# Ay 190 Assignment 6

Linear Systems of Equations

February 7, 2013

## 1 Solving Large Linear Systems of Equations

### 1.1 The given linear systems of equations

Five linear systems of equations (LSEs) downloaded from `http://www.tapir.caltech.edu/~cott/ay190/stuff/LSEi_m.dat` and `http://www.tapir.caltech.edu/~cott/ay190/stuff/LSEi_bvec.dat` with `i` = 1..5. The properties of the LSEs and whether or not a unique solution exists (i.e. $\det A \neq 0$) can be found in Table 1.

### 1.2 Solving the LSEs

A routine written by Isaac Evan (available at `https://github.com/ievans/GaussianElimination/blob/master/gaussianelimination.py`, see also Appendix A) was used to solve the LSEs using Gauss elimination and back-substitution. As a comparison, the LSEs were also solved using NumPy's `linalg.solve` routine, which employs LU decomposition, partial pivoting, and row swapping. The performance of both routines on the 5 LSEs are shown in Table 1.

**Table 1:** Dimensions and solution speeds of the 5 given LSEs

| i | $\dim A$ | $\dim b$ | Gauss Elimination | NumPy solver |
|---|---|---|---|---|
| 1 | $10 \times 10$ | 10 | 0.000598 s | 0.000256 s |
| 2 | $100 \times 100$ | 100 | 0.256901 s | 0.002186 s |
| 3 | $200 \times 200$ | 200 | 1.979687 | 0.006756 s |
| 4 | $1000 \times 1000$ | 1000 | 247.190892 s | 0.278832 s |
| 5 | $2000 \times 2000$ | 2000 | 1940.300075 | 1.819025 |

## 2 Stiff ODE Systems

Here we will solve the stiff ODE

$$\begin{aligned} \frac{d}{dt}Y_1 &= \quad Y_1 - 99Y_2 \\ \frac{d}{dt}Y_2 &= \ -Y_1 + 99Y_2 \end{aligned}$$

over $t = 0$ to $t = 4$ for the initial conditions $Y_1(0) = 1$, $Y_2(0) = 0$.

### 2.1 Analytic solution

Plugging in the ansatz $\mathbf{Y} = \mathbf{v} \exp \lambda t$ into the ODEs gives

$$\begin{aligned} v_1 \lambda e^{\lambda t} &= v_1 e^{\lambda t} - 99 v_2 e^{\lambda t} \\ v_1 \lambda &= v_1 - 99 v_2 \\ v_2 &= \frac{1}{99}(1-\lambda) v_1 \end{aligned} \qquad \begin{aligned} v_2 \lambda e^{\lambda t} &= -v_1 e^{\lambda t} + 99 v_2 e^{\lambda t} \\ v_2 \lambda &= -v_1 + 99 v_2 \\ v_1 &= (99-\lambda) v_2 \end{aligned}$$

$$\begin{aligned} v_1 &= (1 - \frac{\lambda}{99})(1-\lambda) v_1 \\ 1 &= (1 - \frac{\lambda}{99} - \lambda + \frac{\lambda^2}{99} \\ 0 &= \lambda(-100 + \lambda). \end{aligned}$$

Thus $\lambda = 0$ or 100. $\lambda = 0$ yields a constant solution $\mathbf{Y} = \mathbf{v}$. For $\lambda = 100$, $v_1 = (99 - \lambda) v_2$ yields $v_1 = -v_2$. Solving for the initial conditions and adding a constant term yields the solution

$$\begin{aligned} Y_1(t) &= \frac{1}{100}(99 + e^{100t}) \\ Y_2(t) &= \frac{1}{100}(1 - e^{100t}). \end{aligned}$$

### 2.2 Numerical solutions

The ODEs were integrated using four different integrators: explicit Euler, 2nd and 4th order Runge-Kutta methods, and backwards Euler. Four step sizes $\Delta t = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ were used for each of the integrators to test for convergence. The results are shown below.

