Ordinary Differential Equations

Computational Methods, Oct. 2017, Kenji Doya

A differential equation is an equation that includes a derivative $df(x)/dx$ of a function $f(x)$.
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

$$dy/dt = f(y, t)$$

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

Finding the variable 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*.

References:

- Stephen Wiggins: Introduction to Applied Nonlinear Dynamical Systems and Chaos, 2nd ed., Springer (2003).
- Scipy Lecture Notes: Section 5.8 Numerical Integration

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

$$dy/dt = Ay$$

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

$$dy/dt = ay,$$

the solution is given by

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

where C can be any real value.

- Verify that $y(t)$ satisfies the ODE above.

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 $y(t) = 3e^{at}$.

Another example is a second-order linear ODE

$$d^2y/dt^2 = -a^2y.$$

In this case, 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 dy/dt at certain time.

- Also verify above.

Analytically solvable ODEs

Other cases where analytic solutions are well known are:

- Time-varying linear: $dy/dt = g(t)y + h(t)$
- Separable: $dy/dt = g(t)/h(y(t))$
- Other cases that can be reduced to above by change of variables, etc.

You can use `dsolve()` function of `SymPy` to find some analitic solutions. See Scipy Tutorial, Chapter 16 if you are interested.

Euler Method

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

$$dy/dt = f(y, t)$$

with an ititial condition $y(0) = y_0$, the solution is iteratively approximated by

$$y(t + \Delta t) = y(t) + f(y, t)\Delta t$$

with a small time step Δt.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

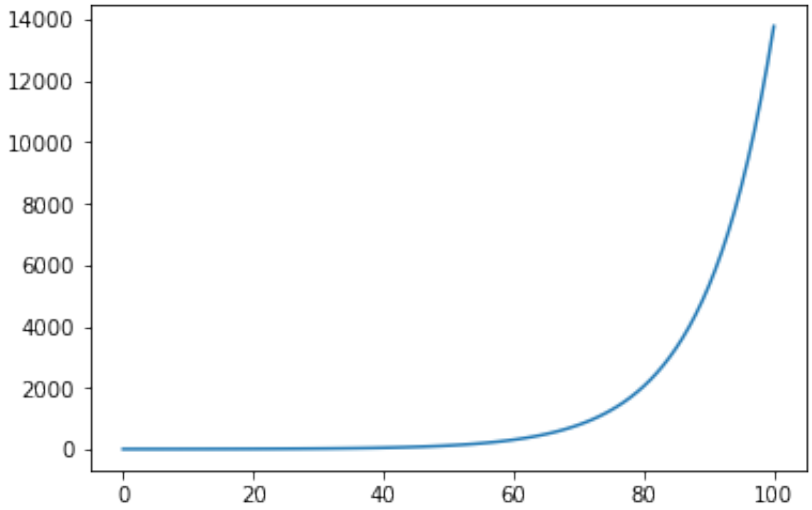
In [2]:
```
def euler(f, y0, dt, n, *args):
    """f: righthand side of ODE dy/dt=f(y,t)
        y0: initial condition y(0)=y0
        dt: time step
        n: iteratons
        args: parameter for f(y,t,*args)"""
    d = np.array([y0]).size  ## state dimension
    y = np.zeros((n+1, d))
    y[0] = y0
    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]:
```
y = euler(first, 1, 0.1, 100, 1)
plt.plot(y)
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x10862acc0>]
```



A second-order ODE

$$d^2y/dt^2 = a_2 dy/dt + a_1 y + a_0$$

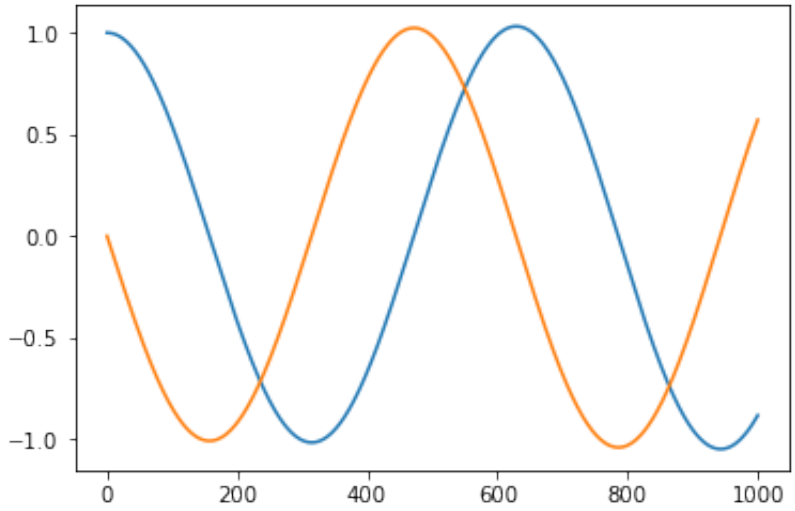
can be converted to a first-order ODE with a 2-dimensional state vector $(y_1, y_2) = (y, dy/dt)$ as

$$dy_1/dt = y_2$$
$$dy_2/dt = a_2 y_2 + a_1 y_1 + a_0$$

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

