CNS 187 - Neural Computation
Handouts

Handed out: 10 Oct 02

This handout contains problems and solutions from last year's class. You'll find it helpful to go through gradient descent and euler integration technique.

1.1 Gradient Descent

You will soon be writing code to simulate artificial neural networks. Very often, this involves some form of parameter optimization with discrete steps, most commonly using a method called "gradient descent." The idea behind this is as follows: suppose you want to find the minimum of some multidimensional surface. Often the mathematical expression for the surface is too complicated to permit a direct algebraic solution, so you have to search for the minimum numerically. If you know the gradient of the surface (and thus the direction in which it goes down most steeply), then a seemingly sensible idea is to start with a "guess" for where the minimum is, and then improve the guess by moving it in the direction of steepest descent; when you can't improve the guess anymore, then you are at a minimum (possibly a local one, of course). We write "seemingly" because we will see in later problem sets that this is often very slow.

This problem and the next one ("Euler Integration") are intimately related, so you might want to look them over before starting in earnest; that will give you an idea of what techniques to use.

Let's first try this with a *very* simple surface, a one-dimensional bowl whose height is:

$$E = x^2/2 \tag{1}$$

(Clearly, the minimum is at $x = 0$, but pretend you don't know that right now – in future problem sets you will test this technique on surfaces you can't solve analytically, rest assured!)

1. Write a program, using `gradesc.m` provided in the MATLAB package, that implements gradient descent for this surface. Initialize x at a random position. Then at each time step,

   - find the gradient $\frac{\partial E}{\partial x}$ at the current position.
   - take a step in a direction negative to the gradient, such that

   $$x_{n+1} = x_n - \eta \frac{\partial E}{\partial x} \tag{2}$$

η is called the "time step". Try running this program for various values of η. Do not hand in Matlab code unless the problem set specifically asks for it.

Hand in three plots of x versus n (for different values of η) that show qualitatively different behaviors. From your numerical experiments, for what range of values of η does this program converge to the correct result?

2. Analyze the procedure to give a theoretical prediction for the values of η which would lead to such results. Does it coincide with your numerical values?

3. Now let's get just a little fancier. Define a new surface by

$$E(x,y) = \frac{1}{25}(13x^2 + 24xy + 13y^2) \tag{3}$$

Plot the surface in MATLAB (but don't hand in this plot). Where is the global minimum?

4. Simulate gradient descent for it, so that now

$$\begin{aligned} x_{n+1} &= x_n - \eta \frac{\partial E}{\partial x} &(4)\\ y_{n+1} &= y_n - \eta \frac{\partial E}{\partial y} &(5) \end{aligned}$$

Hand in three plots, each for a different η, to show qualitatively different behaviors.

5. Repeat the numerics that you did for $x^2/2$, so as to find the values of η for which the procedure doesn't converge. Find a theoretical prediction for this value. (Hint: find x_n and y_n as a function of n.)

Since we've gone this far, we might as well go a little further and consider a general surface $E(\vec{x})$ where $\vec{x}$ is a column vector with N components $(x_1,\ x_2,\ \ldots x_N)$. (The problem you just did would be an example of such a surface, with $\vec{x}$ having 2 components.) Define a vector $\frac{\partial E}{\partial \vec{x}}$ whose i-th component is

$$\left(\frac{\partial E}{\partial \vec{x}}\right)_i = \frac{\partial E}{\partial x_i} \tag{6}$$

This vector is just a shorthand way of writing the gradient. The gradient descent procedure would now be

$$\vec{x}_{n+1} = \vec{x}_n - \eta \frac{\partial E}{\partial \vec{x}} \tag{7}$$

Let

$$H_{ij} \equiv \frac{\partial^2 E}{\partial x_i \partial x_j} \tag{8}$$

The matrix $\vec{H} = [H_{ij}]$ is called the *Hessian.*

The Taylor series expansion for E about some point $\vec{x}_0$ is

$$E(\vec{x}) = E(\vec{x}_0) + \left(\frac{\partial E}{\partial \vec{x}}\right)^T (\vec{x} - \vec{x}_0) + \frac{1}{2}(\vec{x} - \vec{x}_0)^T \vec{H}\,(\vec{x} - \vec{x}_0) + \text{higher order terms} \tag{9}$$

(with both the gradient and the Hessian evaluated at $\vec{x}_0$.)

