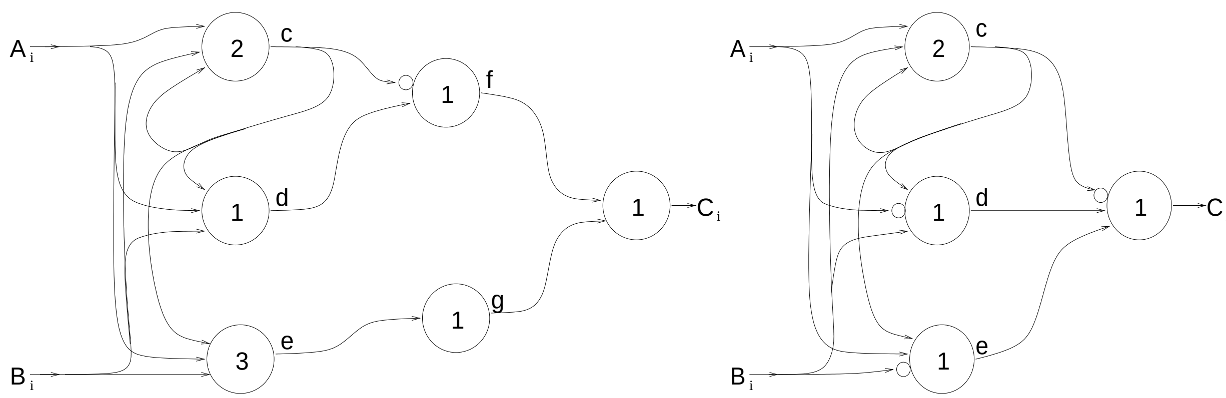
Solution Set 1, CNS 187   Fall 2000

1.1 McCulloch & Pitts neurons

1. There are many ways to solve this problem, but if we restrict ourselves to using 6 neurons, we will probably get something like the network shown in Figure 1(a). Here the two input neurons are not explicitly shown; we simply use the input lines A and B.



(a) Using absolute inhibition   (b) Using relative inhibition

Figure 1: Networks for addition. Threshold values are shown inside the circles.

Just like when we do addition by hand, at each time step we need to sum up the current bit from each of the two numbers with the carry bit from the previous step, yielding a sum of 0, 1, 2, or 3. This sum is detected (in unary) by the three neurons on the left, which have thresholds of 1, 2, and 3. They receive inputs from the A and B input lines, as well as from the upper left unit, which as we will see computed the carry at the previous step. If the sum $A_i + B_i + carry_i$ is 0, then none of the left three neurons will fire. If the sum is 1, then just the one with threshold 1 will fire. If the sum is 2, then the one with threshold 2 will also fire, and if the sum is 3, then all three will fire.

Now, computing the carry bit is easy: We need to carry a 1 iff the sum is 2 or 3, so neuron c correctly computes the carry bit for the next step. Computing the output bit isn't much harder; it should be a 1 iff the sum is 1 or 3. Neuron f fires exactly when the sum is 1, and e fires when the sum is 3, so the output neuron just needs to take the OR of these values, but we need a delay g so that the values are presented to the output neuron at the same time.

Table 1 shows the behavior of the network for the input $11101 + 01110$. The delay in generating the output is 3 time steps. That is, the first bit of the sum is generated on the output line 3 time steps after the first bits of the summands are presented on the input lines.

2. Using relative inhibition, we can merge neurons f and g into a single output neuron with threshold 1, since then the output neuron will fire if the sum is 1 or 3, but not if the sum is 2, which is just what we need. This makes for a network of four neurons, and a delay of just 2 time steps.

   There are other 4-neuron circuits which also work, such as the one shown in Figure 1(b), whose activity is shown in Table 2.

t	A_i	B_i	c	d	e	f	g	C_i
1	**1**	**0**	0	-	-	-	-	-
2	**0**	**1**	0	1	0	-	-	-
3	**1**	**1**	0	1	0	1	0	-
4	**1**	**1**	1	1	0	1	0	**1**
5	**1**	**0**	1	1	1	0	0	**1**
6	0	0	1	1	0	0	1	**0**
7	0	0	0	1	0	0	0	**1**
8	0	0	0	0	0	1	0	**0**
9	0	0	0	0	0	0	0	**1**

Table 1: Activity of network in Figure 1(a) adding $11101 + 01110$

t	A_i	B_i	c	d	e	C_i
1	**1**	**0**	0	-	-	-
2	**0**	**1**	0	0	1	-
3	**1**	**1**	0	1	0	**1**
4	**1**	**1**	1	0	0	**1**
5	**1**	**0**	1	1	1	**0**
6	0	0	1	0	1	**1**
7	0	0	0	1	1	**0**
8	0	0	0	0	0	**1**

Table 2: Activity of network in Figure 1(b) adding $11101 + 01110$

3. Again, there are many possible solutions. We will base our solution on the standard way of doing long multiplication:

   $C_2C_1C_0 = A_1A_0 \times B_1B_0$ is carried out like this:

$$\begin{array}{cccc} & & A_1 & A_0 \\ \times & & B_1 & B_0 \\ \hline & & A_1B_0 & A_0B_0 \\ + & A_1B_1 & A_0B_1 & \\ \hline & C_2 & C_1 & C_0 \end{array}$$

   In Figure 2, we use neurons d and f to store B_0 and B_1. The values in d and f are then used to mask the digits of A, which stream directly through c and then e, by output units a and b, which feed directly into the adder shown previously.

