OCNC 2019 Introductory Session for Biologists:

Numerical Methods for Differential Equations

2019.6.24 by Kenji Doya

Contents

- What is a differential equation
- Euler method
- `ode()` fuction in `scipy`
- Stability and eigenvalus
- Hodgkin-Huxley neuron model

References

- Stephen Wiggins: Introduction to Applied Nonlinear Dynamical Systems and Chaos, 2nd ed., Springer (2003).
- Scipy Lecture Notes (http://www.scipy-lectures.org): Section 1.5.7 Numerical Integration

What is a differential equation

A *differential equation* is an equation that includes a derivative $\frac{dy(x)}{dx}$ of a function $y(x)$.

If the independent variable x is single, such as time, it is called an *ordinary differential equation (ODE)*.

If there are multiple independent variables, such as space and time, the equation includes *partial derivatives* and called a *partial differential equation (PDE)*.

Here we consider ODEs of the form

$$\frac{dy}{dt} = f(y, t)$$

which describes the temporal dynamics of a varibale y over time t. It is also called a *continuous-time dynamical system*.

Finding the variable y as an explicit function of time $y(t)$ is called *solving* or *integrating* the ODE.
When it is done numerically, it is aslo called *simulating*.

Analytic Solutions

Solving a differential equation is an inverse problem of differentiation, for which analytic solution may not be available.

The simplest case where analytic solutions are available is *linear* differential equations

$$\frac{dy}{dt} = Ay$$

where y is a real variable or a real vector, and A is a constant coefficient or matrix.

Linear ODEs

In general, a differential equation can have multiple solutions.

For example, for a scalar linear ODE

$$\frac{dy}{dt} = ay,$$

the solution is given by

$$y(t) = Ce^{at},$$

where C can be any real value.

When the value of y at a certain time is specified, the solution becomes unique.
For example, by specifying $y(0) = 3$, from $e^{a0} = e^0 = 1$, we have $C = 3$ and a particular solution $y(t) = 3e^{at}$.

For a second-order linear ODE

$$\frac{d^2y}{dt^2} = -a^2 y,$$

the solution is given by

$$y(t) = C_1 \sin at + C_2 \cos at$$

where C_1 and C_2 are determined by spedifying y and $\frac{dy}{dt}$ at certain time.

In [ ]:

```
1
```

Euler Method

The most basic way of sovling an ODE numerically is *Euler Method*.

Remember the definition of the derivative is
$$\frac{dy}{dt} = \lim_{\Delta t \to 0} \frac{y(t+\Delta t) - y(t)}{\Delta t}.$$
Thus we can approximate $\frac{dy}{dt}$ with a small time step Δt as
$$\frac{dy}{dt} \simeq \frac{y(t+\Delta t) - y(t)}{\Delta t} = f(y,t).$$

This brings us to an update equation
$$y(t+\Delta t) = y(t) + f(y,t)\Delta t$$
starting from an initial condition $y(t_0) = y_0$.

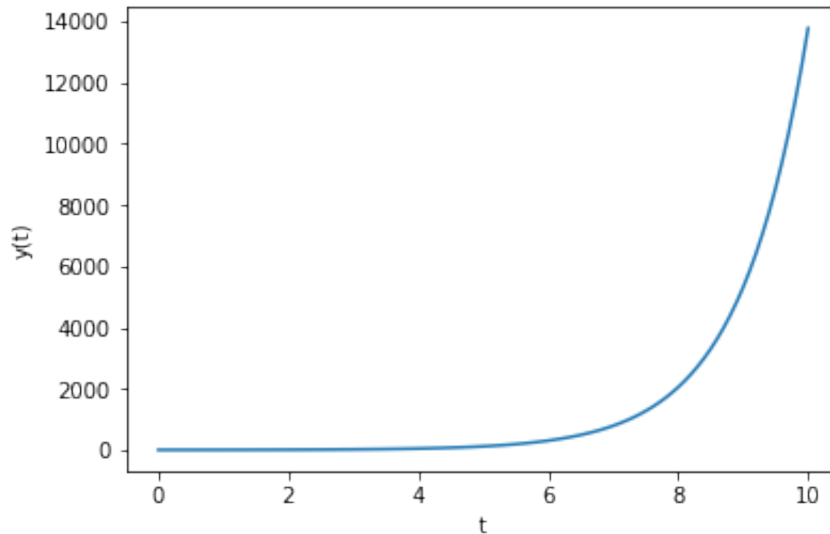
In [1]:
```
# As usual, import numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
def euler(f, y0, dt, n, *args):
    """f: righthand side of ODE dy/dt=f(y,t)
        y0: initial condition y(0)=y0
        dt: time step
        n: iteratons
        args: parameter for f(y,t,*args)"""
    d = np.array([y0]).size  ## state dimension
    y = np.zeros((n+1, d))
    y[0] = y0
    t = 0
    for k in range(n):
        y[k+1] = y[k] + f(y[k], t, *args)*dt
        t = t + dt
    return y
```

Let us test this with a first-order linear ODE.

In [3]:
```
def first(y, t, a):
    """first-order linear ODE dy/dt = a*y"""
    return a*y
```

In [4]:
```
1 dt = 0.1
2 n = 100
3 y = euler(first, 1, dt, n, 1)
4 plt.plot(np.arange(0,n+1)*dt, y)
5 plt.xlabel("t"); plt.ylabel("y(t)");
```



Try different values of `y_0`, `a` and `dt`.

In [ ]:
```
1
```

A second-order ODE

$$\frac{d^2y}{dt^2} = a_2\frac{dy}{dt} + a_1y + a_0$$

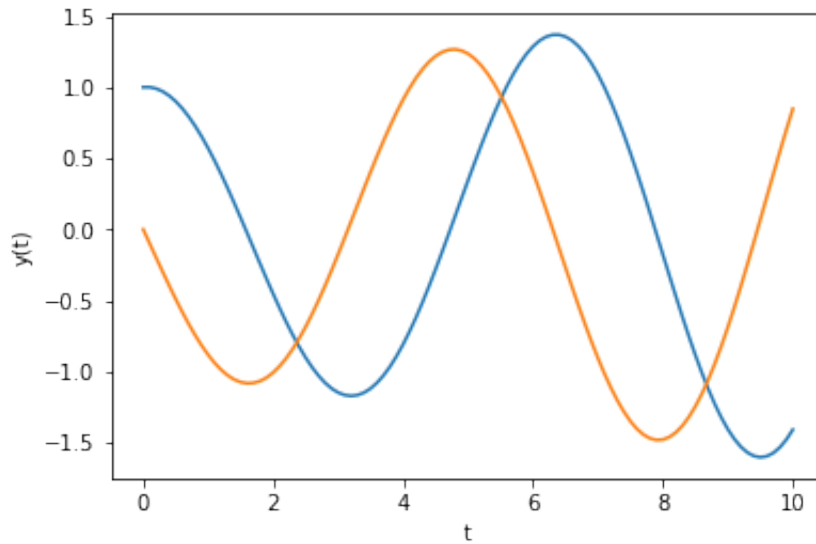
can be converted to a first-order ODE with a 2-dimensional state vector $(y_1, y_2) = (y, \frac{dy}{dt})$ as