6. If $\vec{x}_{min}$ is a local minimum, and we are close enough to it that we can ignore the higher order terms in the Taylor expansion, predict the maximum value of η for convergence in the gradient descent procedure, in terms of the eigenvalues of the Hessian $\vec{H}$ at $\vec{x}_{min}$. You can assume, without loss of generality, that $\vec{x}_{min} = \vec{0}$; this makes manipulating the equations a little more convenient. (Reminder: what's the value of the gradient vector at the minimum?)

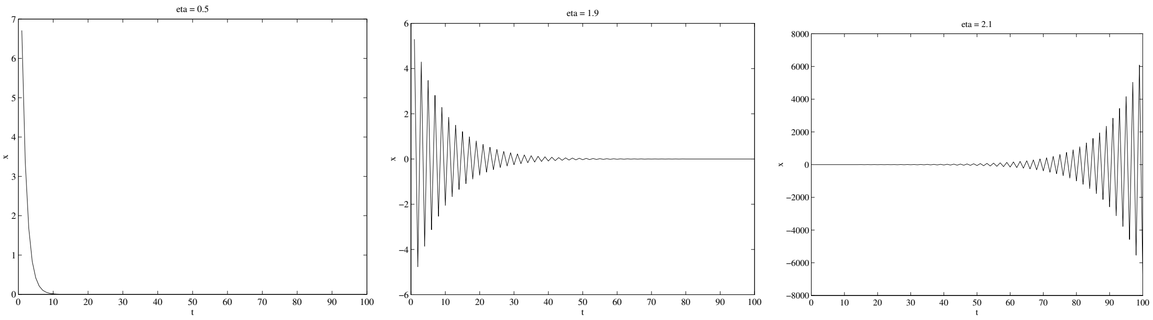
7. (optional) Try this for fun: suppose we are at some point $\vec{x}_n$, and that $\vec{x}_n$ is indeed close enough to the minimum that higher order terms in the Taylor expansion can be ignored, and that we know what $\vec{H}$ is at $\vec{x}_n$. Assume that $\vec{H}$ is invertible. Can you find a closed-form expression for how much we should add to $\vec{x}_n$ so as to jump *directly* to the minimum, in a single bound?

1.2 Gradient Descent Solution

1. To complete the MATLAB program for performing gradient descent on the surface $E = x^2/2$, we can add the lines

```
M = -eye(1);                       % -dE/dx = -1 * x
A = eye(xdimensions) + eta * M;    % x(n+1) = (1 - eta) * x(n)
for n=1:timesteps-1
  xhistory(:,n+1) = A * xhistory(:,n);
end
```

For $\eta = 0.5$, 1.9, and 2.1, we see three different types of behavior: steady approach to 0, oscillatory approach to 0, and oscillatory divergence to ∞.



2. Empirically, the direct convergence occurs for $0 < \eta \leq 1$, the oscillatory convergence occurs for $1 < \eta \leq 2$, and the oscillatory divergence occurs for $\eta > 2$. This result can be derived analytically by noting that our equation is $x_{n+1} = (1 - \eta)x_n$ and hence $x_n = (1 - \eta)^n x_0$; so if $|1 - \eta| < 1$ then $x_n \to 0$ (oscillating depending upon the sign of $(1 - \eta)$), else $x_n \to \pm\infty$.

3. Now let's move to the two dimensional surface. $E(0, 0) = 0$. Is this the global minimum? By examining the gradient

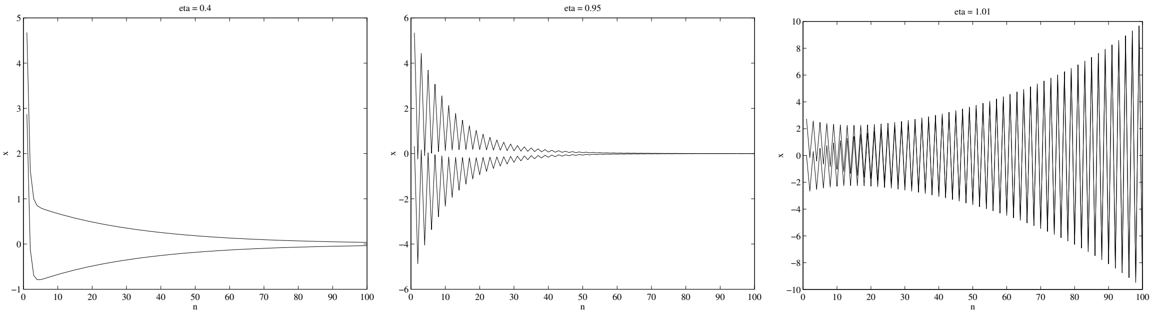
$$\nabla E = \left(\frac{\partial E}{\partial x}, \frac{\partial E}{\partial y} \right) = \left(\frac{1}{25}(26x + 24y), \frac{1}{25}(24x + 26y) \right)$$

we see that the only place where both $\frac{\partial E}{\partial x} = 0$ and $\frac{\partial E}{\partial y} = 0$ is $(0, 0)$.

The last four lines of the MATLAB program for doing gradient descent on E are now:

```
M = -[26/25 24/25; 24/25 26/25];    % -gradE = M * (x,y)
A = eye(xdimensions) + eta * M;     % x(n+1) = (I - eta * M) * x(n)
for n=1:timesteps-1
  xhistory(:,n+1) = A * xhistory(:,n);
end
```

4. Again, we see three different types of behavior: steady approach to the minimum ($\eta = 0.4$ is shown), oscillatory approach to the minimum ($\eta = 0.95$ is shown), and oscillatory divergence ($\eta = 1.01$ is shown).



5. Empirically, the direct convergence occurs for $0 < \eta \leq 0.5$, the oscillatory convergence occurs for $0.5 < \eta \leq 1$, and the oscillatory divergence occurs for $\eta > 1$. To understand the convergence conditions analytically, it is convenient to write the gradient as

$$\nabla E = \begin{pmatrix} x+y \\ x+y \end{pmatrix} + \frac{1}{25} \begin{pmatrix} x-y \\ y-x \end{pmatrix}.$$

This suggests a change of variables, where $u = x + y$ and $v = x - y$. Now the difference equations for the algorithm become:

$$\begin{aligned} x_{n+1} &= x_n - \eta \frac{\partial E}{\partial x} = x_n - \eta \left(x_n + y_n + \frac{x_n - y_n}{25} \right) \\ y_{n+1} &= y_n - \eta \frac{\partial E}{\partial y} = y_n - \eta \left(x_n + y_n - \frac{x_n - y_n}{25} \right) \end{aligned}$$

and thus

$$\begin{aligned} u_{n+1} = x_{n+1} + y_{n+1} &= x_n + y_n - 2\eta \left(x_n + y_n \right) = (1 - 2\eta) u_n \\ v_{n+1} = x_{n+1} - y_{n+1} &= x_n - y_n - \eta \left(\tfrac{2(x_n - y_n)}{25} \right) = (1 - \tfrac{2}{25}\eta) v_n \end{aligned}$$

By induction, $u_n = (1 - 2\eta)^n u_0$ and $v_n = (1 - \frac{2}{25}\eta)^n v_0$. Amazingly, this change of variables has decoupled the dynamics along the two axes u and v, making their behavior obvious. Interestingly, this gives us a richer understanding of convergence and divergence conditions: when $0 < \eta \leq 0.5$, the system converges to the origin; when $0.5 < \eta < 1$,

v still converges directly to 0, while u spirals in; when $1 < \eta < 12.5$, v still converges directly to 0, but u diverges (thus both x and y diverge – and remain exactly equal); when $12.5 < \eta < 25$, v spirals in and u diverges; and when $25 < \eta$, both u and v diverge.

6. Finally, we are prepared to deal with the multivariate version of gradient descent. Near the local minimum $\vec{x}_{min}$, we can approximate $E(\vec{x})$ as:

$$E(\vec{x}) = E(\vec{x}_{min}) \; + \; \vec{D}^T(\vec{x} - \vec{x}_{min}) \; + \; \frac{1}{2}(\vec{x} - \vec{x}_{min})^T \, \vec{H} \, (\vec{x} - \vec{x}_{min})$$

where $\vec{D}$ is the gradient of E evaluated at $\vec{x}_{min}$, and $\vec{H}$ is the Hessian of E evaluated at $\vec{x}_{min}$. Noting that $\vec{D} = \vec{0}$, since $\vec{x}_{min}$ is a local minimum, and assuming without loss of generality that $\vec{x}_{min} = \vec{0}$, we write

$$E(\vec{x}) = E(\vec{0}) + \frac{1}{2}\vec{x}^T \, \vec{H}\vec{x}.$$

Now,

$$\frac{\partial E}{\partial \vec{x}}(\vec{x}) = \vec{H}\vec{x}.$$

(If you're not familiar with derivatives of matrix expressions, write it out explicitly as sum and then take derivatives.) Our gradient descent algorithm will take steps

$$\vec{x}_{n+1} = \vec{x}_n - \eta\frac{\partial E}{\partial \vec{x}} = \vec{x}_n - \eta\vec{H}\vec{x}_n = (\vec{I} - \eta\vec{H})\vec{x}_n = (\vec{I} - \eta\vec{H})^n\vec{x}_1.$$

As before, we have an explicit formula for $\vec{x}_n$, and we want to know for what values of η the expression stays bounded. We saw in the two dimensional example that this is easy to see after the right change of variables. In this case, the right rotation is the eigenbasis of $\vec{I} - \eta\vec{H}$, because it diagonalizes the matrix. (We will assume that the matrix has all N independent eigenvectors.) Let $\vec{\Lambda}$ be the diagonal matrix of eigenvalues λ_i, and let $\vec{R}$ be the corresponding matrix whose columns are eigenvectors, so that

$$\vec{I} - \eta\vec{H} = \vec{R}\vec{\Lambda}\vec{R}^{-1}.$$

Therefore,

$$(\vec{I} - \eta\vec{H})^n = \vec{R}\vec{\Lambda}^n\vec{R}^{-1},$$

where Λ^n is simply a diagonal matrix whose entries are λ_i^n. If all $\lambda_i < 1$, then the algorithm converges to the minimum $\vec{x}_{min}$; otherwise the $\vec{x}_n$ diverges.

How can we easily determine the eigenvalues of $\vec{I} - \eta\vec{H}$ for different values of η? Note that they are related to the eigenvalues γ_i of $\vec{H}$ as follows: if

$$\left(\vec{I} - \eta\vec{H}\right)\vec{e}_i = \lambda_i\vec{e}_i$$

then

$$\vec{H}\vec{e}_i = \frac{1 - \lambda_i}{\eta}\vec{e}_i,$$

and therefore $\gamma_i = \frac{1-\lambda_i}{\eta}$ and $\lambda_i = 1 - \eta\gamma_i$.

In summary, the algorithm converges to the local minimum if for all eigenvalues γ_i of the Hessian at the minimum, $|1 - \eta\gamma_i| < 1$.

7. Just for fun: Again assume that we are near enough to a local minimum $\vec{x}_{min}$ that we can approximate E as

$$E(\vec{x}) = E(\vec{x}_{min}) \; + \; \frac{1}{2}(\vec{x} - \vec{x}_{min})^T \, \vec{H} \, (\vec{x} - \vec{x}_{min})$$

(where, since our surface is quadratic, $\vec{H}$ at our current location is the same as $\vec{H}$ at the minimum). Taking derivatives, we get

$$\vec{D}(\vec{x}) = \vec{H}(\vec{x} - \vec{x}_{min})$$

where $\vec{D}$ is the gradient at $\vec{x}$. Assuming $\vec{H}$ is invertible, we can multiply both sides by $\vec{H}^{-1}$ and solve for $\vec{x}_{min}$:

$$\vec{x}_{min} = \vec{x} - \vec{H}^{-1}\vec{D}.$$

1.3 Euler integration

It's time to look in more detail at the algorithm you used in the previous problem to implement gradient descent. In the case of the two dimensional surface, you might have imagined water flowing downhill, or a ball rolling downhill to the bottom. This illustrates an important theme in this class, namely, finding physical systems whose intrinsic dynamics carry out a computation. In the case of gradient descent, this way of thinking suggests a continuous version of gradient descent, where

$$\frac{\partial \vec{x}(t)}{\partial t} = -\frac{\partial E(\vec{x}(t))}{\partial \vec{x}} \tag{10}$$

In a sense, this differential equation represent the *true* dynamics we are trying to incorporate into the algorithm used in problem 1.1, which we now see boils down to numerically integrating the differential equation using a difference equation approximation. In the more general context of ordinary differential equations, this naive algorithm is known as *Euler's method.* It approximates any differential equation of the form

$$\frac{dy}{dt} = f(y) \tag{11}$$

with the difference equation

$$\frac{\Delta y}{\Delta t} = f(y) \tag{12}$$

which can be trivially solved for Δy,

$$\Delta y = f(y)\Delta t \tag{13}$$

Based on this difference equation, we can iterate our simulation forward one step at a time.

$$y_{t+\Delta t} = y_t + f(y_t)\Delta t \tag{14}$$

The value of Δt is known as the *step size* of the simulation. As $\Delta t \to 0$, the difference equation becomes a more and more accurate model of the real differential equation (until the effects of finite machine precision come into play).

We can convert any high–order ordinary differential equation[1] to a multidimensional first–order ordinary differential equation, and thus simulate it with Euler's method. This is done by creating a state vector

$$\vec{y} = \begin{pmatrix} y \\ y' \\ y'' \\ \vdots \end{pmatrix} \tag{15}$$

The primes denote differentiation by time. Notice that Euler's method can be applied to vectors just as easily as scalar variables:

$$\Delta\vec{y} = f(\vec{y})\Delta t \tag{16}$$

For example, the linear differential equation $y'' + y' + y = 0$ would be written in vector form as

$$\vec{y}' = \vec{M}\vec{y}$$
$$\begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}\begin{pmatrix} y \\ y' \end{pmatrix} \tag{17}$$

1. Show that in the general linear case $\vec{y}' = \vec{M}\vec{y}$, where $\vec{y} = (y_1, y_2, y_3, \ldots, y_N)^T$,
$$\vec{y}_{n\Delta t} = (\vec{I} + \vec{M}\Delta t)^n \vec{y}_0 \tag{18}$$

2. Convert the following differential equations into the state vector form above:
   a) $y'' + y = 0,\quad y(0) = 1,\ y'(0) = 0$
   b) $y'' + 0.1y' + y = 0,\quad y(0) = 1,\ y'(0) = 0$
   c) $y'' + 101y' + 100y = 0\quad y(0) = 1,\ y'(0) = -1$

   Find the closed form solutions for these equations using the given initial conditions.

3. Using the program `eulerint.m` provided in the MATLAB package, write a program which can simulate the differential equations above using the given initial conditions. Plot $y(t)$ versus $y'(t)$ for each equation, using a reasonable value for Δt.

4. For each equation, what is the largest step size ($\Delta t_{\max}$) for which you get non-divergent behavior? Determine this numerically by trying different values of Δt.

   There are two related but distinct issues to consider here: instability (i.e. unboundedness as time increases) and accumulated error during stable operation. The second issue is much more involved than the first, and the relationship between discrete and continuous dynamical systems has evolved into a branch of applied mathematics in its own right. For the purposes of this homework set, as will become clear when you have solved 1), 2), and 3) analytically, we are only concerned here with the first issue: instability.

5. Using the same techniques as in problem 1.1, analyze your procedure to obtain the theoretical maximum stepsize, $\Delta t_{\max}$, for which the Euler method converges. That is, we want to know the $\Delta t_{\max}$ for which the solution doesn't explode (go infinite) as $t \to \infty$. How do the analytically obtained values compare with the empirical ones obtained from simulation?

[1] or even a multidimensional higher–order ODE

For more thoughts on accumulated error, consult *Numerical Recipes in C*, by W. Press et al. (Cambridge: Cambridge University Press).

