Theory HW4

Problem 1: linearize a feed-forward NN

Let $\mathbf{x}$ input units, $\mathbf{W} \in \mathbb{R}^{M \times N}$ weight matrix, and $\mathbf{r} \in \mathbb{R}^{M}$ output unit.

The feedforward neural netork is then a dot product followed by a non linearity

$$\mathbf{r} = f(\mathbf{W}\mathbf{x})$$

The i'th output unit sees a weighted sum of all inputs

$$r_i = f(\sum_{j=1}^{N} w_{ij} x_j)$$

First define the Jacobian matrix $\boldsymbol{J}(\mathbf{x}) \in \mathbb{R}^{M \times N}$

$$\boldsymbol{J}(\mathbf{x}) = \frac{\partial r_i}{\partial x_j} = \begin{bmatrix} \frac{\partial r_1}{\partial x_1} & \frac{\partial r_1}{\partial x_2} & \cdots & \frac{\partial r_1}{\partial x_N} \\ \frac{\partial r_2}{\partial x_1} & \frac{\partial r_2}{\partial x_2} & \cdots & \frac{\partial r_2}{\partial x_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial r_M}{\partial x_1} & \frac{\partial r_m}{\partial x_2} & \cdots & \frac{\partial r_M}{\partial x_N} \end{bmatrix}$$

each entry of the Jacobian $\boldsymbol{J}_{ij} \in \mathbb{R}$ is

$$\boldsymbol{J}_{ij} = \frac{\partial r_i}{\partial x_j} = \frac{\partial}{\partial x_j}(r_i) = \frac{\partial}{\partial x_j}(f(\sum_{j=1}^{N} w_{ij} x_j)) = f'(\sum_{j=1}^{N} w_{ij} x_j) w_{ij}$$

where we have used the chain rule. note that

$$\boldsymbol{J}_{ij} = \underbrace{f'(\sum_{j=1}^{N} w_{ij} x_j))}_{:=a_i(\mathbf{x}),\text{ "gain" of } r_i} \underbrace{w_{ij}}_{\text{orig. weight for } x_j}$$

and we have M gains ($\mathbf{r} \in \mathbb{R}^{M}$) and and N input weights for each r_i, ($\mathbf{x} \in \mathbb{R}^{N}$). in matrix notation:

$$\boldsymbol{J}(\mathbf{x}) = \begin{bmatrix} a_i(\mathbf{x})\mathbf{W}_i \\ \cdots \\ a_m(\mathbf{x})\mathbf{W}_m \end{bmatrix} = \text{diag}_M(\mathbf{a}(\mathbf{x}))\mathbf{W}$$

now linearize about $\mathbf{r}_0$:

$$\mathbf{r} = \mathbf{r}_0 + \Delta\mathbf{r} = f(\mathbf{W}\mathbf{x}_0) + \underbrace{\Delta\mathbf{x}\mathbf{J}(\mathbf{x}_0)}_{=\text{diag}_M(\mathbf{a}(\mathbf{x}_0))\mathbf{W}\Delta\mathbf{x}}$$

Problem 2: forward simulate a randomly-connected RNN

In [327]:

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

In [328]:

```
# define params
N = 500
g_small = 0.9
g_large = 1.1
```

In [329]:

```
# define a random weight matrix with variance(W) = g**2 / N
standard_normal_mat = np.random.normal(size=(500,500))
W_small_g = (g_small*(1/np.sqrt(N)))*standard_normal_mat
W_large_g = (g_large*(1/np.sqrt(N)))*standard_normal_mat
```

In [330]:

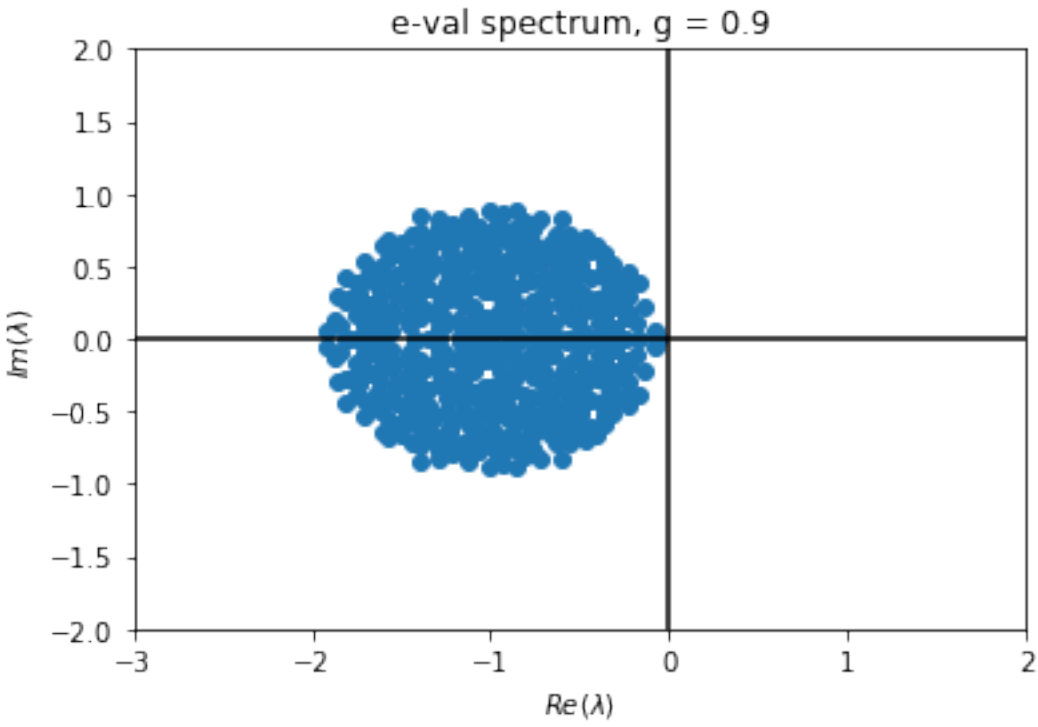
```
linearized_mat_g_small = - np.eye(N) + W_small_g
linearized_mat_g_large = - np.eye(N) + W_large_g
```

In [331]:

```
def plot_evals(matrix, g):
    w,v = np.linalg.eig(matrix)
    plt.vlines(x=0, ymin = -2, ymax=2)
    plt.hlines(y=0, xmin = -3, xmax = 2)
    plt.scatter(np.real(w), np.imag(w))
    plt.xlim(-3,2)
    plt.ylim(-2,2)
    plt.xlabel('$Re(\lambda)$')
    plt.ylabel('$Im(\lambda)$')
    plt.title('e-val spectrum, g = %.1f' %g);
```

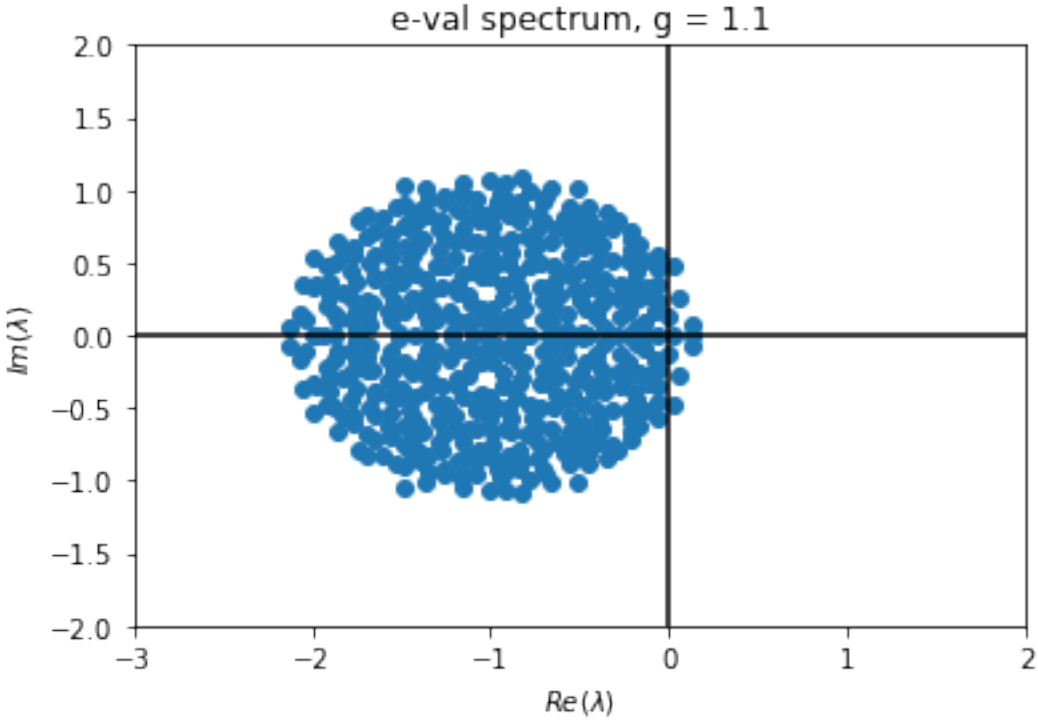
In [332]:

```
plot_evals(linearized_mat_g_small, g_small)
```