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = a_2y_2 + a_1y_1 + a_0$$

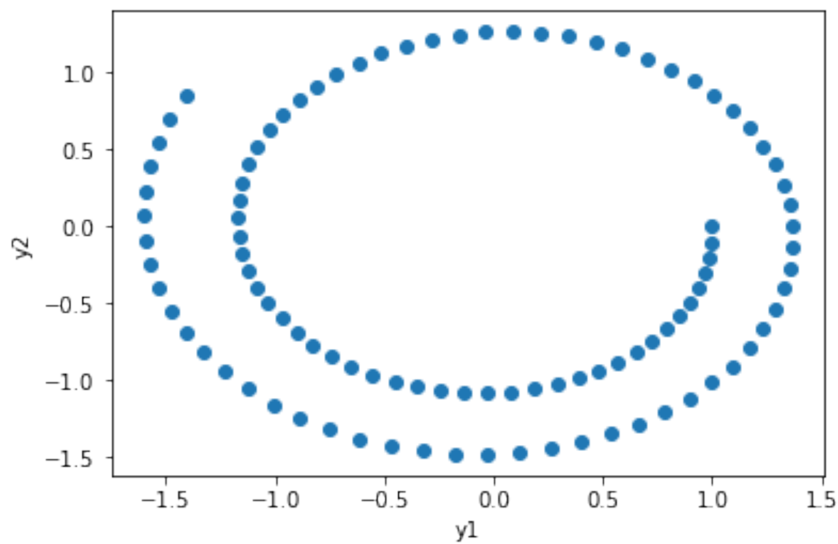
In [5]:
```
1 def second(y, t, a):
2     """second-order linear ODE """
3     y1, y2 = y
4     return np.array([y2, a[2]*y2 + a[1]*y1 + a[0]])
```

In [6]:
```
1 y = euler(second, [1, 0], dt, n, [0, -1, 0])
2 plt.plot(np.arange(0,n+1)*dt, y)
3 plt.xlabel("t"); plt.ylabel("y(t)");
```



The waves look growing even though `a1=-1`. Why?

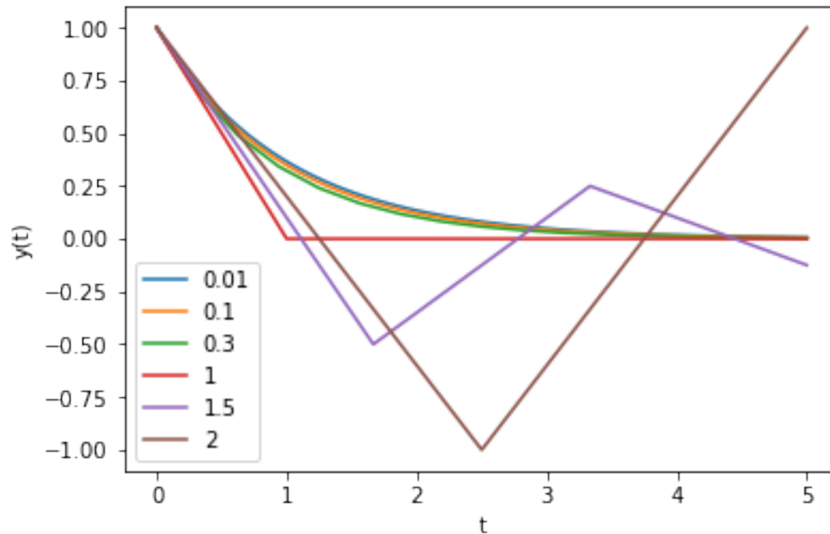
In [7]:
```
1 plt.plot(y[:,0], y[:,1], "o")
2 plt.xlabel("y1"); plt.ylabel("y2");
```



In [ ]:
```
1
```

Let us see how the time step affects the accuracy of the solution.

In [8]:
```
dts = [0.01, 0.1, 0.3, 1, 1.5, 2]
tend = 5
a = -1
for dt in dts:
    y = euler(first, 1, dt, int(tend/dt), a)
    plt.plot(np.linspace(0, tend, len(y)), y)
plt.xlabel("t"); plt.ylabel("y(t)");
plt.legend(dts);
```



In [ ]:
```

```

Scipy's Integrate package

For any serious integration, it is better to use a well tested and proven library, such as `odeint()` in `scipy`.

In [9]:
```
from scipy.integrate import odeint
```

In [10]:
```
1 help(odeint)
```

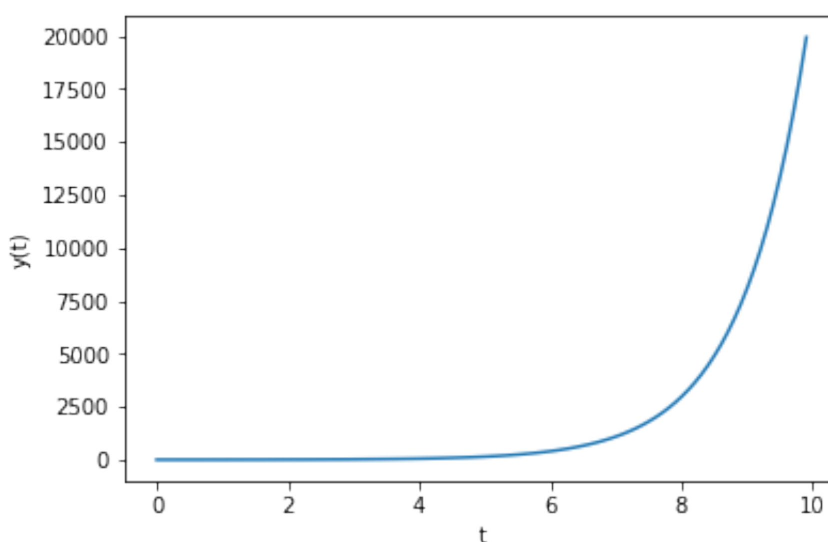
```
        'mused'   a vector of method indicators for each successful
time step:
                  1: adams (nonstiff), 2: bdf (stiff)
        =======   =================================================
===========

    Other Parameters
    ----------------
    ml, mu : int, optional
        If either of these are not None or non-negative, then the
        Jacobian is assumed to be banded.  These give the number o
f
        lower and upper non-zero diagonals in this banded matrix.
        For the banded case, `Dfun` should return a matrix whose
        rows contain the non-zero bands (starting with the lowest
diagonal).
        Thus, the return matrix `jac` from `Dfun` should have shap
e
        ``(ml + mu + 1, len(y0))`` when ``ml >=0`` or ``mu >=0``.
        The data in `jac` must be stored such that ``jac[i - j + m
```

`odeint()` internally uses adaptive time steps, and returns values of `y` for time points specified in `t` by interpolation.

