Amath 586: Homework 3

Due on April 30, 2015

Brian de Silva

Problem 1

Let $g(x) = 0$ represent a system of s nonlinear equations in s unknowns, so $x \in \mathbb{R}^s$ and $g : \mathbb{R}^s \to \mathbb{R}^s$. A vector $\bar{x} \in \mathbb{R}^s$ is a *fixed point* of $g(x)$ if

$$\bar{x} = g(\bar{x}). \tag{1}$$

One way to attempt to compute $\bar{x}$ is with *fixed point iteration*: from some starting guess x^0, compute

$$x^{j+1} = g(x^j) \tag{2}$$

for $j = 0,\ 1,\ \ldots$.

(a) Show that if there exists a norm $\|\cdot\|$ such that $g(x)$ is Lipschitz continuous with constant $L < 1$ in a neighborhood of $\bar{x}$, then fixed point iteration converges from any starting value in this neighborhood. **Hint:** Subtract equation (1) from (2).

(b) Suppose $g(x)$ is differentiable and let $g'(x)$ be the $s \times s$ Jacobian matrix. Show that if the condition of part (a) holds then $\rho(g'(\bar{x})) < 1$, where $\rho(A)$ denotes the spectral radius of a matrix.

(c) Consider a predictor-corrector method (see Section 5.9.4) consisting of forward Euler as the predictor and backward Euler as the corrector, and suppose we make N correction iterations, i.e., we set

$$\begin{aligned}
&\hat{U}^0 = U^n + kf(U^n)\\
&\text{for } j = 0,\ 1,\ \ldots,\ N-1\\
&\qquad \hat{U}^{j+1} = U^n + kf(\hat{U}^j)\\
&\qquad \text{end}\\
&U^{n+1} = \hat{U}^N.
\end{aligned}$$

Note that this can be interpreted as a fixed point iteration for solving the nonlinear equation

$$U^{n+1} = U^n + kf(U^{n+1})$$

of the backward Euler method. Since the backward Euler method is implicit and has a stability region that includes the entire left half plane, as shown in Figure 7.1(b), one might hope that this predictor-corrector method also has a large stability region.

Plot the stability region S_N of this method for $N = 2,\ 5,\ 10,\ 20$ (perhaps using `plotS.m` from the webpage) and observe that in fact the stability region does not grow much in size.

(d) Using the result of part (b), show that the fixed point iteration being used in the predictor-corrector method of part (c) can only be expected to converge if $|k\lambda| < 1$ for all eigenvalues λ of the Jacobian matrix $f'(u)$.

(e) Based on the result of part (d) and the shape of the stability region of Backward Euler, what do you expect the stability region S_N of part (c) to converge to as $N \to \infty$?

Solution:

(a) Suppose there exists a norm $\|\cdot\|$ such that $g(x)$ is Lipschitz continuous with constant $L < 1$ in a neighborhood, N, of $\bar{x}$. Let $x^0 \in N$ and let $x^1 = g(x^0),\ x^2 = g(x^1) = g(g(x^0))$, etc.. Notice that $g(N) \subset N$ since for any $x \in N, |g(x) - g(\bar{x})| \leq L|x - \bar{x}| \leq |x - \bar{x}| \implies g(x) \in N$. Hence $x^j \in N,\ j = 0, 1, 2, \ldots$. Hence we have

$$\begin{aligned}
\|x^j - \bar{x}\| &= \|g(x^{j-1}) - g(\bar{x})\| \\
&\leq L\|x^{j-1} - \bar{x}\| \\
&= L\|g(x^{j-2}) - g(\bar{x})\| \\
&\leq L^2\|x^{j-2} - \bar{x}\| \\
&\vdots \\
&\leq L^j\|x^0 - \bar{x}\|.
\end{aligned}$$

It follows that as $j \to \infty,\ \|x^0 - \bar{x}\| \to 0$.

(b) Suppose there exists a norm $\|\cdot\|$ such that $g(x)$ is Lipschitz continuous with constant $L < 1$ in a neighborhood of $\bar{x}$. Fix $\epsilon > 0$ small enough so that $\bar{x} + y \in N$ for any $\|y\| < \epsilon$, where N is the neighborhood around $\bar{x}$ in which g is Lipschitz continuous. Taylor expanding $g(\bar{x} + y)$ about $\bar{x}$ gives

$$g(\bar{x} + y) = g(\bar{x}) + g'(\bar{x})y + O(\|y\|^2) = g(\bar{x}) + g'(\bar{x})y + v$$

for some v with $\|v\| \leq C\|y\|^2$, some $C \in \mathbb{R}$. From this it follows that

$$g'(\bar{x})y = -v + g(\bar{x} + y) - g(\bar{x}),$$

and so

$$\begin{aligned}
\|g'(\bar{x})y\| &= \| - v + g(\bar{x} + y) - g(\bar{x})\| \\
&\leq \|v\| + \|g(\bar{x} + y) - g(\bar{x})\| \\
&\leq C\|y\|^2 + L\|y\|.
\end{aligned}$$

Since this holds for any $\|y\| < \epsilon$, choose y to be the unit-length eigenvector corresponding to the largest (in magnitude) eigenvalue of $g'(\bar{x}),\ \lambda$, scaled by $\frac{\epsilon}{2}$, i.e. $\rho(g'(\bar{x})) = \|\lambda\|$. Then the above becomes

$$\begin{aligned}
\|g'(\bar{x})y\| = |\lambda|\|y\| &\leq C\|y\|^2 + L\|y\| \\
\Rightarrow |\lambda| &\leq C\|y\| + L \\
&= C\frac{\epsilon}{2} + L.
\end{aligned}$$

As ϵ can be made arbitrarily small and $L < 1$, we conclude that $\rho(g'(\bar{x})) = |\lambda| < 1$.

(c) Applied to the test problem $u' = \lambda u$ this method yields

$$\hat{U}^0 = U^n + kf(U^n) = U^n + k\lambda U^n = (1 + z)U^n,$$

where $z = k\lambda$. Next

$$\begin{aligned}
\hat{U}^1 &= U^n + kf(\hat{U}^0) = U^n + k\lambda(1 + z)U^n = (1 + z + z^2)U^n, \\
\hat{U}^2 &= U^n + kf(\hat{U}^1) = U^n + k\lambda(1 + z + z^2)U^n = (1 + z + z^2 + z^3)U^n \\
&\vdots \\
\hat{U}^N &= (1 + z + z^2 + \cdots + z^{N+1})U^n.
\end{aligned}$$

Problem 1 continued on next page...

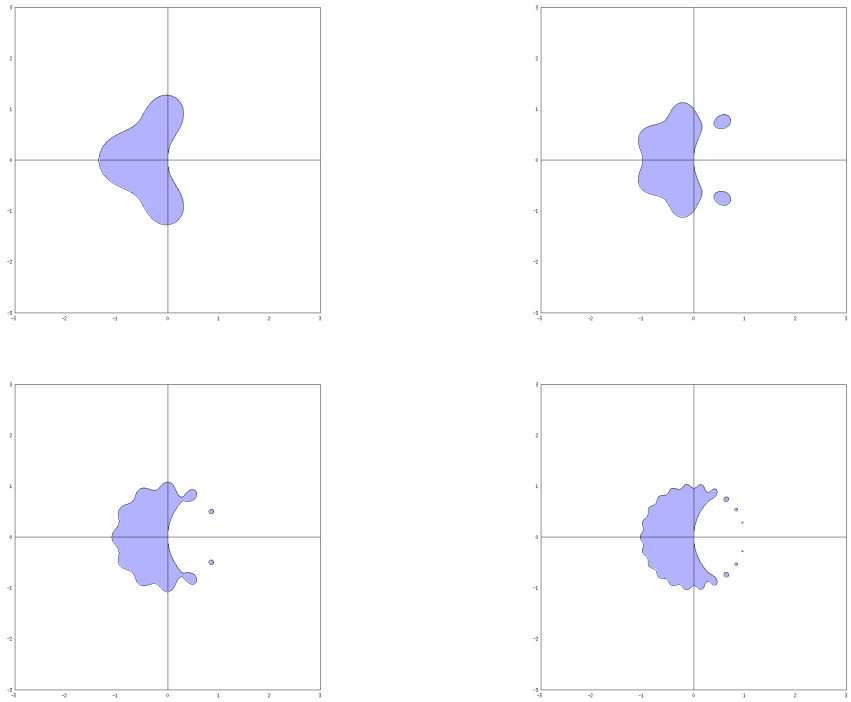


Figure 1: Top left: S_2, Top right: S_5, Bottom left: S_{10}, Bottom right: S_{20}

Thus $R(z) = 1 + z + z^2 + \cdots + z^{N+1}$.

Using `plotS.m` as suggested, we generate the following plots showing the stability regions S_N for $N = 2,\ 5,\ 10,\ 20$ (Figure 1). Notice that the stability region does not grow much in size as N is increased.

(d) Interpreting the method presented in (c) as a fixed point iteration to solve the nonlinear equation

$$U^{n+1} = U^n + kf(U^{n+1})$$

we take

$$g(x) = U^n + kf(x).$$

If the result from (a) is to hold and $g(x)$ is to converge to the fixed point U^{n+1} as desired, the condition from (a) must be satisfied. If this is the case, then the result from (b) holds, namely

$$\rho(g'(U^{n+1})) = \rho((U^n + kf(u))') = \rho(kf'(u)) < 1.$$

But $\rho(kf'(u)) = k\rho(f'(u))$, so this condition is equivalent to requiring that $|k\lambda| < 1$ for all eigenvalues λ of $f'(u)$. Notice that this requirement is slightly different from the result of (b) in that we now require that $\rho(kf'(u)) < 1$ for all value u, not just U^{n+1}. This is because we modify the contraction map g at each time step so that its fixed point becomes the next approximation we wish to compute. Therefore we must ensure that the entire family of contractions maps converge to their respective fixed points. If this condition is not satisfied then the result in (b) does not hold, so neither does the result in (a) and the map cannot be expected to converge.

(e) Based on part (d) we expect that the stability region S_N will approach the unit circle, i.e. the set on which $|\lambda k| = |z| < 1$ as this is where we expect the method to converge. The plots in Figure 1 support this expectation. As N is increased the stability regions look more and more like the unit circle.

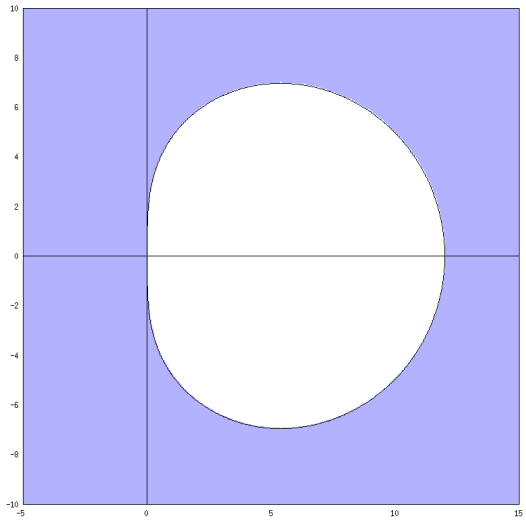


Figure 2: Stability region for TR-BDF2

Problem 2

Use the Boundary Locus method to plot the stability region for the TR-BDF2 method (8.6). You can use the Matlab script `makeplotS.m` from the book website, or a Python script as illustrated in `$AM586/codes/notebook3.ipynb` Observe that the method is A-stable and show that it is also L-stable.

Solution:

The TR-BDF2 method is given by

$$U^* = U^n + \frac{k}{4}(f(U^n) + f(U^*))$$
$$U^{n+1} = \frac{1}{3}(4U^* - U^n + kf(U^{n+1})).$$

For the test problem, $u' = \lambda u$ we can derive the relations

$$U^* = \left(\frac{1 + \frac{z}{4}}{1 - \frac{z}{4}}\right) U^n$$

and

$$U^{n+1} = \left(\frac{12 + 5z}{12 - 7z + z^2}\right) U^n,$$

where $z = \lambda k$. Hence $R(z) = \frac{12+5z}{12-7z+z^2}$.
Using `plotS.m` we plot the stability region for the TR-BDF2 method in Figure 2. It is easy to see from this plot that the method is A-stable. To see that it is also L-stable, simply notice that

$$\lim_{z \to \pm\infty} R(z) = \lim_{z \to \pm\infty} \frac{12 + 5z}{12 - 7z + z^2} = 0.$$

Problem 3

The goal of this problem is to write a very simple adaptive time step ODE solver based on the Bogacki–Shampine Runge–Kutta method. This is a third-order accurate 4-stage method that also produces a second-order accurate approximation each time step that can be used for error estimation. (This is the method used in the Matlab routine `ode23`.)

Note that it appears to require 4 evaluations of f each time step, but the last one needed in one step can be re-used in the next step. You should take advantage of this to reduce it to 3 new f-evaluations each step.

In lecture, I discussed the method (5.42) in which U^{n+1} is used for the next time step and is second-order accurate, while $\hat{U}^{n+1}$ is "first-order accurate", which means the 1-step error is $\mathcal{O}(k^2)$, and this is what is approximated by the difference $|U^{n+1} - \hat{U}^{n+1}|$.

For the Bogacki–Shampine method, the corresponding difference $|U^{n+1} - \hat{U}^{n+1}|$ will be approximately equal to the 1-step error of the second order method, so we expect it to have the form $k_n \tau^n \approx C_n k_n^3$, where k_n is the time step just used in the nth step, and C_n is a constant that will depend on how smooth the solution is near time t_n. After each time step, we can estimate this by

$$C_n \approx |U^{n+1} - \hat{U}^{n+1}|^{1/3}. \tag{3}$$

If we were simply using the second order method and trying to achieve an absolute error less than some ϵ over the time interval $t_0 \le t \le T$, then we would want to choose our time steps so that

$$|\tau^n| \le \epsilon/(T - t_0)$$

Then if we assume the method is behaving stably and we estimate the global error at time T by the sum of the one-step errors, this is roughly

$$\sum_m k_m |\tau^m| \le \left(\sum k_m\right) \frac{\epsilon}{T - t_0} = \epsilon.$$

Convince yourself that this suggests choosing the next time step as

$$\left(\frac{\epsilon}{C_n(T - t_0)}\right)^{1/2}. \tag{4}$$

Note that we have already taken a step with time step k_n in order to approximate C_n, so the simplest approach is just to use (4) to define our *next* time step k_{n+1}, with C_n estimated from (3). (A more sophisticated method would go back and re-take a smaller step if the estimate indicates we took a step that was too large.)

Note that the error estimate is based on the second-order method, but the value we take for our next step is U^{n+1}, which we hope is even more accurate.

Problem 3 continued on next page...

Implement this idea, with the following steps:

- First implement a function that takes a single time step with the B–S method and confirm that it is working, e.g. if you apply it to $u' = -u$ for a single step of size k the errors behave as expected when you reduce k.
- Then implement a fixed time step method based on this and verify that the method is third-order accurate.
- Implement an adaptive time version and test it on various problems where you know the solution.

Note: You will need to choose a time step for the first step. To keep it simple, just use $k_0 = 10^{-4}$. You should also set some maximum step size that's allowed since there may be times when the error estimate happens to be very small. Take this to be $k_{max} = 10\epsilon^{1/3}$.
Once your code is working, try it on problems of the form

$$u'(t) = \lambda(u - v(t)) + v'(t) \tag{5}$$

where $v(t)$ is a desired exact solution and the initial data is chosen as $u(0) = v(0)$.
In particular, try the following. Produce some sample plots of the solutions and also plots of the error and step size used as functions of t, as illustrated in the figures below.

(a) Exact solution

$$v(t) = \cos(t)$$

for $0 \le t \le 10$ and $\lambda = -1$. Try different tolerances ϵ in the range 10^{-2} to 10^{-10}.

(b) Try the above problem with more negative λ, e.g. $\lambda = -100$. What happens if ϵ is large enough that $k_{acc} > k_{stab}$? Does the method go unstable?

(c) Exact solution

$$v(t) = \cos(t) + \exp\left(-80(t-2)^2\right),$$

for $0 \le t \le 4$ and $\lambda = -1$. The Gaussian gives a region where the solution is much more rapidly varying and smaller time steps are required.

Solution:

See attached Julia notebook for verification that the single step and fixed time step methods achieve proper orders of accuracy, method implementations, and extra plots.

(a) First we apply the adaptive time-stepping method to the differential equation above with exact solution

$$\nu(t) = cos(t).$$

The results are summarized in Figure 3. As the error tolerance is decreased, the number of time steps taken increases rapidly. The method does a fairly good job of staying within the error tolerances provided. The error is at worst the same order of magnitude as the specified tolerances.

Figure 4 shows the numerical approximation, absolute error, and time step size for this problem when $\epsilon = 10^{-3}$. We see that the size of the time step is lengthened around the areas where the exact solution is fairly flat, where its derivative is zero. This is the type of behavior we expect from an adaptive time stepping method–larger time steps when the solution is changing slowly and smaller ones when it is rapidly varying.

Problem 3 continued on next page...

ϵ	Absolute Error	Steps taken
10^{-2}	4.2e-3	22
10^{-3}	1.2e-3	94
10^{-4}	1.1e-5	330
10^{-5}	5.3e-7	1065
10^{-6}	4.6e-8	6403
10^{-7}	1.2e-9	10777
10^{-8}	1.0e-10	34110
10^{-9}	2.2e-12	1.1e6
10^{-10}	9.0e-10	2.1e6

Figure 3: Problem 3(a)

(b) Next we test the method on a more stiff version of the problem above, taking $\lambda = -100$ rather than -1.

Figure (5) summarizes the results over the same range of ϵ as was used in part (a). In response to the stiffness of the problem, more time steps are needed to stay within each given error tolerance. This particular method is unable to keep the absolute error under 10^{-9} and 10^{-10} when it is asked to, although it achieves these levels of accuracy when ϵ is as large as 10^{-6} and 10^{-7}, respectively.

When ϵ is made large enough that $k_{acc} > k_{stab}$, the method is allowed to take a large enough time step that the approximate solution blows up temporarily until the time step is reduced appropriately. Then the accuracy improves substantially. This phenomenon is demonstrated on the right in Figure (6). When $\epsilon = 10^{-1}$ the first few approximations are much too large, but they quickly settle down as the method decreases the step size. The error then oscillates just below 10^{-1} until the final time is reached. If ϵ is taken to be too large then k_max becomes larger than $T - t_0$ and so the method takes only two steps (an initial step of 10^{-4} then another of length $T - t_0 - 10^{-4}$) to obtain an approximation at the final time.

Problem 3 continued on next page...

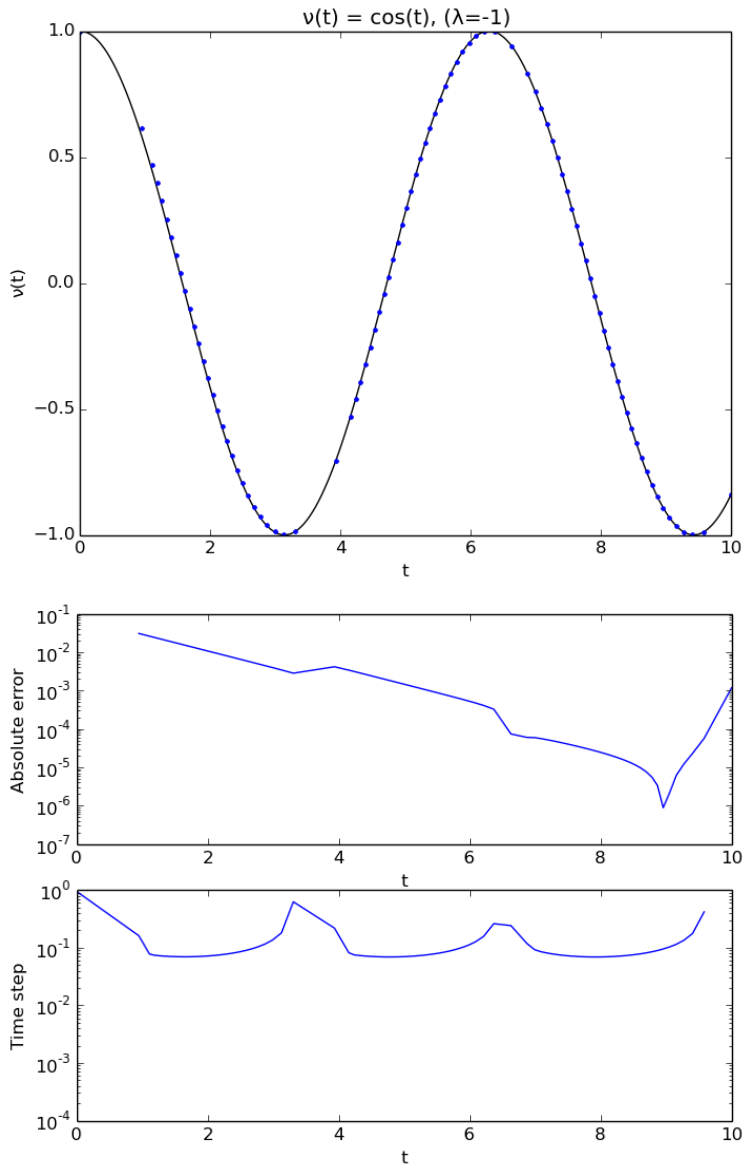


Figure 4: Plot, error, and time step size for 3(a), $\epsilon = 10^{-3}$

(c) Finally we apply the Bogacki-Shampine method to the above differential equation with exact solution

$$v(t) = \cos(t) + \exp\left(-80(t-2)^2\right).$$

We found that we needed to take either k_{max} or ϵ to be slightly smaller than was specified in the assignment to obtain plots that look similar to those on the handout. Taking $\epsilon = 10^{-3}$ rather than 10^{-2} gives the approximation, error, and time step sizes shown on the left of Figure (8) and taking $k_{max} = \epsilon^{\frac{1}{3}}$ instead of $10(\epsilon)^{\frac{1}{3}}$ gives those on the right. Both approaches show k being decreased at the time when the extra gaussian term causes an abrupt change in the solution behavior. Notice how large the second time step is for the larger choice of k_{max}. In fact, if we choose $\epsilon = 10^{-2}$ along with the larger k_{max}, we get $k_{max} = 10(\epsilon)^{\frac{1}{3}} \approx 2.15$ so when the second time step is taken to be k_{max} (as it was observed to do in every instance tested), the approximate solution is computed at a point past the rapidly varying part of the exact solution. The adaptive method takes too large a time step and

ϵ	Absolute Error	Steps taken
10^{-2}	1.8e-3	710
10^{-3}	3.2e-6	1038
10^{-4}	3.4e-7	1968
10^{-5}	3.9e-8	4085
10^{-6}	4.3e-9	8623
10^{-7}	5.1-10	18678
10^{-8}	7.3-11	44637
10^{-9}	2.8e-7	1.2e6
10^{-10}	3.6e-7	1.0e6

Figure 5: Problem 3(b)

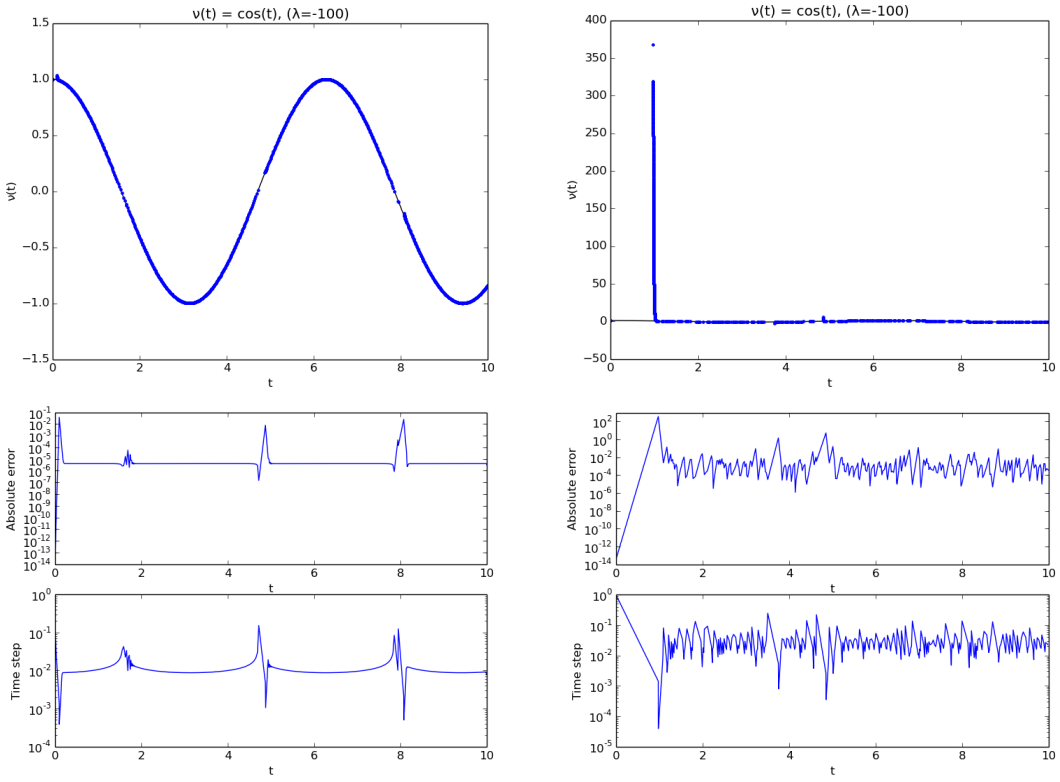


Figure 6: Plot, error, and time step size for 3(b) (Left: $\epsilon = 10^{-3}$, Right: $\epsilon = 10^{-1}$)

misses the "interesting" part of the solution entirely. It is for this reason that we compare the method's performance using two different formulas for k_{max}. Figure (7) shows the results of both versions of the B-S method. Notice that for the more restrictive definition of k_{max} the method requires roughly the same number of time steps to reach the final time and achieves slightly better accuracy for every trial. If given a choice between the two, we would use the method using the more constrained maximum time step (they both run almost instantaneously in any case).

	$k_{max} = 10\epsilon^{\frac{1}{3}}$		$k_{max} = \epsilon^{\frac{1}{3}}$	
ϵ	Absolute Error	Steps taken	Absolute Error	Steps taken
10^{-2}	4.0e-2	10	6.7e-4	63
10^{-3}	3.0e-3	164	1.5e-3	158
10^{-4}	2.2e-5	564	2.7e-6	586
10^{-5}	3.3e-6	1801	1.3e-7	1827
10^{-6}	2.5e-7	5739	3.6e-7	5715
10^{-7}	9.7e-10	18190	4.2e-10	18207
10^{-8}	6.5e-11	57513	2.3e-11	57563
10^{-9}	1.8e-12	1.8e6	1.1e-12	1.8e6
10^{-10}	5.2e-8	94885	7.8e-11	1.2e6

Figure 7: Problem 3(c)

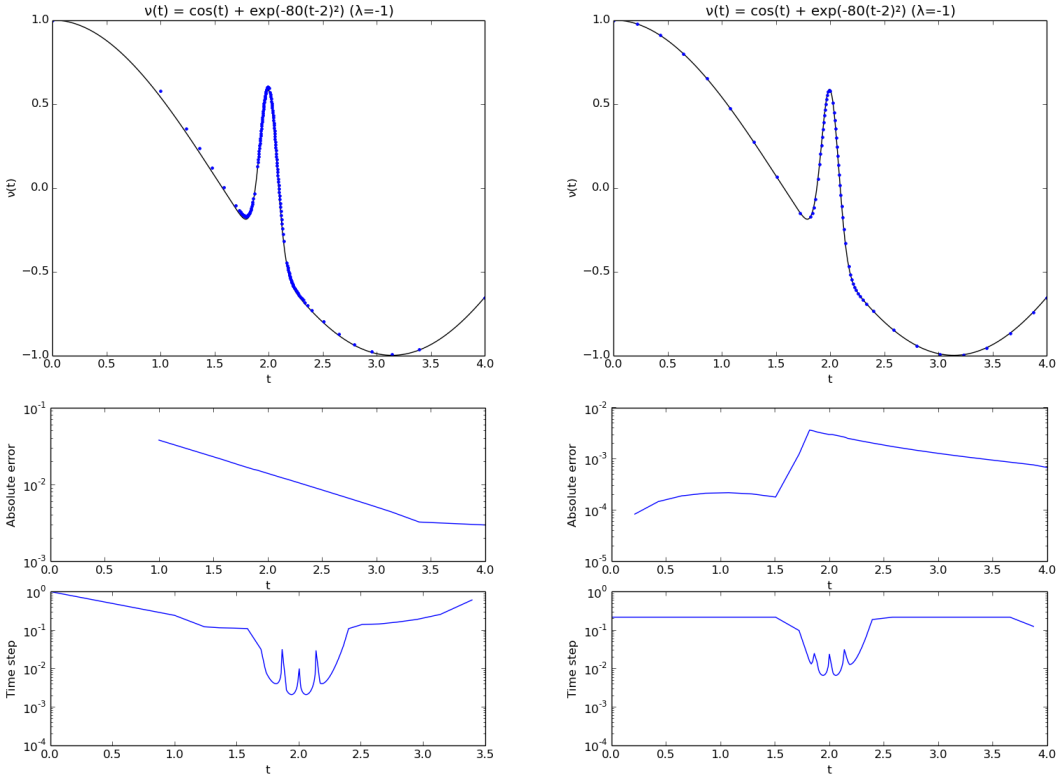


Figure 8: Plot, error, and time step size for 3(c) (Left: $\epsilon = 10^{-3}$, $k_{max} = 10\epsilon^{\frac{1}{3}}$, Right: $\epsilon = 10^{-2}$, $k_{max} = \epsilon^{\frac{1}{3}}$)

Amath 586 HW 3 Julia code

April 30, 2015

1 AMath 586 Homework 3

1.0.1 Brian de Silva

1.0.2 Problem 3 - Bogacki-Shampine Runge-Kutta method

In [58]:
```
# Step 1: Single timestep with B-S method

# Perform one step of B-S method with timestep k (u'(t) = f(t,u), u(t_0) = u_0)
function singleBSStep(f,u_0,t_0,k)
    k_1 = f(t_0,u_0);
    k_2 = f(t_0 + k/2, u_0 + (k*k_1)/2);
    k_3 = f(t_0 + (3/4)*k, u_0 + (3*k*k_2)/4);
    u_hat = u_0 + (2*k*k_1)/9 + (k*k_2)/3 + (4*k*k_3)/9;           # Third order approximation
    k_4 = f(t_0 + k, u_hat);
    u = u_0 + (7*k*k_1)/24 + (k*k_2)/4 + (k*k_3)/3 + (k*k_4)/8; # Second order approximation
    return (u_hat,u)
end

# This f corresponds to u' = lambdau, which has exact solution u(t) = exp(lambdat),
# for IC u(0) = 1.
lambda = -3;
f = (t,u) -> lambda * u;

# Print ratio between absolute errors of the two approximations as stepsize is halved
(u_hat,u) = singleBSStep(f,1,0,1/2);
u_hat_prev_err = abs(u_hat - exp(lambda/2));
u_prev_err = abs(u - exp(lambda/2));

err2ord = Float64[];
err3ord = Float64[];
for j=2:10
    k = 2.0^(-j);
    (u_hat,u) = singleBSStep(f,1,0,k);
    u_hat_err = abs(u_hat - exp(k * lambda));
    u_err = abs(u - exp(k * lambda));
    push!(err2ord,u_prev_err / u_err);
    push!(err3ord,u_hat_prev_err / u_hat_err);
    u_hat_prev_err = u_hat_err;
    u_prev_err = u_err;
end
println("Third order method error ratio as stepsize is halved:")
for ratio in err3ord
```

```
        println(ratio)
    end

    println("\nSecond order method error ratio as stepsize is halved:")
    for ratio in err2ord
        println(ratio)
    end
```

```
Third order method error ratio as stepsize is halved:
14.05454710800195
14.923526422174445
15.432261325423378
15.70824783875262
15.852085435630233
15.925524350789642
15.96263042156919
15.981287767341858
15.99051823972206

Second order method error ratio as stepsize is halved:
9.208203329622291
9.381371574197148
9.010134912108915
8.613245882907943
8.338703950904382
8.178138161436115
8.091371860291467
8.04627568271947
8.02328670442059
```

This output suggests that the "second order" part of the method is in fact third order and the "third order" part is fourth order if we only take one time step from the initial time, at least on the test problem $u' = \lambda u$.

In [59]:
```
# Step 2: Fixed time step B-S

# Computes third order accurate approximation to solution of
# u'(t) = f(t,u), with IC u(t_0) = u_0 up to time T using N timesteps,
# i.e. t_n = t_0 + k*n, UN ~ u(t_n), k = T/N, UN ~ u(T).
function fixedStepBS(f,u_0,t_0,N,T)
    k = T/N
    U = zeros(N+1);
    U[1] = u_0;

    # Prime k4 for use in the first iteration
    k4 = f(t_0,U[1]);
    for n=2:N+1
        t_n = t_0 + k*(n-2);
        # Reuse final stage from prev. iteration
        k1 = k4;
        k2 = f(t_n + k/2, U[n-1] + (k*k1)/2);
        k3 = f(t_n + (3/4)*k, U[n-1] + (3*k*k2)/4);
        # Third order approximation
        U[n] = U[n-1] + (2*k*k1)/9 + (k*k2)/3 + (4*k*k3)/9;
        k4 = f(t_n + k, U[n]);
```

```
            # Second order approximation
            U_2ord = U[n-1] + (7*k*k1)/24 + (k*k2)/4 + (k*k3)/3 + (k*k4)/8;
        end
        return U
    end


    # This f corresponds to u' = lambdau, which has exact solution u(t) = exp(lambdat),
    # for IC u(0) = 1.
    lambda = -3;
    f = (t,u) -> lambda * u;


    # Print ratio between absolute error at T as the stepsize is halved
    T = 10
    U = fixedStepBS(f,1,0,10,T);
    U_prev_err = abs(U[end] - exp(lambda*T));
    fixedsteperr = Float64[];
    for j=2:10
        N = convert(Int64,2.0^(j) * 5);
        U = fixedStepBS(f,1,0,N,T);
        U_err = abs(U[end] - exp(lambda*T));
        push!(fixedsteperr,U_prev_err/U_err);
        U_prev_err = U_err;
    end

    println("Error ratio as stepsize is halved:")
    for ratio in fixedsteperr
        println(ratio);
    end
```

```
Error ratio as stepsize is halved:
1.094294997157778e16
1.6010731121185087
7.319503958662112
8.951853052317007
8.589898214653301
8.302380172440731
8.151116475854781
8.075338974332153
8.03759208291063
```

This output suggests the method achieving third order accuracy on the test problem $u' = \lambda u$.

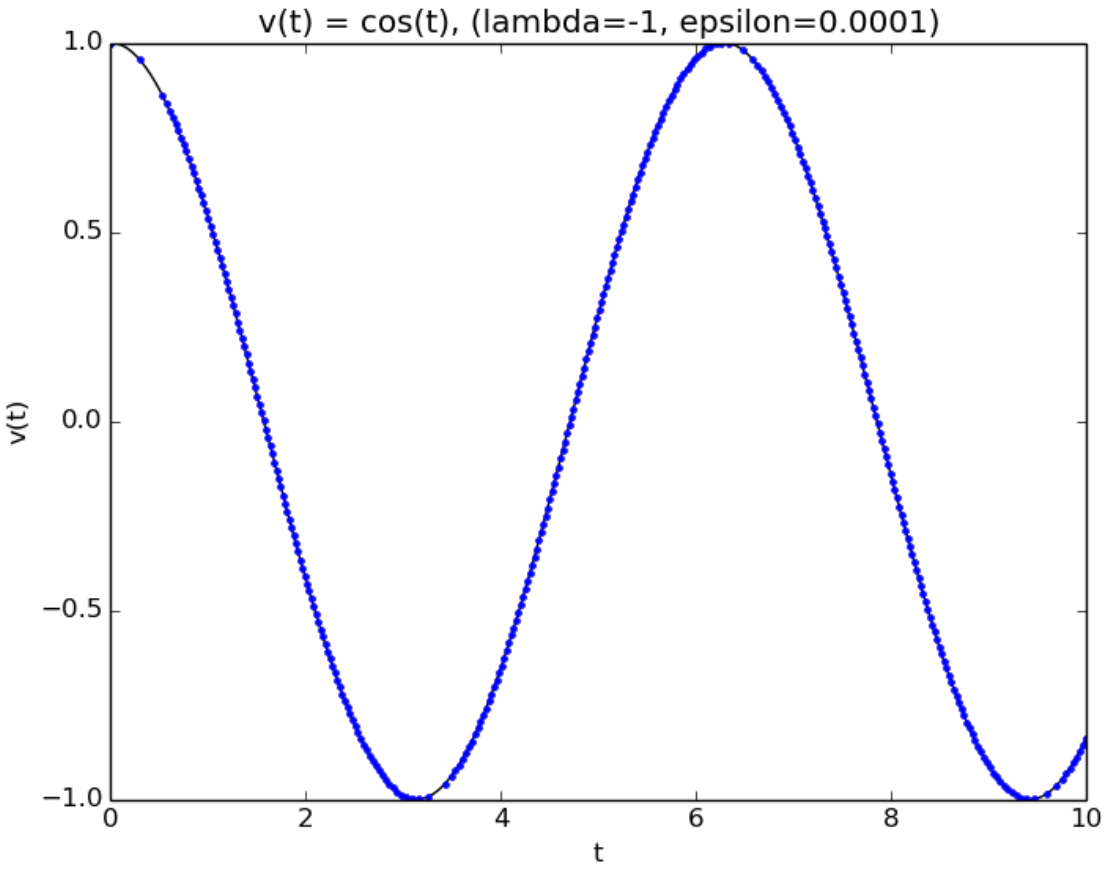
In [60]:
```
# Step 3 Adaptive time method

# Import plotting package
using PyPlot;

# Computes a (hopefully!) second order accurate approximation to solution of
# u'(t) = f(t,u), with IC u(t_0) = u_0 up to time T using adaptive timestepping (k_0 = initial
# Attempts to bound absolute error |UN - u(T)| < epsilon (using Bogacki-Shampine method)
function adaptiveStepBS(f,u_0,t_0,k_0,T,epsilon)
    k_max = 10*(epsilon)^(1/3);
    k = k_0
```

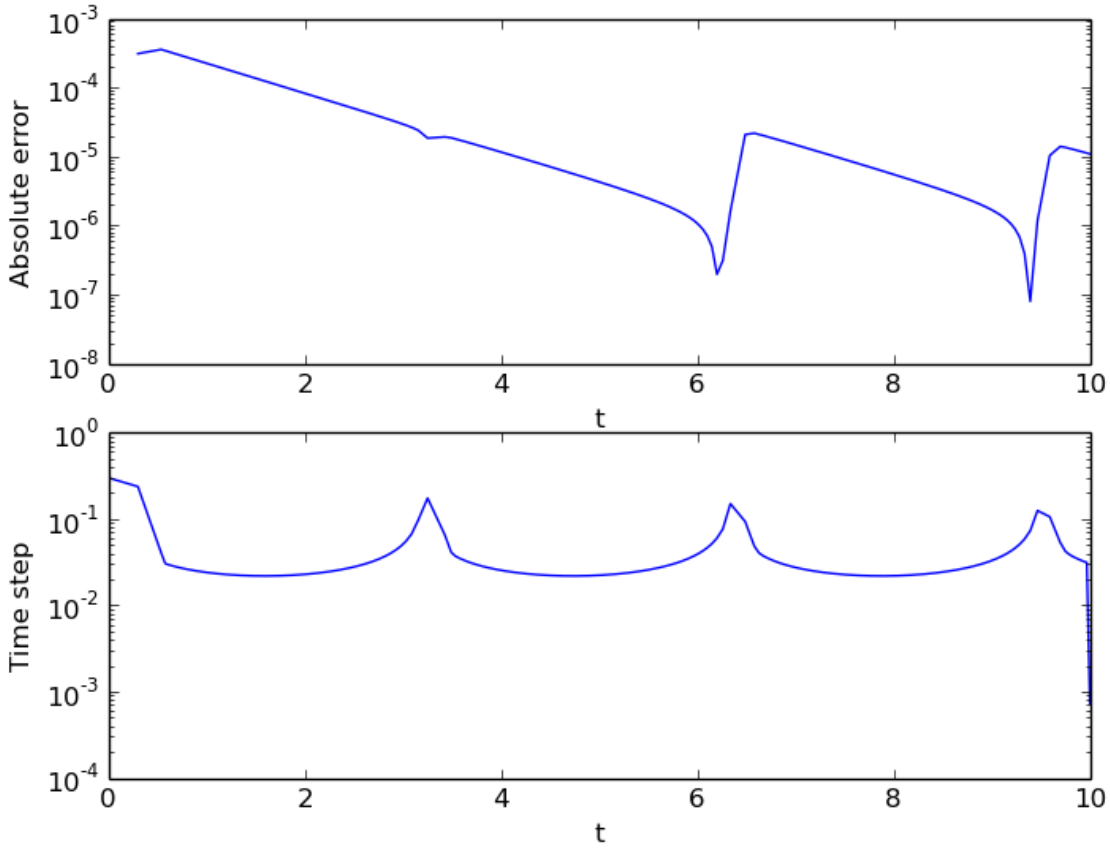
```
    U = Float64[];
    push!(U,u_0);
    tVec = Float64[];
    push!(tVec,t_0)

    # Prime k4 for use in the first iteration
    k4 = f(t_0,U[1]);
    n = 2;
    while tVec[end] < T
        t_n = tVec[n-1];
        push!(tVec,t_n + k);
        # Reuse final stage from prev. iteration
        k1 = k4;
        k2 = f(t_n + k/2, U[n-1] + (k*k1)/2);
        k3 = f(t_n + (3/4)*k, U[n-1] + (3*k*k2)/4);
        # Third order approximation
        push!(U, U[n-1] + (2*k*k1)/9 + (k*k2)/3 + (4*k*k3)/9);
        k4 = f(t_n + k, U[n]);
        # Second order approximation
        U_2ord = U[n-1] + (7*k*k1)/24 + (k*k2)/4 + (k*k3)/3 + (k*k4)/8;

        # Choose next time step
        C = abs(U[n] - U_2ord) / (k^3);
        k_new = sqrt(epsilon / (C*(T-t_0)));

        # Make sure new k isn't too large
        if k_new > k_max
            k_new = k_max;
        end

        # Make sure we don't overshoot the final time T
        if T - t_n - k < k_new
            k_new = T - t_n - k
        end
        k = k_new;

        n += 1;
    end
    return (U,tVec)
end;
```

In [69]:
```
#  Solve problems of the form u'(t) = lambda(u - v(t)) + v'(t)

# Problem 3(a): v(t) = cos(t), lambda=-1
lambda = -1;
v = t -> cos(t);
v_prime = t -> -sin(t);
T = 10;
f = (t,u) -> lambda*(u - v(t)) + v_prime(t);
epsilon = 1e-4;


# # Print LateX-friendly error and time steps used
# errVec = Float64[]
```

```
# iterVec = Int64[]
# for m=2:10
#     (U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,1/(10^m))
#     push!(errVec,abs(U[end] - v(T)));
#     push!(iterVec,size(tVec)[1]);
# end
# for k=1:size(iterVec)[1]
#     println("\£10^{-£(k+1)}\£ & £(errVec[k]) & £(iterVec[k]) \\\\ \\hline")
# end


# Print absolute error at time T and total number of time steps
(U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,epsilon);
println("Error: $(abs(U[end] - v(T)))");
println("Used $(size(tVec)[1]) time steps")

# Plot exact solution vs. approximation;
PyPlot.figure();
PyPlot.plot(linspace(0,T,10000),v(linspace(0,T,10000)),"k-");
PyPlot.plot(tVec, U, "b.");
PyPlot.xlabel("t");
PyPlot.ylabel("v(t)");
PyPlot.title("v(t) = cos(t), (lambda=$lambda, epsilon=$epsilon)");

# Plot Error and time step size
PyPlot.figure();
PyPlot.subplot(2,1,1)
PyPlot.semilogy(tVec, abs(v(tVec) - U), "b-")
PyPlot.ylabel("Absolute error")
PyPlot.xlabel("t")
PyPlot.subplot(2,1,2)
PyPlot.semilogy(tVec[1:end-1],tVec[2:end] - tVec[1:end-1],"b-");
PyPlot.ylabel("Time step")
PyPlot.xlabel("t");
```

```
Error: 1.0902762976439107e-5
Used 330 time steps
```

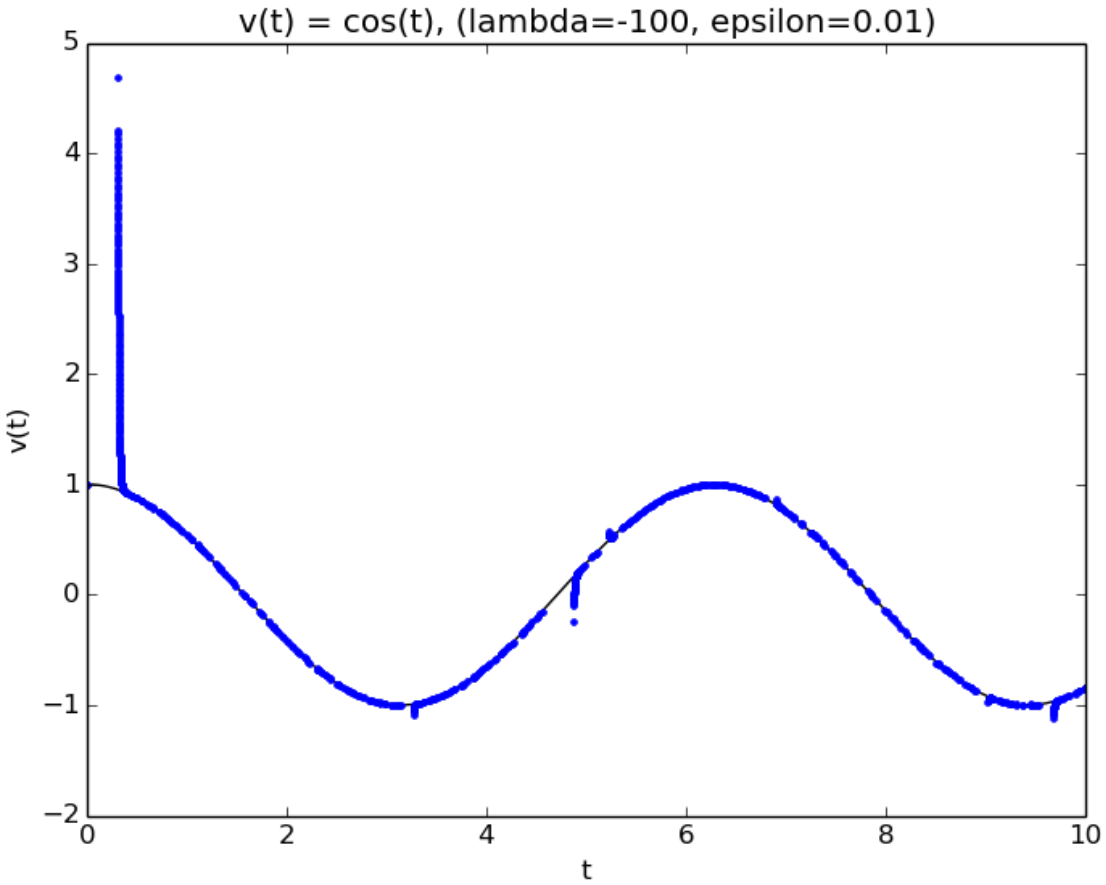

v(t) = cos(t), (lambda=-1, epsilon=0.0001)
1.0
0.5
0.0
−0.5
−1.0
v(t)
0
2
4
6
8
10
t

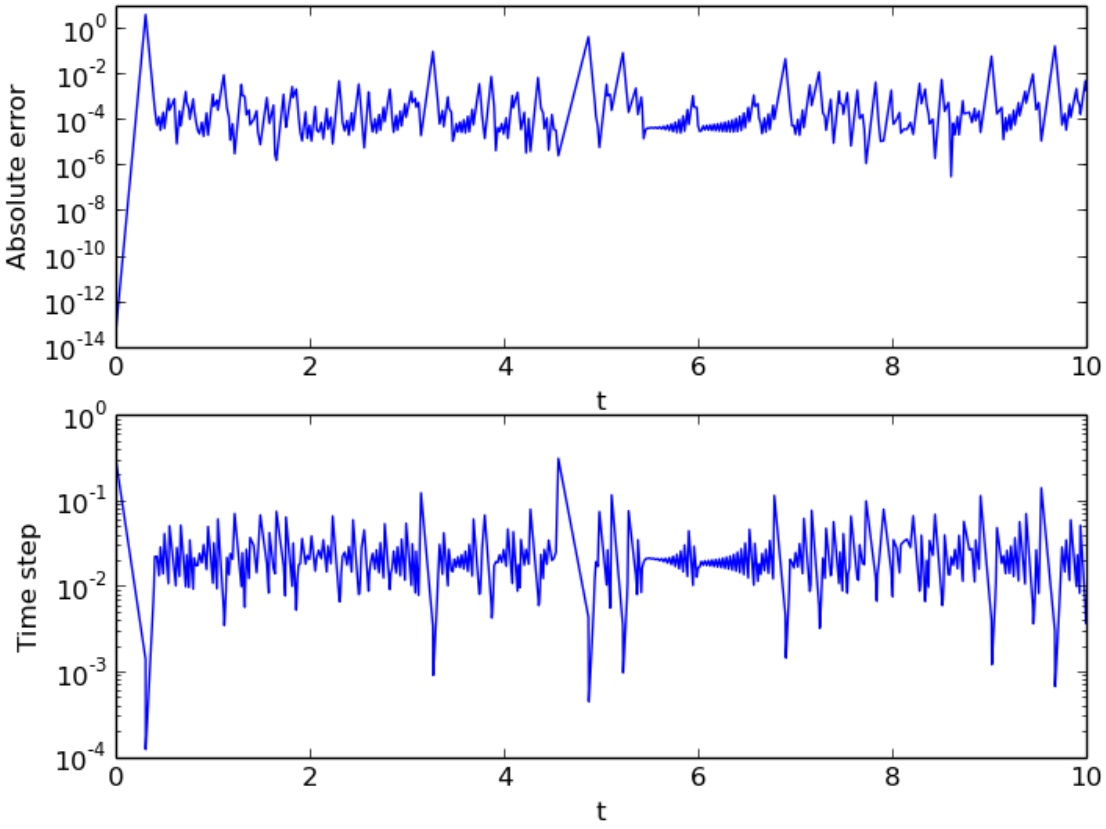


In [67]:
```
# Problem 3(b): v(t) = cos(t), lambda<-1
lambda = -100;
v = t -> cos(t);
v_prime = t -> -sin(t);
T = 10;
f = (t,u) -> lambda*(u - v(t)) + v_prime(t);
epsilon = 1e-2;

# # Print LateX-friendly error and time steps used
# errVec = Float64[]
# iterVec = Int64[]
# for m=2:10
#     (U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,1/(10^m))
#     push!(errVec,abs(U[end] - v(T)));
#     push!(iterVec,size(tVec)[1]);
# end
# for k=1:size(iterVec)[1]
#     println("\£10^{-£(k+1)}\£ & £(errVec[k]) & £(iterVec[k]) \\\\ \\hline")
# end


# Print absolute error at time T and total number of time steps
(U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,epsilon);
```

```
println("Error: $(abs(U[end] - v(T)))");
println("Used $(size(tVec)[1]) time steps")

# Plot exact solution vs. approximation;
PyPlot.figure();
PyPlot.plot(linspace(0,T,10000),v(linspace(0,T,10000)),"k-");
PyPlot.plot(tVec, U, "b.");
PyPlot.xlabel("t");
PyPlot.ylabel("v(t)");
PyPlot.title("v(t) = cos(t), (lambda=$lambda, epsilon=$epsilon)");

# Plot Error and time step size
PyPlot.figure();
PyPlot.subplot(2,1,1)
PyPlot.semilogy(tVec, abs(v(tVec) - U), "b-")
PyPlot.ylabel("Absolute error")
PyPlot.xlabel("t")
PyPlot.subplot(2,1,2)
PyPlot.semilogy(tVec[1:end-1],tVec[2:end] - tVec[1:end-1],"b-");
PyPlot.ylabel("Time step")
PyPlot.xlabel("t");
```

```
Error: 0.0017645787780279365
Used 710 time steps
```