In [333]:

```
plot_evals(linearized_mat_g_large, g_large)
```



note that with $g < 1$ we do not get any positive real part, so the linearized system is stable with this effective connectivity. At the second plot, we do see e-cals with positive real part.

In [334]:

```
# ToDo: check that and FW sim to solve 2/3
def forward(w_mat, x_init, num_timesteps, dt, nonlinear = True
):
    x_vals = np.zeros((x_init.shape[0], num_timesteps)) # save
room
    x_vals[:,0] = x_init
    for i in range(1,num_timesteps): # Euler discretization
        if nonlinear:
            x_dot = - x_vals[:,i-1] + \
                w_mat.dot(np.tanh(x_vals[:,i-1]))
        else: # linearized, assume w_mat = -I + W
            x_dot = w_mat.dot(x_vals[:,i-1])
        x_vals[:,i] = x_vals[:,i-1] + dt*x_dot
    return x_vals
```

In [336]:

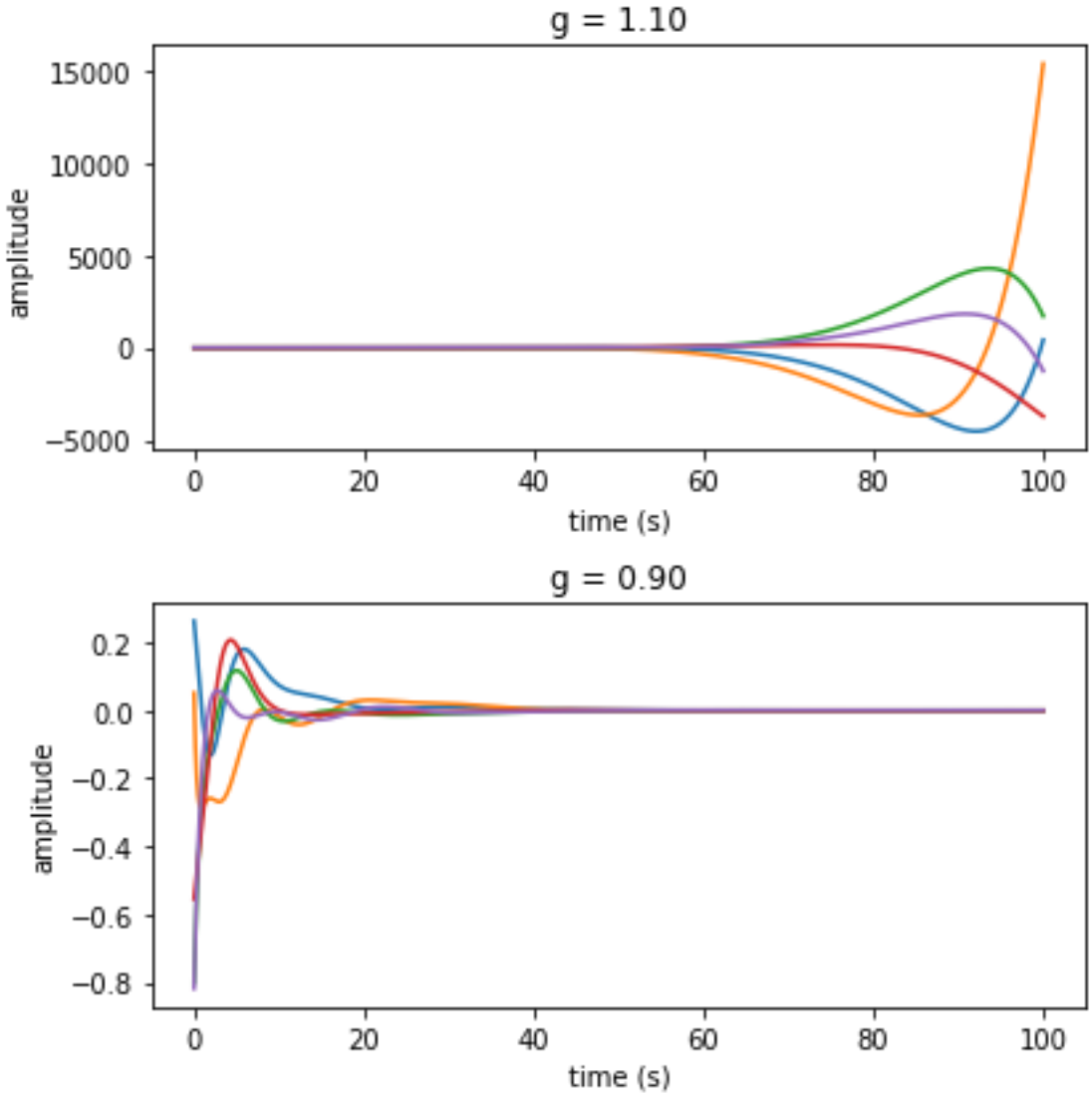
```
# init
x_init = np.random.normal(size=(N))
num_timesteps = 10000
dt = 0.01
```

