Numerical Treatment of Differential Equations: Homework 7

Truman Ellis

March 25, 2011

We wish to solve the second order differential equation

$$\begin{aligned} -u(x)'' + cu(x) &= f(x)\,, \quad \text{in } (0,1) \\ u(0) = u_L\,, &\qquad u(1) = u_R\,, \end{aligned}$$

for $c \geq 0$. We will solve this with continuous first order basis functions. This boils down to a linear system of equations

$$\mathbf{A}\boldsymbol{\alpha} = \mathbf{b}\,,$$

where

$$\begin{aligned} A_{ij} &= (\phi_i', \phi_j') + (c\phi_i, \phi_j) \\ b_i &= (f, \phi_i) - u_L[(\phi_0', \phi_i') + (c\phi_0, \phi_i)] - u_R[(\phi_N', \phi_i') + (c\phi_N, \phi_i)]\,. \end{aligned}$$

In one dimension, we can calculate most of these integrals explicitly. For uniform spacing, we get $(\phi_i', \phi_i') = \frac{2}{h}$, $(\phi_i', \phi_{i\pm1}') = -\frac{1}{h}$, $(\phi_i, \phi_i) = \frac{2h}{3}$, $(\phi_i, \phi_{i\pm1}) = \frac{h}{6}$, and all other terms are zero. Using the trapezoid rule, we can approximate $(f, \phi_i) = hf(x_i)$.

Using the method of manufactured solutions, we can test the convergence of our method. If we wish to converge to an exact solution $u_{ex} = \sin(4\pi x^3) + 5x - 3$ with boundary conditions $u_L = -3$ and $u_R = 2$, we can set our forcing function to be $f(x) = -u_{ex}(x)'' + cu_{ex}(x)$.

We achieve quadratic convergence in both the quasi-optimal error norm and the L_2 norm as we see in the following figures. This is higher than expected for the quasi-optimal error norm, and there are several possible explanations. With this uniform mesh, I could be seeing a case of superconvergence. Alternatively the solution at the half node values could be second order accurate and using these exclusively to calculate the error integral could be artificially reducing my error calculation. The third option is that there is a bug in the code, but since everything else is working so well, I think one of the first two options is more likely.

We have no issue solving with zero or large c, in fact the convergence plots look nearly identical. However, as the final three figures demonstrate, the solution degrades for negative c.

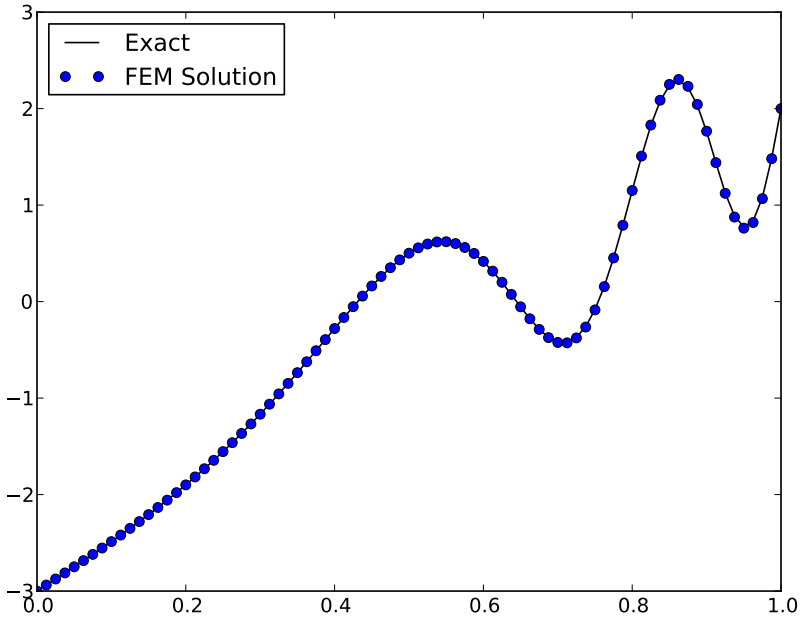

Figure 1: Finite element vs exact solution, for $c = 1$

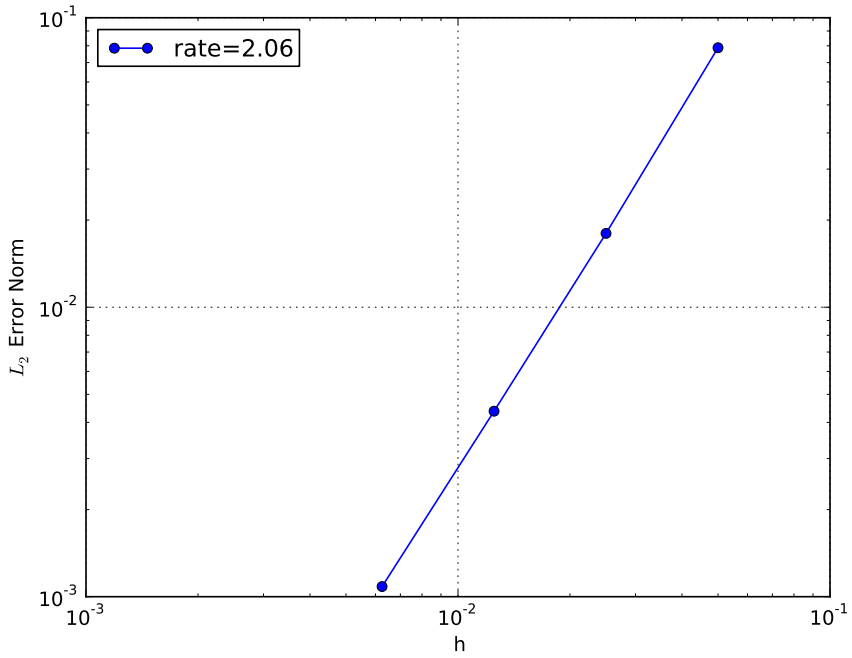

Figure 2: L_2 convergence for $c = 1$

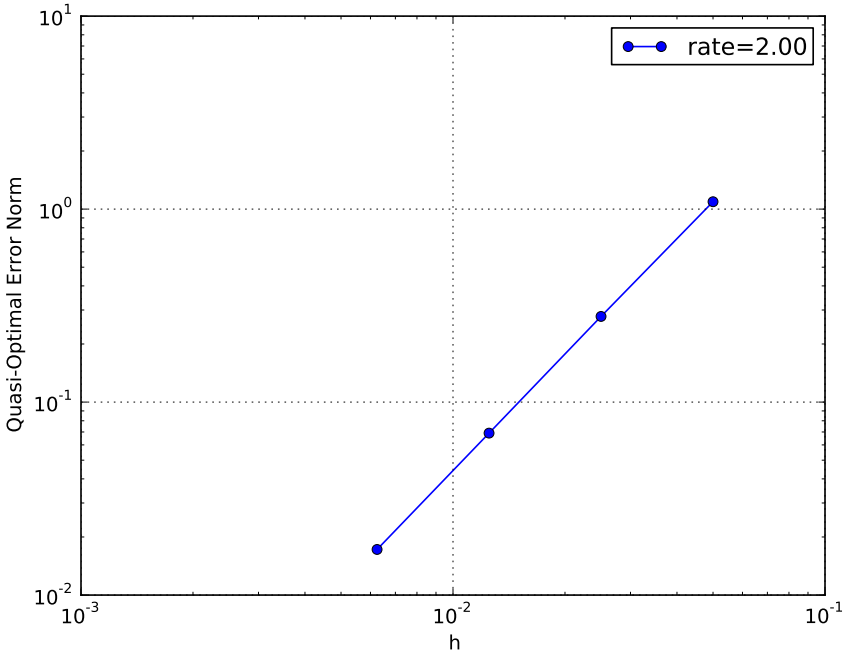

Figure 3: Quasi-optimal convergence for $c = 1$

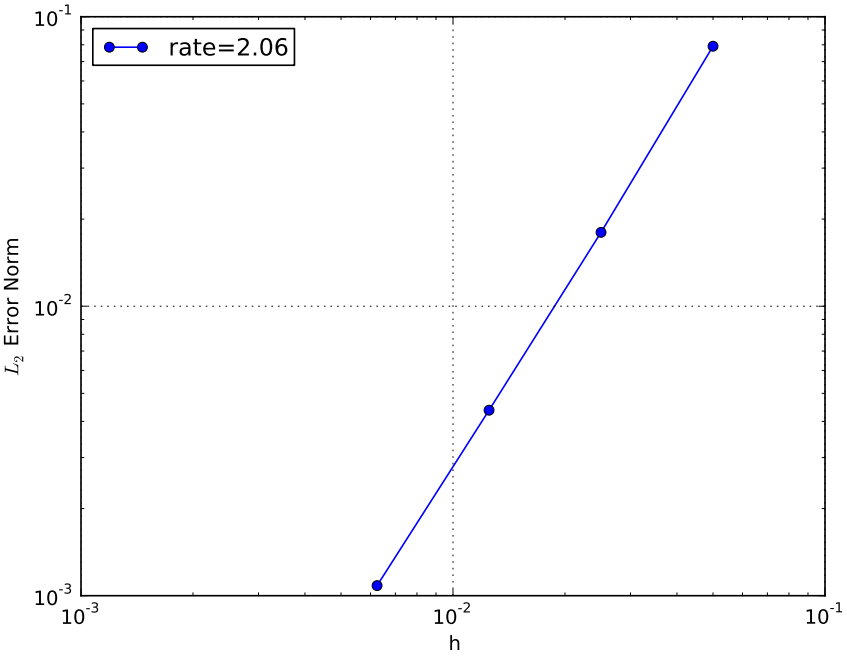

Figure 4: L_2 convergence for $c = 0$

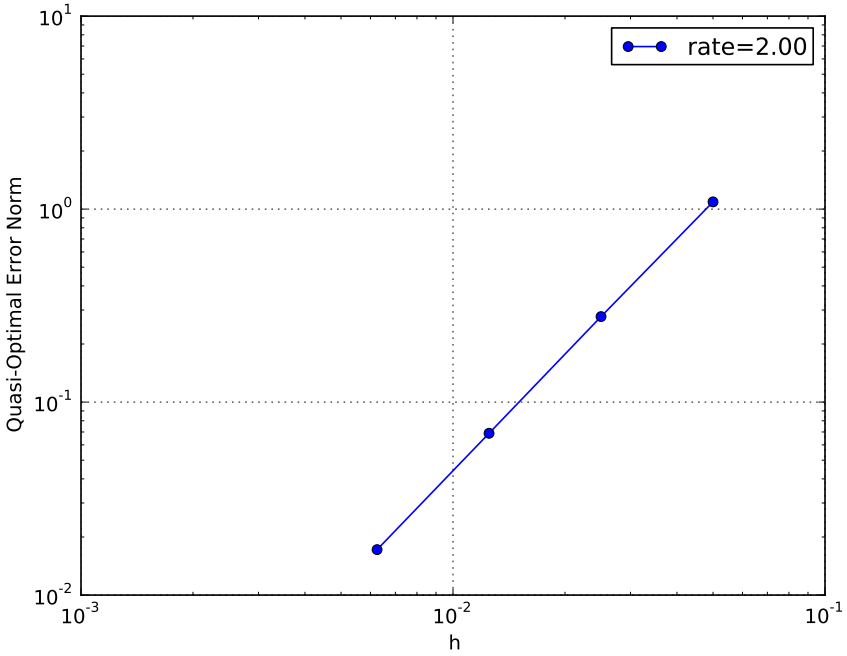

Figure 5: Quasi-optimal convergence for $c = 0$

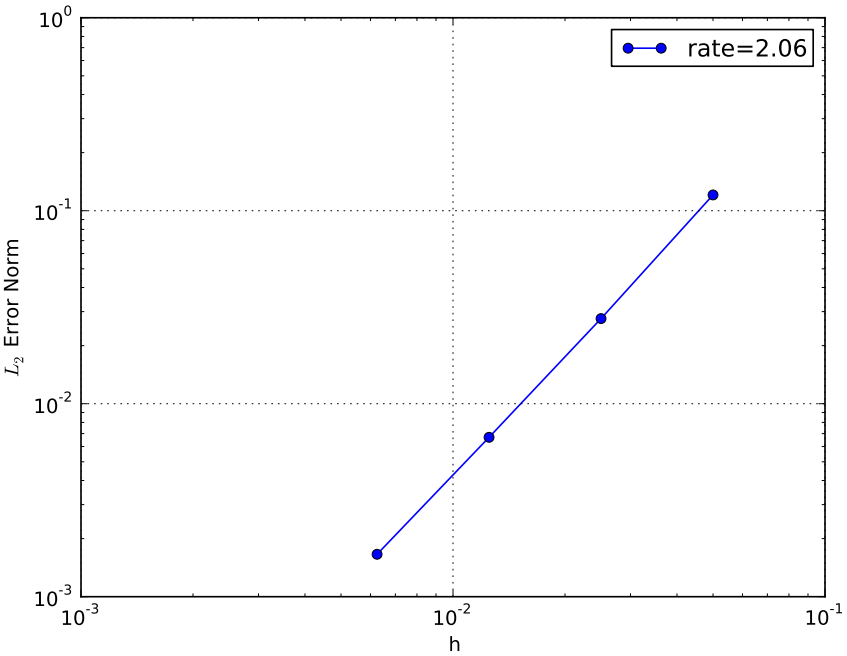

Figure 6: L_2 convergence for $c = 1000$

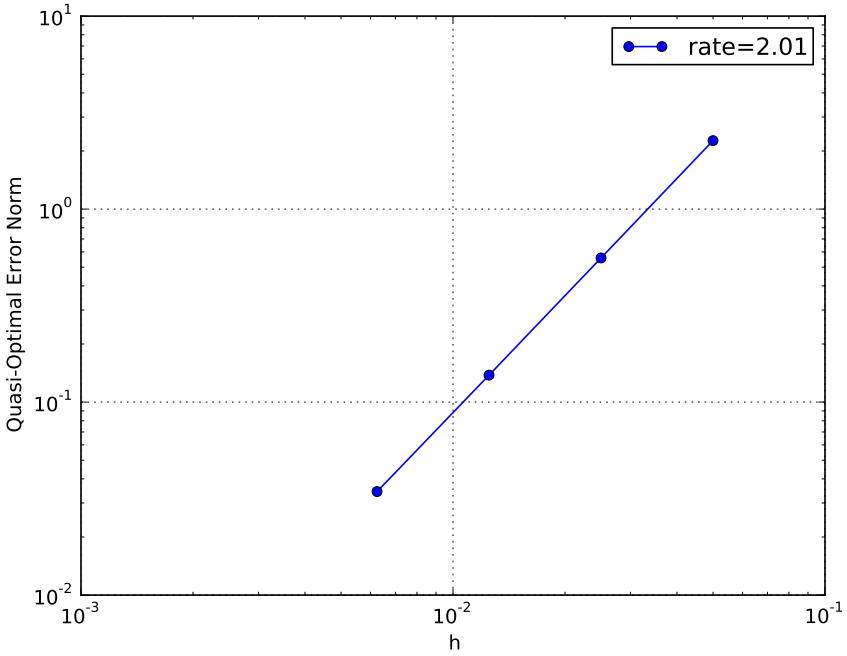

Figure 7: Quasi-optimal convergence for $c = 1000$

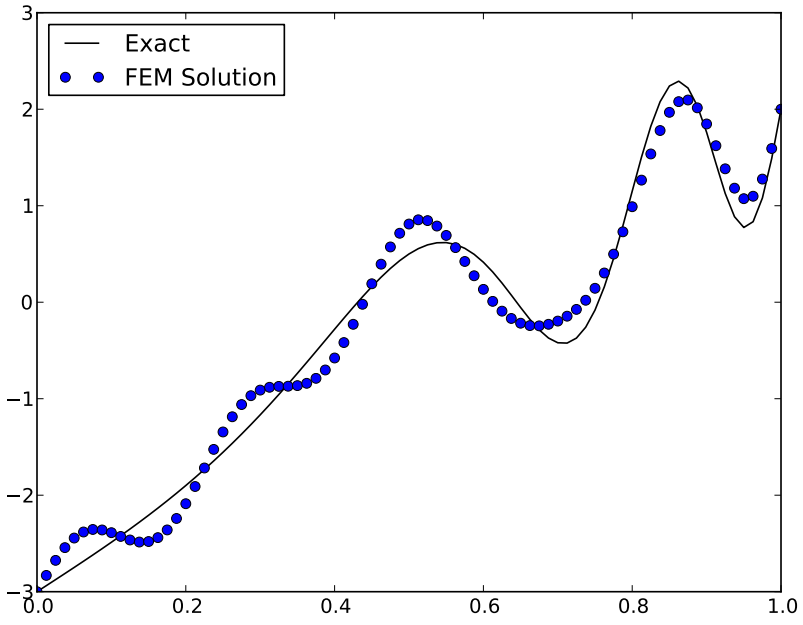

Figure 8: Finite element vs exact solution, for $c = -800$

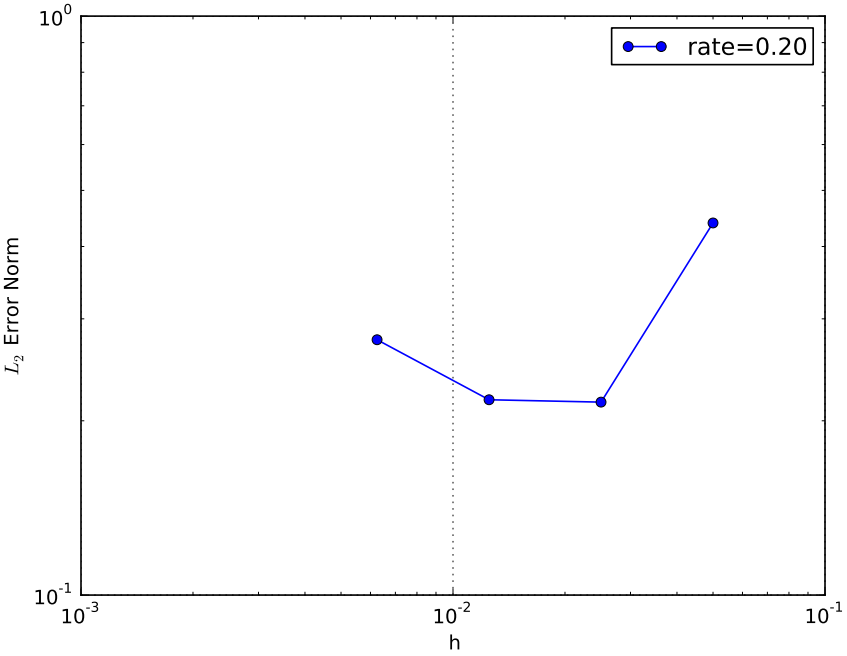

Figure 9: L_2 convergence for $c = -800$

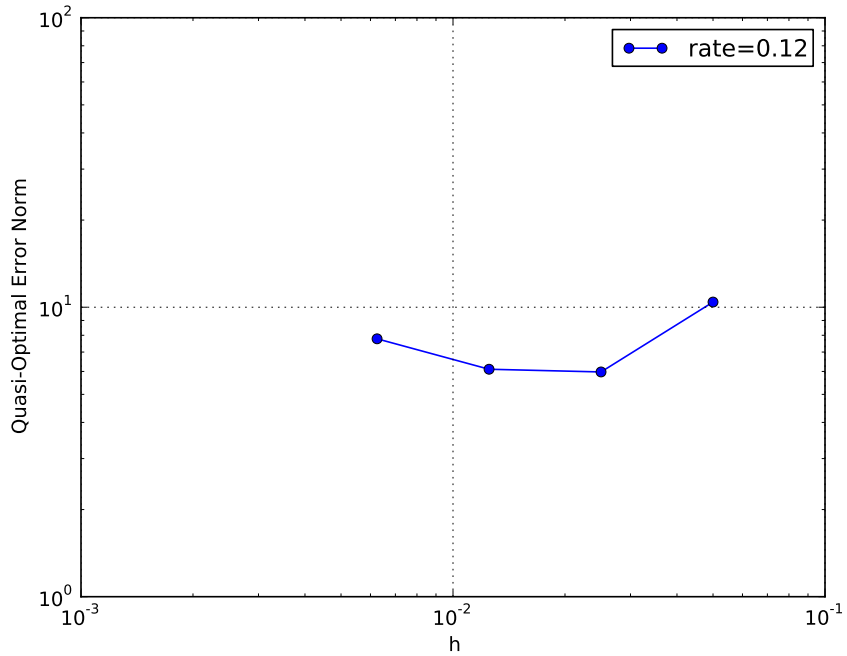

Figure 10: Quasi-optimal convergence for $c = -800$

FESolver.py

```
# One-dimensional Finite Element Solver
# Solves -u(x)'' + cu(x)  = f(x,y)
# Written by: Truman Ellis
# Numerical Treatment of Differential Equations
# Spring 2011

from pylab import *

close('all')

# Define Initial Refinement
nx = 19

# Define problem domain
xmin = 0.
xmax = 1.

# Define boundary conditions
uL = -3.
uR = 2.

# Define scalar c
c = 0.

# Define exact solution for convergence tests
def exact(x):
    return sin(4*pi*x**3) + 5*x - 3

def dexact(x):
    return 4*pi*cos(4*pi*x**3)*3*x**2 + 5

def d2exact(x):
    return -16*pi**2*sin(4*pi*x**3)*9*x**4 + 4*pi*cos(4*pi*x**3)*6*x

# Define forcing function to allow for convergence tests
def f(x):
    return -d2exact(x) + c*exact(x)

# Set number of refinement steps for convergence test
nref = 4
error = zeros(nref)
derror = zeros(nref)
ndofs = zeros(nref)
hs = zeros(nref)
for r in range(0,nref):
    if r > 0:
        nx = 2*(nx+1)-1
    ndofs[r] = nx
    h = (xmax - xmin)/(nx + 1)
    hs[r] = h

    X = linspace(xmin, xmax, nx+2)
    x = X[1:-1]
    xh = (X[1:] + X[0:-1])/2.
    A = zeros((nx,nx))
    b = zeros(nx)
    u = zeros(nx)
    du = zeros(nx+1)
    u_ex = zeros(nx)
    du_ex = zeros(nx+1)

    for i in range(0,nx):
            A[i,i] = 2./h + c*2.*h/3.
            b[i] += h*f(x[i])
            if (i < nx-1):
                A[i,i+1] = -1./h + c*h/6.
            else:
                b[i] += -uR*(-1./h + c*h/6.)
            if (i > 0):
                A[i,i-1] = -1./h + c*h/6.
            else:
                b[i] += -uL*(-1./h + c*h/6.)

            # Calculate exact solution
            u_ex[i] = exact(x[i])

    # Solve for finite difference solution
    u = linalg.solve(A,b)
    du[1:-1] = (u[1:] - u[0:-1])/h
    du[0] = (u[0]-uL)/h
    du[-1] = (uR - u[-1])/h
    du_ex = dexact(xh)
    error[r] = sqrt(((u-u_ex)**2).sum()/nx)
    derror[r] = sqrt(((du-du_ex)**2).sum()/(nx+1))
```

```
    # Plotting
    if r == 2:
        U = zeros(nx+2)
        E = zeros(nx+2)
        U[1:-1] = u
        U[0] = uL
        U[-1] = uR
        E[1:-1] = u_ex
        E[0] = uL
        E[-1] = uR

        figure(1)
        plot(X, E, 'k')
        plot(X, U, 'o')
        legend(('Exact','FEM Solution'),loc='best')
        show()

        figure(2)
        plot(xh, du_ex, 'k')
        plot(xh, du, 'o')


figure()
loglog(hs,error,'-o')
(m,b) = polyfit(log(hs),log(error),1)
xlabel('h')
ylabel('$L_2$ Error Norm')
grid()
legend(('rate=%.2f' % (m,),), loc='best')

figure()
loglog(hs,derror,'-o')
(dm,b) = polyfit(log(hs),log(derror),1)
xlabel('h')
ylabel('Quasi-Optimal Error Norm')
grid()
legend(('rate=%.2f' % (dm,),), loc='best')
```