#### 2.2.1 Explicit Euler and Runge-Kutta methods

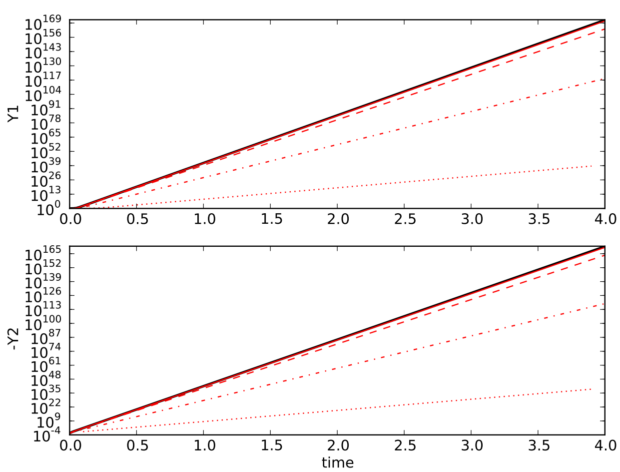


**Figure 1:** Solution computed using the explicit Euler method. The exact solution for both $Y_1$ (top) and $Y_2$ (bottom) is depicted in black. The solution computed with a step size of $10^{-1}$ is depicted with a dotted line, a step size of $10^{-2}$ with a dot-dashed line, a step size of $10^{-3}$ with a dashed line, and a step size of $10^{-4}$ with a solid red line. The explicit Euler method required a step size of at least $10^{-4}$ for convergence.

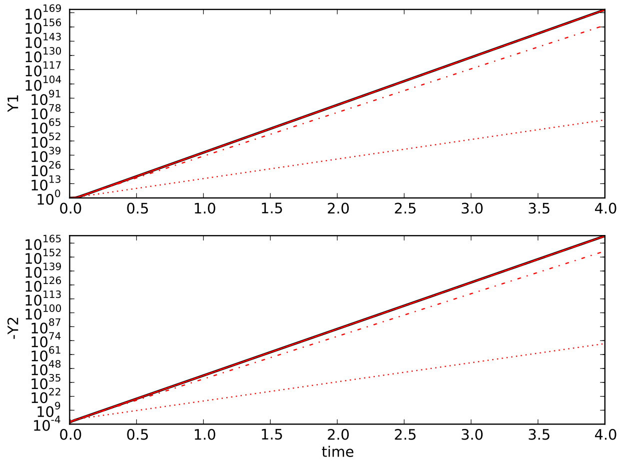


**Figure 2:** Similar to Figure 1, but for solutions computed using the 2nd order Runge-Kutta method. A step size of at least $10^{-3}$ was required for convergence.

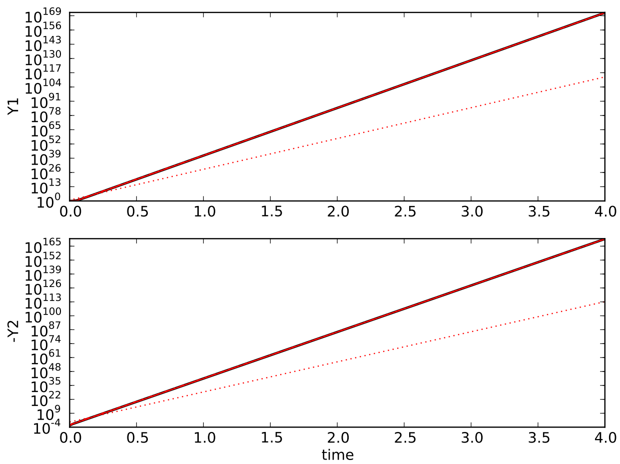


**Figure 3:** Similar to Figure 1, but for solutions computed using the 4th order Runge-Kutta (RK4) mthod. The superior convergence rate of RK4 is demonstrated here; a step size as large as $10^{-2}$ was sufficient for convergence.

### 2.2.2 Backwards Euler

For the backwards Euler method, the ODEs were written as

$$\frac{1}{\Delta t}\begin{pmatrix} Y_1^{i+1} - Y_1^i \\ Y_2^{i+1} - Y_2^i \end{pmatrix} = \begin{pmatrix} 1 & -99 \\ -1 & 99 \end{pmatrix}\begin{pmatrix} Y_1^{i+1} \\ Y_2^{i+1} \end{pmatrix}$$

Solving this system of equations for $Y_1^{i+1}$ and $Y_2^{i+1}$ yields

$$Y_1^{i+1} = \frac{Y_1^i(1-99\Delta t) - 99Y_2^i\Delta t}{1-100\Delta t}, \qquad Y_2^{i+1} = \frac{Y_2^i(1-\Delta t) - Y_1^i\Delta t}{1-100\Delta t}.$$