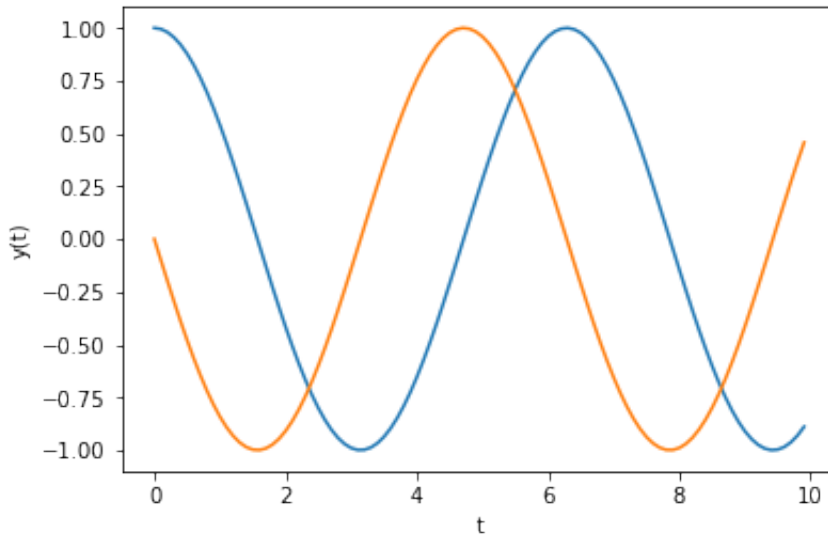
Try with the first order linear equaiton.

In [11]:
```
1 t = np.arange(0, 10, 0.1)  # time points
2 y = odeint(first, 1, t, args=(1,))
3 plt.plot(t, y)
4 plt.xlabel("t"); plt.ylabel("y(t)");
```



And the second order.

In [12]:
```
1 t = np.arange(0,10,0.1)  # time points
2 y = odeint(second, [1, 0], t, args=([0, -1, 0],))
3 plt.plot(t, y)
4 plt.xlabel("t"); plt.ylabel("y(t)");
```



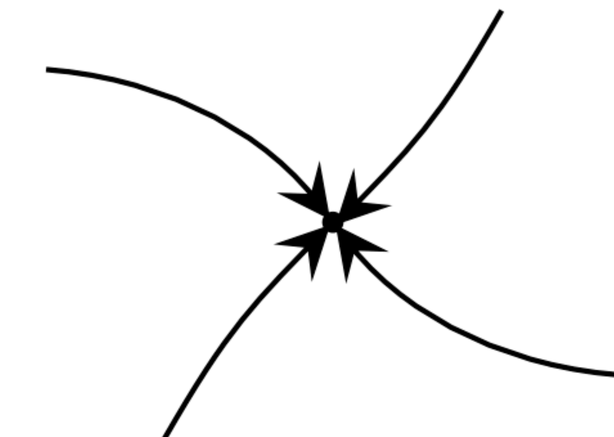
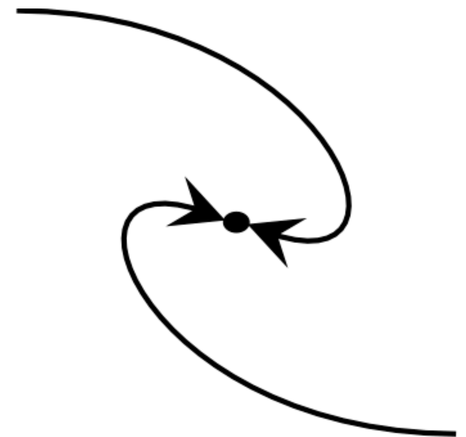
In [ ]:
```
1
```

Fixed Point and Stability

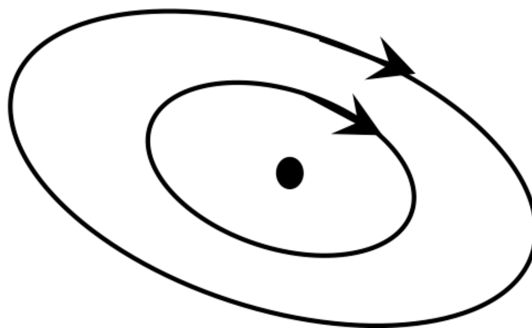
A point y where $\frac{dy}{dt} = 0$ is called a *fixed point*.

A fixed point is characterized by its *stability*:

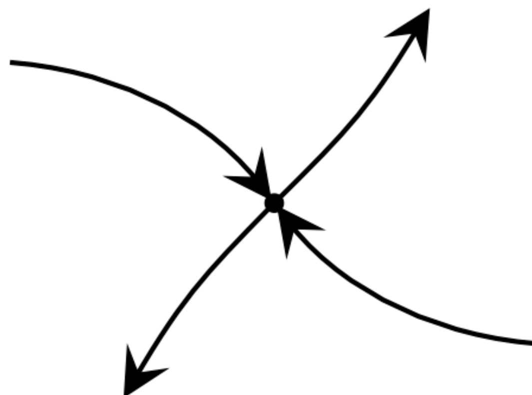
- Stable
    - Attractor

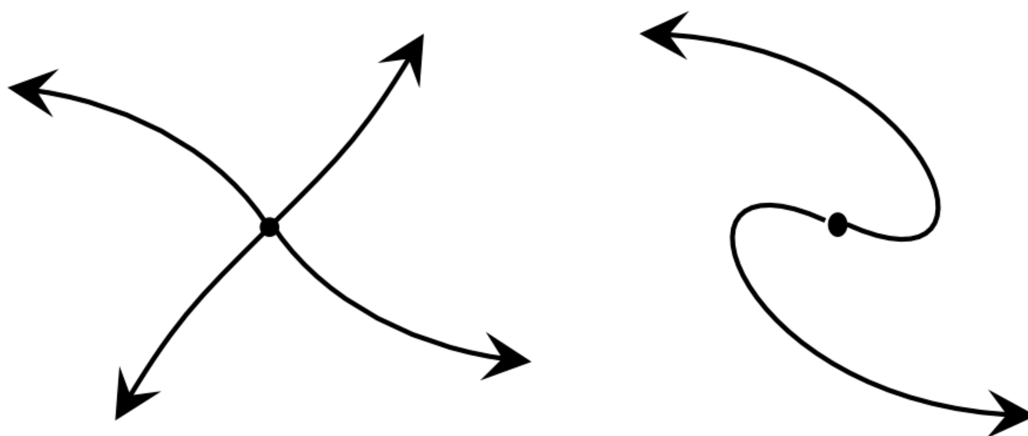
    - Neutrally stable



- Unstable
  - Saddle



  - Repellor



For a linear dynamical system

$$\frac{dy}{dt} = Ay$$

where y is an n dimensional vector and A is an $n \times n$ matrix, the origin $y = 0$ is a fixed point. Its stability is determined by the eigenvalues of A.

```
In [ ]: 1
```

