MAT351 - HW1 - Part 2

Tom Bertalan

February 25, 2014

Contents

1 *Problem 5: Wilson Exercise 5.1, p70, et al.*

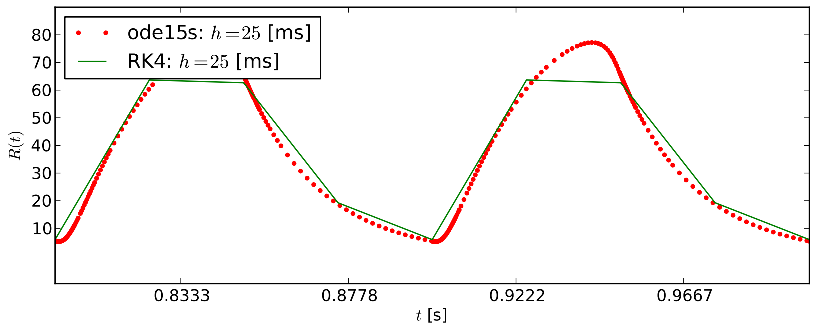

Figure 1: Adaptive timestepping vs widely-stepped RK4. Since Matlab's `ODE45` and SciPy's equivalent `dopri5` actually don't do vanilla RK4, but a more sophisticated adaptive method, I thought it would be interesting to show this graphically. The same timestep is given to both solvers, but the ode15s-analog adaptively adds extra timesteps where the rate-of-change is high.

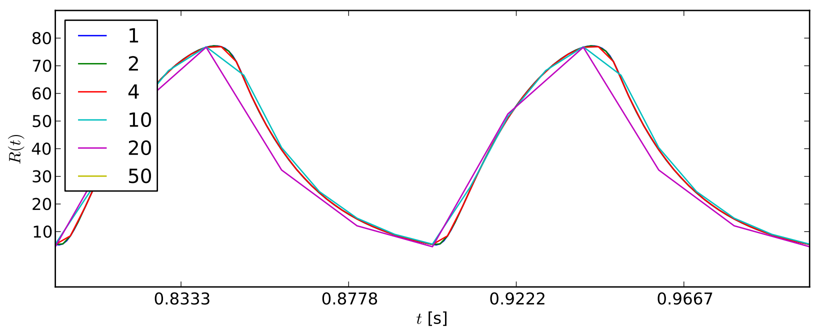

Figure 2: Wilson, problem 5.1. simple Naka-Rushton neuron via Runge-Kutta $\mathcal{O}(4)$ time-integration. Only the last 0.20 [s] of integration time is shown. Legend gives timestep in [ms]. Analytical solution for $P(t) = 1$ (constant forcing) is $R(t) = \frac{50}{13}(1 - e^{-50t})$

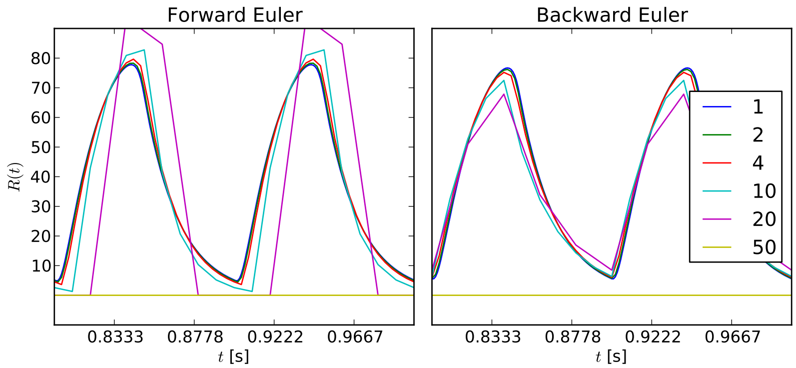

Figure 3: Naka-Rushton neuron via forward and backward Euler-integration. Only the last 0.20 [s] of integration time is shown. Legend gives timestep in [ms]. All but the largest of the timesteps seem to be in the domain of stability for backwards Euler for this problem.