In [337]:

```
x_vals_large_g = forward(linearized_mat_g_large, x_init,\
                         num_timesteps, dt, nonlinear=False)
x_vals_small_g = forward(linearized_mat_g_small, x_init,\
                         num_timesteps, dt, nonlinear=False)
```

In [338]:

```
fig = plt.figure(figsize=(6,6))
units_to_plot = np.array([1,10,16,92,450])
#plt.suptitle('linearized networks, %i timesteps' %num_timeste
ps)
plt.subplot(211)
plt.plot(np.arange(num_timesteps)*dt, \
         x_vals_large_g[units_to_plot,:].T);
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title('g = %.2f' % g_large);
plt.subplot(212)
plt.plot(np.arange(num_timesteps)*dt, \
         x_vals_small_g[units_to_plot,:].T);
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title('g = %.2f' % g_small);
plt.tight_layout()
#plt.suptitle('linearized networks, %i timesteps' %num_timeste
ps)
```



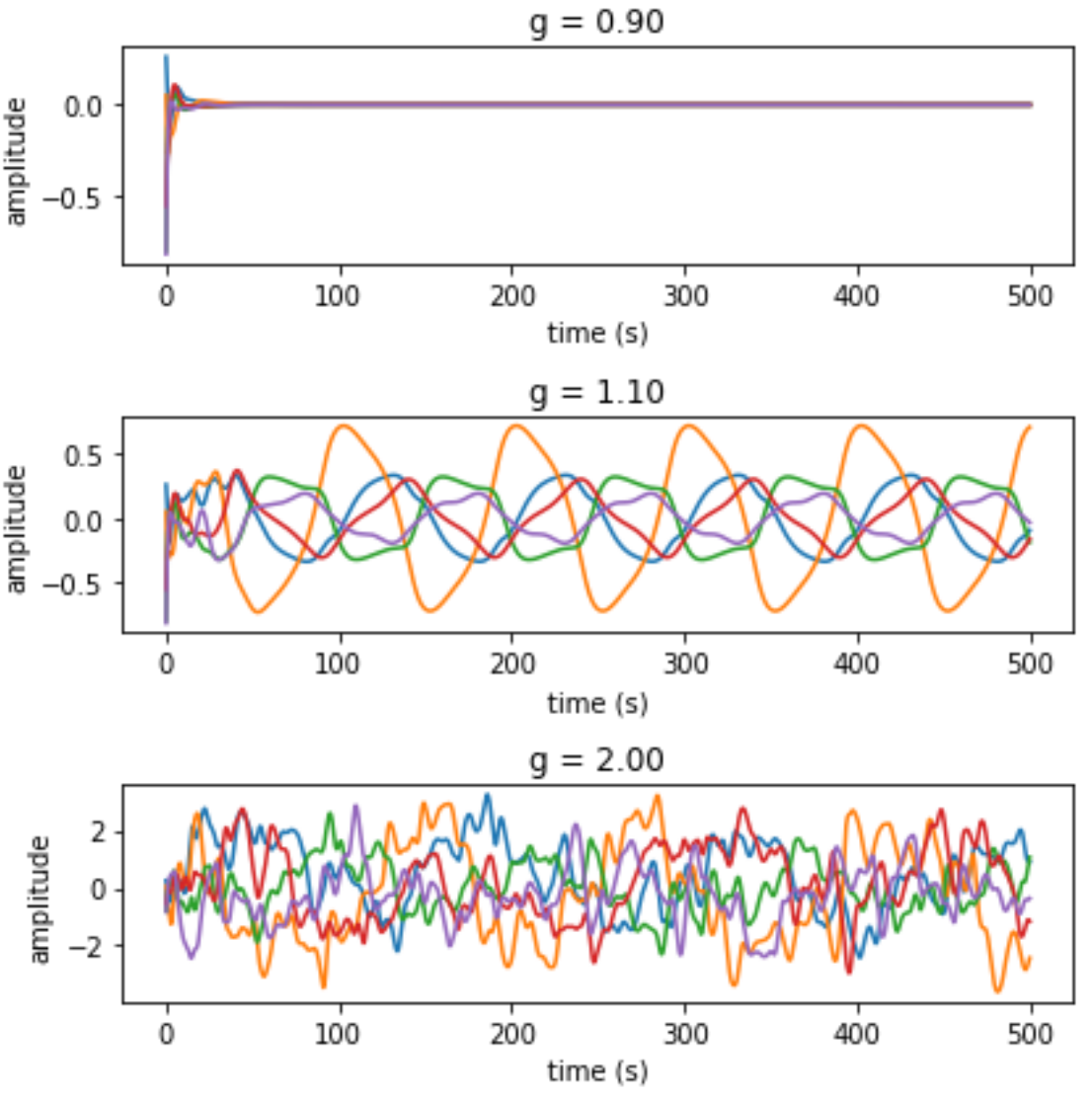
In [339]:

```
num_timesteps = 50000
non_lin_x_vals_small_g = forward(W_small_g, x_init,\
                         num_timesteps, dt, nonlinear=True)
non_lin_x_vals_large_g = forward(W_large_g, x_init,\
                         num_timesteps, dt, nonlinear=True)
# add g=2
W_g_2 = (2*(1/np.sqrt(N)))*standard_normal_mat

non_lin_x_vals_g_2 = forward(W_g_2, x_init,\
                         num_timesteps, dt, nonlinear=True)
```

In [340]:

```
fig = plt.figure(figsize=(6,6))
units_to_plot = np.array([1,10,16,92,450])
plt.subplot(311)
plt.plot(np.arange(num_timesteps)*dt, \
         non_lin_x_vals_small_g[units_to_plot,:].T);