Linear differential equation system

Here we take a vector-matrix notation

$$\frac{d}{dt}\mathbf{y} = A\mathbf{y}$$

where

$$\mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \quad A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

In [13]:

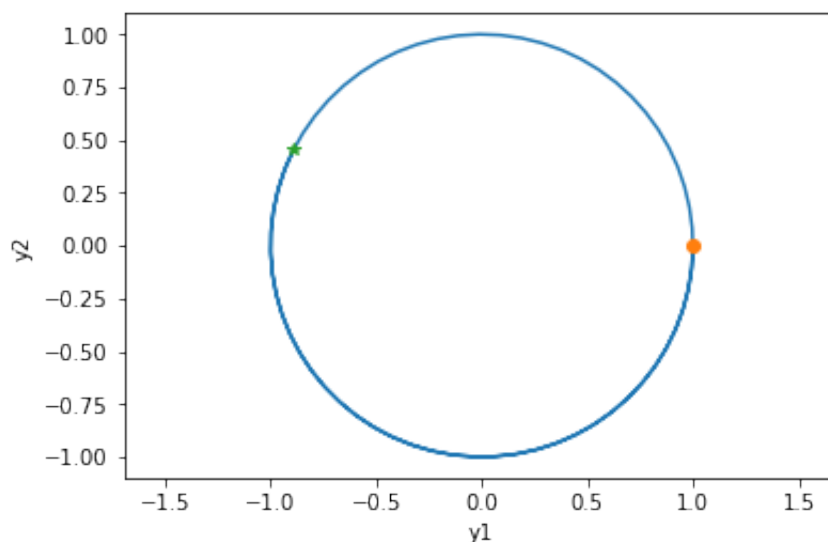
```
def linear(y, t, A):
    """Linear dynamcal system dy/dt = Ay
    y: n-dimensional state vector
    t: time (not used, for compatibility with odeint())
    A: n*n matrix"""
    # y is an array (row vector), A is a matrix
    return A@y
```

In [14]:

```
A = np.array([[0, 1], [-1, 0]])
```

In [15]:

```
y0 = np.array([1, -0.001])
t = np.arange(0, 10, 0.1)
y = odeint(linear, y0, t, args=(A,))
plt.plot(y[:,0], y[:,1])  # trajectory
plt.plot(y[0,0], y[0,1], 'o', y[-1,0], y[-1,1], '*')   # start/end
plt.axis('equal'); plt.xlabel("y1"); plt.ylabel("y2");
```



Eigenvalues and eigenvectors

The behavior of the linear differential equation is determined by the *eigenvalues* and *eigenvectors* of the coefficient matrix A.

With a matrix multiplication, a vector $\mathbf{x}$ is mapped to $A\mathbf{x}$, which can chenge the direction and size of the vector.
An *eigenvector* of A is a vector that keeps its direction after multiplication and its scaling coefficient is called the *eigenvalue*.

Eigenvalues and eigenvectors are derived by solving the equation

$$A\mathbf{x} = \lambda\mathbf{x}.$$

For the 2x2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, the eivenvalues are given from

$$\det(A - \lambda I) = (a - \lambda)(d - \lambda) - bc = 0$$

as

$$\lambda = \frac{a+d}{2} \pm \sqrt{\left(\frac{a-d}{2}\right)^2 + bc}$$

Complex eigenvalues makes an oscillatory solution.
The signs of the real part determines the stability.

You can check eigenvalues and eigenvectors by `linalg.eig()` function.

In [16]:

```
np.linalg.eig(A)
```

Out[16]:

```
(array([0.+1.j, 0.-1.j]),
 array([[0.70710678+0.j        , 0.70710678-0.j        ],
        [0.        +0.70710678j, 0.        -0.70710678j]]))
```

Try different settings of A and corresponding solutions.

In [17]:

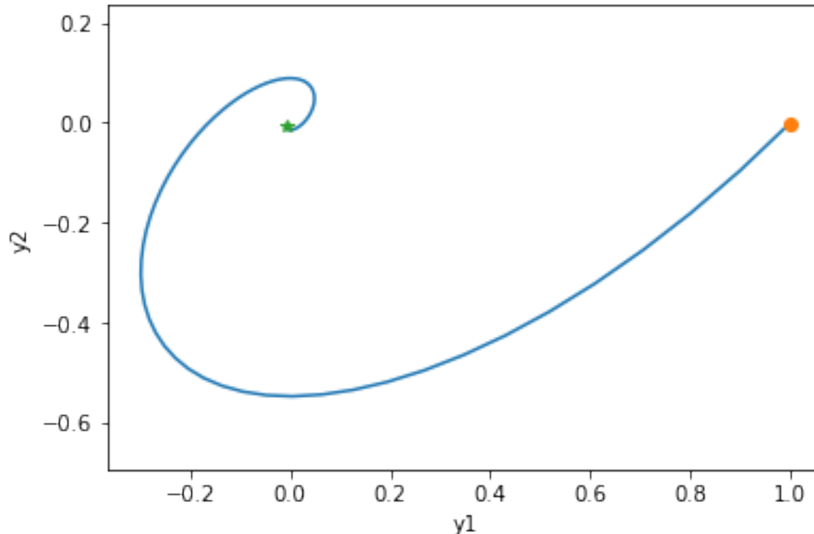
```
# Spiral in
A = np.array([[-1, 1], [-1, 0]])
print(A)
np.linalg.eig(A)
```

```
[[-1  1]
 [-1  0]]
```

Out[17]:

```
(array([-0.5+0.8660254j, -0.5-0.8660254j]),
 array([[0.35355339-0.61237244j, 0.35355339+0.61237244j],
        [0.70710678+0.j        , 0.70710678-0.j        ]]))
```

In [18]:
```
y0 = np.array([1, -0.0001])
t = np.arange(0, 10, 0.1)
y = odeint(linear, y0, t, args=(A,))
plt.plot(y[:,0], y[:,1])  # trajectory
plt.plot(y[0,0], y[0,1], 'o', y[-1,0], y[-1,1], '*')   # start/end
plt.axis('equal'); plt.xlabel("y1"); plt.ylabel("y2");
```



In [19]:
```
# Spiral out
A = np.array([[1, 1], [-1, 0]])
print(A)
np.linalg.eig(A)
```

```
[[ 1  1]
 [-1  0]]
```

Out[19]:
```
(array([0.5+0.8660254j, 0.5-0.8660254j]),
 array([[ 0.35355339+0.61237244j,  0.35355339-0.61237244j],
        [-0.70710678+0.j        , -0.70710678-0.j        ]]))
```

In [20]:
```
# Saddle
A = np.array([[-1, 0], [0, 1]])
print(A)
np.linalg.eig(A)
```

```
[[-1  0]
 [ 0  1]]
```

Out[20]:
```
(array([-1.,  1.]), array([[1., 0.],
        [0., 1.]]))
```

In [ ]:
```

```

Response to time varying input

Alpha function

$$\alpha(t) = \frac{t}{\tau} e^{-\frac{t}{\tau}}$$

is often used to approximate EPSP.

This is a solution of a second-orde ODE with an impulse input $\delta(t)$

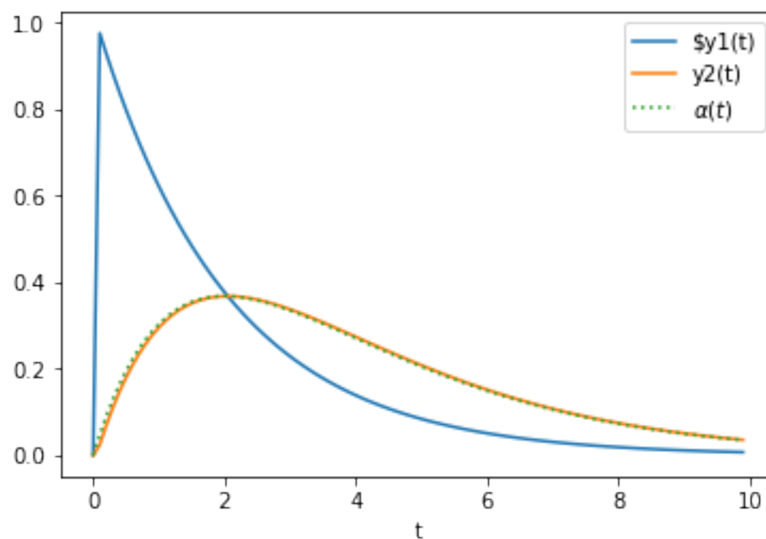
$$\frac{dy_1}{dt} = \frac{-y_1}{\tau} + \delta(t)$$
$$\frac{dy_2}{dt} = \frac{y_1 - y_2}{\tau}$$

In [21]:

```
def alpha(y, t, tau=1):
    I = (0<t)*(t<0.1)*10   # short pulse input
    return np.array([I-y[0]/tau, (y[0]-y[1])/tau])
```

In [22]:

```
tau = 2
t = np.arange(0, 10, 0.1)
y = odeint(alpha, [0,0], t, (tau,))
plt.plot(t, y)
plt.plot(t, t/tau*np.exp(-t/tau), ":")
plt.xlabel("t");
plt.legend(("$y1(t)", "y2(t)", r"$\alpha(t)$"));
```

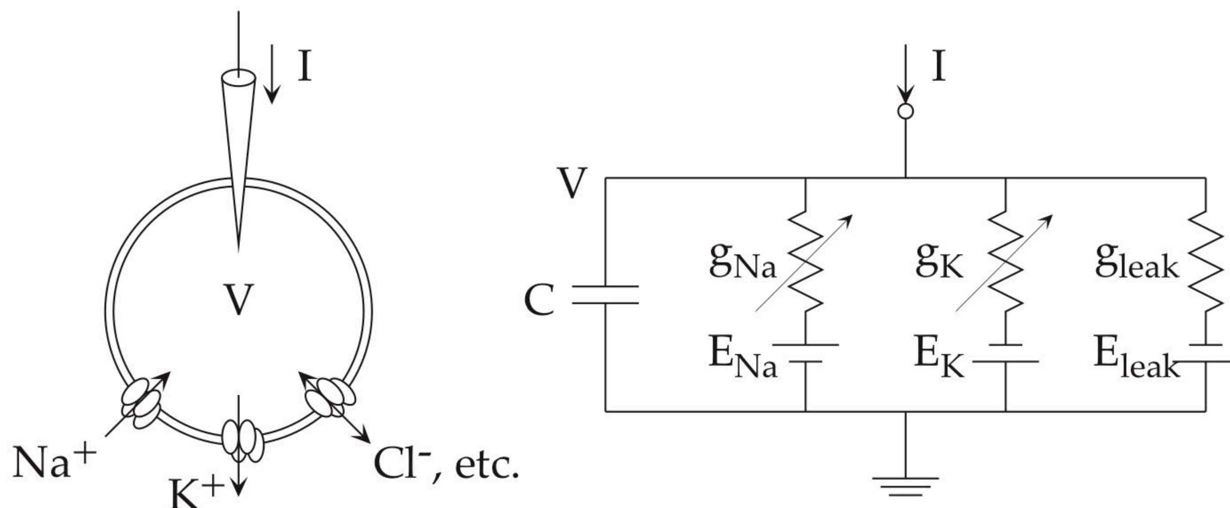


In [ ]:

```
```

Hodgkin-Huxley neuron models

The Hodgkin-Huxley (HH) model considers a neuron as an electric circuit as depicted below.



On the cellular membrane, there are *ionic channels* that pass specific type of ions. Sodium ions (Na^+) are scarce inside the cell, so that when sodium channel opens, positive charges flood into the cell to cause excitation. Potassium ions (K^+) are rich inside the cell, so that when potassium channel opens, positive charges flood out of the cell to cause inhibition. The HH model assumes a 'leak' current that put together all other ionic currents.

The ingeniety of Hodgkin and Huxley is that they inferred from careful data analysis that a single sodium channel consists of three *activation* gates and one *inactivation* gate, and a single potassium channel consists of four activation gates. Such structures were later confirmed by genomics and imaging.

The electric potential inside the neuron V follows the following equation:

$$C\frac{dV}{dt} = g_{Na}m^3h(E_{Na} - V) + g_K n^4(E_K - V) + g_L(E_L - V) + I$$

Here, m, h, and n represent the proportions of opening of sodium activation, sodium inactivation, and potassium activation gates, respectively. They follow the following differential equations with their rates of opening and closing, $\alpha(V)$ and $\beta(V)$, depending on the membrane voltage V.

$$\frac{dm}{dt} = \alpha_m(V)(1 - m) - \beta_m(V)m$$

$$\frac{dh}{dt} = \alpha_h(V)(1 - h) - \beta_h(V)h$$

$$\frac{dn}{dt} = \alpha_n(V)(1 - n) - \beta_n(V)n$$

These compose a system of four-dimensional non-linear differential equations. Another amazing thing about Hodgkin and Huxley is that they could simulate the solutions of these differential equations by a hand-powered computer.

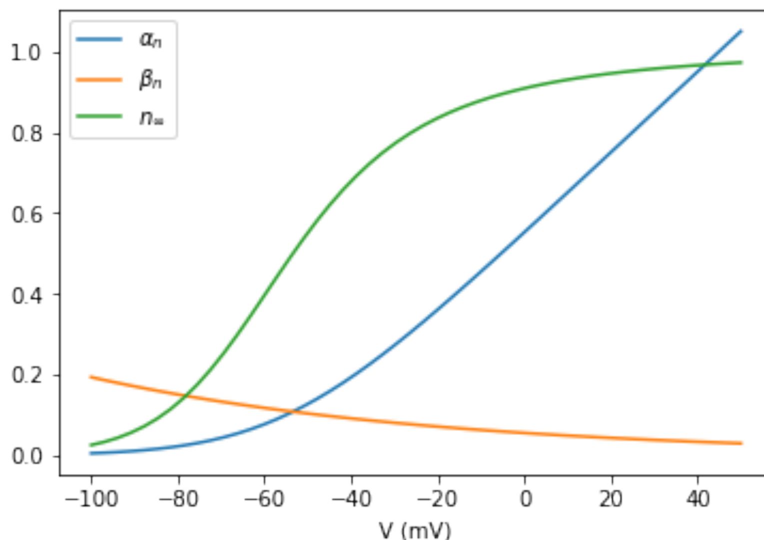
Below is a code to simulate the HH model by Python. Much easier!

First, let us look into the potassium activation function n. The asymptotic value for a constant potential V is given from $\frac{dn}{dt} = 0$ as

$$n_\infty(V) = \frac{\alpha_n(V)}{\alpha_n(V) + \beta_n(V)}$$

In [23]:

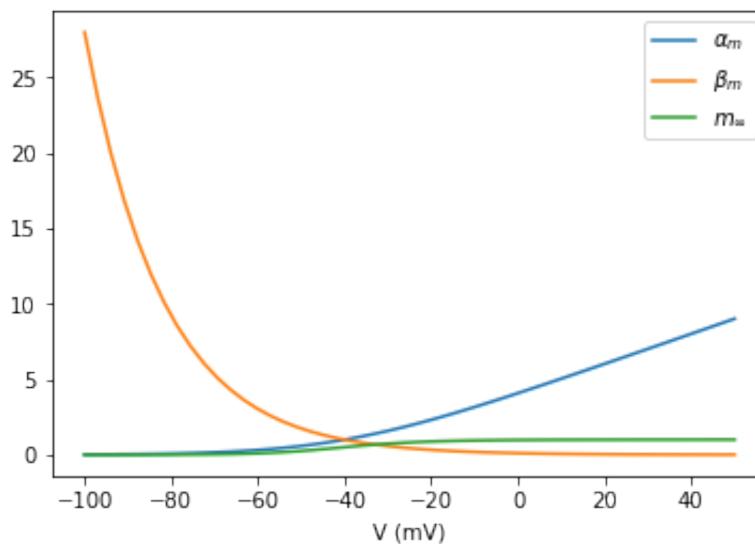
```
def alpha_n(v):
    """opening rate of potassium activation gate"""
    return 0.01*(v+55)/(1-np.exp(-(v+55)/10))
def beta_n(v):
    """closing rate of potassium activation gate"""
    return 0.125*np.exp(-(v+65)/80)
def n_inf(v):
    """asymptotic sodium activation"""
    return alpha_n(v)/(alpha_n(v)+beta_n(v))
# Let us plot them
v = np.linspace(-100, 50)  # from -100mV to +50mV
plt.plot(v, alpha_n(v))
plt.plot(v, beta_n(v))
plt.plot(v, n_inf(v))
plt.legend((r"$\alpha_n$", r"$\beta_n$", r"$n_\infty$"));
plt.xlabel("V (mV)");
```



We can see the same for m and h for sodium current.

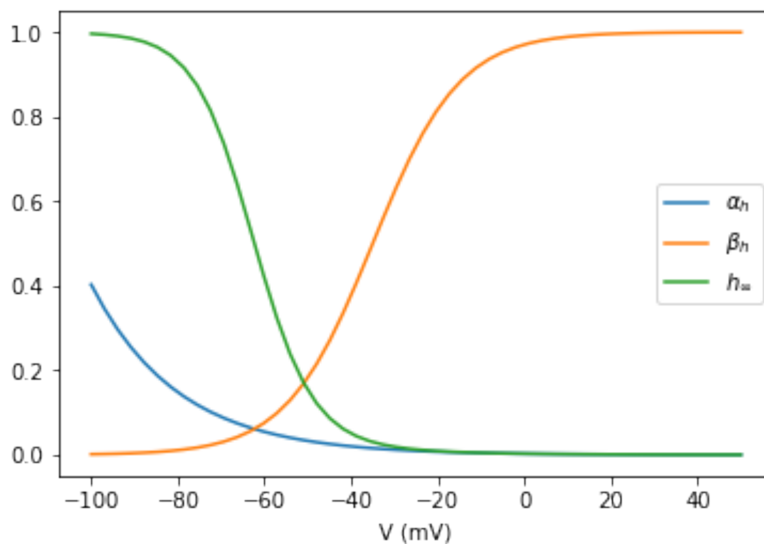
In [24]:

```
def alpha_m(v):
    """opening rate of sodium activation gate"""
    return 0.1*(v+40)/(1-np.exp(-(v+40)/10))
def beta_m(v):
    """closing rate of sodium activation gate"""
    return 4*np.exp(-(v+65)/18)
def m_inf(v):
    """asymptotic sodium activation"""
    return alpha_m(v)/(alpha_m(v)+beta_m(v))
# Let us plot them
plt.plot(v, alpha_m(v))
plt.plot(v, beta_m(v))
plt.plot(v, m_inf(v))
plt.legend((r"$\alpha_m$", r"$\beta_m$", r"$m_\infty$"));
plt.xlabel("V (mV)");
```



In [25]:

```
def alpha_h(v):
    """opening rate of sodium inactivation gate"""
    return 0.07*np.exp(-(v+65)/20)
def beta_h(v):
    """closing rate of sodium inactivation gate"""
    return 1/(1+np.exp(-(v+35)/10))
def h_inf(v):
    """asymptotic sodium inactivation"""
    return alpha_h(v)/(alpha_h(v)+beta_h(v))
# Let us plot them
plt.plot(v, alpha_h(v))
plt.plot(v, beta_h(v))
plt.plot(v, h_inf(v))
plt.legend((r"$\alpha_h$", r"$\beta_h$", r"$h_\infty$"));
plt.xlabel("V (mV)");
```



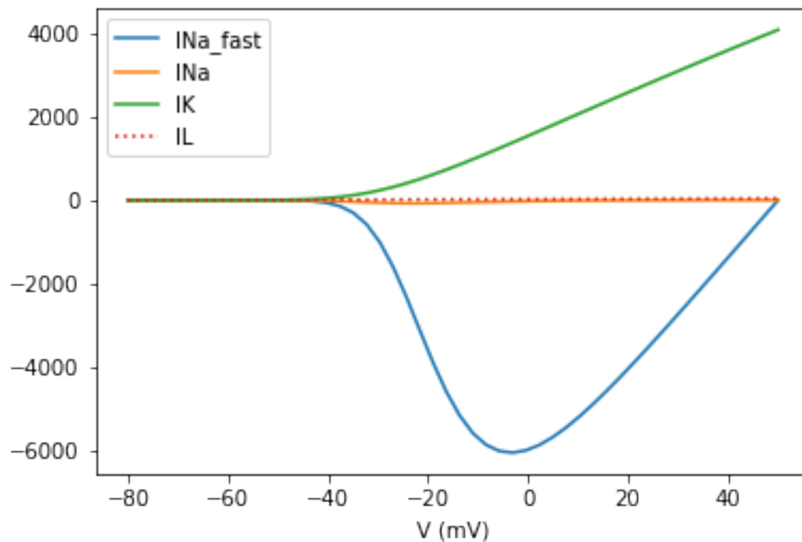
With reversal potentials and max conductance, we can visulaize the I_V curves for different ions.

In [26]:

```
# reversal potentials (mV)
Ena = 50    # sodium
Ek = -77    # potassium
El = -54.4 # leak
# maximum conductances (uS/cm^2)
gna = 120   # sodium
gk = 36     # potassium
gl = 0.3    # leak
```

In [27]:

```
Inaf = gna*m_inf(v)**3*(v-Ena)  # Fast component of Na current
Ina = Inaf*h_inf(v)   # Na current after inactivation
Ik = gk*n_inf(v)**4*(v-Ek)
Il = gl*(v-Ek)
v = np.linspace(-80, 50)  # from -80mV to +50mV
plt.plot(v, Inaf, v, Ina)
plt.plot(v, Ik, v, Il, ":")
plt.legend(("INa_fast", "INa", "IK", "IL"))
plt.xlabel("V (mV)");
```



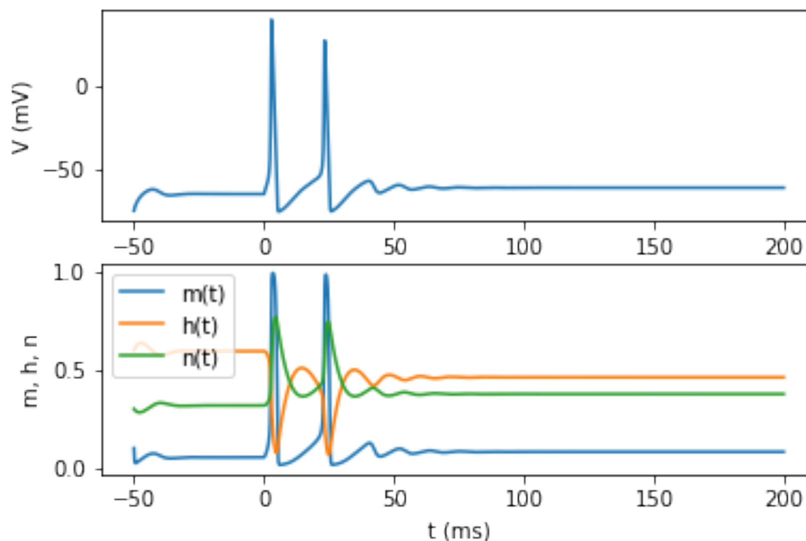
Now let us define the HH model and simulate it!

In [28]:

```
# membrabe capactance (uF/cm^2)
Cm = 1
def hh(y, t, I=0):
    """Hodgkin-Huxley (1952) model
    I: input current (uA/cm^2) for t>0"""
    v, m, h, n = y
    I = 0 if t<0 else I  # no current for t<0
    # time derivatives
    return np.array([ (I - gna*m**3*h*(v-Ena) - gk*n**4*(v-Ek) - g
            alpha_m(v)*(1-m) - beta_m(v)*m,
            alpha_h(v)*(1-h) - beta_h(v)*h,
            alpha_n(v)*(1-n) - beta_n(v)*n])
```

In [29]:

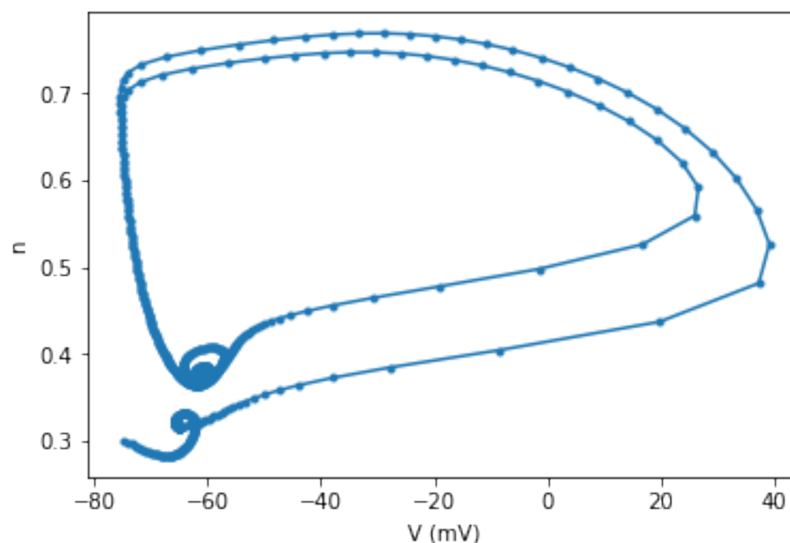
```
tt = np.arange(-50, 200, 0.1)  # time
y0 = np.array([ -75, 0.1, 0.6, 0.3])  # initial state
yt = odeint(hh, y0, tt, args=(6,))  # current (uA)
#plt.plot(tt,stim(tt))
plt.subplot(2, 1, 1)
plt.plot(tt, yt[:,0])   # plot V separetely in different scale
plt.ylabel("V (mV)");
plt.subplot(2, 1, 2)
plt.plot(tt, yt[:,1:])  # m, h, n
plt.legend(("m(t)", "h(t)", "n(t)"), loc='upper left')
plt.ylabel("m, h, n");
plt.xlabel("t (ms)");
```



Let us see the trajectory in a *phase space*.

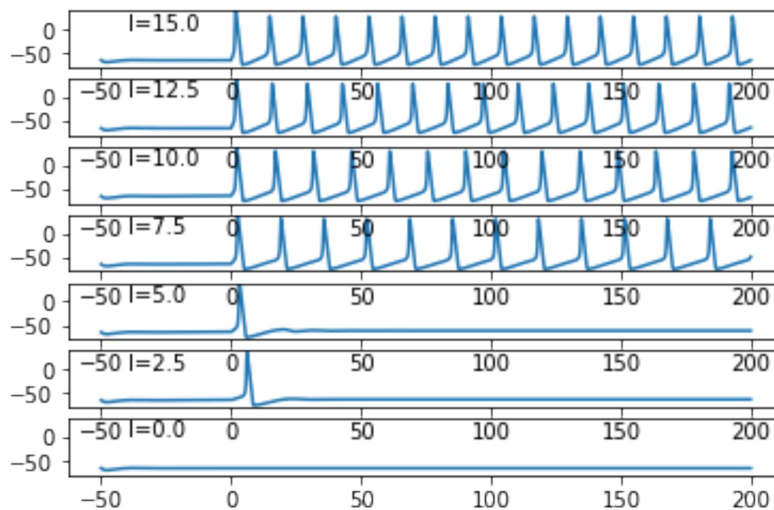
In [30]:

```
plt.plot(yt[:,0], yt[:,3], ".-")
plt.xlabel("V (mV)")
plt.ylabel("n");
```



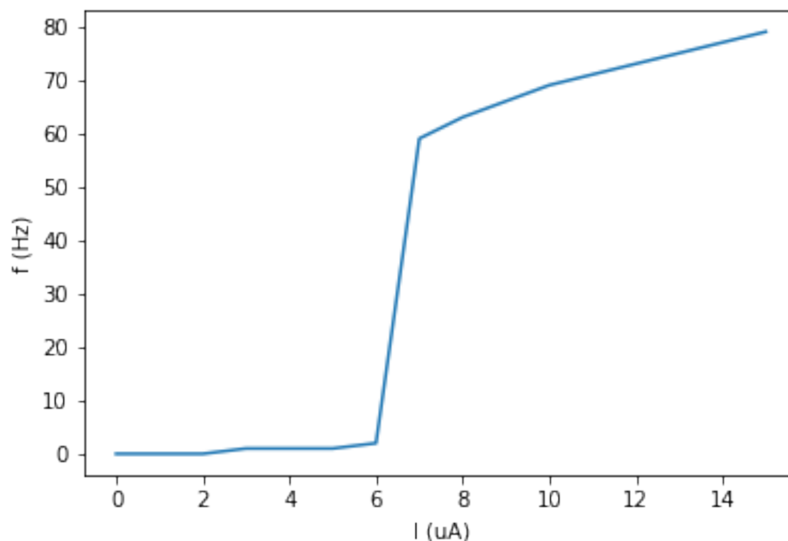
Let us see how the behavior changes with the input current.

In [31]:
```
tt = np.arange(-50, 200, 0.1)  # time
y0 = np.array([ -65, 0.1, 0.6, 0.4])  # initial state
n = 7  # levels
Ir = np.linspace(0, 15, n)  # range of current
for i, I in enumerate(Ir):
    yt = odeint(hh, y0, tt, args=(I,))
    plt.subplot(n, 1, n-i)
    plt.plot(tt, yt[:,0])   # plot V
    plt.ylim(-80, 40)  # same scale
    plt.text(-40, 0, "I={0:1.1f}".format(I));
```



Let's plot a F-I curve.

In [*]:
```
T = 1000    # run length (ms)
tt = np.arange(-50, T, 0.2)
y0 = np.array([ -65, 0.1, 0.5, 0.4])  # initial state
n = 16  # levels
Ir = np.linspace(0, 15, n)  # range of current (uA)
fs = np.zeros(n)   # to store spike #
for i, I in enumerate(Ir):
    yt = odeint(hh, y0, tt, args=(I,))
    st = (yt[1:,0]<0) & (yt[:-1,0]>=0) # zero crossing
    fs[i] = sum(st)*1000/T
plt.plot(Ir, fs)   # F-U curve
plt.ylabel("f (Hz)");
plt.xlabel("I (uA)");
```



In the standard HH model, the firing frequency is around 60 Hz once a current goes above the threshold. This is called *type-II* behavior, associated with *Hopf bifurcation*.

The HH model can show *type-I* behavior, associated with *saddle-node bifurcation* with some change in the parameter, e.g., E_K (Guckenheimer & Labouliau, 1993).

In [*]:
```
Ek = -60    # changed from the standard -77 mv
T = 1000    # run length (ms)
tt = np.arange(-50, T, 0.2)
y0 = np.array([ -65, 0.1, 0.5, 0.4])  # initial state
n = 16  # levels
Ir = np.linspace(-10, 5, n)  # range of current (uA)
fs = np.zeros(n)   # to store spike #
for i, I in enumerate(Ir):
    yt = odeint(hh, y0, tt, args=(I,))
    st = (yt[1:,0]<0) & (yt[:-1,0]>=0) # zero crossing
    fs[i] = sum(st)*1000/T  # frequency
plt.plot(Ir, fs)   # F-I curve
plt.ylabel("f (Hz)");
plt.xlabel("I (uA)");
```

```
In [ ]:  1
```

Further readings

- Hodgkin AL, Huxley AF (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. Journal of Physiology, 117, 500-544. http://doi.org/10.1113/jphysiol.1952.sp004764 (http://doi.org/10.1113/jphysiol.1952.sp004764)
- Guckenheimer J, Labouriau IS (1993). Bifurcation of the Hodgkin and Huxley Equations - a New Twist. Bulletin of Mathematical Biology, 55, 937-952. http://doi.org/10.1007/Bf02460693 (http://doi.org/10.1007/Bf02460693)
- Rinzel J, Ermentrout B (1998). Analysis of neural excitability and oscillations. Koch C, Segev I, Methods in Neuronal Modeling: From Ions to Networks, MIT Press, 251-292.
- Catterall WA, Raman IM, Robinson HP, Sejnowski TJ, Paulsen O (2012). The Hodgkin-Huxley heritage: from channels to circuits. J Neurosci, 32, 14064-73. http://doi.org/10.1523/JNEUROSCI.3403-12.2012 (http://doi.org/10.1523/JNEUROSCI.3403-12.2012)


```
In [ ]:  1
```