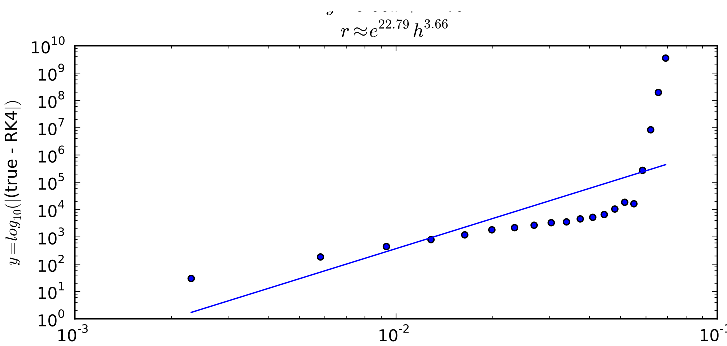

Figure 4: Growth of RK4 error as stepsize h increases. Error is measured as the L_2 norm of the difference between the solution for $h = 10^{-5}$ [s] (assumed to be the "true" solution) and a linear interpolant of the solution $R(t)$ for the given value of h. In contrast to the theoretical prediction of $r \sim \mathcal{O}(4)$, I regressed an exponent better than 2 for this problem (below the limiting h value where RK4 diverged). This is consistent, I think with the $\mathcal{O}(4)$ estimate being an upper bound on the error.

```python
'''
Hugh R. Wilson, "Spikes, decisions, and actions: dynamical foundations of
neuroscience", chapter 5, problem 1

With additions from Holmes 323HW14.1.pdf, problem 5.

@author: bertalan@princeton.edu
'''
import numpy as np
import matplotlib.pyplot as plt

from Integrators import integrate, logging


def dXdt(X, t):
    R = X[0]  # len(X) = 1
    P = 20. * np.sin(np.pi * 20 * t)
    if P < 0:
        S = 0
```

```
    else:
        S = 100. * P**2 / (25 + P**2)
    return np.array([(-R + S) / .02])


tf = 1.0



# First, a plot showing adaptive timestepping. Not that it was requested.
adapFig = plt.figure(figsize=(8.5,3.5))
adapAx = adapFig.add_subplot(1, 1, 1)
h = .025
for method, label, style in zip(
                                ('ode15s', 'rungekutta'),
                                ('ode15s', 'RK4'),
                                ('r.', 'g')):
    X, T = integrate(dXdt, h, t0=0.0, tf=tf, X0=(0.0,), method=method)
    adapAx.plot(T, X[0,:], style, label='%s: $h=%.0f$ [ms]' % (label, h*1e3))

# Now, Runge-Kutta O(4) with various timesteps.
tstepFig = plt.figure(figsize=(8.5,3.5))
tstepAx = tstepFig.add_subplot(1, 1, 1)

hs = .001, .002, .004, .01, .02, .05
# To get something to compare against, we'll use a super-fine timestep and just
#  hope that we don't get subtractive machine-precision error.

for h in hs:
    X, T = integrate(dXdt, h, 0.0, tf, (0.0,), method='rungekutta')
    tstepAx.plot(T, X[0,:], label='%.0f' % (h*1e3))

# Let's explicitly look at the norm of the error
def linInterp(T, X, Treq):
    '''
    Produce linear interpolation of X(T) at the requested T values
    '''
    def findClosest(vec, val):
        """Return the index where vec[index] is closest to val.
        >>> findClosest([2, 8, 3, 6], 5)
        3
        """
        distances = np.abs([val - x for x in vec])
        return distances.tolist().index(np.min(distances))
    assert T[0] == Treq[0], 'I am lazy.'
```

```
    j = 0
    built = np.empty(Treq.shape)
    for i in range(len(T) - 1):
        t1 = T[i]
        t2 = T[i + 1]

        while j < len(Treq) and t1 <= Treq[j] < t2:
            dxdt = (X[i+1] - X[i]) / (T[i+1] - T[i])
            built[j] = dxdt * (Treq[j] - T[i]) + X[i]
            j += 1

    return built

errFig = plt.figure(figsize=(8.5, 3.5))
errAx = errFig.add_subplot(1, 1, 1)