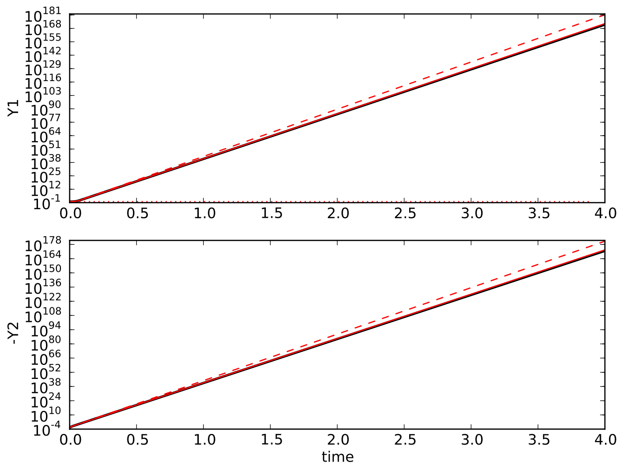


**Figure 4:** Similar to Figure 1, but for solutions computed using the backwards Euler method. Note that the solution is significantly erroneous (remaining constant at about 1 for $Y_1$ and at about 0.01 for $Y_2$) for a step size of 0.1. Due to a division by zero error, the solution was not calculated for a step size of $10^{-2}$. At a step size of $10^{-4}$, the solution was nearly (but not quite) convergent, as demonstrated in the following figure.

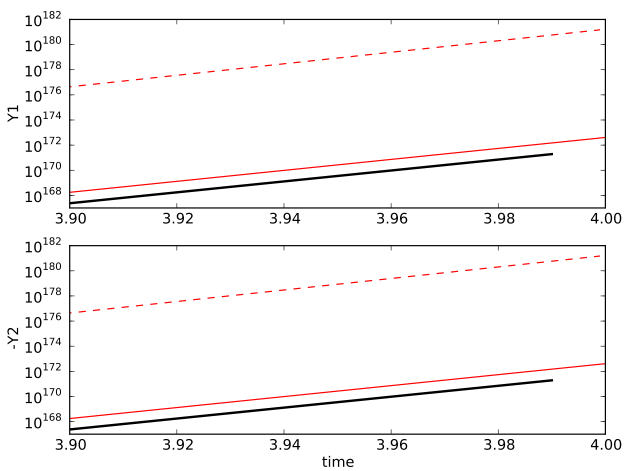


**Figure 5:** Close up of the tail of the solutions computed using the backwards Euler method. Even with a step size of $10^-4$ (solid red line), the solution did not converge to the exact solution.

# Appendices

The Python modules and scripts used in this assignment include

A The Gauss Elimination LSE Solver: `gaussianelimination.py` (see page 5)
B The Runge-Kutta Routines: `rungekutta.py` (see pages 5-8)
C Script for Set 6: `set6.py` (see pages 8-11)

## A The Gauss Elimination LSE Solver: gaussianelimination.py

```
# Copyright (c) Isaac Evans 2011
# All rights reserved.

def myGauss(m):
    # eliminate columns
    for col in range(len(m[0])):
        for row in range(col+1, len(m)):
            r = [(rowValue * (-(m[row][col] / m[col][col]))) for rowValue in m[col]]
            m[row] = [sum(pair) for pair in zip(m[row], r)]
    # now backsolve by substitution
    ans = []
    m.reverse() # makes it easier to backsolve
    for sol in range(len(m)):
            if sol == 0:
                ans.append(m[sol][-1] / m[sol][-2])
            else:
                inner = 0
                # substitute in all known coefficients
                for x in range(sol):
                    inner += (ans[x]*m[sol][-2-x])
                # the equation is now reduced to ax + b = c form
                # solve with (c - b) / a
                ans.append((m[sol][-1]-inner)/m[sol][-sol-2])
    ans.reverse()
    return ans
```

## B The Runge-Kutta Routines: rungekutta.py

```
# rk4.py
# created 11/8/11 by stacy kim
#
# Computes a function given its derivative func with initial conditions x using the
# Runge-Kutta method.  2nd, 3rd, and 4th order routines and adaptive stepping are
# implemented.
#
#
# modified 11/10/11 to include adaptive step size control
# modified 1/24/12
#    corrected 2-step rungekutta call (t+h/2 in second call)
```

```
#    corrected error comparison in stepper routine (element vs. list comparison)
# modified 1/31/12
#    modified error comparison in stepper (max(list) vs. element)
#    removed calculation complete status output from driver
#    added debug print stmts to stepper to check error decrease
# modified 2/6/12: removed shebang line
# modified 2/12/12
#    added t0 to the driver's parameter list
#    create min step size (abort if reached)
# modified 1/21/13
#    renamed module from rk4 to rungekutta and 4th order routine
#    implemented 2nd and 3rd order runge-kutta methods
#    modified driver and stepper routines to use the rk method of the given order
#    switched support of vector arith. from homegrown Vector module to numpy


import sys, math

MAXITER=1000
MINH=1e-8
h=1.0


# TIME-STEPPING ROUTINES ------------------------------------------------------

def driver(rk,x,t0,tf,h,f,err,fn):
    """
    Solves the differential equation 'f' using the given Runge-Kutta method
    'rk' from t0 to tf.  Takes time steps of 'h' or adaptively steps to satisfy
    the given accuracy criteria 'err'.  Results are printed to the file fn.
    """

    ADAPTIVE = 1 if h==0 else 0

    # Open output file and write headers
    file=open(fn,'w')
    file.write(fn)
    file.write('\nh={0},t0={5},tf={1},err={4}\n{2}{3}'.format(h,tf,'t'.rjust(19),'h'.rjust(19),err,t0))
    for i in range(len(x)): file.write('x{0}'.format(i+1).rjust(19))
    file.write('\n')


    #Iteratively calculate position and velocities using given time steps
    t=t0
    i=0
    while (t<=tf):
        # Write results to file
        file.write('{0} {1} '.format(str(t).rjust(19),str(h).rjust(19)))
        for j in range(len(x)): file.write('{0} '.format(str(x[j]).rjust(19)))
        file.write('\n')

        # Calculate function value at next step
        if (ADAPTIVE):
            x,t,h=stepper(rk,x,t,f,err)
```

```
        else:
            t=h*i
            x=rk(x,t,h,f)
            i+=1

    file.close()

    return


def stepper(rk,x,t,f,err):
    """
    Computes the next step of the function f using the given Runge-Kutta method
    rk to the given accuracy 'err'.  Only called if adaptively time stepping.
    """

    global h
    h0=h
    n=0

    while (1):
        x1=rk(x,t,h,f)
        x2=rk(rk(x,t,h/2,f),t+h/2,h/2,f)

        if (max(abs(x2-x1)) > err[0]):
            h/=2
            if h < MINH:
                print 'Exceeded max precision.'
                sys.exit()
        else:
            h_old=h
            if max(abs(x2-x1)) < err[0]/10.: h*=2
            return x2,t+h_old,h_old

        n+=1
        if ((MAXITER-n)<10):
            print "abs(x2-x1)={0},err={1}\n".format(abs(x2-x1),err)
            if (n==MAXITER):
                print "Failed to converge within {0} iterations.".format(MAXITER)
                print "[x={0},err={1},h0={2},hf={3},t={4}]".format(x,abs(x2-x1),h0,h,t)
                sys.exit()



# RUNGE-KUTTA ROUTINES ---------------------------------------------------------

def rk4(x,t,h,f):
    """
    Computes the function value x with derivative f at the time t + h using
    the fourth order Runge-Kutta method.
    """

    k1=h*f(x,t)
    k2=h*f(x+k1/2,t+h/2)
```

```
    k3=h*f(x+k2/2,t+h/2)
    k4=h*f(x+k3,t+h)

    return x + (k1 + 2*k2 + 2*k3 + k4)/6


def rk3(x,t,h,f):
    """
    Computes the function value x with derivative f at the time t + h using
    the third order Runge-Kutta method.
    """

    k1=h*f(x,t)
    k2=h*f(x+k1/2,t+h/2)
    k3=h*f(x-k1+2*k2,t+h)

    return x + (k1 + 4*k2 + k3)/6


def rk2(x,t,h,f):
    """
    Computes the function value x with derivative f at the time t + h using
    the second order Runge-Kutta method.
    """

    k1=h*f(x,t)
    k2=h*f(x+k1/2,t+h/2)

    return x + k2
```

## C Script for Set 6: set6.py

```
# set6.py
# created 1/29/13 by stacy kim

import numpy as np, time
import matplotlib.pyplot as plt
from gaussianelimination import *
from rungekutta import *


# EXERCISE 1 ---------------------------------------------------------------------
print 'reading in LSEs with dimensions'
m,b=[],[]
for i in range(1,6):
    b.append([float(line[1:])
              for line in open('LSE'+str(i)+'_bvec.dat').read().split('\n')[:-1]])
    m.append([[float(n) for n in line.split(' ')[1:] if n != ''] \
              for line in open('LSE'+str(i)+'_m.dat').read().split('\n')[:-1]])

    print '  ',i, '\tm:',len(m[-1]), len(m[-1][0]),'\tb:',len(b[-1])
```

```
for i in range(len(m)):
    sy=[m[i][j]+[b[i][j]] for j in range(len(m[i]))]
    #t1=timeit.timeit(stmt='x1=myGauss(sy)',setup='from __main__ import myGauss, sy,x1',number=1)
    t1=time.clock()
    ans1=myGauss(sy)
    t1=time.clock()-t1

    t2=time.clock()
    ans2=np.linalg.solve(m[i],b[i])
    t2=time.clock()-t2
    print i,':',t1, t2

    if np.all([ans1[j]==ans2[j] for j in range(len(ans1))]):
        print 'Does not match for LSE',j+1,'!'


# EXERCISE 2 -------------------------------------------------------------------
# Stiff ODE Systems

# exact solution
t_exact =np.arange(0,4,0.01)
y1_exact=np.array([(99+math.exp(100*tt))/100 for tt in t_exact])
y2_exact=np.array([(1-math.exp(100*tt))/100 for tt in t_exact])

def plot_exact():
    plt.subplot(211)
    plt.plot(t_exact,y1_exact,'k',lw=2)
    plt.ylabel('Y1')
    plt.yscale('log')
    plt.xlim([0,4])

    plt.subplot(212)
    plt.plot(t_exact,-y2_exact,'k',lw=2)
    plt.xlabel('time')
    plt.ylabel('-Y2')
    plt.yscale('log')
    plt.xlim([0,4])


# RHS of our stiff ODE system
def rhs(x,t):
    # x = [y1, y2]
    dy1= x[0] - 99.0*x[1]
    dy2=-x[0] + 99.0*x[1]
    return np.array([dy1,dy2])


# Inputs
h=[0.1,0.01,1e-3,1e-4] # step size
x0=np.array([1.,0.])   # initial condition, x=[y1,y2]
t0=0.0                 # initial time
tf=4.0                 # time to integrate until
err=[1.0,1.0]          # desired accuracy for each element of x
```

```
ls=[':','-.','--','-']      # line styles (one for each each step size)


# explicit Euler
print 'explicit euler'
plot_exact()
for hh in h:
    x=[x0[0],x0[1]]
    y1,y2=[x[0]],[x[1]]
    t=np.arange(t0,tf,hh)
    for tt in t[1:]:
        x+=hh*rhs(x,tt)
        y1.append(x[0])
        y2.append(x[1])

    plt.subplot(211)
    plt.plot(t,y1,'r',ls=ls[h.index(hh)])
    plt.subplot(212)
    plt.plot(t,-np.array(y2),'r',ls=ls[h.index(hh)])

plt.savefig('stiff_explicit_euler.pdf')
plt.show()


# RK2 integrator
print 'rk2'
plot_exact()
for hh in h:
    fn='stiff_rk2_h{0}.dat'.format(hh)
    driver(rk2,x0,t0,tf,hh,rhs,err,fn)

    dat = np.array([[float(el) for el in line.split(' ') if el != '']
                    for line in open(fn).read().split('\n')[3:-1]])
    t,y1,y2=dat[:,0],dat[:,2],dat[:,3]

    plt.subplot(211)
    plt.plot(t,y1,'r',ls=ls[h.index(hh)])
    plt.subplot(212)
    plt.plot(t,-y2,'r',ls=ls[h.index(hh)])

plt.savefig('stiff_rk2.pdf')
plt.show()


# RK4 integrator
print 'rk4'
plot_exact()
for hh in h:
    fn='stiff_rk4_h{0}.dat'.format(hh)
    driver(rk4,x0,t0,tf,hh,rhs,err,fn)

    dat = np.array([[float(el) for el in line.split(' ') if el != '']
                    for line in open(fn).read().split('\n')[3:-1]])
    t,y1,y2=dat[:,0],dat[:,2],dat[:,3]
```

```
    plt.subplot(211)
    plt.plot(t,y1,'r',ls=ls[h.index(hh)])
    plt.subplot(212)
    plt.plot(t,-y2,'r',ls=ls[h.index(hh)])

plt.savefig('stiff_rk4.pdf')
plt.show()


# backward Euler
print 'bkwd euler'
plot_exact()
for hh in h:
    if hh==0.01: continue # will cause zero division error
    x=[x0[0],x0[1]]
    y1,y2=[x[0]],[x[1]]
    t=np.arange(t0,tf,hh)
    for tt in t[1:]:
        y1.append(-(y1[-1]-99*y1[-1]*hh-99*y2[-1]*hh)/(100*hh-1))
        y2.append(-(y2[-1]-y1[-1]*hh-y2[-1]*hh)/(100*hh-1))

    plt.subplot(211)
    plt.plot(t,y1,'r',ls=ls[h.index(hh)])
    plt.xlim([3.9,4])
    plt.ylim([1e167,1e182])
    plt.subplot(212)
    plt.xlim([3.9,4])
    plt.ylim([1e167,1e182])
    plt.plot(t,-np.array(y2),'r',ls=ls[h.index(hh)])

plt.savefig('stiff_bkwd_euler_closeup.pdf')
plt.show()
```