1.4 Euler integration solution

1. In the general linear case $\vec{y}' = \vec{M}\vec{y}$, Euler's method uses the iteration

$$\vec{y}_{t+\Delta t} = \vec{y}_t + \Delta t \vec{M}\vec{y}_t = (\vec{I} + \vec{M}\Delta t)\vec{y}_t$$

   and so by induction,

$$\vec{y}_{n\Delta t} = (\vec{I} + \vec{M}\Delta t)^n \vec{y}_0.$$

   To solve a linear second order ODE with constant coefficients in closed form, one simply proposes the exponential $e^{\alpha t}$ as a solution. Plugging this solution into the equation yields a quadratic in α which in general will have two roots. (Actually, we may end up with a single double root in which case our strategy is to propose a second solution of the form $te^{\alpha t}$ which will work.)

   The general solution of the ODE is then simply given by:

$$y = C_1 e^{\alpha_1 t} + C_2 e^{\alpha_2 t} + y_P$$

   where C_1 and C_2 are complex constants determined by the initial conditions, and y_P is any particular solution to the non-homogeneous part of the equation. (If the right hand side is zero as in all our cases, this term drops out.)

2. For the given examples, the state vector form and closed form solutions obtained by following the methods explained above (making use of $e^{it} = cos(t) + isin(t)$) are given below:

   (a) Original equation: $y'' + y = 0$

$$\begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix}$$

   Solution given initial conditions: $y = cos(t)$.

   (b) Original equation: $y'' + 0.1y' + y = 0$

$$\begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & -0.1 \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix}$$

   Solution given initial conditions:

$$y = e^{-0.05t} cos(t\sqrt{1-(0.05)^2}) + \frac{0.05}{\sqrt{1-0.05^2}} e^{-0.05t} sin(t\sqrt{1-(0.05)^2}).$$

   (c) Original equation: $y'' + 101y' + 100y = 0$

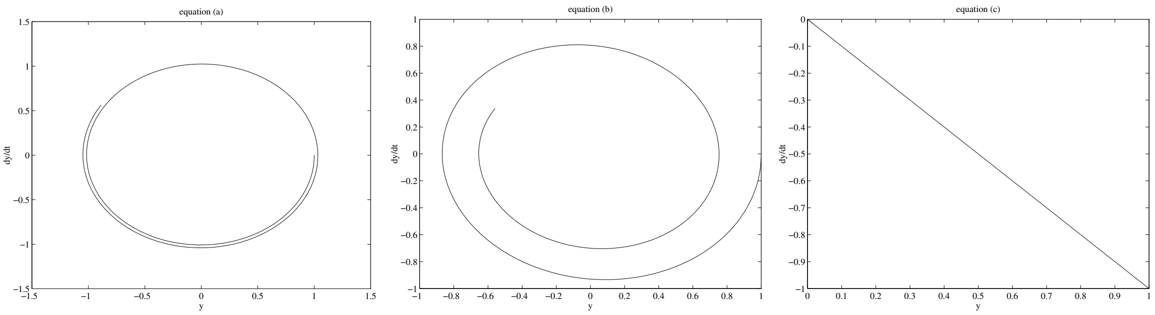
$$\begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -100 & -101 \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix}$$

   Solution given initial conditions: $y = e^{-t}$.

3. The MATLAB program is completed with the following lines (for equation (a); the other equations are similar):

```
M = [0 1; -1 0];
A = eye(ydimensions) + deltat * M;
for t=1:timesteps-1
  yhistory(:,t+1) = A * yhistory(:,t);
end
```

Plots of $\vec{y}(t)$ vs $\vec{y}'(t)$ from the simulation, with $\Delta t = 0.01$, look similar, but not identical, to the desired analytical solutions. In particular, (a) should give rise to a circle, but in the simulations an outward spiral is always produced.



4. Empirically, the greatest values of Δt which were found to preserve stability for the three equations were:

$$(a)\ \Delta t_{max} = 0.0\ (b)\ \Delta t_{max} = 0.1\ (c)\ \Delta t_{max} = 0.02$$

Notice that for the first example, the maximum limit is zero. This means that the Euler method is not powerful enough to accurately simulate this equation no matter how small the step size is. In other words, if we let the simulation run for long enough, this example will always blow up. For other examples, values of Δt_{max} near the limit (but slightly over it) may appear to be stable for a very long time before diverging, but they will eventually diverge.