trueSoln, trueTimes = integrate(dXdt, .00001, 0.0, tf, (0.0,), method='rungekutta')
errs = []
hsErr = np.log(np.logspace(.001, .03, 20))
# hsErr = np.linspace(.001, .03, 20)
for h in hsErr:
    logging.info('error plotting: h=%f' % h)
    X, T = integrate(dXdt, h, 0.0, tf, (0.0,), method='rungekutta')
    interpolated = linInterp(T, X.ravel(), trueTimes)
    errs.append(np.linalg.norm(trueSoln[0, :] - interpolated))
errAx.scatter(hsErr, errs)
errAx.set_xlabel('$x=log_{10}(h)$')
errAx.set_ylabel('$y=log_{10}(|$(true - RK4$|)$')
errAx.set_xscale('log')
errAx.set_yscale('log')
# fit a line:
from scipy.optimize import minimize
def minimizeThis(MB):
    m = MB[0]
    b = MB[1]
    #l10(err)0 = m * l10(h) + b
    return np.linalg.norm(m * np.log(hsErr) + b - np.log(np.asarray(errs)))
MB = minimize(minimizeThis, np.array([4,.001])).x
xlim = errAx.get_xlim()
ylim = errAx.get_ylim()
m, b = tuple(MB)
errAx.plot(hsErr, np.exp(b)*hsErr**m)
logging.info('m=%f, b=%f' % (m, b))
errAx.set_title(
```

```
                r'$y\approx%.2fx+%.2f$' % (m, b) + '\n' +
                r'$r\approx e^{%.2f}h^{%.2f}$' % (b, m)
                )

showTime = 0.20  # We're not going to plot the whole thing.

# Now, forward/backward Euler with the same timesteps:
fbFig = plt.figure(figsize=(8.5,4))
ax2forward  = fbFig.add_subplot(1, 2, 1)
ax2backward = fbFig.add_subplot(1, 2, 2)
for axis, method in zip(
                        (ax2forward, ax2backward),
                        ('Forward Euler', 'Backward Euler')
                        ):
    axis.set_title(method)
    for h in hs:
        X, T = integrate(dXdt, h, 0.0, tf, (0.0,), method=method
                                                          .replace(' ', '')
                                                          .replace('Forward', ''))
        axis.plot(T, X[0,:], label='%.0f' % (h*1e3))


# Fix up the plots a bit.
for axis in tstepAx, ax2forward, ax2backward, adapAx:
    d = showTime / 6
    maxy = 90
    miny = -10

    if axis is not ax2backward:  # We'll do something different with this.
        axis.set_ylabel('$R(t)$')
        axis.set_yticks(np.arange(10, maxy, 10))
    else:
        axis.set_yticks([])
    axis.set_ylim(miny,maxy)

    axis.set_xlabel('$t$ [s]')
    axis.set_xticks(np.linspace(tf-showTime+d, tf-d, 4))
    axis.set_xlim(tf - showTime, tf)
tstepAx.legend(loc='upper left')
adapAx.legend(loc='upper left')
ax2backward.legend(loc='right')
fbFig.subplots_adjust(bottom=.14, top=.9, left=.1, right=.99, wspace=.05)
tstepFig.subplots_adjust(bottom=.18, top=.99, left=.08, right=.99)
adapFig.subplots_adjust(bottom=.18, top=.99, left=.08, right=.99)
```

```
errFig.subplots_adjust(top=.9)

# Save them.
tstepFig.savefig('hw1b-wils5_1.pdf')
fbFig.savefig('hw1b-5-forwardBackward.pdf')
adapFig.savefig('hw1b-5-adaptive.pdf')
errFig.savefig('hw1b-5-error.pdf')

# The analytical solution for P=1 is
#  R(t) = \frac{50}{13}(1 - e^{-50t})

# plt.show()
```

