Lab 5 – Sage Tutorial

1. Linearization

What?: Linearization refers to the local approximation of a nonlinear function by a linear one. Here, 'local' refers to the fact that the approximation is 'good' only in a (small) neighborhood of a reference point.

Why?: Linear problems are always explicitly solvable (think of algebraic or dfferential equation) while nonlinear problems aren't. A good linear approximation will thus provide valuable insights into the more complicated nonlinear 'geometry'.

How?: The linear approximation is constructed by using the first terms of a Taylor expansion $f(x) = f(x^*) + f'(x^*)(x - x^*) + HOT$ (i.e., by discarding Higher Order Terms)). For vector-valued functions of several variables, $f'(x^*)$ is replaced by its high-dimensional counterpart, the Jacobi matric (or Jacobian).

Sage does provide a specialized command for computing the linearization/Jacobian

```
x, y = var('x y')
f = (2*x-x^2-x*y, -y+x*y)
jacobian(f, (x,y))
```

```
Out[70]: [-2*x - y + 2          -x]
         [           y       x - 1]
```

Notice the fact that this is a (matrix-valued) function (of x and y).

After inserting (different) numerical values for of x and y one obtains (different) matrices

```
In [7]: jac = jacobian(f, (x,y))
        A1 = jac.subs(x=0,y=0)
        A2 = jac.subs(x=2,y=0)
        A3 = jac.subs(x=1,y=1)
        A1, A2, A3
```

```
Out[7]: (
        [ 2  0]  [-2 -2]  [-1 -1]
        [ 0 -1], [ 0  1], [ 1  0]
        )
```

These can be analyzed with standard Linear Algebra methods (eigenvalues, eigenvectors, etc.).

2. Plotting direction fields and orbits

One of the most powerful methods for understanding nonlinear (planar dynamics) is based on representing the direction(vector) filed which generates the dynamics.

Geometrically: the systems orbits are tangent to the generating vector field.

Sage has inbuild commands for computing and plotting both direction fields and orbits.

For **plotting the direction field** we first define the field (RHS of the system) to be plotted (*f* has two components, f[0] and f[1] in **Sage**)

```
In [71]: x, y = var('x y')
         f = (2*x-x^2-x*y, -y+x*y)
```

then use the appropriate plot command (with good limits for the *x* and *y* ranges)

```
In [57]: # the direction field
         plot_vector_field((f[0],f[1]), (x,-1,2.2),(y,-1,1.2))
```

In order to **compute orbits** (as most systems are **not** explicitly solvable in terms of simple functions) one resorts to a numerical ODE solver

```
sol = desolve_odeint(f, ics, times, [x, y])
line(zip(sol[:,0],sol[:,1]))
```

Here, the arguments of the solve command are: 'f' the nonlinear RHS of the equation, the initial conditions 'ics', 'times' a 3-tuple containing the initial time, final time and time step.

The 'line' command is used in generating the orbit (ODE solver only generates a discrete sequence of points).

Finally, one can superimpose orbits over a direction field (using a for loop to generate multiple orbits). Mind the '+=' syntax inside the for loop and the fact that

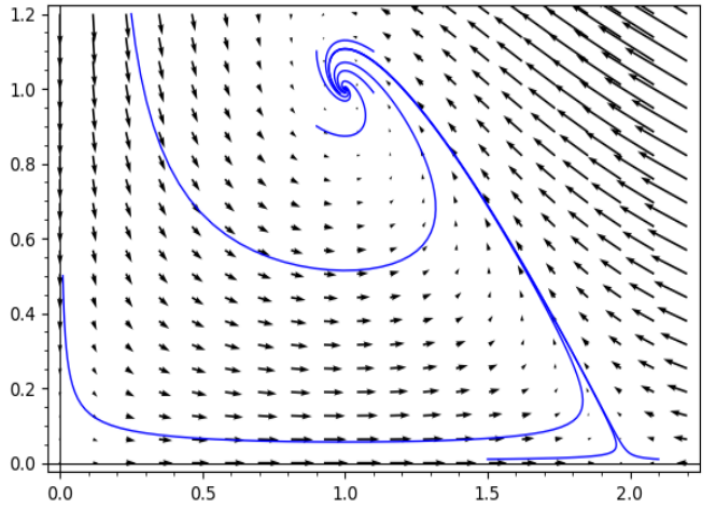
the ‘show’ command must remain outside the for loop (orherwise multiple plots are generated).

```
In [65]: # this is the solve block
times = srange(0,10,0.1)
ics   = (-1.5, 3)
sol = desolve_odeint(f, ics, times, [x, y])

# the direction field
p_tot = plot_vector_field((f[0],f[1]), (x,0,2.2),(y,0,1.2))

# orbits are added to the same plot
for ics in [(0.01, 0.5), (2.1,0.01), (1.5,0.01), (0.9,1.1), (1.1, 1.1), (1.1, 0.99), (0.9,0.9), (0.25, 1.2)]:
    sol = desolve_odeint(f, ics, times, [x, y])
    p_tot += line(zip(sol[:,0],sol[:,1]))

show(p_tot)
```



3. Representing level curves using contour plots

First integrals of (nonlinear) planar systems can be represented/visualized in terms of their level curves, just as geographical landscapes are plotted on maps (this is why sometimes mathematicians use expressions like ‘energy landscape’).

Sage has a predefined command, called ‘contour_plot’ for plotting level curves.

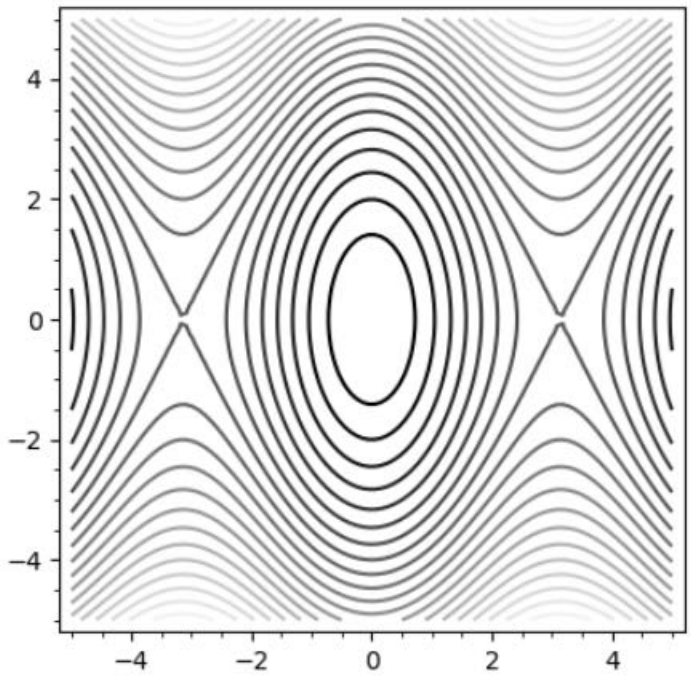
The arguments of the command include: the function to be plotted, the *x* and y ranges, the number of level curves to be plotted as well as visual options such as ‘fill’.

Obviously, the scalar-valued function (energy) to be plotted must be defined first.

In [3]:
```
x, y = var('x y')
H(x,y) = y^2 -8*cos(x)

# we can plot the level curves of H
contour_plot(H, (-5,5), (-5,5), fill=False, contours=20)
```

Out[3]:



Changing the graphical appearance options can be done in numerous ways

In [7]:
```
x, y = var('x y')
H(x,y) = y^2 -8*cos(x)

# we can plot the level curves of H
contour_plot(H, (-5,5), (-5,5), contours=20)
```

Out[7]: