Tag der Talente - Berlin - November 2017

Wellenausbreitung in 1D ¶

Jörn Behrens (http://www.clisap.de/behrens) (joern.behrens@uni-hamburg.de) (mailto:joern.behrens@uni-hamburg.de)

Einführung

In diesem Notebook wollen wir die Ausbreitung einer Wasserwelle (Schwerewelle) in einer Dimension simulieren. Wir stellen uns vor, einen Kanal vorzufinden, in dem sich eine Welle in nur einer Koordinaten-Richtung ausbreiten kann (natürlich kann sie nach rechts und auch nach links laufen).

Die Mathematische Modellierung basiert auf zwei physikalischen Erhaltungsprinzipien:

- Der Massenerhaltung und
- der Impulserhaltung.

Die Massenerhaltung wird durch die sogenannte *Kontinuitätsgleichung* dargestellt:

$$h_t + (hu)_x = 0.$$

Dabei bezeichnet $h = h(x,t)$ die Wellenhöhe (oder Wassertiefe) und $u = u(x,t)$ die Partikelgeschwindigkeit. Die Raum-Zeit-Koordinate ist gegeben durch (x,t) und wir verwenden die vereinfachende Schreibweise

$$h_t = \frac{\partial h}{\partial t} \quad (hu)_x = \frac{\partial (hu)}{\partial x}$$

für zeitliche bzw. räumliche partielle Ableitungen.

Die Impulserhaltung kann (unter Vernachlässigung von Kräften außer der Gravitationskraft) durch die folgende *Impulsgleichung* dargestellt werden:

$$(hu)_t + [(hu)u]_x + (\frac{g}{2}h^2)_x = 0.$$

In dieser Gleichung bezeichnet $g = 9,81$ die Erdbeschleunigungskonstante, die anderen Terme sind analog zu oben zu verstehen.

Diskretisierung

Der Schritt der Diskretisierung besteht nun darin, diese Gleichungen so zu formulieren, dass sie mit Hilfe eines Computer-Algorithmus gelöst werden können. Dazu ist zunächst festzustellen, dass ein Computer eigentlich nur elementare Rechenoperationen ($+, -, \cdot, \div$) ausführen kann. Wir werden also die Differential-Operatoren durch Differenzenoperatoren ersetzen.

Motiviert ist dieses Vorgehen durch die Definition einer Ableitung. Betrachten wir beispielsweise die partielle Ableitung h_t, so ist sie formal definiert als ein Grenzwert

$$\lim_{\tau \to 0} \frac{h(x, t+\tau) - h(x,t)}{\tau}.$$

In der Diskretisierung werden wir nun ein (kleines!) τ fest wählen und statt des Grenzwertes den Differenzenoperator direkt verwenden. Wir bezeichnen mit τ ein kleines Zeitintervall und mit λ ein kleines Raumintervall.

Damit lässt sich die Kontinuitätsgleichung in der folgenden Form schreiben:

$$\frac{h(x, t+\tau) - h(x,t)}{\tau} + \frac{(hu)(x+\lambda, t) - (hu)(x,t)}{\lambda} = 0.$$

Entsprechend kann die Impulsgleichung geschrieben werden als

$$\frac{(hu)(x, t+\tau) - (hu)(x,t)}{\tau} + \frac{((hu)^2/h)(x+\lambda, t) - ((hu)^2/h)(x-\lambda,t)}{\lambda} + \frac{1}{2}\frac{gh^2(x+\lambda,t) - }{\lambda}$$

Gitterfunktion

Um dem Computer die Rechnung weiter zu vereinfachen, werden wir nun sogenannte *Gitterfunktionen* einführen. Dazu definieren wir uns zunächst ein Gitter mit Punkten in Ort und Zeit. Wir werden also unsere Funktionen nicht kontinuierlich an beliebigen (x, t) Koordinaten berechnen, sondern lediglich an endlich vielen Punkten (x_i, t^j), wobei i und j ganze Zahlen ($i, j = 0, 1, 2, \ldots$) sind und

$$x_i = i \cdot \lambda \quad t^j = j \cdot \tau.$$

Nun vereinfachen wir unsere Schreibweise weiter, indem wir setzen $h_i^j = h(x_i, t^j)$ (analog für die anderen Variablen).

Damit noch nicht genug: Wir wenden nun sogenannte zentrale Differenzenquotienten an, die numerisch etwas genauer sind, als die oben dargestellten Vorwärts-Differenzenquotienten. Schließlich schreiben wir alle Terme zum zukünftigen Zeitpunkt ($t + \tau$ bzw. $j + 1$) auf die linke und alle übrigen Terme auf die Rechte Seite des Gleichheitszeichens. Damit ergeben sich

$$h_i^{j+1} = h_i^{j-1} - \frac{\tau}{\lambda}\left[(hu)_{i+1}^j - (hu)_{i-1}^j\right]$$

bzw.

$$(hu)_i^{j+1} = (hu)_i^{j-1} - \frac{\tau}{\lambda}\left[[(hu)^2/h + \frac{1}{2}g(h^2)]_{i+1}^j - [(hu)^2/h + \frac{1}{2}g(h^2)]_{i-1}^j\right]$$

Visualisierung

Bevor wir beginnen, benötigen wir ein Verfahren, um die Ergebnisse unserer Rechnung zu sehen. Dazu implementieren wir eine Funktion, die die Daten als Graphen darstellt. Wir nennen diese Funktion plotswe.

In [1]:
```
def plotswe(fig,x,t,dx,dt,hu,h,u):
    import matplotlib.pyplot as plt

    plt.clf()
    #-- plot the graphs of u and h (we neglect the momentum for the ti
    h1=plt.plot(x,h,linewidth=2, c=[0.7, 0.2, 0.5], label='h')
    h3=plt.plot(x,u,linewidth=2, c=[0.2, 0.2, 0.7], label='u')

    #-- plot figure title, legend and axis lables
    plt.legend(loc='upper left')
    plt.title('SWE, DX=' + '%6.4f' % (dx) + ', DT=' + '%6.4f' % (dt))
    plt.xlabel('x')
    plt.ylabel('h,u')

    filename='swe'+'%6.6d'%(t)+'.png'
    fig.savefig(filename)

    #fig.set_size_inches(10,7)
    #plt.show()
    return
```

FTCS Schema

Zunächst implementieren wir ein FTCS Schema, das ähnlich zu der oben aufgeführten Formel ausschaut, jedoch für alle Terme auf der rechten Seite die Zeit j statt $j-1$ annimmt.

In [2]:
```
from numpy import arange
def ftcs(fig,x,dt,T,huinit,hinit):

    #-- set constant
    g= 9.81       #- gravitational acceleration
    count= 0

    #-- compute spatial step size
    dx= x[1]-x[0]

    #-- initial condition; set up future time array
    hu0 = huinit.copy()
    hu1 = hu0.copy()
    h0  = hinit.copy()
    h1  = h0.copy()
    u0  = hu0/h0

    #-- now the time loop
    for t in arange(0,T,dt):
        count = count+1
        nu= dt/(2.*dx)
        h1[1:(len(x)-2)]  = h0[1:(len(x)-2)] - nu* (hu0[2:(len(x)-1)]-h
        hu1[1:(len(x)-2)] = hu0[1:(len(x)-2)] - nu* (hu0[2:(len(x)-1)]
                                                - (hu0[0:(len(x)-3)]

    #-- periodic boundary handling
        h1[0]  = h0[0] - nu* (hu0[1]-hu0[len(x)-1])
        h1[len(x)-1]  = h0[len(x)-1] - nu* (hu0[0]-hu0[len(x)-2])
        hu1[0] = hu0[0] - nu* (hu0[1]**2/h0[1] + 0.5*g*h0[1]**2 \
                            - (hu0[len(x)-1]**2/h0[len(x)-1] + 0.5*g*h
        hu1[len(x)-1] = hu0[len(x)-1] - nu* (hu0[0]**2/h0[0] + 0.5*g*h
                            - (hu0[len(x)-2]**2/h0[len(x)-2] + 0.5*g*h

    #-- updates for next time step
        u0  = hu1/h1
        h0  = h1.copy()
        hu0 = hu1.copy()
    #-- visualization
        plotswe(fig,x,count,dx,dt,hu0,h0,u0)


    return hu0, h0, u0
```

Leap-Frog Schema

Die Funktion unten implementiert nun die Gleichungen. Allerdings mit einer kleinen Modifikation: Die Zeitableitung ist um einen sogenannten Asselin Filter erweitert. Dies ist notwendig, um das Verfahren stabil zu machen.

In [3]:
```
def leapfrog(fig,x,dt,T,huinit,hinit):
```

```
    #-- set constant
    a= 0.05       #- Asselin time filter
    g= 9.81       #- gravitational acceleration
    count= 0

    #-- compute spatial step size
    dx= x[1]-x[0]

    #-- initial condition; set up future time array
    h0  = hinit.copy()
    h1  = h0.copy()
    h2  = h1.copy()
    hu0 = huinit.copy()
    hu1 = hu0.copy()
    hu2 = hu1.copy()

    #-- first step by FCTS
    hu2, h2, u1 = ftcs(fig,x,dt,dt,hu0,h0)

    #-- now the time loop
    for t in arange(0,T,dt):

    #-- Asselin time filter (swich off, by setting a=0)
        h0  = h1 + a* (h2 -(2.*h1) + h0)
        h1  = h2.copy()
        hu0 = hu1 + a* (hu2 -(2.*hu1) + hu0)
        hu1 = hu2.copy()

    #-- the main step
        count = count+1
        nu= dt/dx
        h2[1:(len(x)-2)]  = h0[1:(len(x)-2)] - nu* (hu1[2:(len(x)-1)]-h
        hu2[1:(len(x)-2)] = hu0[1:(len(x)-2)] - nu* (hu1[2:(len(x)-1)]
                                                - (hu1[0:(len(x)-3)]

    #-- periodic boundary handling
        h2[0]  = h0[0] - nu* (hu1[1]-hu1[len(x)-1])
        h2[len(x)-1]  = h0[len(x)-1] - nu* (hu1[0]-hu1[len(x)-2])
        hu2[0] = hu0[0] - nu* (hu1[1]**2/h1[1] + 0.5*g*h1[1]**2 \
                            - (hu1[len(x)-1]**2/h1[len(x)-1] + 0.5*g*h
        hu2[len(x)-1] = hu0[len(x)-1] - nu* (hu1[0]**2/h1[0] + 0.5*g*h
                            - (hu1[len(x)-2]**2/h1[len(x)-2] + 0.5*g*h

    #-- updates for next time step
        u1  = hu2/h2
    #-- visualization
        plotswe(fig,x,count,dx,dt,hu2,h2,u1)


    #-- set output value
    return hu2, h2, u1
```

McCormack Schema

Als nächstes implementieren wir noch ein sogenanntes McCormack Schema, das bessere Genauigkeit und auch Stabilität besitzt.

In [4]:
```
def mccormack(fig,x,dt,T,huinit,hinit):

    #-- set constant
    g= 9.81       #- gravitational acceleration
    count= 0

    #-- compute spatial step size
    dx= x[1]-x[0]

    #-- initial condition; set up future time array
    hu0    = huinit.copy()
    hu1    = hu0.copy()
    hustar = hu0.copy()
    h0     = hinit.copy()
    h1     = h0.copy()
    hstar  = h0.copy()
    u0     = hu0/h0

    #-- now the time loop
    for t in arange(0,T,dt):
        count = count+1
        nu  = dt/dx
        nu2 = dt/(2.*dx)
    #-- first half step
        hstar[0:(len(x)-2)]  = h0[0:(len(x)-2)] - nu* (hu0[1:(len(x)-1
        hustar[0:(len(x)-2)] = hu0[0:(len(x)-2)] - nu* (hu0[1:(len(x)-
                                                        - hu0[0:(len(x)-2

    #-- second half step
        h1[1:(len(x)-1)]  = 0.5* (h0[1:(len(x)-1)] + hstar[1:(len(x)-1
                                 - nu2* (hustar[1:(len(x)-1)]-hustar[(
        hu1[1:(len(x)-1)] = 0.5* (hu0[1:(len(x)-1)] + hustar[1:(len(x)-
                           - nu2* (hustar[1:(len(x)-1)]**2/hstar[1:(le
                                -  hustar[0:(len(x)-2)]**2/hstar[0:(le

    #-- periodic boundaries
        h1[0]  = h1[(len(x)-1)]
        h1[1]  = h1[(len(x)-2)]
        hu1[0]  = hu1[(len(x)-1)]
        hu1[1]  = hu1[(len(x)-2)]


    #-- updates for next time step
        h0  = h1.copy()
        hu0 = hu1.copy()
        u0  = hu0/h0
    #-- visualization
        plotswe(fig,x,count,dx,dt,hu0,h0,u0)

    return hu0, h0, u0
```

Lax-Friedrichs Schema

Und schließlich noch ein weitere Schema, das auch bedingt stabil ist, aber recht stark dämpft, das Lax-Friedrichs Verfahren.

In [5]:
```
def laxfriedrichs(fig,x,dt,T,huinit,hinit):

    #-- set constant
    g= 9.81       #- gravitational acceleration
    count= 0      #- counter for the plots

    #-- compute spatial step size
    dx= x[1]-x[0]

    #-- initial condition; set up future time array
    hu0 = huinit.copy()
    hu1 = hu0.copy()
    h0  = hinit.copy()
    h1  = h0.copy()
    u0  = hu0/h0

    #-- now the time loop
    for t in arange(0,T,dt):
        count = count+1
        nu= dt/(2.*dx)
        h1[1:(len(x)-2)]  = 0.5*(h0[2:(len(x)-1)]+h0[0:(len(x)-3)]) - 
        hu1[1:(len(x)-2)] = 0.5*(hu0[2:(len(x)-1)]+hu0[0:(len(x)-3)]) 
                                                - (hu0[0:(len(x)-3)]

    #-- periodic boundary handling
        h1[0]  = 0.5*(h0[1]+h0[len(x)-1]) - nu* (hu0[1]-hu0[len(x)-1])
        h1[len(x)-1]  = 0.5*(h0[0]+h0[len(x)-2]) - nu* (hu0[0]-hu0[len
        hu1[0] = 0.5*(hu0[1]+hu0[len(x)-1]) - nu* (hu0[1]**2/h0[1] + 0
                            - (hu0[len(x)-1]**2/h0[len(x)-1] + 0.5*g*h
        hu1[len(x)-1] = 0.5*(hu0[0]+hu0[len(x)-2]) - nu* (hu0[0]**2/h0
                            - (hu0[len(x)-2]**2/h0[len(x)-2] + 0.5*g*h

    #-- updates for next time step
        h0  = h1.copy()
        hu0 = hu1.copy()
        u0  = hu0/h0
    #-- visualization
        plotswe(fig,x,count,dx,dt,hu0,h0,u0)

    return hu0, h0, u0
```

Anfangsbedingungen

Wir werden annehmen, dass eine Anfangswelle ausschließlich durch eine ausgelenkte Wasserhöhe zustande kommt. Wir werden also den Impuls und die Partikelgeschwindigkeit (etwas inkonsistent) zu Null setzen, $hu(x,0) = u(x,0) \equiv 0$, und die Auslenkung der Wasseroberfläche mit einer *Gaußglocke* von geringer Höhe (gegenüber der Wassertiefe) initialisieren.

$$h(x,0) = H + \nu[e^{-\frac{(20x)^2}{2}}],$$

wobei H eine mittlere Wassertiefe ist ($H = 1$) und ν ein Skalierungsfaktor, der dafür sorgt, dass die Wellenhöhe $0.01H$ nicht überschreitet.

In [6]:
```
from numpy import exp, zeros, size
def init(x):
    HH  = 1.0
    nu  = 0.1
    u0  = zeros(size(x))
    hu0 = u0.copy()
    h0  = HH+ nu*(exp(-(20*x)**2/2))
    return hu0, h0, u0
```

Haupt-Program

Nachdem wir nun Funktionen geschrieben haben, welche Anfangsbedingungen setzen und die Wellengleichungen in einem periodischen Gebiet lösen können, wollen wir nun eine Simulation durchführen. Das geschieht in vier Schritten:

1. Setzen von Parametern, wie der Ortsschrittweite λ und des Zeitschrittes τ, etc.
2. Berechnen der Anfangswerte $hu(x,0)$, $h(x,0)$ und $u(x,0)$.
3. Aufruf des Lösungsverfahrens.
4. Visualisierung der Lösung.

In [8]:
```
def testswe():
    from numpy import arange, pi, exp, cos, inf
    import matplotlib.pyplot as plt
#    %matplotlib inline

    #-- initial values/settings
    dx= 1./200.              #- spatial step size
    courant= 0.25             #- Courant No.
    dt= courant* dx          #- time step size
    Tend= 0.2               #- final time
    x= arange(-1,1+dx,dx)    #- spatial grid

    #-- initial conditions
    hu0, h0, u0=init(x)

    #-- initialize visualization
    fig = plt.figure(1)
    plotswe(fig,x,0,dx,dt,hu0,h0,u0)

    #-- leap-frog scheme
    hu, h, u= mccormack(fig,x,dt,Tend,hu0,h0)

    #-- plot the result, the initial condition, and the exact solution
#    plotswe(fig,x,dx,dt,hu,h,u)
    plt.close(fig)

if __name__=="__main__":
    testswe()
```

In [ ]:
```
```