2 Problem 6: Numerical experiments on a nonliner ODE

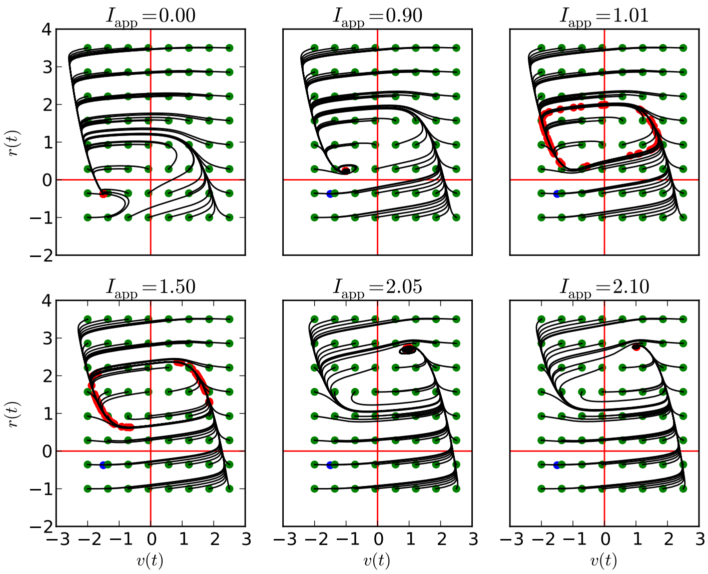

Figure 5: Partial phase portraits for a Fitzhugh-Nagumo neuron for several values of the applied-current parameter I_{app}. Green points are initial conditions; red are final (after 10 [s] of integration). As also happens in the Hodgkin-Huxley neuron, there is a lower and an upper steady state, separated from a stable limit cycle by a pair of Hopf bifurcations. For $I_{\text{app}} = 0$, the fixed point is at $v = \frac{-3}{2}$, $r = \frac{-3}{8}$.

```
'''
Created on Feb 24, 2014, 3:58:19 PM

@author: bertalan
'''

import numpy as np
import matplotlib.pyplot as plt
from Integrators import integrate, logging

def FitzhughNagumo(Iapp):
    '''This closure gives the RHS function for a particular applied current value.'''
    def dXdt(X, t):
        '''t is not used (the ODE is autonomous)'''
        v = X[0]
        r = X[1]
        tv = .1
        tr = 1
        dvdt = (v - v**3 / 3 - r + Iapp) / tv
        drdt = (-r + 1.25 * v + 1.5) / tr
        return np.array([dvdt, drdt])
    return dXdt
```

```
fig = plt.figure()

Iapps = 0, 0.9, 1.01, 1.5, 2.05, 2.1
# for Iapp = 0, the fixed point is (-3/2, -3/8)
axes = []
for i in range(6):
    axes.append(fig.add_subplot(2, 3, i+1))

for ax, Iapp in zip(axes, Iapps):
    logging.info('Iapp=%f' % Iapp)
    dXdt = FitzhughNagumo(Iapp)
    ax.axhline(0, color='red')
    ax.axvline(0, color='red')
    ax.scatter(-3./2, -3./8, color='blue')
    for v0 in np.linspace(-2, 2.5, 8):
#         logging.info('v0=%s' % str(v0))
        for r0 in np.linspace(-1, 3.5, 8):
            X0 = v0, r0
            logging.info('X0=%s' % str(X0))
            # Uncomment these two lines to verify the Euler solution with RK4:
            #X, T = integrate(dXdt, .01, 0, 10.0, X0, method='rungekutta')
            #ax.plot(X[0,:], X[1,:], 'r.')
            X, T = integrate(dXdt, .01, 0, 10.0, X0, method='euler')
            ax.plot(X[0,:], X[1,:], 'k')
            ax.scatter(X[0,0], X[1,0], color='green')  # the initial condition...
            ax.scatter(X[0,-1], X[1,-1], color='red')  # ...and the final point
            ax.set_title('$I_\mathrm{app} = %.2f$' % Iapp)
for i in 0, 1, 2:
    axes[i].set_xticks([])
for i in 1, 2, 4, 5:
    axes[i].set_yticks([])
for i in 0, 3:
    axes[i].set_ylabel('$r(t)$')
for i in 3, 4, 5:
    axes[i].set_xlabel('$v(t)$')
for ax in axes:
    ax.set_xlim(-3, 3)
    ax.set_ylim(-2, 4)

ax.legend()

fig.savefig('hw1bp6-flows.pdf')
```

```
plt.show()
```

3 Integrators.py

```
'''
Created on Feb 24, 2014, 1:04:36 PM

@author: bertalan
'''

import numpy as np  # Gives us matrices and linear algebra.
import logging
logging.basicConfig(format='%(levelname)s: %(message)s',
                    level=logging.INFO, # Turn on/off debug by switching INFO/ERROR.
                    name='log')


def integrate(dXdt, h, t0, tf, X0, method='euler', newtontol=1e-6):
    """
    Integrate the differential equation dXdt from t0 to tf in steps of size h,
    and initial condition X0. method can be one of 'euler', 'rungeKutta', or
    'backwardEuler'; not case-sensitive.

    If 'backwardEuler' is used, newtontol specifies the tolerance for the
    implicit solver.

    Returns the history of states and the times at which they occured.

    dXdt must accept a state and a time. E.g, this linear ODE:

    >>> def dXdt(X, t):
    ...     A = np.array([[1, -2],
    ...                   [3, -4]])  # Eigenvalues are -2 and -1, so the origin is stable.
    ...     X = np.array(X).reshape((2,1))
    ...     return np.dot(A, X)
    >>> h=.1; t0=0; tf=4; X0=(4,8)
    >>> X, T = integrate(dXdt, h, t0, tf, X0)
    >>> print X.shape, T.shape
    (2, 41) (41,)
    >>> assert T[0] == t0
    >>> assert T[-1] == tf
    >>> assert (X[:,0] == X0).all  # ".all" checks that all boolean-valued
    ...                            #  array elements are True
    >>> from matplotlib import pyplot as plt
    >>> f = plt.figure()
    >>> a = f.add_subplot(1, 1, 1)
```

```
Do a phase portrait.
>>> p = a.plot(X[0, :], X[1, :])

Mark the initial condition.
>>> s = a.scatter(X0[0], X0[1])

Try the ODE45 "equivalent"
>>> X, T = integrate(dXdt, h, t0, tf, X0, method='ode45')
>>> assert X.size > 2

"""

# Reshape the initial condition as a column vector. Allows us to accept
#  row-vector numpy arrays, lists, tuples, or other list-like things.
#  This might fail if X0 is something stupid, like a list-of-lists,
#  or a string.
X0 = np.array(X0)
N = X0.size
X0 = X0.reshape((N,1))

# Initialize the history arrays.
#  for more general, adaptive methods, we couldn't do this. At least, not
#  all at once. The +h allows us to include both t0 and tf in the history.
T = np.arange(t0, tf+h, h)
X = np.empty((N, T.size))
X[:,0] = X0.ravel()

f = lambda tn, xn: dXdt(xn, tn)  # symbols chosen to match Wikipedia's RK4
h = h

# some method-codes
ode45 = 'dopri5' 'ode45'
ode15s = 'vode' 'ode15s'

if method.lower() == 'rungekutta':
    for n in xrange(T.size-1):
        xn = X[:, n].reshape((N,1))
        tn = T[n]
        k1 = f(tn, xn)
        k2 = f(tn + h/2., xn + h * k1 / 2.)
        k3 = f(tn + h/2., xn + h * k2 / 2.)
        k4 = f(tn + h,    xn + h * k3)
        X[:, n+1] = (xn + 1/6. * h * (k1 + 2. * k2 + 2. * k3 + k4)).ravel()
```

```
elif method.lower() == 'backwardeuler':
    for k in xrange(T.size-1): # We're using k because Wikipedia does.
        xk = X[:, k].reshape((N,1))
        tk = T[k]
        tk1 = tk + h
        xk10 = xk  # initial iterate
        # While both Numpy and Scipy have fancier things available, we'll
        #  deliberately use Newton-Rhapson, without putting forth the effort
        #  to write our own.
        from scipy.optimize import newton
        def zeroMe(xk1):
            return xk + h * f(tk1, xk1) - xk1
        xk1 = newton(zeroMe, xk10, tol=newtontol)#, maxiter=...)
        X[:, k+1] = xk1.ravel()

elif method.lower() in ode45 + ode15s:
    # Awkward to use. scipy.integrate.odeint is easier. This code is adapted
    #  from the [SciPy-User] mailing list:
    #  http://mail.scipy.org/pipermail/scipy-user/2011-March/028683.html
    # It's safe to use a quite large h value for the ode15s method, since
    #  the real step size is chosen automatically by SciPy. This can be
    #  verified by using this method on the Naka-Rushton problem with a
    #  step size larger than, say 1/6 the oscillatory period, and
    #  scatter-plotting R vs t. The parts with large |dR/h|
    #  automatically get smaller timesteps.
    from scipy.integrate import ode
    solver = ode(f)
    T = [t0]
    X = [X0]
    solver.set_initial_value(X0, t0)
    if method.lower() in ode45:
        solver.set_integrator('dopri5')
    if method.lower() in ode15s:
        solver.set_integrator('vode', method='bdf', order=15)
    while solver.successful() and solver.t < tf:
        solver.integrate(solver.t + h, step=True)  # We should get *at least*
        T.append(solver.t)  #  as many steps as the other methods; perhaps
        X.append(solver.y.reshape((N,)))  #  more.
    T = np.array(T)
    X = np.array(X).T

else:  # assume forward Euler if not ^^. Later, we might add, say,
    #     Adams-Bashforth. Or perhaps backwards Euler.
```

```
        for n in xrange(T.size-1):
            xn = X[:, n].reshape((N,1))
            tn = T[n]
            X[:, n+1] = (xn + f(tn, xn) * h).ravel()

    return X, T


if __name__ == '__main__':
    '''
    The code in this block only runs if this file is invoked as a script;
    not if things from this file are imported elsewhere. Guido van Rossum
    disapproves of this usage, but it's just so handy.
    '''
    # Test that documentation examples are correct.
    import doctest
    doctest.testmod()

    # Do a pseudo-triangle-wave thing.
    def dXdt(X, t):
        X = np.array(X).reshape((2,1))
        x = X[0,0]
        y = X[1,0]
        return (np.array([np.cos(.05423*t), np.sin(t)]) +\
                np.array([np.cos(1.121235432*x), np.sin(.123*y)])).reshape((2,1))
    X, T = integrate(dXdt, h=.1, t0=0, tf=4, X0=(42, 68), method='rungekutta')
    from matplotlib import pyplot as plt
#     f = plt.figure()
#     a = f.add_subplot(1, 1, 1)
#     a.plot(X[0,:], X[1,:])  # will plot both X[0,:], and X[1,:] vs. T

    # Display any plots that may have been generated.
#     plt.show()
```