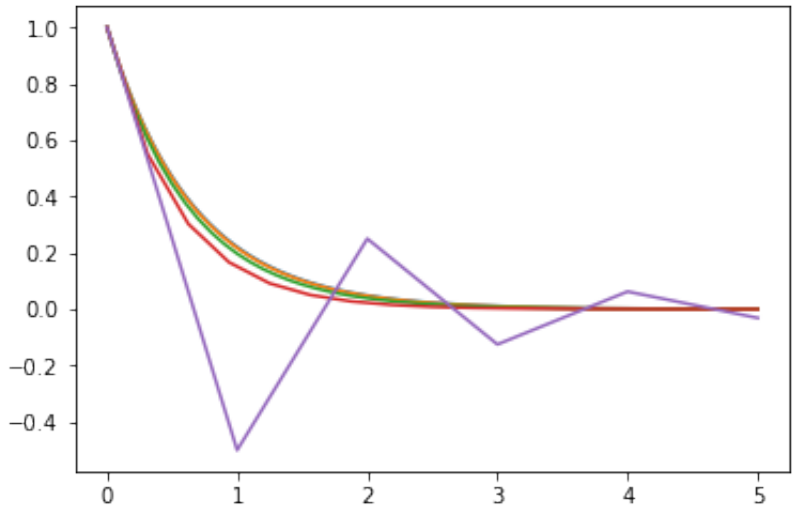
In [6]:
```
y = euler(second, [1, 0], 0.01, 1000, [0, -1, 0])
plt.plot(y)
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x10875eb38>,
 <matplotlib.lines.Line2D at 0x10875ecf8>]
```



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

In [7]:
```
steps = [0.01, 0.03, 0.1, 0.3, 1]
tend = 5
a = -1.5
for dt in steps:
    y = euler(first, 1, dt, int(tend/dt), a)
    plt.plot(np.linspace(0, tend, len(y)), y)
```



In [ ]:

Scipy's Integrate package

To avoid numerical instability and to improve accuracy and efficiency, there are advanced methods for ODE solutions.

- Backward Euler method: solve
$$y(t + \Delta t) = y(t) + f(y(t + \Delta t))\Delta t$$
- Mixture of forward and backward (Crank-Nicolson):
$$y(t + \Delta t) = y(t) + 1/2f(y(t)) + f(y(t + \Delta t))\Delta t$$
- Runge-Kutta method: minimize higher-order erros by Taylor expansion
$$y(t + \Delta t) = y(t) + f(y(t))\Delta t + 1/2df(y(t))/dt\Delta t^2 + \ldots$$