   The activity for some sample multiplications is shown in Table 3. There is a delay of 4 time steps before the output starts coming out. (We could compress the network by adding more inputs to neuron a (from B_i and T) and getting rid of neuron c, but the resulting network would be a little more confusing.)

4. There are three parts to this question.

   (a) Since the answer to part 4b is that a network capable of multiplying two n bit long numbers must have at least 2^{n-1} states, and we know that a k-neuron network can have at most 2^k states, we see that k-neuron network will not be able to always correctly multiply $k+2$ bit

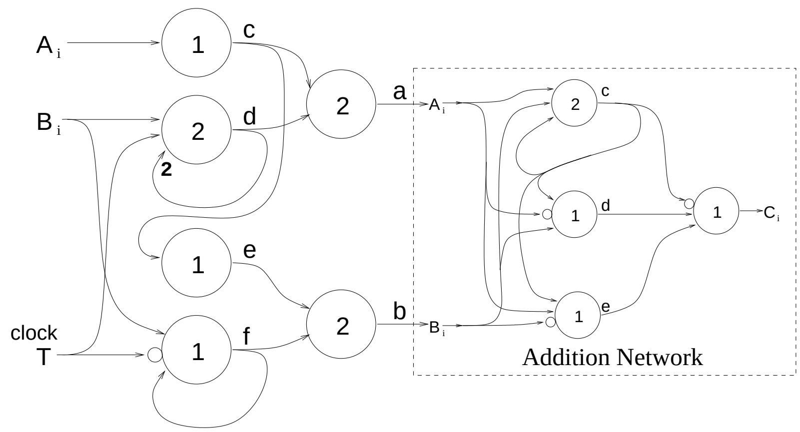


Figure 2: Network for multiplication. The weight from d to itself has weight 2. The outputs of a and b are summed by the addition network.

T	A	B	c	d	e	f	a	b	T	A	B	c	d	e	f	a	b
1	**1**	**1**	0	0	-	-	-	-	1	**0**	**1**	0	0	-	-	-	-
0	**1**	**1**	1	1	0	0	-	-	0	**1**	**0**	0	1	0	0	-	-
0	-	-	1	1	1	1	**1**	0	0	0	0	1	1	0	0	**0**	0
0	-	-	-	1	1	1	**1**	**1**	0	0	0	0	1	1	0	**1**	**0**
0	-	-	-	1	0	1	0	**1**	0	0	0	0	1	0	0	0	**0**

Table 3: Illustrations of multiplication using network in Figure 2. Left: $11 \times 11 = 11 + 110$; right: $10 \times 01 = 10 + 000$;

numbers. This means that any finite network can only correctly multiply all numbers up to a given size. So no, there does not exist any finite network which is capable of multiplying arbitrarily long numbers.

(b) If a network can correctly multiply two n-bit numbers, then it can among other things correctly multiply A and B when

$$A = 1\underbrace{00\ldots0}_{n-1} \qquad B = 0B_{n-2}B_{n-1}\ldots B_1B_0 \qquad A\times B = B_{n-2}B_{n-1}\ldots B_1B_0\underbrace{00\ldots0}_{n-1}$$

for all 2^{n-1} possible values of such B.

Let's consider the state of the network after $n-1$ times steps. At this stage, all of the B_i have been read in, but none of them have been output yet. If the network has fewer than 2^{n-1} internal states, then there must be two different values of B which lead to the same internal state after $n-1$ time steps. Since the input that comes after the first $n-1$ time steps is always the same, these two different values of B must lead to exactly the same network behavior after $n-1$ time steps. However, this means the network is operating incorrectly, since correct multiplication requires that the network behavior be different, after $n-1$ time steps, for different values of B.

Since correct behavior is impossible if the network has fewer than 2^{n-1} internal states, a network that correctly does multiplication of n-bit numbers must have *at least* 2^{n-1} internal states.

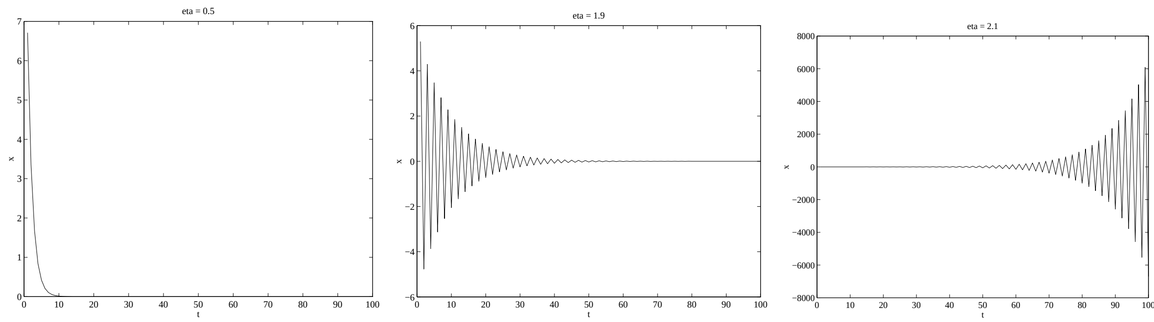
(c) If we generalize our 2-bit multiplication network into an n-bit multiplication network for some particular n, then for a given input pair (A, B) after some number t of time steps the network will be in some state. In fact, every reachable state of the network will exist for some triple (A, B, t), so since there are 2^n possibilities for A and 2^n possibilities for B and $2n$ possibilities for t, there can be no more than $n2^{2n+1}$ states of the network. This is an upper bound. Can we get a lower bound, or find the exact answer? For most network architectures, at time $t = n$ all the digits of both A and B are stored separately in the network, so there are at least 2^{2n} states. So this gives us a lower bound. Unfortunately, finding the exact number of states does not appear to be tractable for most network architectures.

1.2 Gradient Descent

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 $0 < \eta \le 1$, the oscillatory convergence occurs for $1 < \eta \le 2$, 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

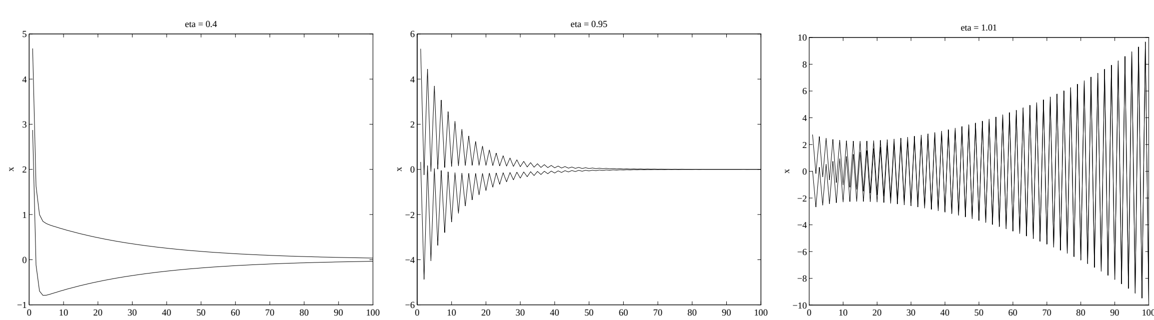
$$\nabla E = \left(\frac{\partial E}{\partial x}, \frac{\partial E}{\partial y}\right) = \left(\frac{1}{25}(26x + 24y), \frac{1}{25}(24x + 26y)\right)$$

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

$$\nabla E = \left(\begin{array}{c} x+y \\ x+y \end{array} \right) + \frac{1}{25} \left(\begin{array}{c} x-y \\ y-x \end{array} \right).$$

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

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

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

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: $y = e^{-t}$.

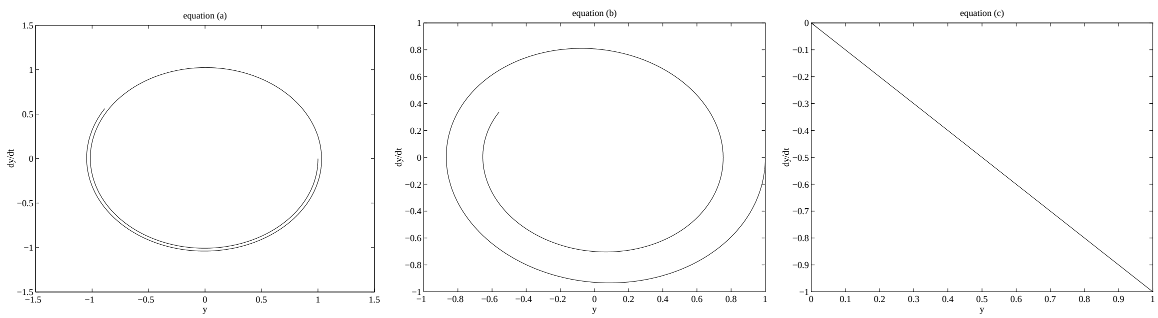
3. The MATLAB program is completed with the following lines (for equation (a); the other equations are similar):

```
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:

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



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 $\Delta t_{max} = 0.02$