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title('g = %.2f' % g_small);
plt.subplot(312)
plt.plot(np.arange(num_timesteps)*dt, \
         non_lin_x_vals_large_g[units_to_plot,:].T);
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title('g = %.2f' % g_large);
plt.subplot(313)
plt.plot(np.arange(num_timesteps)*dt, \
         non_lin_x_vals_g_2[units_to_plot,:].T);
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title('g = %.2f' % 2);
plt.tight_layout()
```



Optional Question 3: FitzHugh-Nagumo model

$\dot{v} = v - \frac{v^3}{3} - w + I,$ [non-linear dyn. on v]

$\tau\dot{w} = v + a - bw,$ [linear dyn.]

where:

1. v := membrane voltage
2. w := recovery variable
3. I := magnitude of external current
4. with constants: $\tau = 12.5, a = 0.7, b = 0.8$

In [164]:

```
tau = 12.5
a = 0.7
b = 0.8
```

find the fixed point as the intersection of the null-clines, where $\dot{v} = \dot{w} = 0$:

after some algebra, we obtain the following polynomial:

$$\frac{b}{3}v^3 + (1-b)v + (-bI + a) = 0$$

after we solve for v, we compute

$$w = \frac{v+a}{b}$$

In [177]:

```
def find_roots_for_fp(b, a, I):
    coeff = [b/3,0,1-b, -b*I + a]
    v = np.roots(coeff)
    w = (v + a) / b
    #w = v - v**3 + I # same, just for test
    return v, w
```

In [312]:

```
I = 0.3 # around 0.3 should be stable, above 0.4 unstable.
```

When you change the value of `I`, you'll get a different third order polynomial with different roots, which will in turn change the fixed point and the Jacobian evaluated at that fixed point.