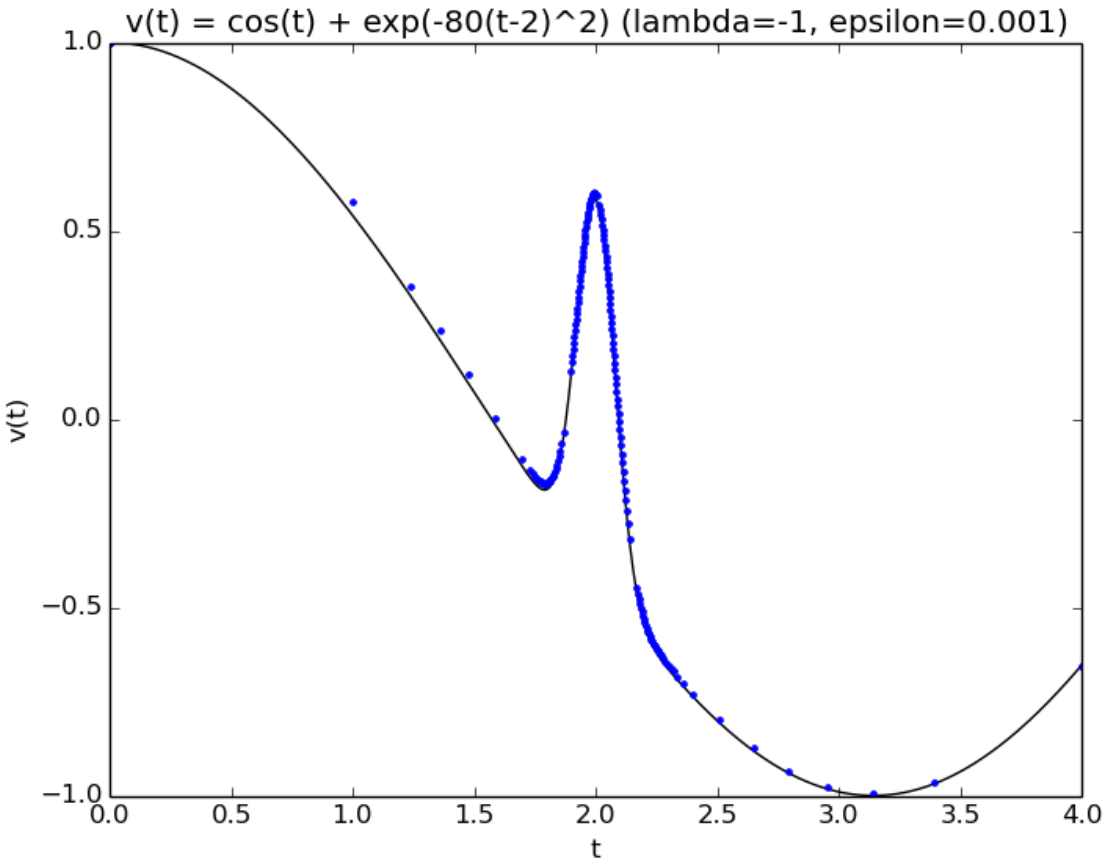


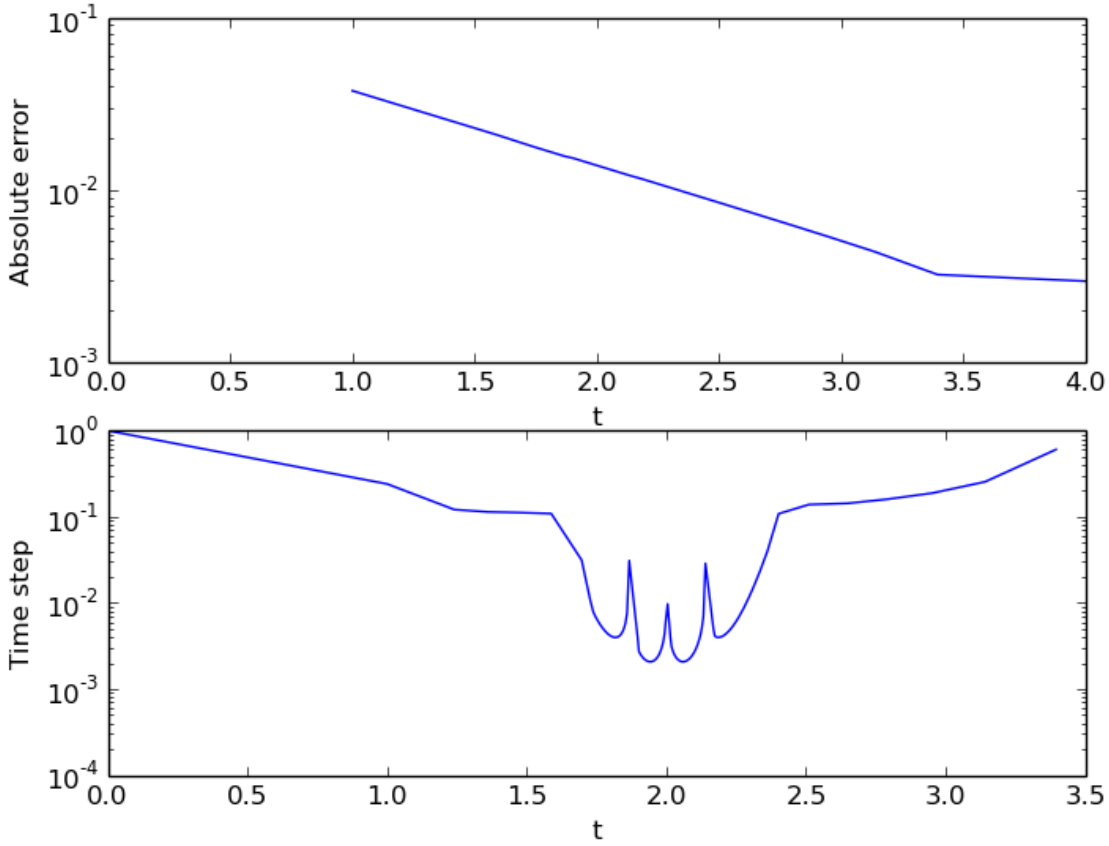


In [68]:
```
# Problem 3(c): v(t) = cos(t) + exp(-80(t-2)^2)
lambda = -1;
v = t -> cos(t) + exp(-80 * (t-2).^2);
v_prime = t-> -sin(t) - 160*(t-2)*exp(-80 * (t-2)^2);


f = (t,u) -> lambda*(u - v(t)) + v_prime(t);
T = 4;
epsilon = 1e-3;


# Print LateX-friendly error and time steps used
# errVec = Float64[]
# iterVec = Int64[]
# for m=2:10
#     (U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,1/(10^m))
#     push!(errVec,abs(U[end] - v(T)));
#     push!(iterVec,size(tVec)[1]);
# end
# for k=1:size(iterVec)[1]
#     println("\£10^{-£(k+1)}\£ & £(errVec[k]) & £(iterVec[k]) \\\\ \\hline")
# end
```

```
# Print absolute error at time T and total number of time steps
(U,tVec) = adaptiveStepBS(f,1,0,1e-4,T,epsilon);
println("Error: $(abs(U[end] - v(T)))");
println("Used $(size(tVec)[1]) time steps")

# Plot exact solution vs. approximation;
PyPlot.figure();
PyPlot.plot(linspace(0,T,10000),v(linspace(0,T,10000)),"k-");
PyPlot.plot(tVec, U, "b.");
PyPlot.xlabel("t");
PyPlot.ylabel("v(t)");
PyPlot.title("v(t) = cos(t) + exp(-80(t-2)^2) (lambda=$lambda, epsilon=$epsilon)");


# Plot Error and time step size
PyPlot.figure();
PyPlot.subplot(2,1,1)
PyPlot.semilogy(tVec, abs(v(tVec) - U), "b-")
PyPlot.ylabel("Absolute error")
PyPlot.xlabel("t")
PyPlot.subplot(2,1,2)
PyPlot.semilogy(tVec[1:end-1],tVec[2:end] - tVec[1:end-1],"b-");
PyPlot.ylabel("Time step")
PyPlot.xlabel("t");
```

```
Error: 0.0029402070388248047
Used 164 time steps
```


v(t) = cos(t) + exp(-80(t-2)^2) (lambda=-1, epsilon=0.001)
v(t)
t
1.0
0.5
0.0
−0.5
−1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Absolute error
10^{-1}
10^{-2}
10^{-3}
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
t
Time step
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
t