Ay 190 Assignment 8

Advection Equation

February 21, 2013

The advection equation

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0$$

was numerically solved for the Gaussian $\Phi(x, t = 0) = \exp\left(-(x - x_0)^2/(2\sigma^2)\right)$ with $x_0 = 30$ and $\sigma = \sqrt{15}$.

The implementations of the various first- and second-order solvers used to integrate the advection equation for this Gaussian in this assignment is shown below.

```
# advect.py
# created 2/18/13 by stacy kim
#
# Contains a suite of solvers for the advection equation. The solution is not
# computed at the boundary points.


def upwind(y):
    """Implements the upwind method (O[h] in space and time)."""
    return [y[i]-v*dt/dx*(y[i]-y[i-1]) for i in range(1,len(y)-1)]

def downwind(y):
    """Implements the downwind method (O[h] in space and time)."""
    return [y[i]-v*dt/dx*(y[i+1]-y[i]) for i in range(1,len(y)-1)]

def ftcs(y):
    """Implements the FTCS method (O[h^2] in space, O[h] in time)."""
    return [y[i]-v*dt/(2*dx)*(y[i+1]-y[i-1]) for i in range(1,len(y)-1)]

def lax_friedrich(y):
    """Implements the Lax-Friedrich method (O[h^2] in space, O[h] in time)."""
    return [0.5*(y[i+1]+y[i-1]) - v*dt/(2*dx)*(y[i+1]-y[i-1])
            for i in range(1,len(y)-1)]

def leapfrog(y,yold2):
    """Implements the leapfrog method (O[h^2] in space and time)."""
    return [yold2[i] - v*dt/dx*(y[i+1]-y[i-1]) for i in range(1,len(y)-1)]

def lax_wendroff(y):
    """Implements the Lax-Wendroff method (O[h^2] in space and time)."""
    return [y[i]-v*dt/(2*dx)*(y[i+1]-y[i-1])+0.5*(v*dt/dx)**2*(y[i-1]-2*y[i]+y[i+1])
            for i in range(1,len(y)-1)]
```

1 The Upwind Method

1.1 The original Gaussian

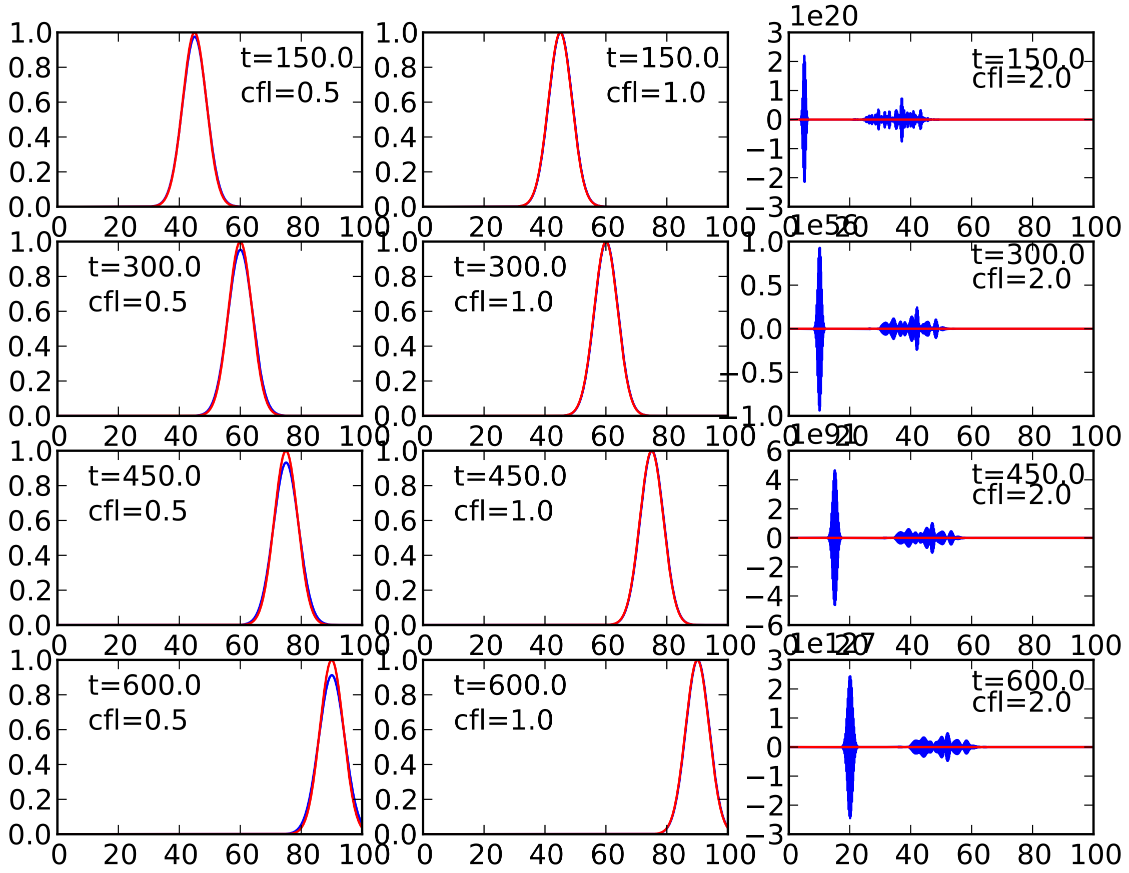

Figure 1: The solution produced by the upwind method at times $t = 0, 150, 300,$ and 450 (rows top to bottom) for $\alpha = 0.5$, 1, and 2 (columns left to right). The analytic solution is plotted in red, and the numeric solution is plotted in blue. Note that the best performance occurs when $\alpha = 1$. When $\alpha = 0.5$, though the Gaussian translated at the appropriate speed, it decayed in height and grew in width. When $\alpha = 2$, the Gaussian is translated to the right too quickly and eventually develops large errors throughout the domain.

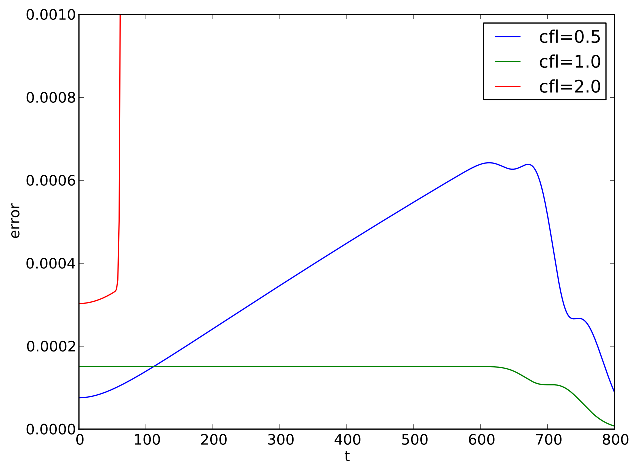

Figure 2: Error over time in the numerical solution, calculated as the root mean square of the deviation of the numerical from the analytic result. The Gaussian begins to leave the domain around $t = 750$, causing the error (which arises primarily near the Gaussian) to dip. When $\alpha = 2.0$, errors eventually exceed 10^{140}.

1.2 The gaussian with $\sigma/5$

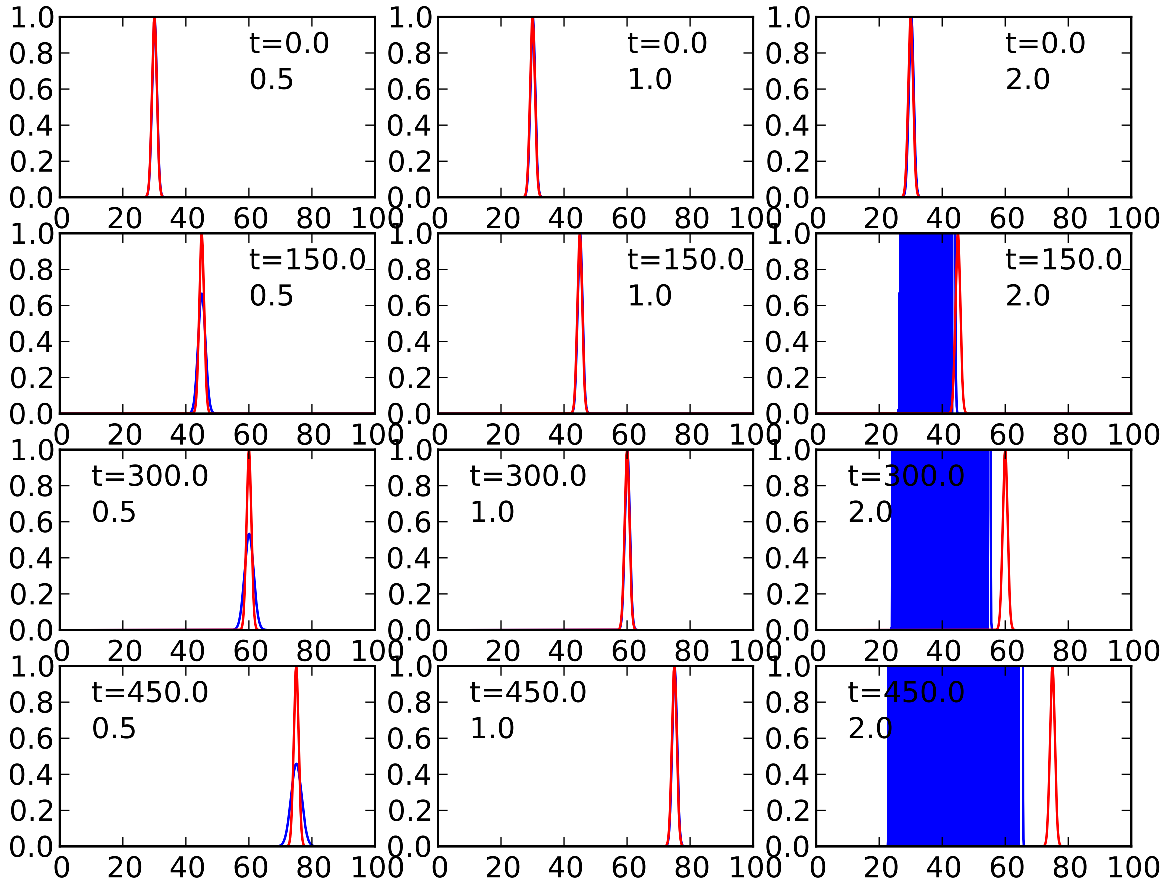

Figure 3: Similar to Figure 1, but for a Gaussian with width $\sigma/5$. For $\alpha = 0.5$, the height and width of the gaussian changes more dramatically over time compared to the wider Gaussian. For $\alpha = 1$, the solution again well matches the analytic result, and for $\alpha = 2$, large errors again result.

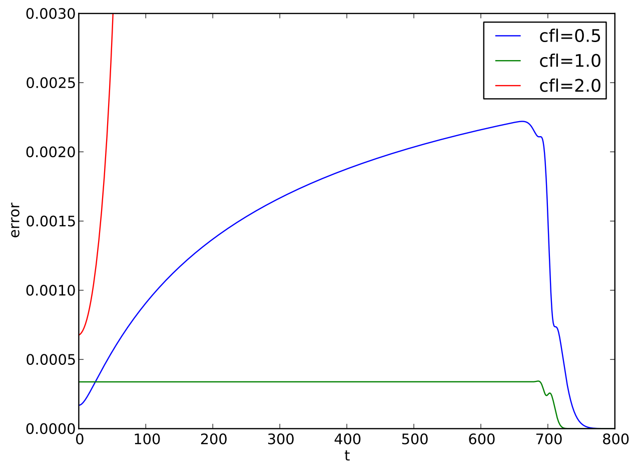

Figure 4: Error over time in the numerical solution.

2 The Downwind Method

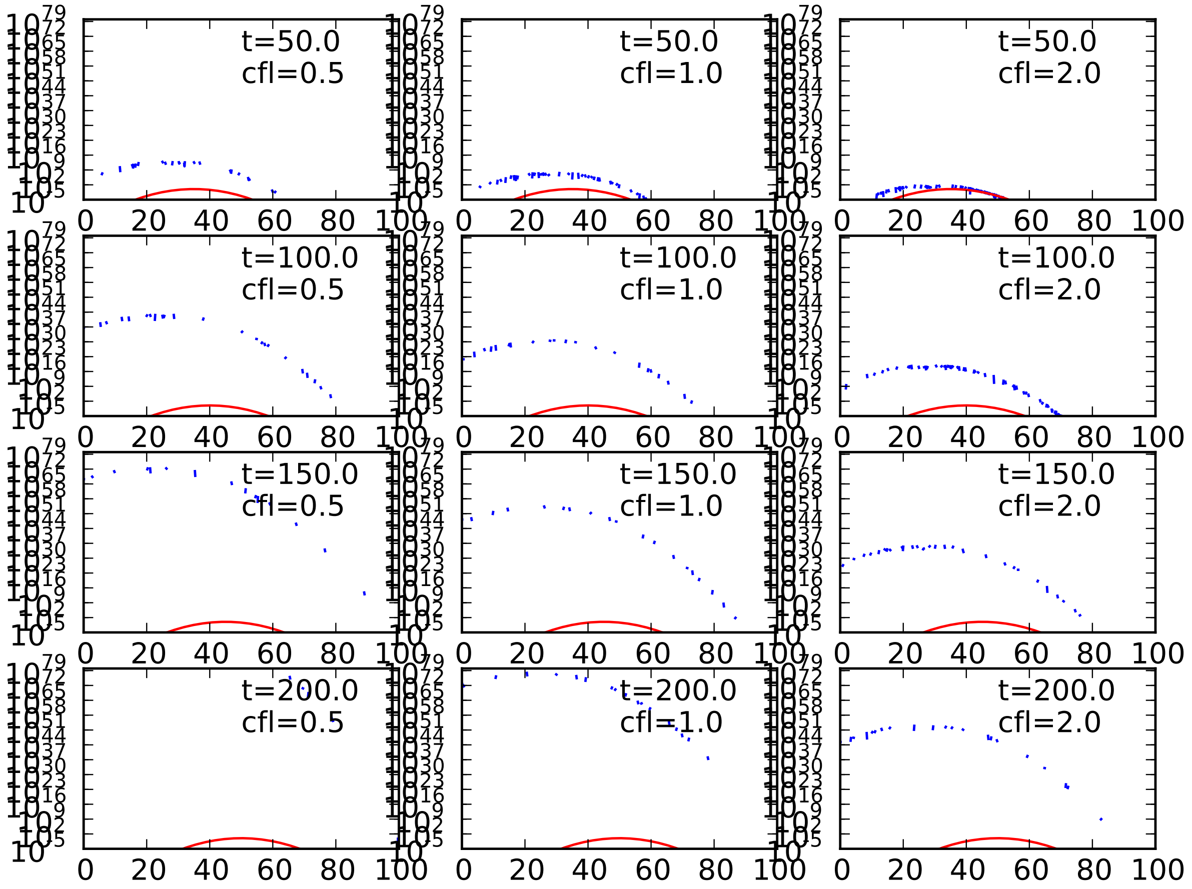

Figure 5: The numerical solution (blue), with the analytical solution overplotted in red. Note the logarithmic scaling of the y axis. The numerical solution oscillates with position, and (erroneously) increases by tens to a couple hundred orders of magnitude in amplitude over time. The growth in the amplitude of the numerical solution is suppressed until later times and is greatly reduced with larger α.

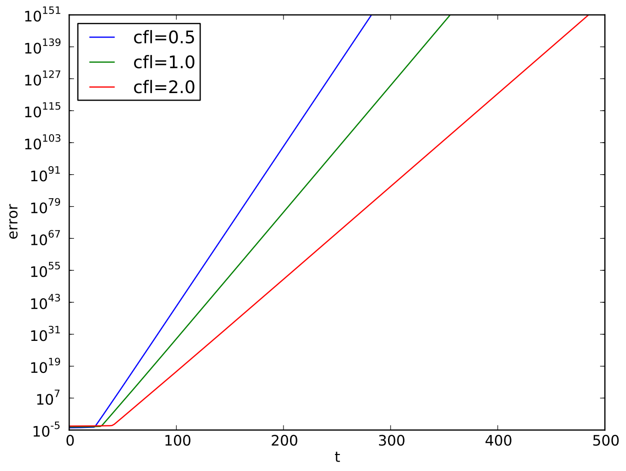

Figure 6: Error over time in the numerical solution calculated via the downwind scheme. The error in the solution increases dramatically as early as $t = 20 - 40$, with later "blow-up" times for larger α (though by number of integration steps, larger α blows up more quickly).

3 The FTCS Method

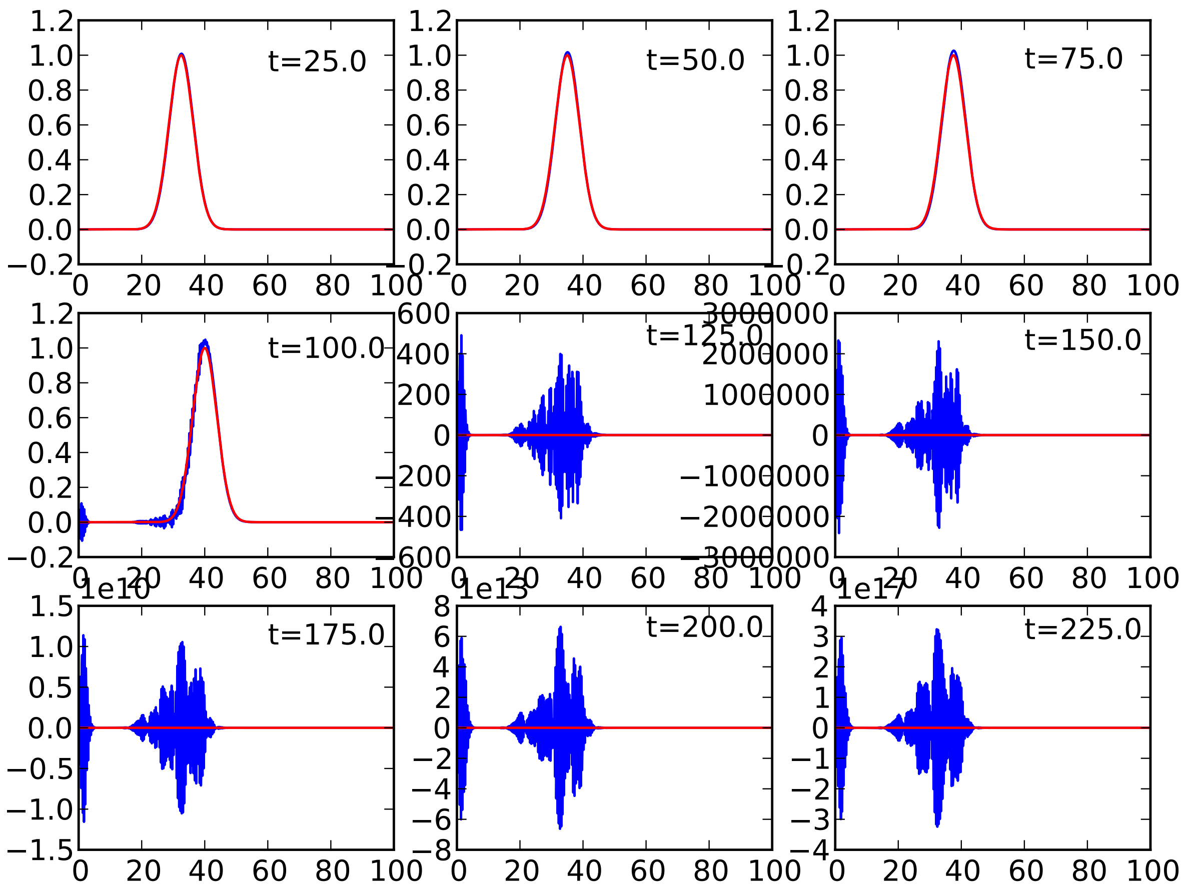

Figure 7: The numerical solution (blue) calculated using the FTCS scheme with $\alpha = 1$ ($t = 1$), with the analytical solution overplotted in red. Before about $t = 90$, the height of the numerically advected Gaussian increases slowly, then rapidly develops spatial oscillatory behavior primarily at the beginning of the domain and the back half of the Gaussian. Similar behavior is observed for different values of α, though the deviation occurs earlier for larger values of α.

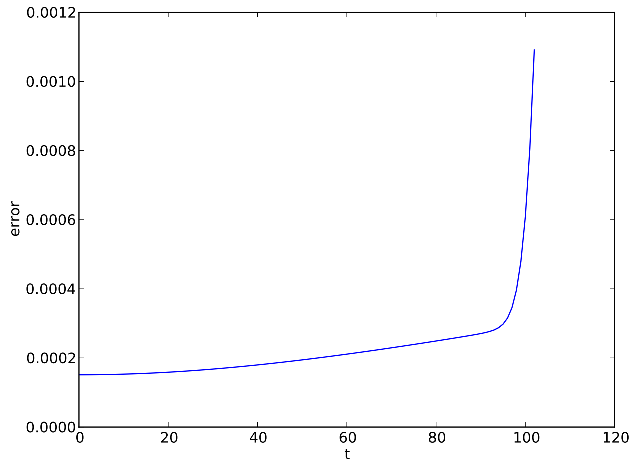

Figure 8: Error over time in the numerical solution calculated via the FTCS scheme for $\alpha = 1$ ($t = 1$). The error increases exponentially, but with an e-folding time of 61 before $t =$ 90 and 3 after.

4 The Lax-Friedrich Method

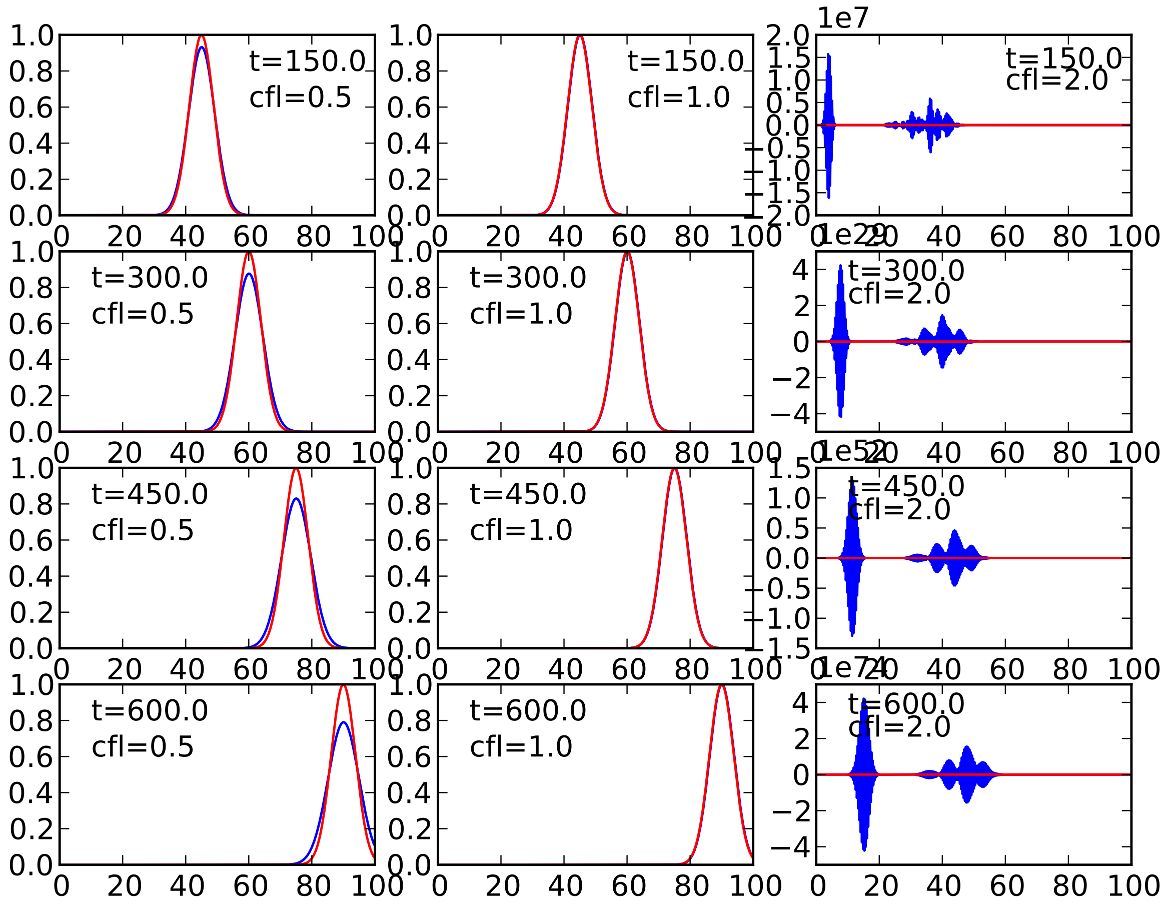

Figure 9: The numerical solution (blue) calculated with the Lax-Friedrich method with $\alpha = 1$ ($t = 1$), with the analytical solution overplotted in red. The Lax-Freidrich method performs similarly to the upwind method.

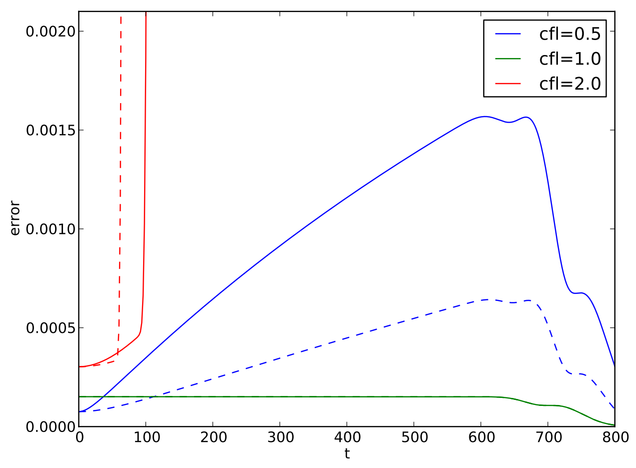

Figure 10: Error over time in the numerical solution calculated via the Lax-Friedrich scheme (solid lines) compared with the error in the upwind scheme (dotted) for various α. The Lax-Friedrich scheme is more stable for $\alpha > 1$ (red) and less stable for $\alpha < 1$ (blue).

5 The Leapfrog and Lax-Wendroff methods

The leapfrog and Lax-Wendroff methods, both second-order in both time and space, produced results that agreed well with the analytical solution for $\alpha \leq 1$. However, under the leapfrog method, the solution developed a ripple in the following edge of the Gaussian beginning at around $t = 30$ that flowed away from the Gaussian and eventually exited the domain. The parity of the feature oscillated over time when $\alpha = 1$.