- Adative time step: adjust Δt depending on the scale of $df(y(t))/dt$.

The implementation and choice of these methods require a good expertise, but fortunately `scipy` includes `integrate` package which has been well tested and optimized.
`odeint()` implements automatic method switching and time step adaptation.
`ode()` is a class interface for multiple methods.

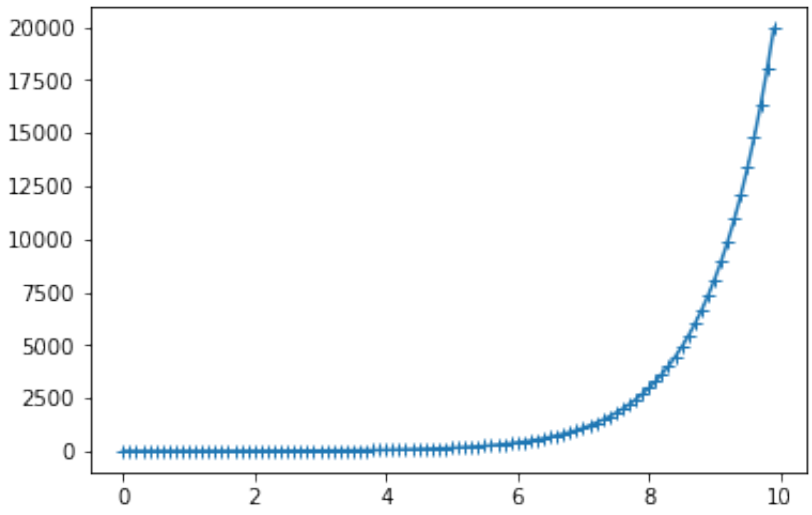
In [8]:
```
from scipy.integrate import odeint
```

In [9]:
```
odeint?
```

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

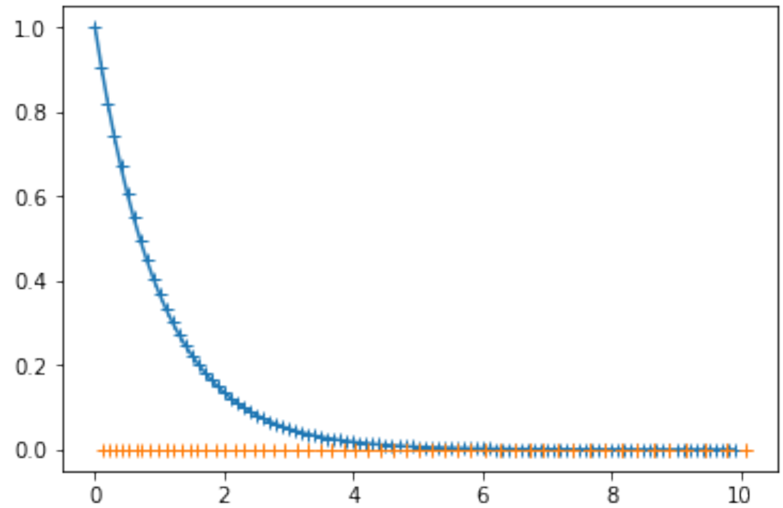
In [10]:
```
t = np.arange(0, 10, 0.1)  # time points
y = odeint(first, 1, t, args=(1,))
plt.plot(t, y, '+-')
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x150fe71d68>]
```



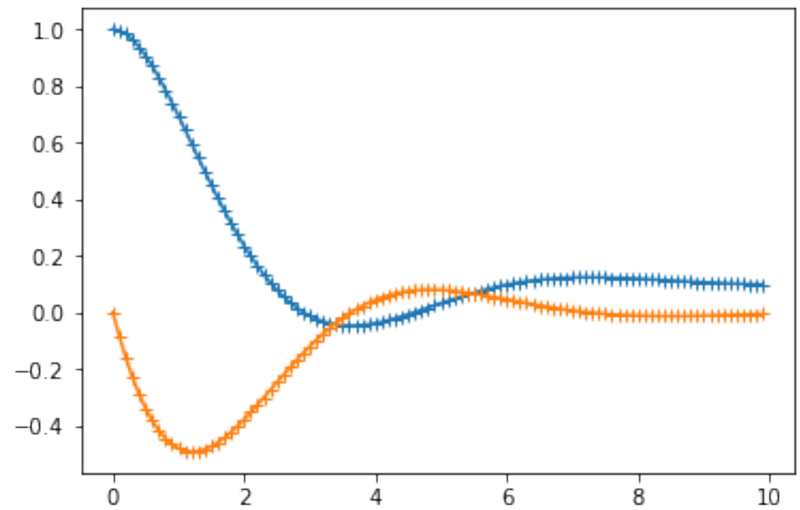
In [11]:
```python
# If you are interested in details of computing
y, info = odeint(first, 1, t, args=(-1,), full_output=1)
plt.plot(t, y, '+-')
plt.plot(info['tcur'], np.zeros_like(info['tcur']), '+')
print(info['mused'])
```

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```



In [12]:
```python
t = np.arange(0,10,0.1)  # time points
y = odeint(second, [1, 0], t, args=([0.1, -1, -1],))
plt.plot(t, y, '+-')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x151006b400>,
 <matplotlib.lines.Line2D at 0x151006b588>]
```



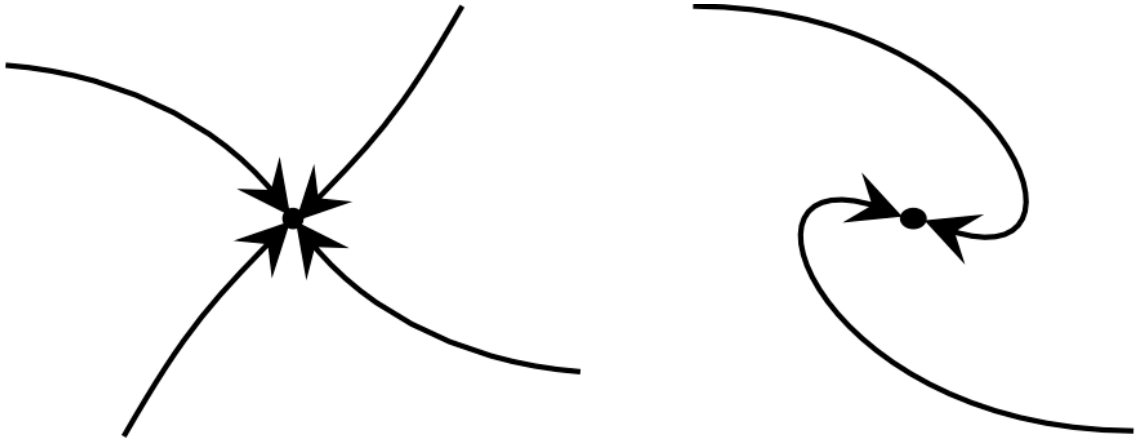
In [ ]:

Fixed Point and Stability

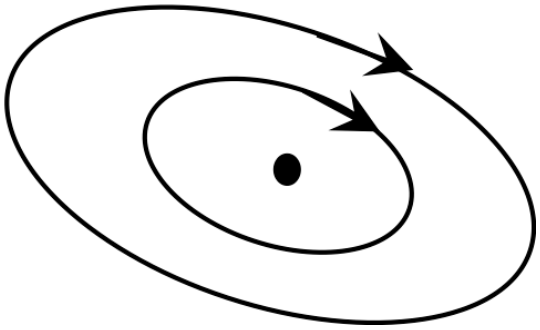
A point y where $dy/dt = f(y) = 0$ is called a *fixed point*.

A fixed point is characterized by its *stability*:

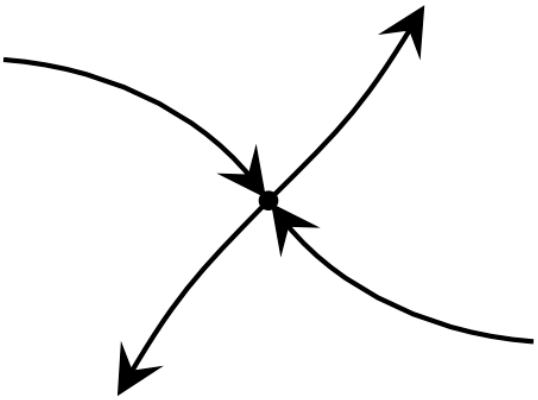
- Stable
  - Attractor



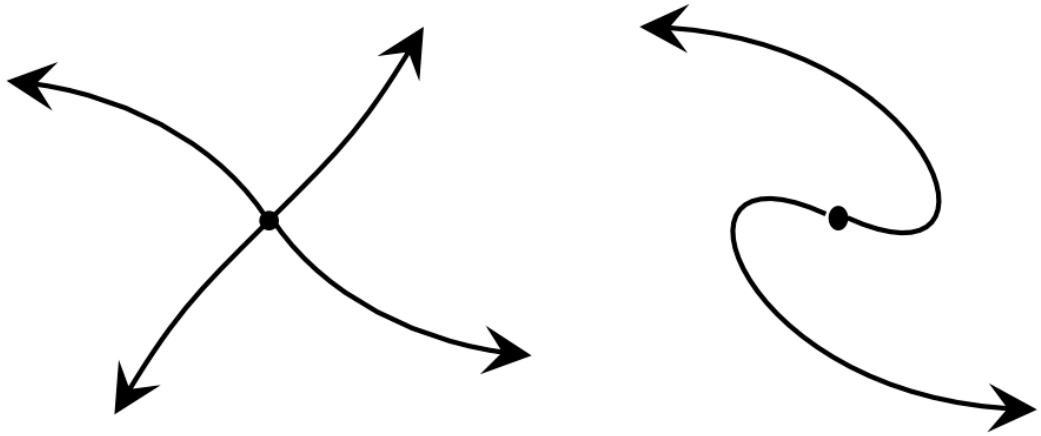
  - Neutrally stable



- Unstable
  - Saddle



  - Repellor



For a linear dynamical system

$$dy/dt = Ay$$

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 [ ]:

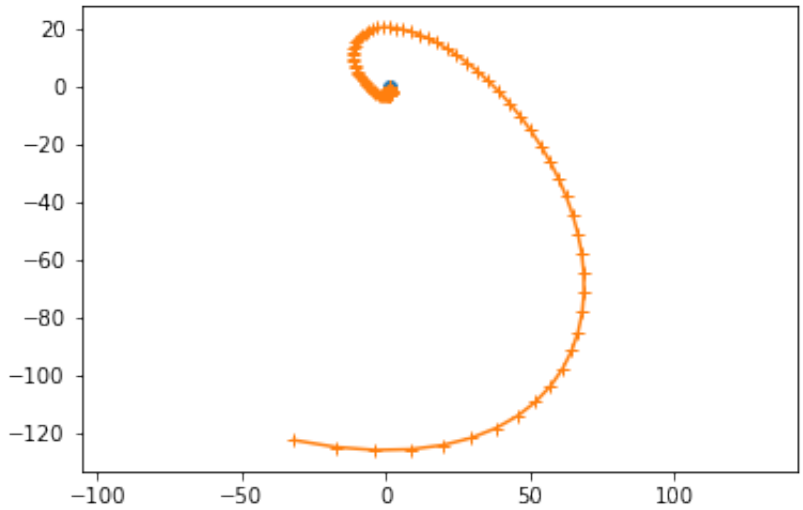
Linear differential equation system

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(np.dot(A, y))
```

In [14]:
```
A = np.array([[1, 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[0,0], y[0,1], 'o')   # starting point
plt.plot(y[:,0], y[:,1], '+-')  # trajectory
plt.axis('equal')
np.linalg.eig(A)
```

Out[15]:
```
(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        ]]))
```



Try different settings of A.

In [16]:
```
A = np.array([[-1, 1], [-1, 0]])
```

In [17]:
```
A = np.array([[1, 1], [-1, 0]])
```

In [18]:
```
A = np.array([[-1, 0], [0, 1]])
```

In [ ]:
```

```

Nonlinear ODEs

While the dynamics of a linear ODE can show only convergence, divergence, or neutrally stable oscillations, nonlinear ODEs can show limit-cycle oscillation and chaos.

Van der Pol oscillator

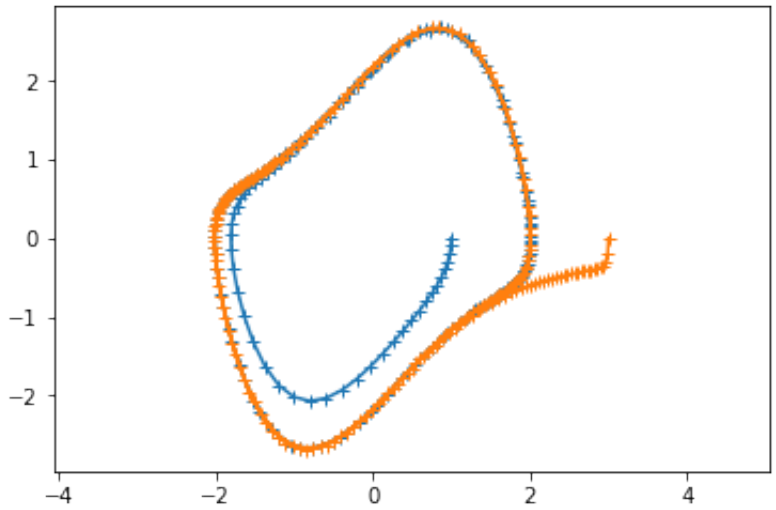
This is a classic equation describing an oscillator circuit with a vacuume tube:

$$dy^2/dt^2 - \mu(1 - y^2)dy/dt + y = 0$$

In [19]:
```
def vdp(y, t, mu):
    """Van der Pol equation"""
    y1, y2 = y
    return(np.array([y2, mu*(1 - y1**2)*y2 - y1]))
```

In [20]:
```
yinit = np.array([[1, 0], [3, 0]])
t = np.arange(0, 20, 0.1)
for y0 in yinit:
    y = odeint(vdp, y0, t, args=(1,))
    plt.plot(y[:,0], y[:,1], '+-')  # trajectory
plt.axis('equal')
```

Out[20]:
```
(-2.263390116298972,
 3.2506376245856652,
 -2.9617599782357917,
 2.946892226839148)
```



In [ ]:
```
```

Periodic orbit and Limit cycle

If a trajectry comes back to itself $y(t+T) = y(t)$ after some period T, it is called a *periodic orbit*.
If trajectories around it converges to a periodic orbit, it is called a *limit cycle*.

Poincaré-Bendixon theorem: In a continuous 2D dynamical system, if a solution stay within a closed set with no fixed point, it converges to a periodic orbit.
It implies that there is no chaos in a continuous 2D dynamic system.

Lorenz attractor

Edward Lorenz derived a simplified equation describing the convection of atmosphere and found that it shows non-periodic oscillation.

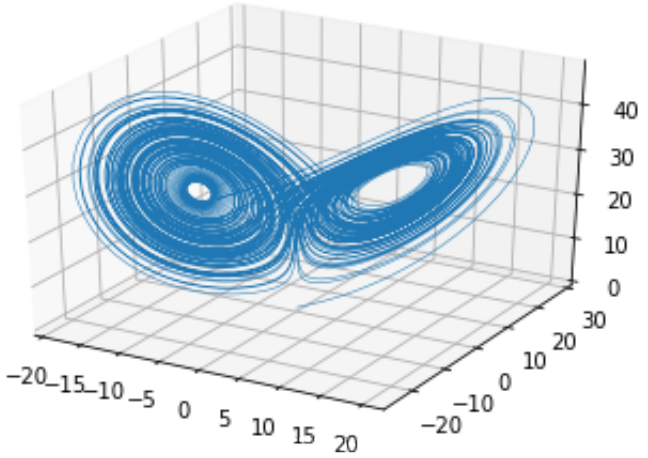
$$dx/dt = p(y - x)$$
$$dy/dt = -xz + rx - y$$
$$dz/dt = xy - bz$$

In [21]:
```
def lorenz(xyz, t, p=10., r=28., b=8./3):
    """Lorenz equation"""
    x, y, z = xyz
    dxdt = p*(y - x)
    dydt = -x*z + r*x - y
    dzdt = x*y - b*z
    return(np.array([dxdt, dydt, dzdt]))
```

In [22]:
```
from mpl_toolkits.mplot3d import Axes3D
```

In [23]:
```
y0 = np.array([0.1, 0, 0])
t = np.arange(0, 100, 0.01)
y = odeint(lorenz, y0, t, args=(10., 28., 8./3))
# Plot in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(y[:,0], y[:,1], y[:,2], lw=0.5)
```

Out[23]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x15103338d0>]
```



In [ ]:

Exercise

1. Linear ODEs

Like the exponential of a real number x is given by

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \ldots = \sum_{k=0}^{\infty} \frac{1}{k!}x^k,$$

the *exponential of a matrix* X

$$e^X = I + X + \frac{1}{2}X^2 + \frac{1}{6}X^3 + \ldots = \sum_{k=0}^{\infty} \frac{1}{k!}X^k.$$

For one dimensional linear ODE

$$dy/dt = ay$$

the solution is given by

$$y(t) = e^{at}y(0),$$

where $y(0)$ is the initial state.

For an n dimensional linear ODE

$$dy/dt = Ay$$

where A is an $n \times n$ matrix, the solution is given by the matrix exponential

$$y(t) = e^{At}y(0),$$

where $y(0)$ is an n-dimensional initial state.

- Verify this by expanding e^{At} accordint to the definition and differentiating each term by t.

In [ ]:

Remember the *eigendecomposition*

$$A = V\Lambda V^{-1}$$

where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is a diagonal matrix of eigenvalues of A and $V = (v_1, \ldots, v_n)$ is a matrix made of corresponding eigenvectors of A in the columns.

The matrix exponential of At can be simplified as

$$e^{At} = Ve^{\Lambda t}V^{-1} = V\mathrm{diag}(e^{\lambda_1 t}, \ldots, e^{\lambda_n t})V^{-1}.$$

Then the solution is given by

$$y(t) = V\mathrm{diag}(e^{\lambda_1 t}, \ldots, e^{\lambda_n t})V^{-1}y(0),$$

where $V^{-1}y(0)$ determines the amplitudes of exponential components and $V = (v_1, \ldots, v_n)$ represents the directions of the components in the state space.

Let us visualize solutions for different eigenvalues.

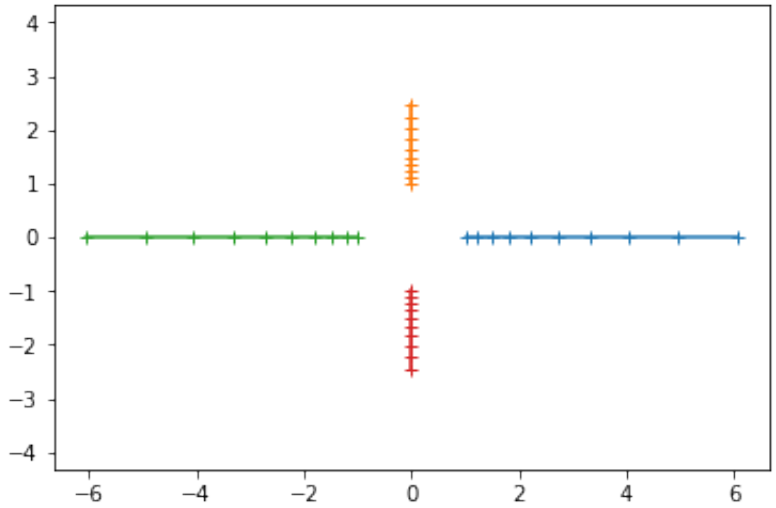
In [24]:
```
def linear(y, t, A):
    """Linear dynamcal system dy/dt = Ay
    y: n-dimensional state vector
    t: time (not used, for compatibility with odeint())
    A: n*n matrix"""
    # y is an array (row vector), A is a matrix
    return(np.dot(A, y))
```

In [25]:
```
def linear2D(A, yinit=np.array([[1,0], [0,1],[-1,0],[0,-1]]), t=np.arar
    """Visualizing linear 2D dynamical system"""
    for y0 in yinit:
        y = odeint(linear, y0, t, args=(A,))
        #plt.plot(y[0,0], y[0,1], 'o')   # starting point
        plt.plot(y[:,0], y[:,1], '+-')  # trajectory
    plt.axis('equal')
    print(np.linalg.eig(A))
```

1) Real eigenvalues $\lambda_1 > \lambda_2 > 0$

In [26]:
```
A = np.array([[2, 0], [0, 1]])  # modift this!
linear2D(A, t=np.arange(0, 1, 0.1))
```

```
(array([ 2.,  1.]), array([[ 1.,  0.],
       [ 0.,  1.]]))
```



2) Real eigenvalues $\lambda_1 > 0 > \lambda_2$

In [ ]:

3) Real eigenvalues $0 > \lambda_1 > \lambda_2$

In [ ]:

4) Complex eigenvalues $\lambda_1 = a + ib$ and $\lambda_2 = a - ib$ with $a > 0$

In [ ]:

5) Complex eigenvalues $\lambda_1 = a + ib$ and $\lambda_2 = a - ib$ with $a < 0$

In [ ]:

Option) Make a 3D version of the function to visualize the trajectories and see how they depend on the three eigen values.

In [27]:
```
from mpl_toolkits.mplot3d import Axes3D
```

In [28]:
```
def linear3D(A, yinit=... ):
```

```
  File "<ipython-input-28-02b364656895>", line 1
    def linear3D(A, yinit=... ):
                                ^
SyntaxError: unexpected EOF while parsing
```

In [ ]:

2. Nonlinear ODEs

1) Implement a nonlinear system, such as a pendulum with friction μ:

$$d\theta/dt = \omega$$
$$ml^2 d\omega/dt = -\mu\omega - mgl\sin\theta$$

In [ ]:
```
def pendulum(y, t, args):
```

In [ ]:

2) Run a simulation by `odeint()` and show the trajectory as (t, y(t))

In [ ]:

3) Show the trajectory in the 2D state space (θ, ω)

In [ ]:

Option) Implement a nonlinear system with time-dependent input, such as a forced pendulum:

$$d\theta/dt = \omega$$
$$ml^2 d\omega/dt = -\mu\omega - mgl\sin\theta + a\sin bt$$

and see how the behavior changes with the input.

In [ ]:

In [ ]:

3. Bifurcation

FitzHugh-Nagumo model is an extension of Van der Pol model to approximate spiking behaviors of neurons.

$$dv/dt = v - \frac{v^3}{3} - w + I$$
$$dw/dt = \phi(v + a - bw)$$

1) Implement a function and see how the behaviors at different input current I.

In [ ]:
```
def fhn(y, t, I=0, a=0.7, b=0.8, phi=0.08):
    """FitzHugh-Nagumo model"""
    v, w = y
```

In [ ]:
```
y0 = np.array([0, 0])
t = np.arange(0, 100, 0.1)
y = odeint(fhn, y0, t, args=(0.5,))
plt.plot(t, y, '.-')  # trajectory
```

In [ ]:
```
plt.plot(y[:,0], y[:,1], '+-')  # phase plot
```

In [ ]:

In [ ]: