Assignment 1 - Advanced Computational Science

Liam O'Sullivan – 10309537

March 12, 2014

Question 1

(1)

We are interested in solving the 1-d wave equation with $c = 1$,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} \tag{1}$$

using central differences,

$$\frac{\partial^2 u}{\partial t^2} \approx \frac{\delta_t^2 u_j^n}{\tau^2}, \qquad \frac{\partial^2 u}{\partial x^2} \approx \frac{\delta_x^2 u_j^n}{h^2} \tag{2}$$

so our equation becomes

$$\frac{\delta_t^2 u_j^n}{\tau^2} = \frac{\delta_x^2 u_j^n}{h^2} \tag{3}$$

where τ is the time step size, and h is the x step size.

The operator δ_x^2, when applied to u_j^n, gives

$$\delta_x^2 = u_j^{n-1} - 2u_j^n + u_j^{n+1}$$

allowing us to expand out for both spatial and temporal derivatives and rearrange for u_j^{n+1}, giving

$$u_j^{n+1} = \nu^2 \left[u_{j-1}^n + u_{j+1}^n\right] - u_j^{n-1} + 2(1-\nu^2)u_j^n \tag{4}$$

with $\nu = \frac{\tau}{h}$.

A program was written in Python 3.3 to solve this problem with Dirichlet boundary conditions, $u(-7,t) = u(7,t) = 0$, on the intervals $x \in [-7,7]$, $t \in [0,14]$. The initial condition is given by $u(x,0) = e^{-x^2}$. The program is included in appendix A.

The solutions at various times are shown in figure 1. The wave is observed to split in two, propagating half in either direction. At the boundaries, the wave is inverted, travelling back to superimpose into a mirror image of the initial condition at $t = 14$.

(2)

Our expression for u_j^n using Central Differences is given in equation 4. To perform Fourier/von Neumann stability analysis, we take $u_j^n = \xi^n e^{ikhj}$ as a single Fourier mode solution, and see how our *amplification factor*, ξ, behaves as a function of τ, h, k. Substituting this into equation 4 gives

$$\xi^{n+1}e^{ikhj} = \nu^2\left[\xi^n e^{ikh(j-1)} + \xi^n e^{ikh(j+1)}\right] - \xi^{n-1}e^{ikhj} + 2(1-\nu^2)\xi^n e^{ikhj}.$$

Dividing across by $\xi^{n-1}e^{ikhj}$ and cleaning up gives

$$\xi^2 + \alpha\xi + 1 = 0 \tag{5}$$

with

$$\alpha = -\nu^2\left[e^{-ikh} + e^{ikh} - 2\left(1 - \frac{1}{\nu^2}\right)\right] = -2\nu^2\left[\cos(kh) - \left(1 - \frac{1}{\nu^2}\right)\right]$$

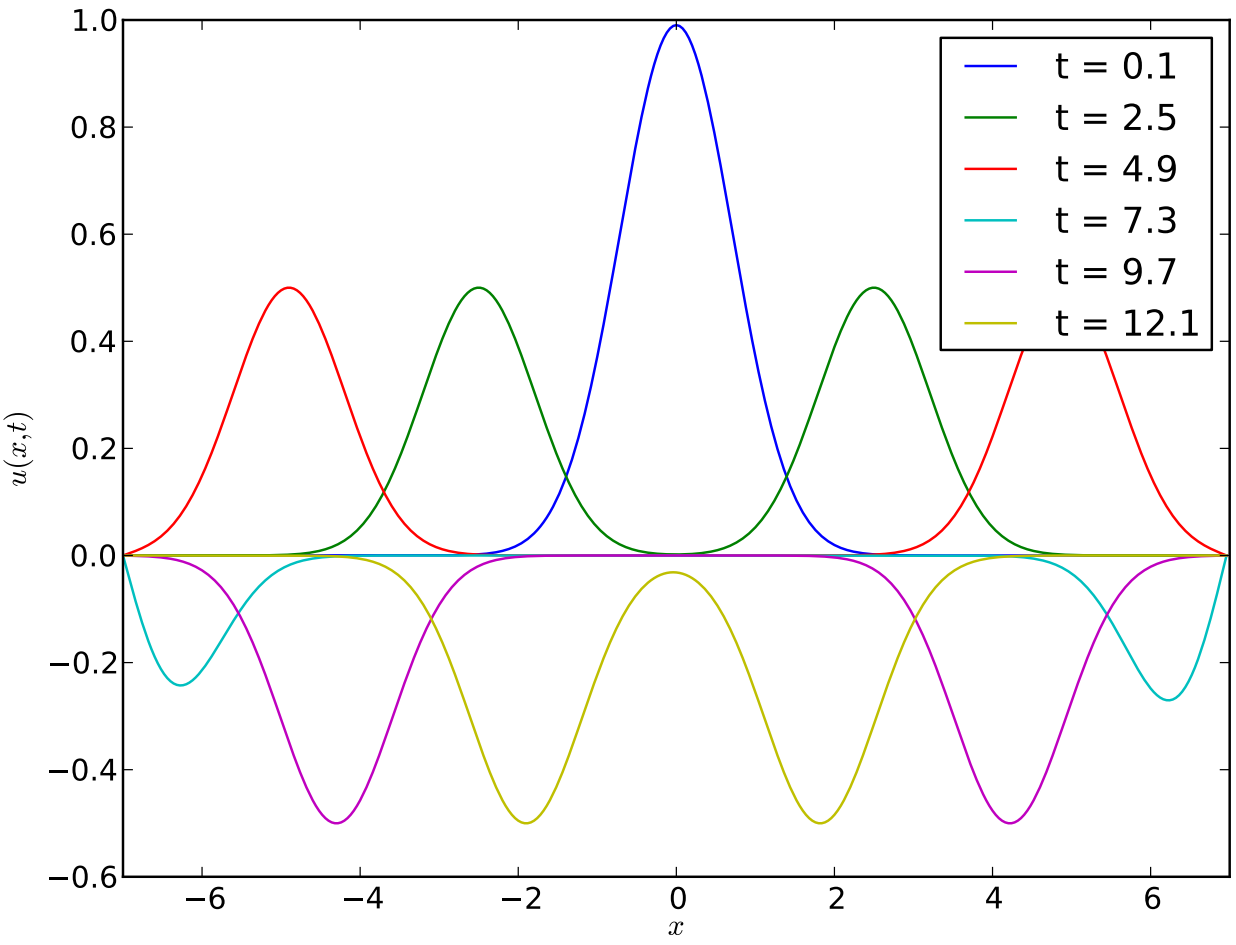

Figure 1: Solution to the wave equation with Gaussian initial condition.

$$= -2\nu^2 \left[1 - \sin^2\left(\frac{kh}{2}\right) - \left(1 - \frac{1}{\nu^2}\right)\right] = 2\nu^2 \left[\sin^2\left(\frac{kh}{2}\right) - \frac{1}{\nu^2}\right]$$

This is a quadratic equation with roots

$$\xi_\pm = \frac{1}{2}\left(-\alpha \pm \sqrt{\alpha^2 - 4}\right) = \frac{1}{2}\left(-\alpha \pm i\sqrt{(2-\alpha)(2+\alpha)}\right).$$

Inserting α into the above gives

$$\xi_\pm = -\frac{1}{2}\left[2\nu^2 \sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \frac{1}{2}\sqrt{\left(4\nu^2 \sin^2\left(\frac{kh}{2}\right) - 2\right)^2 - 4}$$

$$= -\left[2\nu^2 \sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \nu \sin\left(\frac{kh}{2}\right)\sqrt{4\nu^2 \sin^2\left(\frac{kh}{2}\right) - 2}$$

We consider $\tau > 0$ and $h > 0$, and by extension $\nu > 0$. From here, it is immediately apparent that if $\nu > 1$, both terms will be greater than 1 for some values of k, making the method unstable. For $\nu = 1$, ξ_- can for some values of k, be greater than one. A good choice seems to be $\nu = \frac{1}{\sqrt{2}}$, as then the above reduces to

$$\xi_\pm = -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \sin\left(\frac{kh}{2}\right)\sqrt{\sin^2\left(\frac{kh}{2}\right) - 1}$$

$$= -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm \sin\left(\frac{kh}{2}\right)\sqrt{-\cos^2\left(\frac{kh}{2}\right)}$$

$$= -\left[\sin^2\left(\frac{kh}{2}\right) - 1\right] \pm i\sin\left(\frac{kh}{2}\right)\cos\left(\frac{kh}{2}\right)$$

In the limits of $h \to 0$ or $k \to 0$, the sin terms go to 0, and we are left with $\xi_\pm = 1$. Elsewhere, the function moves between 0 and 1 as a function of k. Otherwise, when considering $|\xi_\pm|$, it is calculated by

$$\xi_\pm = \sqrt{\mathrm{Re}\left(\xi_\pm\right)^2 + \mathrm{Im}\left(\xi_\pm\right)^2}.$$

This demonstrates that ξ_+ and ξ_- are equivalent for $\nu = \frac{1}{\sqrt{2}}$. It is worth noting that in the above workings on the expression for $\xi_\pm$, many $\pm$s were absorbed by the initial $\pm$ due to working with the square root. This leads me to believe that both $\xi_\pm$ are not physically significant. In the quadratic form, we consider what happens when we require $\xi \in [-1, 1]$. It is clear that α as defined earlier must be bound to the region $[-2, 2]$, which sets a clear requirement that $\nu \in (0, 1]$[1] is a limit beyond which stability will not be achieved. This constrains $\tau \leq h$ for stability. However, I have chosen to take $\nu = \frac{1}{\sqrt{2}}$ for the computations in this report, a situation where $\xi \in [0, 1]$ is guaranteed , as this avoids the possibility of one of $\xi_\pm$ blowing the solution up, should I be wrong.

(3)

The program used is included in appendix B.

Question 2

(1)

We are now asked to solve the wave equation with $c = 1$ and external potential $V(x) = \frac{1}{\cosh^2(x)}$. I chose to use central differences – equation 2 – once again as it is simple to program and showed no significant shortcomings in the previous section which cannot be fixed by computational power. By the same method as for question 1.1, we can use central differences for this equation. This gives

$$u_j^{n+1} = \nu^2\left[u_{j-1}^n + u_{j+1}^n\right] - u_j^{n-1} + 2(1-\nu^2)u_j^n - \tau^2 V_j u_j^n \tag{6}$$

where V_j is the potential corresponding to the x-value j. The effect of V can be seen to scale with the time step τ, which is intuitive.

The program was written in Python 3 again, and is in appendix C. A plot of the waves behaviour is shown in figure 2. The wave interacts with the barrier, and some is transmitted, while more is reflected.

[1] ν could be negative, but we have previously constrained it to be positive.

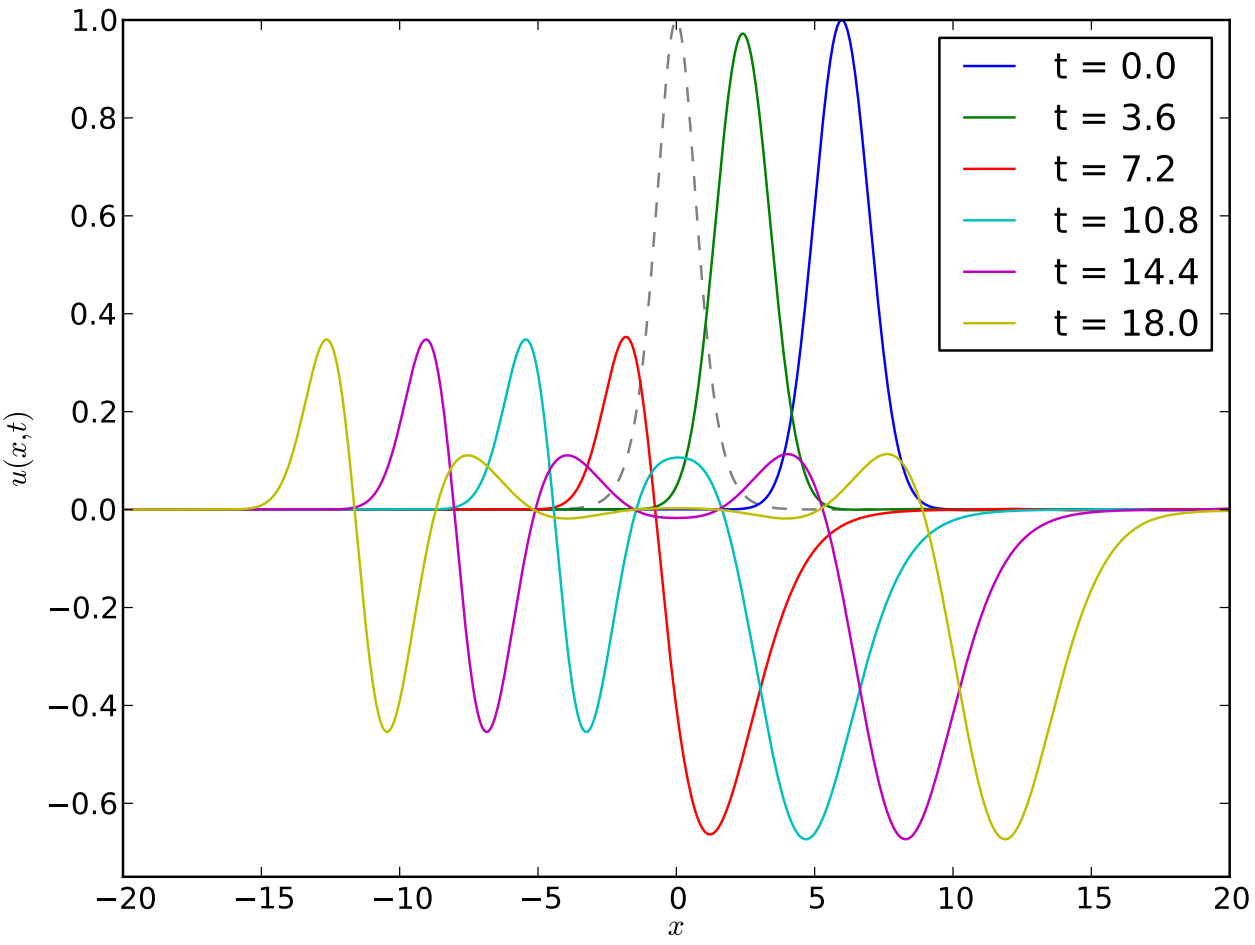

Figure 2: The wave at various times, with the potential plotted in the dashed red line.

(2)

(3)

(4)

A Program for 1.1

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Dirichlet boundaries are used, and imposed implicitly, by allocating an array
# of zeros and not writing to the edge points.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

# Define our space and time step sizes.
dx = 0.004
dt = dx

# Define our Temporal and Spatial lengths.
Time = [0,14]
Dist = [-7,7]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range(), but should be integers
# anyway, since I'll only be setting dx as a nice value, I swear...
nt = int((Time[1] - Time[0])/dt)
nx = int((Dist[1] - Dist[0])/dx)

# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)
# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx

# Initialise the t_0 part of the array.
for j in range(1,nx-1):
    U[0][j] = e**(-x[j]*x[j])

# c is not c from the wave equation:
c = (dt/dx)**2

# Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
for j in range(1,nx-1):
    U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]
    if ((n-2)%600) is 0:
        plot(x,U[n],label="t = "+str(round(14*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-7,7])
legend()
savefig("dirichlet.pdf",format="pdf")
show()
```

B Program for 1.3

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 using central differences.
# Dirichlet boundaries are used, and imposed implicitly, by allocating an array
# of zeros and not writing to the edge points.
# A variety of h values are used.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

def run(dt,dx,T,D):
    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range(), but should be integers
    # anyway, since I'll only be setting dx as a nice value, I swear...
    nt = int((T[1] - T[0])/dt)
    nx = int((D[1] - D[0])/dx)

    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)
    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx

    # Initialise the t_0 part of the array.
    for j in range(1,nx-1):
        U[0][j] = e**(-x[j]*x[j])

    # c is not c from the wave equation, just a useful constant.
    c = (dt/dx)**2

    # Since U_j^n-1 is unknown, I take is equivalent to U_j^n for the first run.
    for j in range(1,nx-1):
        U[1][j] = c*(U[0][j-1] - 2*U[0][j] + U[0][j+1]) + U[0][j]

    # Now perform our norm computation.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j]

    Ltime = nt*10.5/(T[1] - T[0])
    norm = 0
    for k in U[Ltime]:
        norm += k**2
    norm = sqrt(dx*norm)
    return norm


# Define our Temporal and Spatial lengths.
Time = [0,14]
Dist = [-7,7]

# Define our space and time step sizes.
dxs = [(0.02 + o/250) for o in range(0,200)]
dts = [0.01 for o in dxs] #(o**2)/4 for o in dxs] # The limit for stability.
err = []

for i in range(0,len(dxs)):
    Lfull = run(dts[i],dxs[i],Time,Dist)
    Lhalf = run(dts[i]/4,dxs[i]/2,Time,Dist)
    err.append(1 - (Lhalf/Lfull))
    print("h = "+str(round(dxs[i],4))+", err = "+str(err[-1]))

xlabel("h")
ylabel("$\epsilon(h)$")
semilogy(dxs,err,'.')
```

```
#plot(dxs,err,'.')
savefig("Error.pdf",format='PDF')
show()
```

C Program for 2.1

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

# Define our space and time step sizes.
dx = 0.02
dt = 0.01

# Define our Temporal and Spatial lengths.
Time = [0,21]
Dist = [-20,20]

# Define variables with the number of space/time steps for ease later.
# They must be cast as ints for use with range(), but should be integers
# anyway, since I'll only be setting dx as a nice value, I swear...
nt = int((Time[1] - Time[0])/dt)
nx = int((Dist[1] - Dist[0])/dx)

# Define our initial arrays.
U = zeros((nt, nx), dtype=float64)
V = zeros(nx, dtype=float64)
t = zeros(nt, dtype=float64)
x = zeros(nx, dtype=float64)

# Initialise the tand x arrays.
for n in range(0,nt):
    t[n] = Time[0] + n*dt
for j in range(0,nx):
    x[j] = Dist[0] + j*dx

# Initialise the t_0 part of the array.
# Gaussian with sigma=1 and mu=6.
for j in range(1,nx-1):
    U[0][j] = e**(-((x[j]-6)**2)/2)

# Initialise our potential array.
for j in range(0,nx):
    V[j] = 1/(cosh(x[j])**2)

# Plot the potential with a broken line for visual purposes.
plot(x,V,'--',color='grey')

# c is not c from the wave equation, but nu^2.
c = (dt/dx)**2

# I manually impose the initial derivative condition, by computing the second
# time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
for j in range(1,nx-1):
    U[1][j] = U[0][j] + dt*(-(x[j]-6)*(e**(-((x[j]-6)**2)/2)))

# Now perform our integration.
for n in range(2,nt):
    for j in range(1,nx-1):
        U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j] - (dt**2)*V[j]*U[
            n-1][j]
        # Impose the Neumann condition here:
        U[n][0] = U[n][1]
        U[n][nx-1] = U[n][nx-2]
    if ((n-2)%360) is 0:
        plot(x,U[n],label="t = "+str(round(Time[1]*(n/nt),1)))
xlabel("$x$")
ylabel("$u(x,t)$")
xlim([-20,20])
ylim([-0.75,1])
legend()
```

```
savefig("cosh.pdf",format="pdf")
show()
```

D Program for 2.2

```
#!/usr/bin/env python3
# Program to solve the wave equation with c=1 and external potential using
# central differences. Neumann boundaries are used.
# Liam O'Sullivan

from matplotlib.pyplot import *
from numpy import *

def run(dt,dx,Time,Dist):
    # Define variables with the number of space/time steps for ease later.
    # They must be cast as ints for use with range(), but should be integers
    # anyway, since I'll only be setting dx as a nice value, I swear...
    nt = int((Time[1] - Time[0])/dt)
    nx = int((Dist[1] - Dist[0])/dx)

    # Define our initial arrays.
    U = zeros((nt, nx), dtype=float64)
    V = zeros(nx, dtype=float64)
    t = zeros(nt, dtype=float64)
    x = zeros(nx, dtype=float64)

    # Initialise the tand x arrays.
    for n in range(0,nt):
        t[n] = Time[0] + n*dt
    for j in range(0,nx):
        x[j] = Dist[0] + j*dx

    # Initialise the t_0 part of the array.
    # Gaussian with sigma=1 and mu=6.
    for j in range(1,nx-1):
        U[0][j] = e**(-((x[j]-6)**2)/2)

    # Initialise our potential array.
    for j in range(0,nx):
        V[j] = 1/(cosh(x[j])**2)

    # c is not c from the wave equation, but nu^2.
    c = (dt/dx)**2

    # I manually impose the initial derivative condition, by computing the second
    # time step as d/dx of the analytic u_0, using the Euler method(Forgive me).
    for j in range(1,nx-1):
        U[1][j] = U[0][j] + dt*(-(x[j]-6)*(e**(-((x[j]-6)**2)/2)))
        U[1][0] = U[1][1]
        U[1][nx-1] = U[1][nx-2]

    # Now perform our integration.
    for n in range(2,nt):
        for j in range(1,nx-1):
            U[n][j] = c*(U[n-1][j-1] - 2*U[n-1][j] + U[n-1][j+1]) - U[n-2][j] + 2*U[n-1][j] - (dt**2)*V[j
                ]*U[n-1][j]
            # Impose the Neumann condition here:
            U[n][0] = U[n][1]
            U[n][nx-1] = U[n][nx-2]
        if (n%20 == 0):
            plot(x,U[n])
            show()


    # The time t_{max}:
    Ltime = 18.*nt/(Time[1] - Time[0])

    norm = 0
    for b in U[Ltime]:
        norm += b**2
    norm = sqrt(dx*norm)
    return norm

# Define our space and time step sizes.
```

```
dxs = [0.2] #linspace(0.02,0.22,10)
dts = [0.1] #linspace(0.02,0.22,10)

# Define our Temporal and Spatial lengths.
aTime = [0,19]
aDist = [-20,20]
hs = []

for i in range(0,len(dxs)):
    n1 = run(dts[i],dxs[i],aTime,aDist)
    n2 = run(dts[i],dxs[i]/2.,aTime,aDist)
    hs.append(1 - n2/n1)

plot(dxs,hs,'.')
xlabel("$h$")
ylabel("$\epsilon(h)$")
savefig("epsilon.pdf",format="pdf")
show()
```