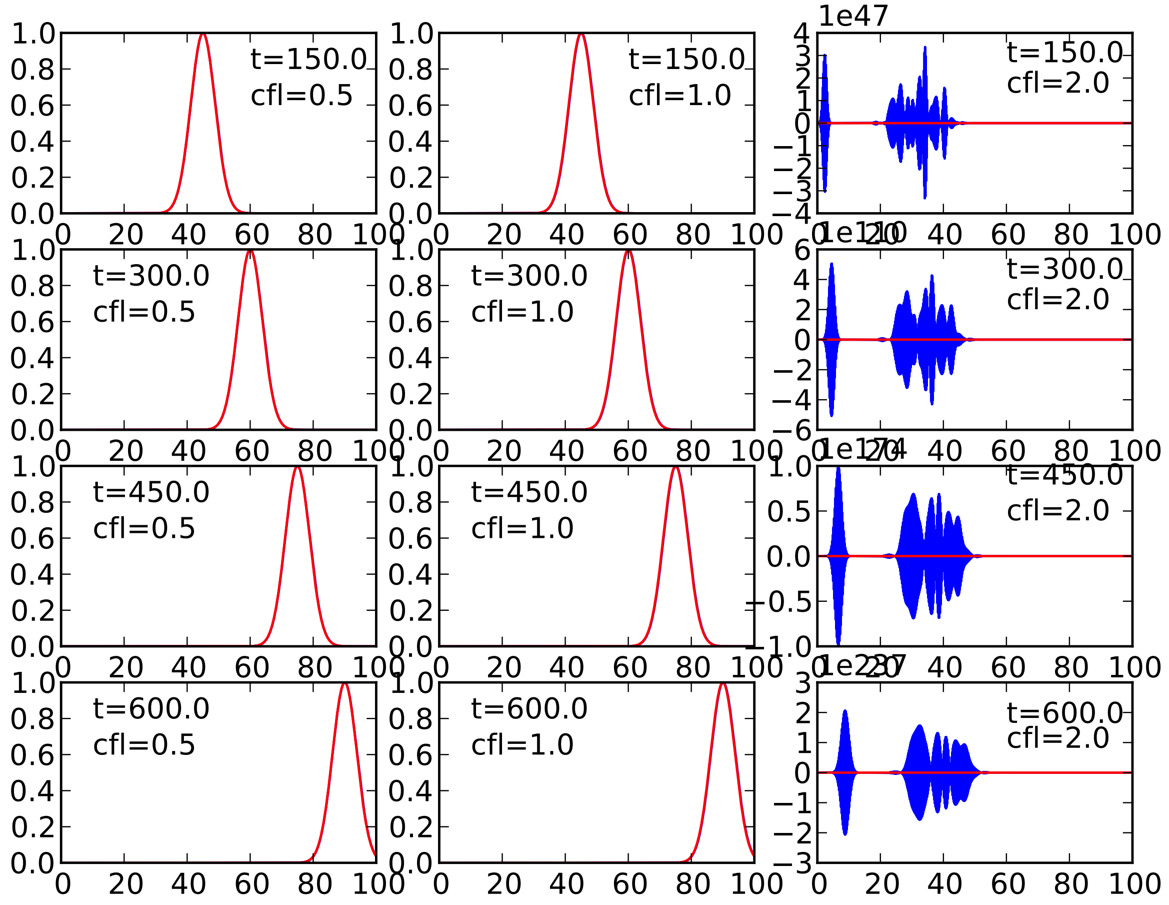

Figure 11: The numerical solution (blue) calculated using the fully second-order Lax-Wendroff method for various α. The analytical solution overplotted in red. For $\alpha \leq 1$, the numerical solution closely matches the analytic solution, and in particular, the shape of the Gaussian is preserved for $\alpha = 0.5$, unlike previous methods that were first-order in time.

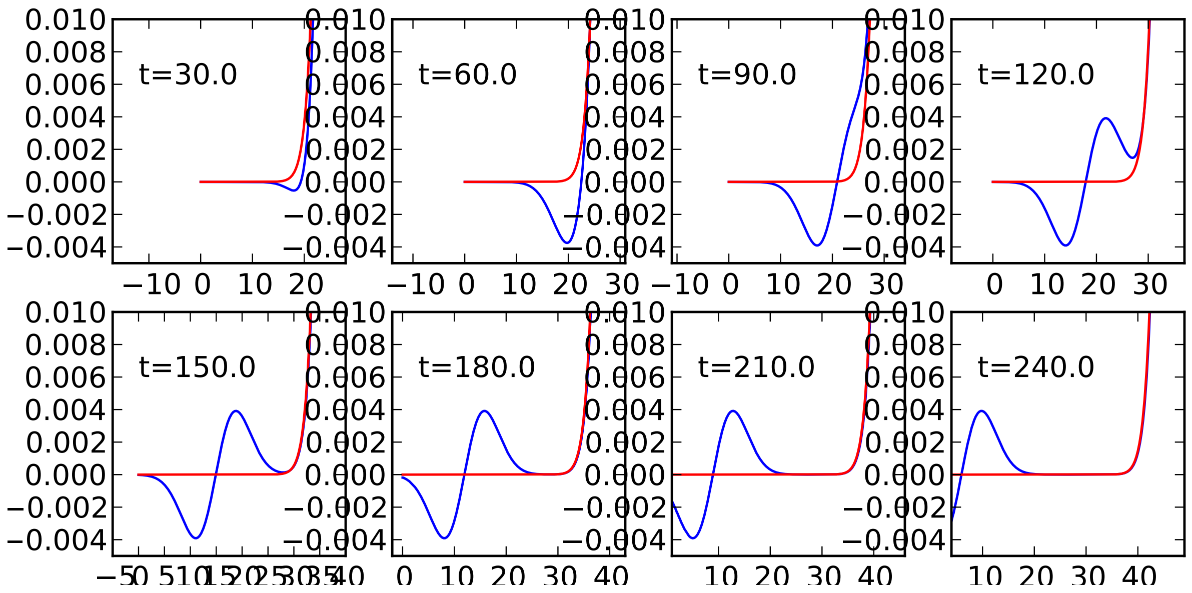

Figure 12: A close up of the small ripple (with a height of $\tilde{0}.004$) that develops under the leapfrog method in the following edge of the Gaussian for $\alpha = 0.5$ (similar behavior occurs for $\alpha = 1$). As usual, the numerical solution is in blue and the analytical solution is overplotted in red.

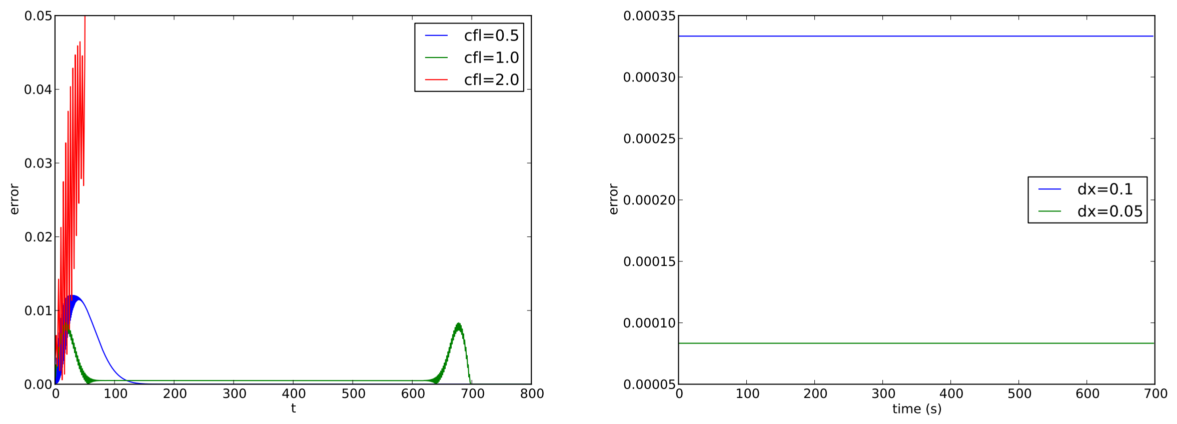

Figure 13: Errors under leapfrog method (left) when varying α and under the Lax-Wendroff method (right) when varying the spatial and temporal resolution for $\alpha = 1$. The error was defined to be the deviation from the analytic solution at the single point at the peak of the Gaussian. For the Lax-Wendroff method, a halving in the resolution quadrupules the accuracy, demonstrating second-order convergence.

Appendices

The Python modules and scripts used in this assignment include

The Advection Routines: `advect.py` (see page 1)
A The Upwind Method: `advect_gauss_upwind.py` (see pages 9-11)
The Downwind Method: `advect_gauss_downwind.py` (not shown)
The FTCS Method: `advect_gauss_ftcs.py` (not shown)
The Lax-Friedrich Method: `advect_gauss_lax_friedrich.py` (not shown)
The Leapfrog Method: `advect_gauss_leapfrog.py` (not shown)
The Lax-Wendroff Method: `advect_gauss_lax_wendroff.py` (not shown)

A The Upwind Method

```
import sys,math
from pylab import *


# THE UPWIND ADVECTION EQUATION SOLVER -------------------------------------------

def upwind(y):
    """
    Implements the downwind method to solve the advection equation.  The
    solution is not computed at the boundary points (call apply_bcs to do this).
    """
    return [y[i]-v*dt/dx*(y[i]-y[i-1]) for i in range(1,len(y)-1)]

def apply_bcs(x,y):
    """
    Applies 'outflow' boundary conditions. Assumes boundary points not
    included in y and adds them to y.
    """
    return [y[0]] + y + [y[-1]]

def analytic(x,t):
    """Analytic solution to advection equation for a Gaussian."""
    return exp(-(x-v*t-x0)**2/(2*sigma**2))


# MAIN ---------------------------------------------------------------------------

# parameters
dx = 0.1
v = 0.1
# set up grid
x = arange(0,100,dx)
n = len(x)
cfl = array([0.5,1.0,2.0])
t = 0.0
tf=800.0

# for initial data
```

```
sigma = sqrt(15.0)
x0 = 30.0

#set up initial conditions
y = analytic(x,0)
yold2 = y
yold = y

# evolve (and show evolution)
ion()  # "interaction on" - required for animation via draw()
figure()
ylim([0,1])
plot(x,y,'x-') # numerical data
plot(x,analytic(x,t),'r-') # analytic data
show()

err=[]
for idt in range(len(cfl)):
    #set up initial conditions
    y = analytic(x,0)
    yold2 = y
    yold = y

    dt = cfl[idt]*dx/abs(v)
    ntmax = (int)(tf/dt+1)
    err.append(zeros(ntmax))
    nplot=idt-2

    for it in range(ntmax):
        t = it*dt
        # save previous and previous previous data
        yold2 = yold
        yold = y

        y=upwind(y)           # integrate advection equation
        y=apply_bcs(x,y)      # after update, apply boundary conditions
        yana = analytic(x,t) # get analytic result for time t

        # compute error estimate
        err[idt][it] = sqrt(sum([(yana[i]-y[i])**2 for i in range(len(x))]))/n

        if t%150==0 and t>=150 and t<=600:
            nplot+=3
            subplot(4,3,nplot)
            plot(x,y,'b-')     # plot numerical result
            plot(x,yana,'r-')  # plot analytic results
            ha = 60 if (nplot <=3 or idt==2) else 10
            va2 = 0.5 if idt==2 else 0.6
            pos= 60 if nplot <= 6 else 10
            text(ha,0.8*max(max(y),max(yana)),'t={0}'.format(t))
            text(ha,va2*max(max(y),max(yana)),'cfl={0}'.format(cfl[idt]))
            draw()
```

```
savefig('upwind_snapshots.pdf')
ioff()
show()
clf()
for idt in range(len(cfl)):
    dt=cfl[idt]*dx/abs(v)
    plot(arange(0,dt*(int)(tf/dt+1),dt),err[idt])
xlabel('t')
ylabel('error')
ylim([0,0.001])
legend(['cfl={0}'.format(cfl[0]),'cfl={0}'.format(cfl[1]),
        'cfl={0}'.format(cfl[2])],loc='best')
savefig('upwind_err.pdf')
show()
```