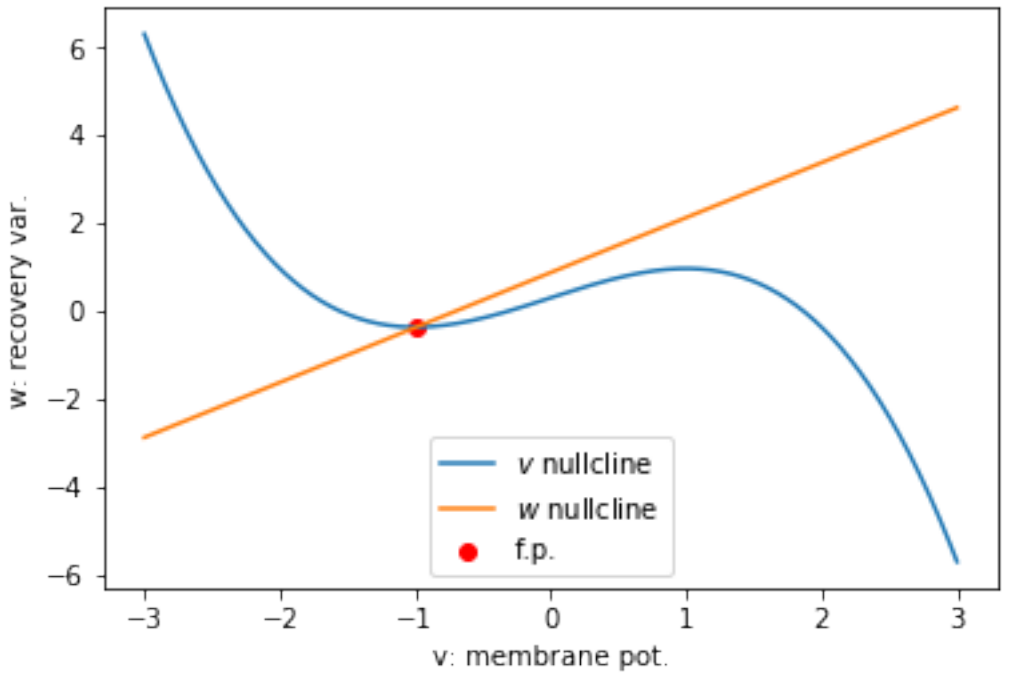
In [313]:

```
v, w = find_roots_for_fp(b, a, I)
v_fp = v[-1]
w_fp = w[-1]
print(v_fp)
print(w_fp)
# we'll take the last entry of which, which is real.
```

```
(-0.9932974745495973+0j)
(-0.3666218431869664+0j)
```

In [314]:

```
v_range = np.linspace(-3,3,1000)
plt.plot(v_range, v_range - (v_range**3)/3+I, label = '$v$ nul
lcline');
plt.plot(v_range, (v_range + a)/b, label = '$w$ nullcline');
plt.scatter(np.real(v_fp), np.real(w_fp), marker='o', c='red',
label = 'f.p.');
plt.xlabel('v: membrane pot.')
plt.ylabel('w: recovery var.')
plt.legend();
```



In [315]:

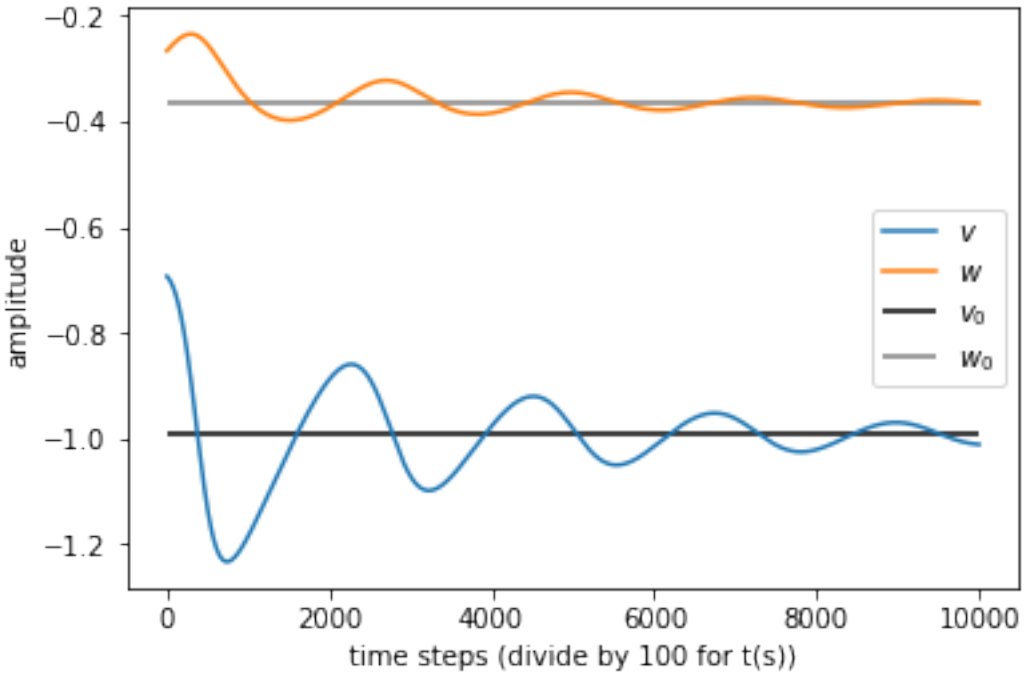
```
def FN_forward(v_init, w_init, num_timesteps, dt, I, nonlinear
= True):
    v_vals = np.zeros((1, num_timesteps))
    v_vals[:,0] = v_init
    w_vals = np.zeros((1, num_timesteps))
    w_vals[:,0] = w_init
    for i in range(1,num_timesteps): # Euler discretization
        if nonlinear:
            v_dot = v_vals[:,i-1] - \
                (v_vals[:,i-1]**3)/3 - w_vals[:,i-1] + I
            w_dot = 1/tau*(v_vals[:,i-1] + a - b*w_vals[:,i-1]
)
        else: # linearized
            v_dot = 0 # not needed
        v_vals[:,i] = v_vals[:,i-1] + dt*v_dot
        w_vals[:,i] = w_vals[:,i-1] + dt*w_dot
    return v_vals, w_vals
```

In [316]:

```
# note that we init slightly up and to the right of the f.p
v_vals, w_vals = FN_forward(v_init=np.real(v_fp)+0.3,
                            w_init=np.real(w_fp)+0.1,
                            num_timesteps=10000, dt=0.01,
                            I=I, nonlinear = True)
```

In [325]:

```
plt.hlines(xmin=0, xmax = 10000, y = np.real(v_fp),
           color = 'black' , label = '$v_0$')
plt.hlines(xmin=0, xmax = 10000, y = np.real(w_fp),
           color = 'gray', label = '$w_0$')
plt.plot(v_vals[0,:], label = '$v$');
plt.plot(w_vals[0,:], label = '$w$');
plt.xlabel('time steps (divide by 100 for t(s))')
plt.ylabel('amplitude')
plt.legend();
```



playing with different values of I we find that for approximately $I > 0.4$ we get repetetive spikes and for values below that we converge to our fixed point.

The Jacobian matrix for this system is:

$$J(v, w) = \begin{bmatrix} \partial f/\partial v & \partial f/\partial w \\ \partial g/\partial v & \partial g/\partial w \end{bmatrix} = \begin{bmatrix} 1 - v^2 & -1 \\ 1/\tau & -b/\tau \end{bmatrix}$$

which we evaluate at the fixed point $v = v_0$ that we found above using our function.

In [318]:

```
def Jac(v,b,tau):
    return np.array(([1-v**2, -1], [1/tau, -b/tau]))
```

In [319]:

```
# compute jacobian evaluated at fp
jac_fp = Jac(np.real(v_fp),b, tau) # jacobian eval at fp
print(jac_fp)
```

```
[[ 0.01336013 -1.        ]
 [ 0.08       -0.064     ]]
```

In [320]:

```
# compute evals and evecs of the jacobian
evals_J, evecs_J = np.linalg.eig(jac_fp)
print(np.round(evals_J,2)) # see that both e-vals have negativ
e real parts
```

```
[-0.03+0.28j -0.03-0.28j]
```

inspect that for $I = 0.4 : Re(\lambda_1) = Re(\lambda_2) > 0$ and the linearized system is unstable as we observe in the simulations,

and that for $I = 0.3 : Re(\lambda_1) = Re(\lambda_2) < 0$ the system converges to the fixed point with oscilations.

In [ ]:

```

```