5. To obtain theoretical conditions for convergence, we can apply the same mathematics that we worked our for our gradient descent algorithm. In particular, since

$$\vec{y}_{n\Delta t} = (\vec{I} + \vec{M}\Delta t)^n \vec{y}_0.$$

where $\vec{M}$ is a constant matrix, the solution will converge so long as all the eigenvalues λ_i of $\vec{M}$ satisfy $|1 + \Delta t \lambda_i| < 1$. (Note, of course, that in some equations it is *correct* for the solution to not converge, for example (a) (which shouldn't diverge either).)

We apply this analysis in turn to the three equations:
(a) The eigenvalues of M are $\pm i$. $\Delta t_{max} = 0$ since $|1 \pm i\Delta t| > 1$.
(b) The eigenvalues of M are $-.05 \pm i\sqrt{1 - (0.05)^2}$, and $\Delta t_{max} = 0.1$.
(c) The eigenvalues of M are -100 and -1, and $\Delta t_{max} = 0.02$

1.5 NUMERICAL INTEGRATION

This section contains no questions for you to answer, merely some additional notes on integrating ODEs. For more detail, consult chapter 8 of *Elementary Differential Equations and Boundary Value Problems* by W.E. Boyce and R.C. DiPrima. For even more detail, see chapter 7 of *Introduction to Numerical Analysis* by J. Stoer and R. Bulirsh.

Consider the initial value problem

$$y' = f(y), y(x_0) = y_0, \tag{19}$$

which we will denote IVP. This is just a single first order equation, but all the techniques discussed here apply to systems (where y is a vector) as well. Let the solution of this ODE be

$$y_n = \phi(x_n) \tag{20}$$

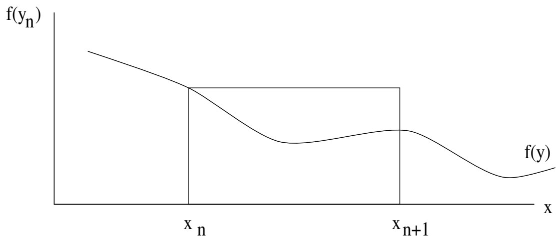
We can integrate IVP from one step to the next and obtain

$$\phi(x_{n+1}) = \phi(x_n) + \int_{x_n}^{x_{n+1}} f(\phi(x))dx \tag{21}$$

If we knew what the value of the integral was, we would have a solution for ϕ. Since we don't, we have to resort to approximation. The Euler method makes the simplest possible approximation:

$$\int_{x_n}^{x_{n+1}} f(\phi(x))dx \approx hf(y_n), \tag{22}$$

where $h = x_{n+1} - x_n$ is the stepsize (denoted Δt in problem 1.1). Graphically,



One obvious improvement on this is to use a trapezoidal approximation:

$$\int_{x_n}^{x_{n+1}} f(\phi(x))dx \approx (h/2)(f(y_n) + f(y_{n+1})) \tag{23}$$

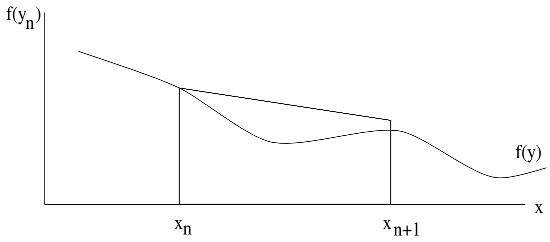
How do we know $f(y_{n+1})$ you're wondering? Once again, we make an approximation (look familiar?):

$$f(y_{n+1}) \approx f(y_n + hf(y_n)) \tag{24}$$

Putting all this together, we obtain the *Heun* formula

$$y_{n+1} = y_n + (h/2)[f(y_n) + f(y_n + hf(y_n))], \tag{25}$$

which, remarkably, has a discretization error of $O(h^3)$ as opposed to $O(h^2)$ for the Euler method. (An approximation is called n-th order if its error term is $O(h^{n+1})$. Thus, Euler is *first-order*, Heun is *second-order*.) Graphically,



Higher order methods can be obtained using similar approximations. Numerical accuracy and efficiency are always a concern when doing simulations. When solving problem 1.3, it would be helpful to write your code so that you can try various integrations techniques just by calling different functions. Even higher order methods, such as fourth order Runge-Kutta, although somewhat tricky to derive, are not difficult to implement.