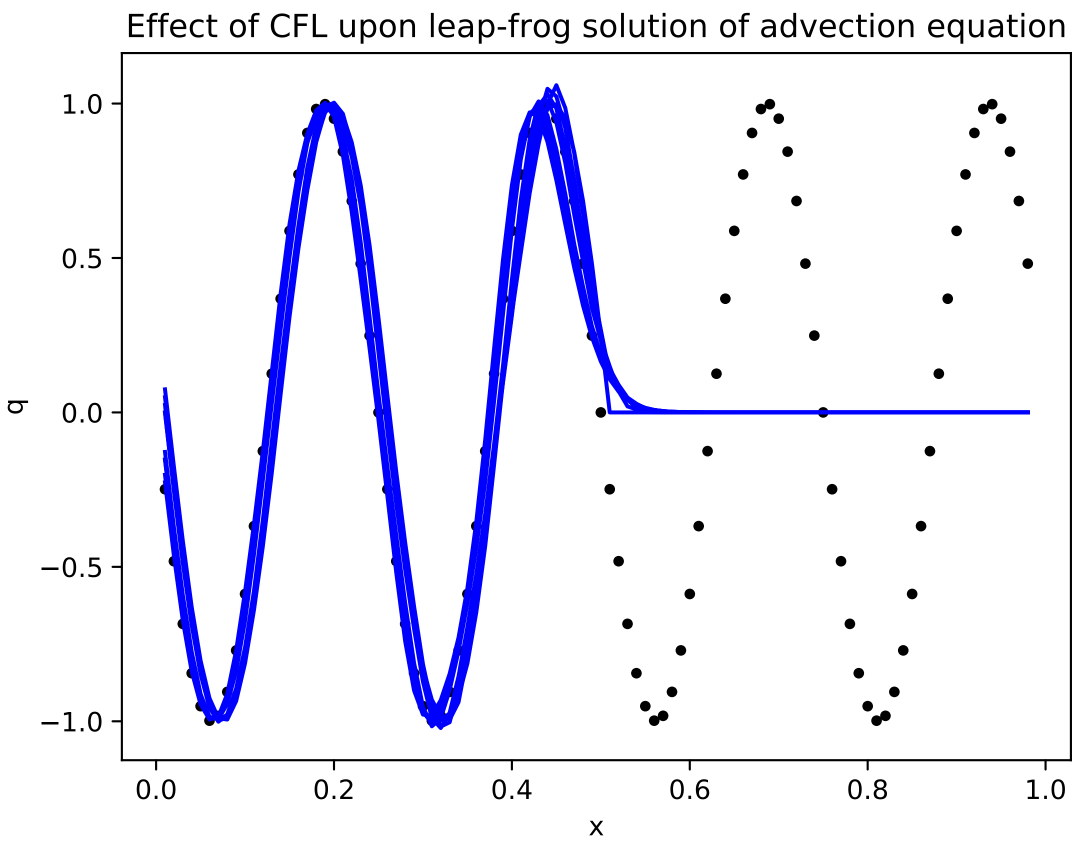

Figure 3. Upwind scheme

3.1.4 Lax-Wendroff

```
Python] method=LaxWendroff;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.2; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x,kappa); Q1=zeros(size(x));
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Effect of CFL upon Lax-Wendroff solution of advection equation');
        savefig("Lab04LaxWendroffSmoothInitCond.pdf");
Python]
```

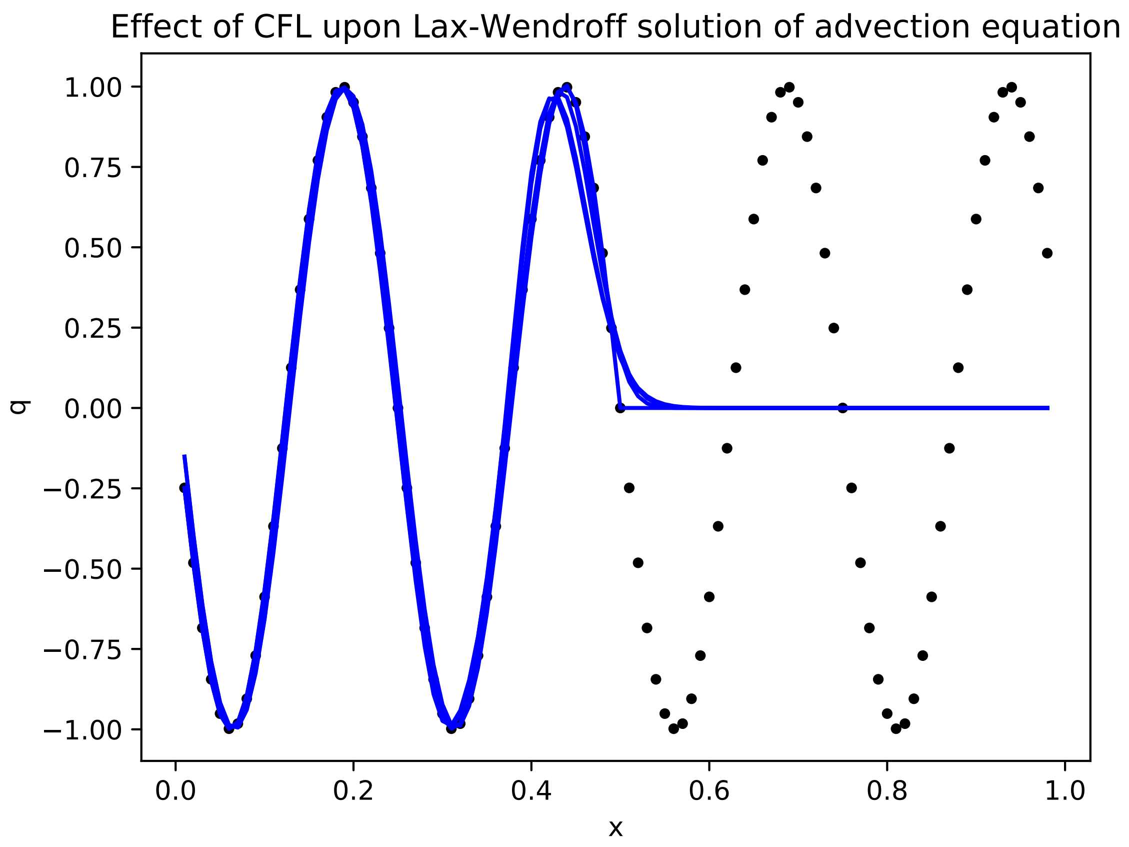

Figure 4. Lax-Wendroff

3.2 Discontinuous boundary condition

Study now the behavior for a discontinuous (shock) initial condition

```
Python] def f(x,kappa):
          return zeros(size(x))
        def g(t,kappa):
          return (sign(t)+1)/2
        u=1;
Python] kappa=4
Python]
```

3.2.1 Upwind

```
Python] method=Upwind;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.1; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x,kappa); Q1=zeros(size(x));
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Shock propagation by upwind scheme');
        savefig("Lab04UpwindShockInitCond.pdf");
```

```
Python]
```

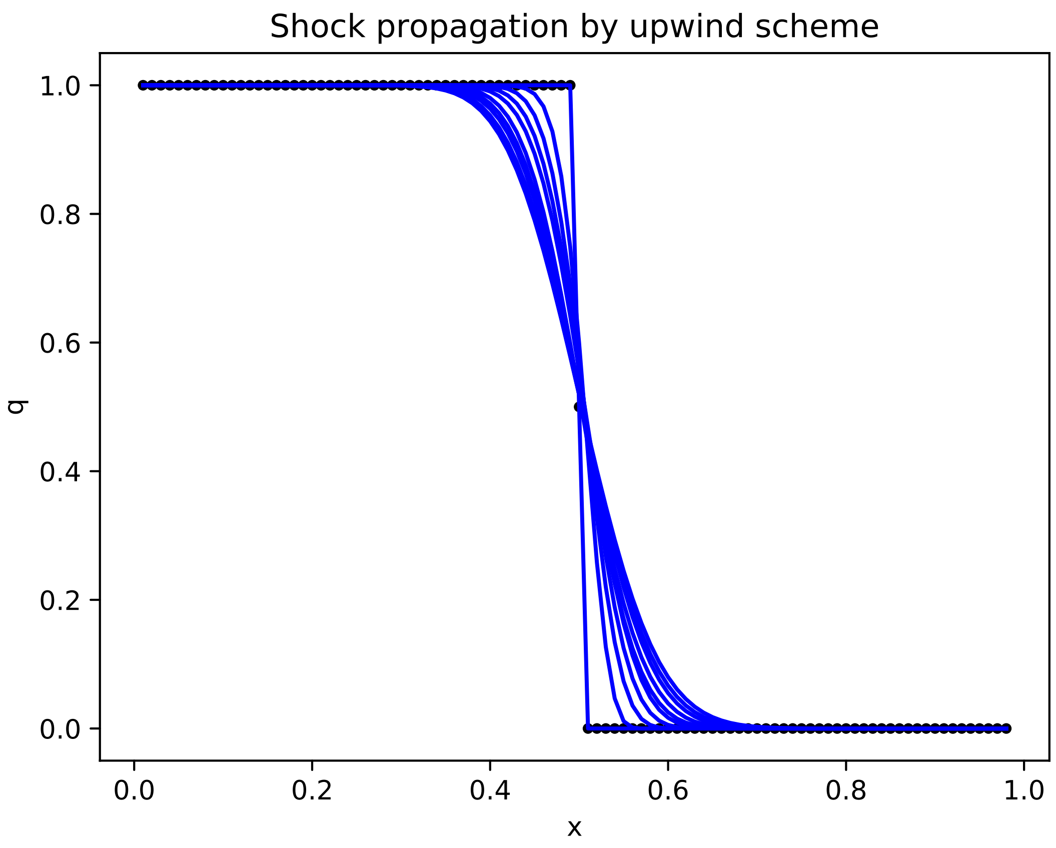

Figure 5. Upwind scheme exhibits diffusion but predicts correct shock position

3.2.2 Leapfrog

```
Python] method=LeapFrog;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.2; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x-u*k,kappa); Q1=f(x,kappa);
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Shock propagation by leap-frog scheme');
        savefig("Lab04LeapFrogShockInitCond.pdf");
```

```
Python]
```

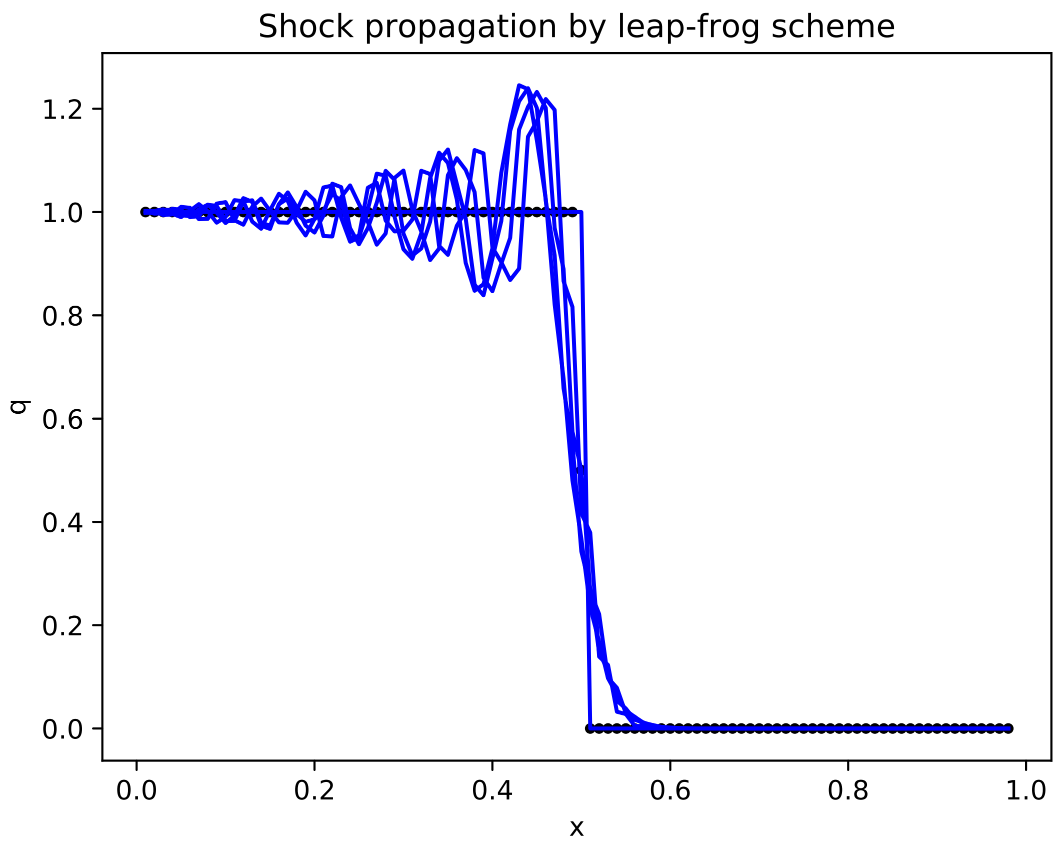

Figure 6. Leap-frog exhibits artificial diffusion, dispersion, Gibbs oscillations

3.2.3 Lax-Wendroff

```
Python] method=LaxWendroff;
        m=99; h=1./(m+1); x=arange(m+2)*h;
        dcfl=0.2; tfinal=0.5;
        clf();
        qex=g(tfinal-x/u,kappa);
        plot(x[1:m],qex[1:m],'k.');
        for cfl in arange(dcfl,1+dcfl,dcfl):
          k=h*cfl/u; nSteps=ceil(tfinal/k); t=arange(nSteps+1)*k; kappa=4;
          Q0=f(x,kappa); Q1=zeros(size(x));
          Qlft=g(t,kappa); Qrgt=zeros(size(t));
          Q1=scheme(method,cfl,Q0,Q1,Qlft,Qrgt);
          plot(x[1:m],Q1[1:m],'b');
        xlabel('x'); ylabel('q'); title('Shock propagation by Lax-Wendroff');
        savefig("Lab04LaxWendroffShockInitCond.pdf");
Python]
```

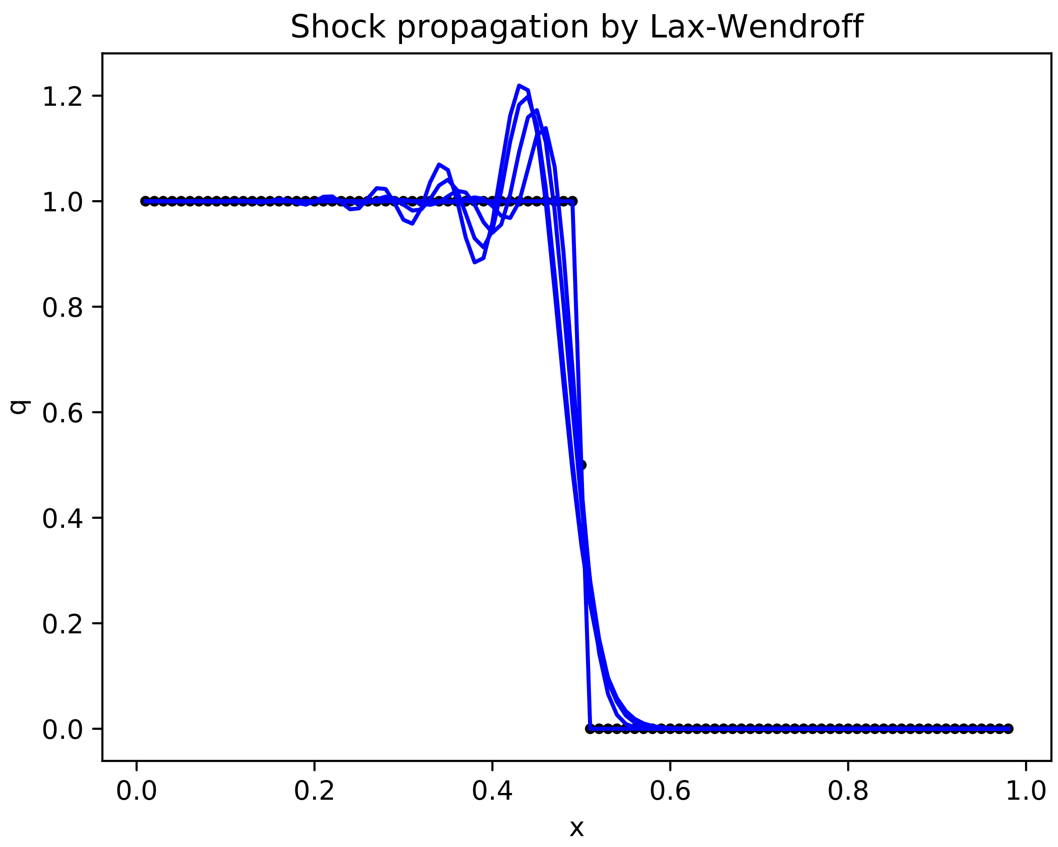

Figure 7. Lax-Wendroff is similar to leap-frog with more rapidly decaying Gibbs oscillations