Workshop 5: Further graphs

So far, you have only plotted the standard 'graph of a function' type of graph, but today we shall meet other types of graph.

Important note: In this workshop, there will be some lengthy blocks of code that will need to be run so that we can plot our own graphs in a much easier way. To those of you who are new to programming, these blocks of code may look a little intimidating. Don't worry, you are not expected to understand them! However if you are interested in programming, feel free to read through these bits of code - it's all good practice and you may be able to follow more than you think!

Any block of code that starts with the comment:

In [1]:

```
## Background code ##
```

does not have to be understood, or even read by you. However, it will still need to be run!

If you haven't already, now is a good time to go to the top toolbar and click: `Cell -> Run All`.

In [2]:

```
## Background code ##

import matplotlib.pyplot as plt
import numpy as np
from sympy import *
from numpy import pi      # override SymPy object "pi"
%matplotlib inline
init_printing()

x,y = symbols('x y')

def eqn_to_mesh(eqn,X,Y):
    '''For substituting SymPy symbols x,y into Numpy meshgrids X,Y.'''

    str_eqn = str(eqn)
    sub_eqn = ''

    count = 0
    for i in str_eqn:

        # Change case of x and y:
        if i == 'x' and str_eqn[count+1] != 'p':
            sub_eqn += 'X'
        elif i == 'y':
            sub_eqn += 'Y'

        # Change trig functions to Numpy trig functions:
        elif i == 's' and str_eqn[count+1] == 'i':
```

```
            sub_eqn += 'np.s'

        elif i == 'c' and str_eqn[count+1] == 'o':
            sub_eqn += 'np.c'
        elif i == 't' and str_eqn[count+1] == 'a':
            sub_eqn += 'np.t'

        # Change other functions to Numpy functions:
        elif i == 's' and str_eqn[count+1] == 'q':
            sub_eqn += 'np.s'
        elif i == 'e' and str_eqn[count+1] == 'x':
            sub_eqn += 'np.e'

        # Keep all other characters the same:
        else:
            sub_eqn += i
        count += 1

    return eval(sub_eqn)


def move_spines():
    '''For centering the axes of a graph.'''

    ax = plt.gca()
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.spines['bottom'].set_position(('data',0))
    ax.yaxis.set_ticks_position('left')
    ax.spines['left'].set_position(('data',0))

    for label in ax.get_xticklabels() + ax.get_yticklabels():
        #label.set_fontsize(16)
        label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65))

    return None
```

5.1. 3-dimensional plots

In [3]:

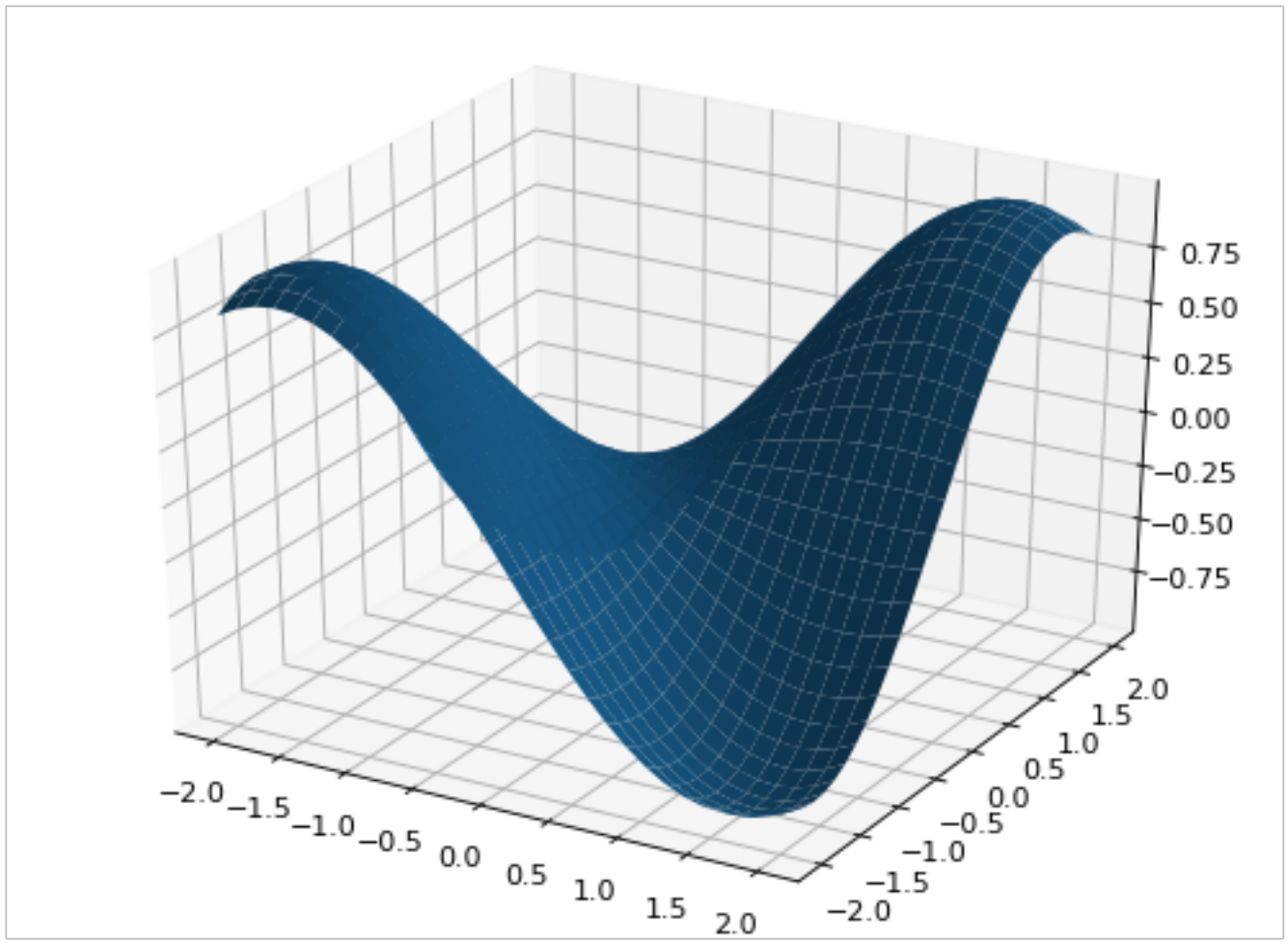
```
## Background code ##

from mpl_toolkits.mplot3d import axes3d

x,y,z = symbols('x y z')

def plot3d(f,xy_range,samples=30):
    xy_lower,xy_upper = xy_range[0],xy_range[1]

    lin = np.linspace(xy_lower,xy_upper,samples)
    sam = np.ones(samples)

    X = np.outer(lin,sam)
    Y = X.copy().T
    Z = eqn_to_mesh(f,X,Y)

    plt.figure(figsize=(8,6), dpi=80)
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z)

    return None
```

If you have a function $f(x, y)$, and you want to see it as a 'height map', then you take your third dimension z and set $z = f(x, y)$. You can do this in Python to see the function, by calling `plot3d`.

The command `plot3d` takes two compulsory arguments: a function $f(x, y)$, and a range for x and y. For example:

In [4]:

```
plot3d(sin(x)*sin(y),[-2.0,2.0])
```



Note: In later functions, you will have the option to set different ranges for x and y, but for `plot3d` x and y always take the same range.

Warning: All graph plotting commands discussed in this workshop should be assumed to be case-sensitive. This means that when you input a function in terms of x and y, you should make sure you have typed `x` and `y` and **not** `X` and `Y`!

5.2. Implicit plots

In [5]:

```
## Background code ##

def implicitplot(f, x_range, y_range, res=1000, center_axes=False, overlay=Fal
se):

    if overlay == False:
        plt.figure(figsize=(8,6), dpi=80)

    x_lower,x_upper = x_range[0],x_range[1]
    y_lower,y_upper = y_range[0],y_range[1]

    Y,X = np.ogrid[y_lower:y_upper:res*1j,x_lower:x_upper:res*1j]
    Z = eqn_to_mesh(f,X,Y)

    plt.contour(X.ravel(),Y.ravel(),Z,[0])
```

```
    if center_axes: move_spines()


    return None


def implicitplot3d(f, xyz_range=[-2.5,2.5]):
    ''' create a plot of an implicit function
    f  ...implicit function (plot where fn==0)
    xyz_range ..the x,y,and z limits of plotted interval'''

    # Lambda function error handling
    try:
        f(1,1,1)
    except TypeError:
        print("Error. Please ensure f is written as a lambda function.")
        return None

    xmin,xmax = xyz_range[0],xyz_range[1]
    ymin,ymax = xyz_range[0],xyz_range[1]
    zmin,zmax = xyz_range[0],xyz_range[1]

    # Graph grid set up
    fig = plt.figure(figsize=(8,6), dpi=80)
    ax = fig.add_subplot(111, projection='3d')
    R = np.linspace(xmin, xmax, 100) # resolution of the contour
    S = np.linspace(xmin, xmax, 15) # number of slices
    G1,G2 = np.meshgrid(R,R) # grid on which the contour is plotted

    for z in S: # plot contours in the XY plane
        X,Y = G1,G2
        Z = f(X,Y,z)
        cset = ax.contour(X, Y, Z+z, [z], zdir='z')
        # [z] defines the only level to plot for this contour for this value o
f z

    for y in S: # plot contours in the XZ plane
        X,Z = G1,G2
        Y = f(X,y,Z)
        cset = ax.contour(X, Y+y, Z, [y], zdir='y')

    for x in S: # plot contours in the YZ plane
        Y,Z = G1,G2
        X = f(x,Y,Z)
        cset = ax.contour(X+x, Y, Z, [x], zdir='x')

    # set plot limits
    ax.set_zlim3d(zmin,zmax)
    ax.set_xlim3d(xmin,xmax)
    ax.set_ylim3d(ymin,ymax)

    plt.show()

    return None
```

You have two functions of two variables, $F(x, y)$ and $G(x, y)$, or of three variables, $U(x, y, z)$ and $V(x, y, z)$, and would like to plot the curve $F(x, y) = G(x, y)$ or surface $U(x, y, z) = V(x, y, z)$.

(Note that these are, in fact, level curves and surfaces respectively, since they can be re-arranged into the form $F(x, y) - G(x, y) = 0$ etc.)

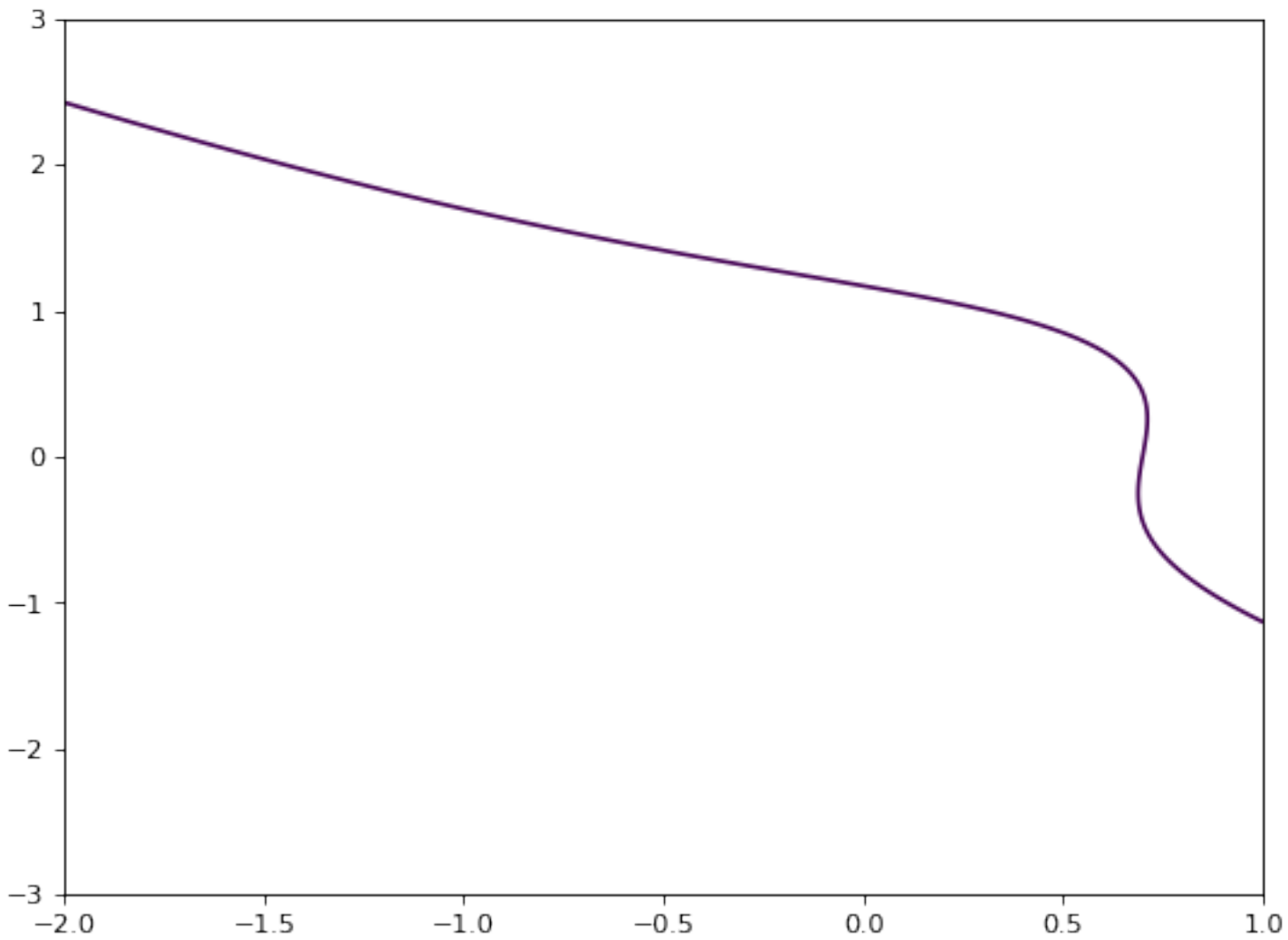
You can do this by calling the `implicitplot` (for two-dimensional graphs) or `implicitplot3d` (for three-dimensional graphs).

When we call `implicitplot`, we need to pass an f, which should be a function written in terms of x and y. This function should be equal to zero, but take care not to include an equals sign. So, for example, if you have $p(x, y) = 4$, rearrange to give $p(x, y) - 4 = 0$ and use this new formulation $q(x, y) = p(x, y) - 4$ as your input.

We also need to pass a range for x and a range for y. As before, these ranges should be enclosed in square brackets. Below we have an example:

In [6]:

```
implicitplot(3*x**3+10*x-2*x**2*y+5*y**3-8,[-2.0,1.0],[-3.0,3.0])
```



For 3D plots things are a little more complicated. For this case we need to pass two arguments; a function f the same as before, and a plot range for x, y, z. We give a single plot range which is then applied to all three (i.e. the grid is always a cube).

This sounds simple enough, but there is a catch when passing f as the first argument - we need to write `lambda x,y,z:` before it. So we call `implicitplot3d` as follows:

```
implicitplot3d((lambda x,y,z: f), plot_range)
```

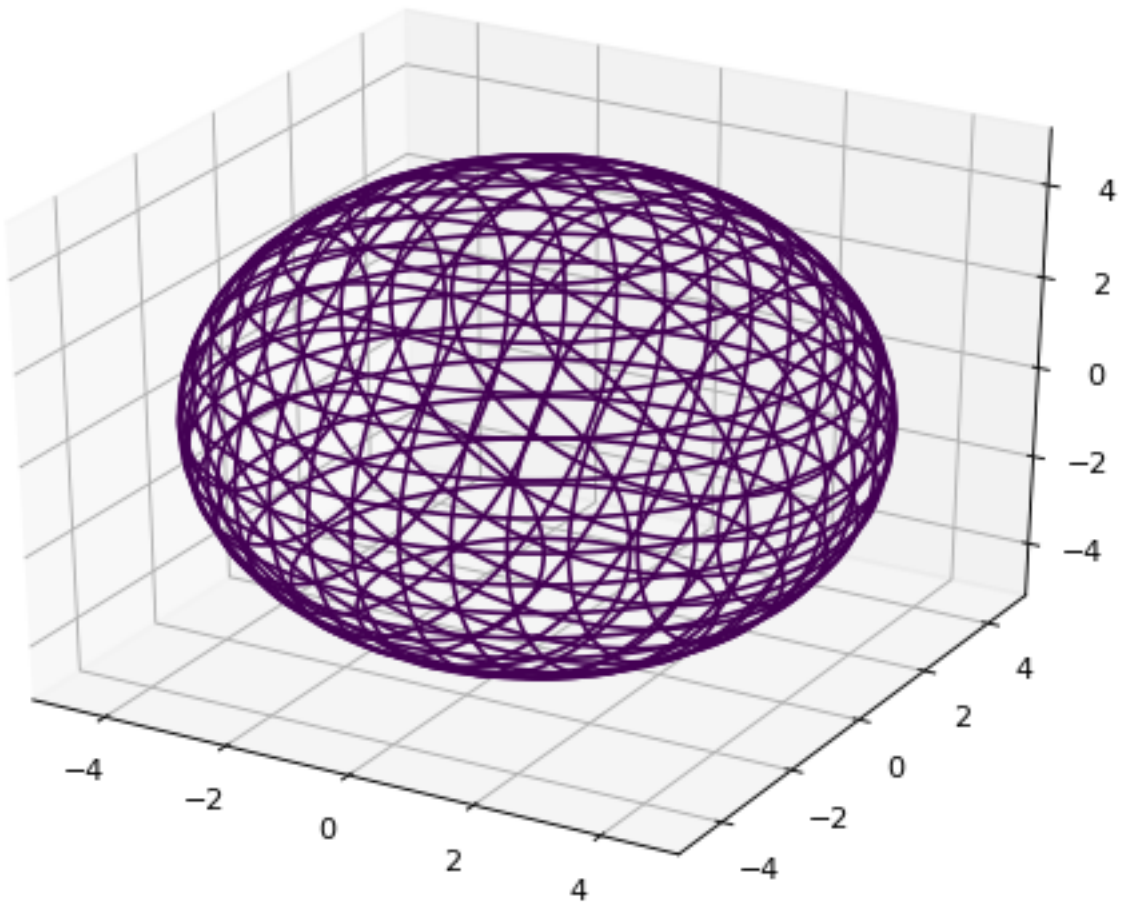
Note: Roughly speaking, the code `lambda x,y,z:` tells Python to write a "temporary" function that does not need to be given a name. This is most likely the only time you will ever see a *lambda* function on this course, but if you are interested in learning more they will be covered in greater depth in *MATH1920 - Computational Mathematics*.

Below is an example of how to use `implicitplot3d` to implicitly plot a sphere:

In [7]:

```
implicitplot3d((lambda x,y,z: x**2 + y**2 + z**2 - 5.0**2),[-5.0,5.0])
```

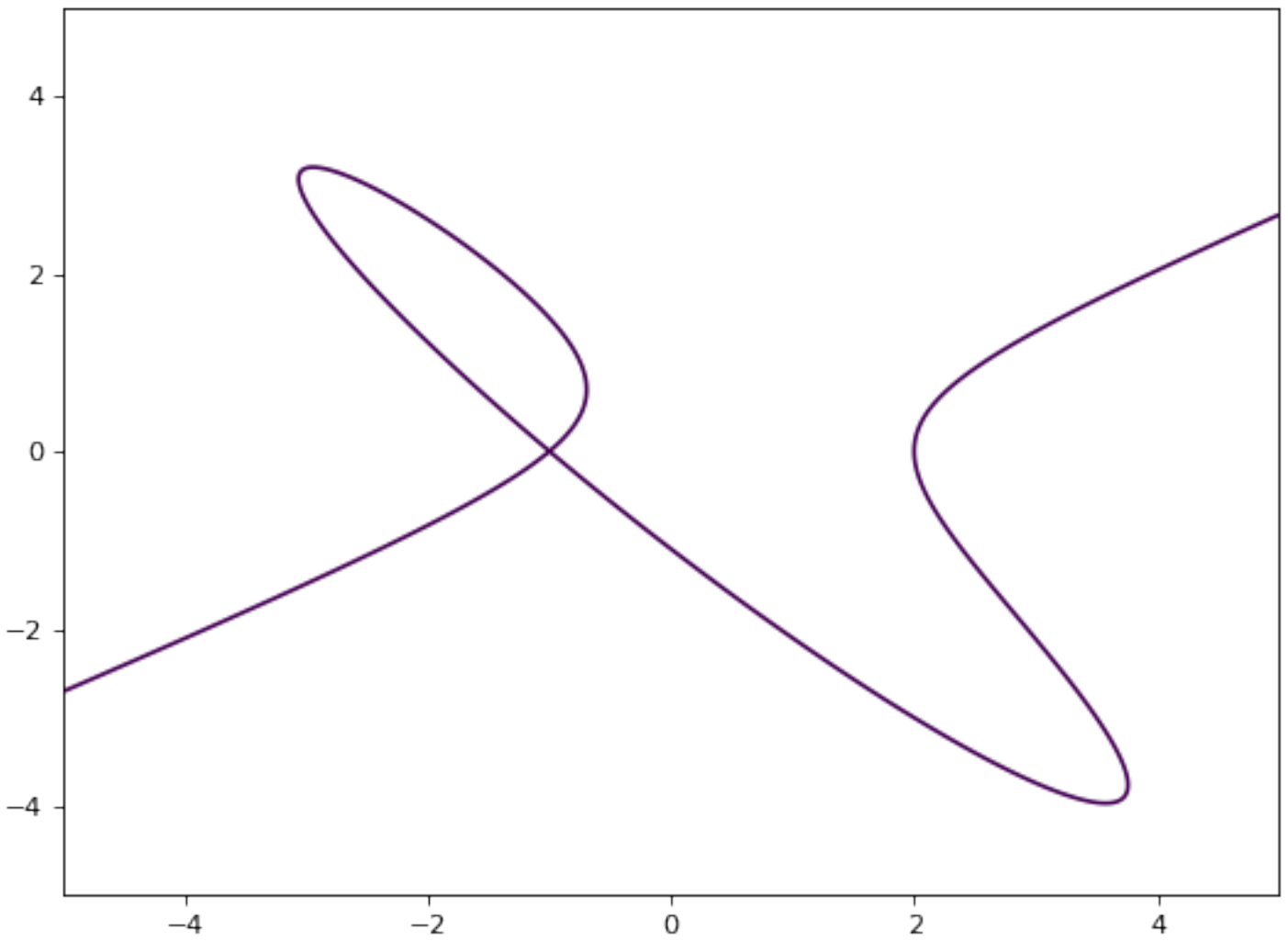
```
/Users/elliesleightholm/anaconda3/lib/python3.7/site-packages/ipyk
ernel_launcher.py:47: UserWarning: No contour levels were found wi
thin the data range.
/Users/elliesleightholm/anaconda3/lib/python3.7/site-packages/ipyk
ernel_launcher.py:53: UserWarning: No contour levels were found wi
thin the data range.
/Users/elliesleightholm/anaconda3/lib/python3.7/site-packages/ipyk
ernel_launcher.py:58: UserWarning: No contour levels were found wi
thin the data range.
```



Use these commands to find the level curve $4x^3 - 12x - 9xy^2 + 6y^3 = 8$, and the level surface $x^2 - y^2 + z^2 = 1$, for $|x| \leq 5$, $|y| \leq 5$ and (for the 3-D plot) $|z| \leq 5$.

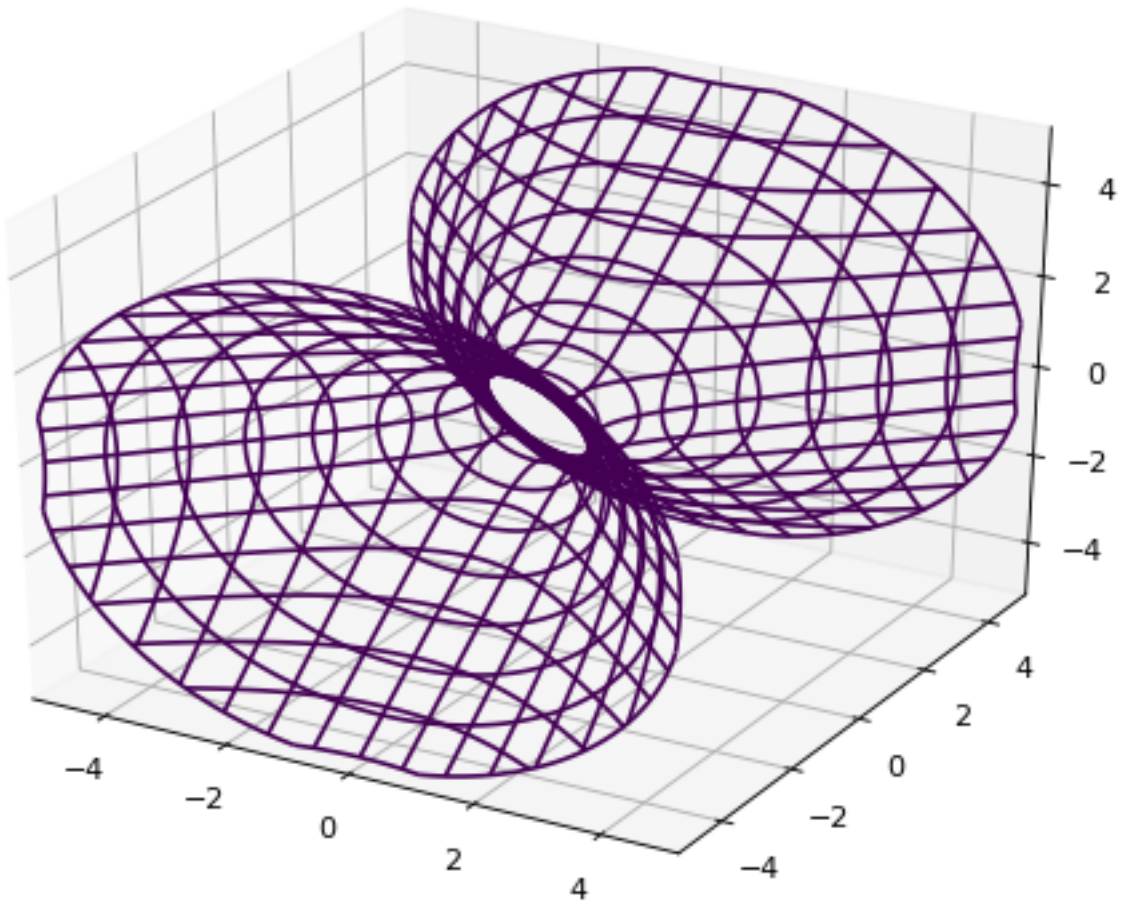
In [15]:

```
implicitplot(4*x**3 - 12*x - 9*x*y**2 - 6*y**3 -8,[-5.0,5.0],[-5.0,5.0])
```



In [18]:

```
implicitplot3d((lambda x,y,z: x**2 - y**2 + z**2 -1),[-5.0,5.0])
```



5.3. Contour plots

In [19]:

```
## Background code ##

def contourplot(f,x_range,y_range,delta=0.01,center_axes=False,overlay=False):

    if overlay == False:
        plt.figure(figsize=(8,6), dpi=80)
    plt.xlim(x_range[0],x_range[1])
    plt.ylim(y_range[0],y_range[1])

    x_lower,x_upper = x_range[0],x_range[1]
    y_lower,y_upper = y_range[0],y_range[1]
    x_vals = np.arange(x_lower,x_upper+delta,delta)
    y_vals = np.arange(y_lower,y_upper+delta,delta)
    X,Y = np.meshgrid(x_vals,y_vals)

    Z = eqn_to_mesh(f,X,Y)

    plt.contour(X,Y,Z)

    if center_axes: move_spines()

    return None
```

A particularly useful kind of implicit plot is the contour plot. This plots a family of level curves for functions $f : \mathbb{R}^2 \to \mathbb{R}$.

To use `contourplot` you need to provide the same 3 arguments as `implicitplot`:

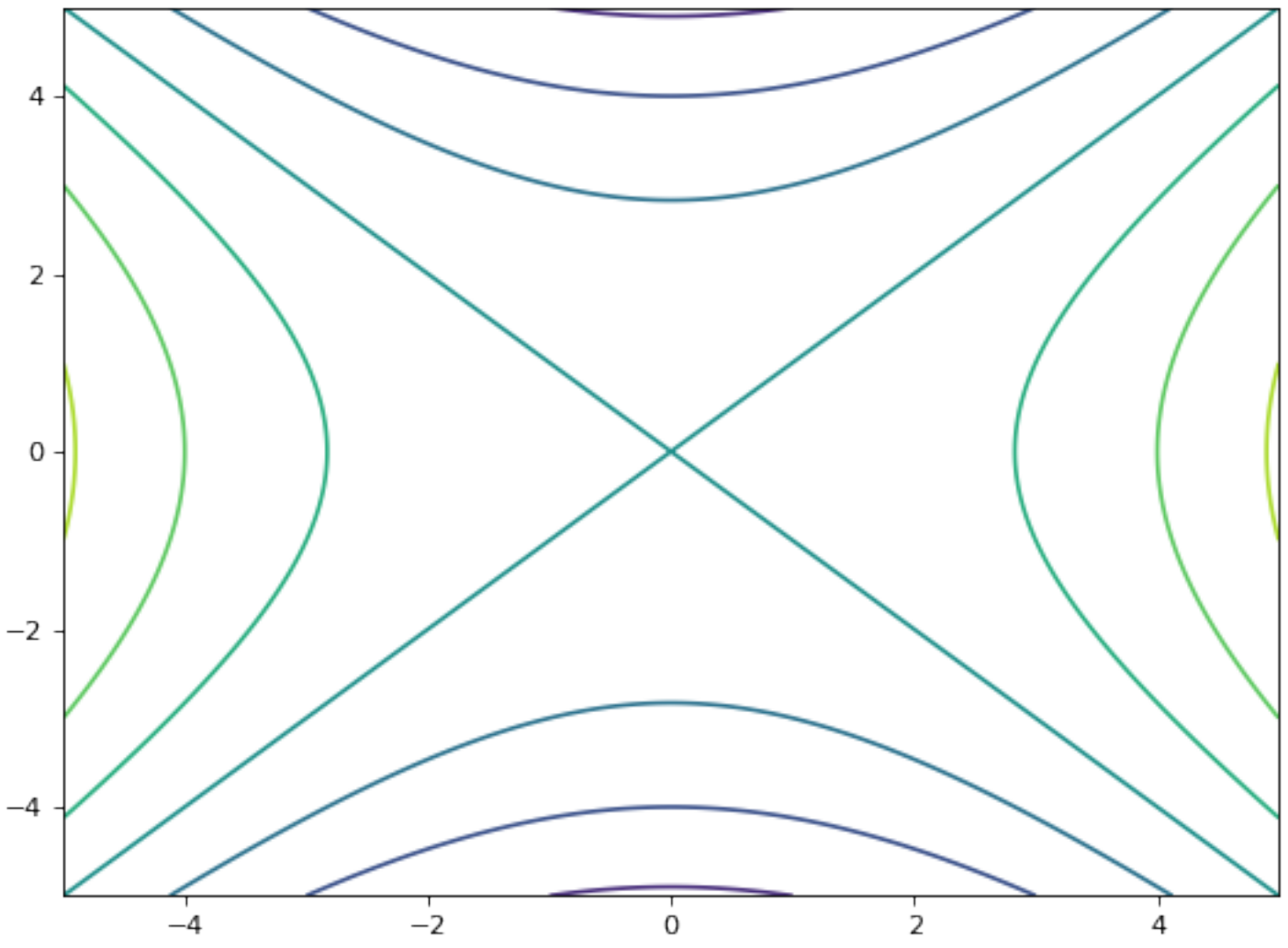
- a function (in terms of x and y)
- a range for x
- a range for y.

Additionally, you may also set a value for `delta`, which is the step size between points. By default `delta` is set to 0.01.

An example is given below:

In [20]:

```
contourplot(x**2-y**2, [-5.0,5.0],[-5.0,5.0])
```



5.4. Gradient plots

In [21]:

```
## Background code ##

def gradplot(f,x_range,y_range,delta=False,colour='r',
             center_axes=False,overlay=False,stream=False):

    if overlay == False:
        plt.figure(figsize=(8,8), dpi=80)
    plt.xlim(x_range[0],x_range[1])
    plt.ylim(y_range[0],y_range[1])

    x_lower,x_upper = x_range[0],x_range[1]
    y_lower,y_upper = y_range[0],y_range[1]

    if delta == False:
        set_delta = (x_upper-x_lower)/20.0
    elif type(delta) == float:
        set_delta = delta
    else:
        print("Please enter a valid number (float) for set_delta.")
        return None

    x_vals = np.arange(x_lower,x_upper+set_delta,set_delta)
    y_vals = np.arange(y_lower,y_upper+set_delta,set_delta)
    X,Y = np.meshgrid(x_vals,y_vals)

    # Find grad(f), and turn into functions
    # for broadcasting over meshgrid
    fx = diff(f,x)
    fy = diff(f,y)
    ffx = lambdify((x,y),fx, "numpy")
    ffy = lambdify((x,y),fy, "numpy")

    # Compute grad(f) vector field meshgrid
    U, V = ffx(X,Y), ffy(X,Y)

    if stream:
        plt.streamplot(X,Y,U,V,color=colour)
    else:
        plt.quiver(X,Y,U,V,color=colour)

    if center_axes: move_spines()

    return None
```

You know that if $\phi : \mathbb{R}^n \to \mathbb{R}$ is a scalar field, then $\nabla\phi : \mathbb{R}^n \to \mathbb{R}^n$ is a vector field. And for $n = 2$, we can visualise this gradient field, which shows the derivative of ϕ. The relevant graphing command is `gradplot`.

Similar to `contourplot` and `implicitplot`, to use `gradplot` you need to provide 3 arguments:

- a function (in terms of x and y)
- a range for x
- a range for y.

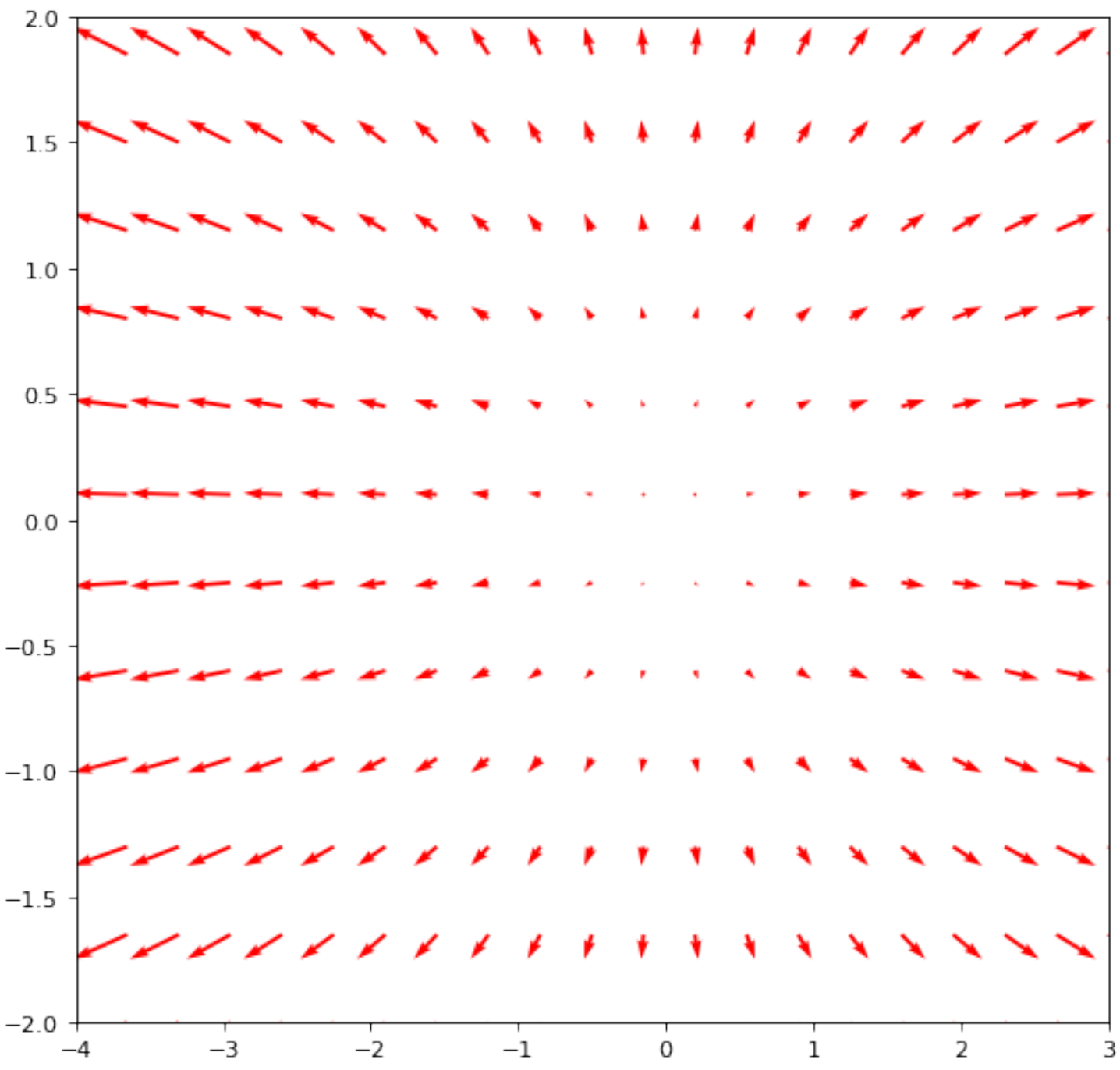
By default the line colour is set to red, and `delta` is set to be automatic.

Note: You will probably never need to change these default values, but if you would like to alter the number and size of arrows, then you should change `delta` to a sensible floating point number.

An example is given below:

In [22]:

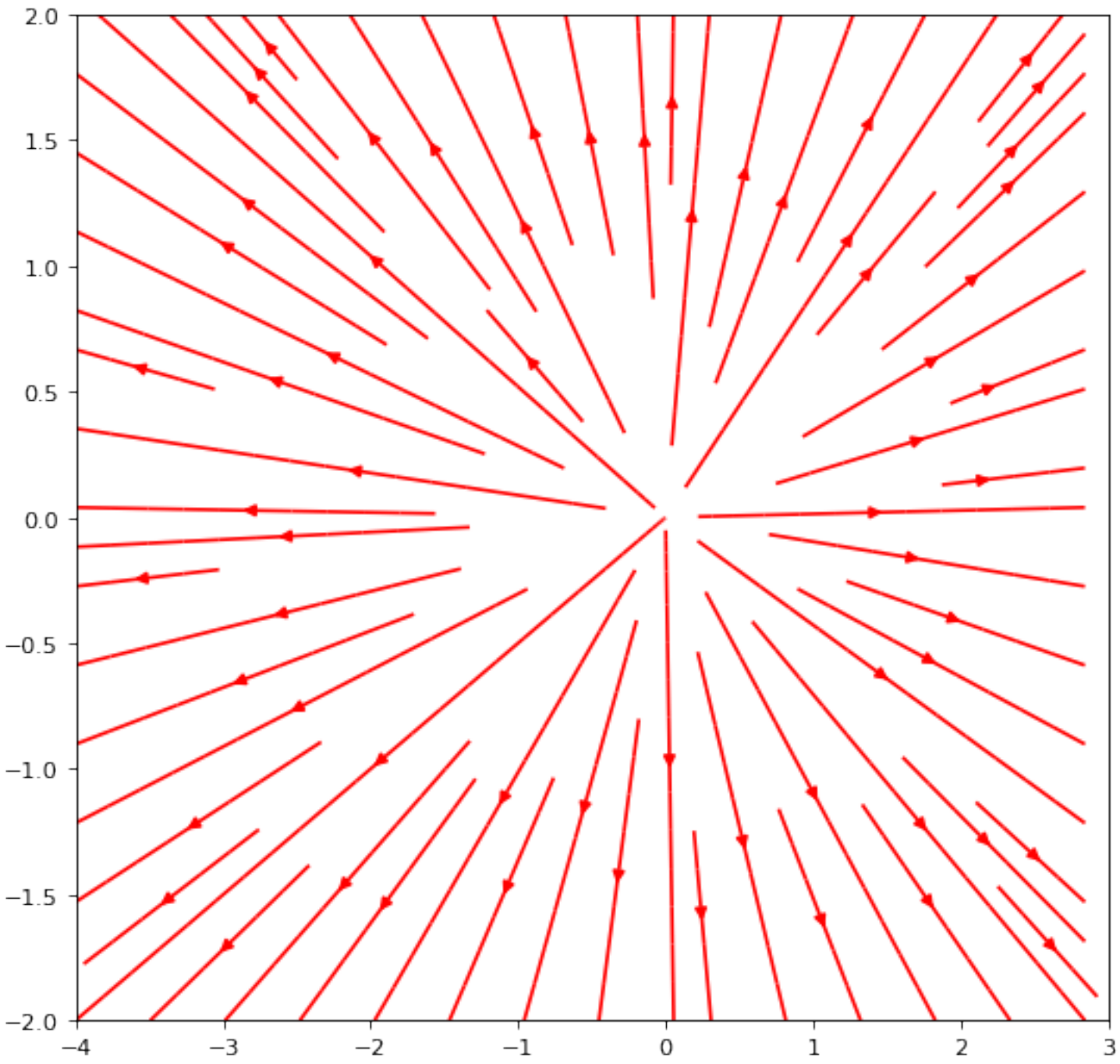
```
gradplot(x**2+y**2,[-4.0,3.0],[-2.0,2.0])
```



If you would like your gradient plot to be joined into smooth lines, you can set simply set `stream = True`:

In [23]:

```
gradplot(x**2+y**2,[-4.0,3.0],[-2.0,2.0],stream=True)
```



5.5. Plot-valued functions

This is a very handy, if rather bizarre, combination of Python's data structures. You already know that one way you can define mathematical functions is to take a variable and map it to something else, like another number, by writing:

`f` = *something involving* `x`

`f.subs(x,a)`.

You have also just been introduced to *lambda* functions (in section 5.2), which are used more generally as 'temporary' functions. Another way to define a function, which is arguably the most useful, is to use `def`. This will be discussed in much more detail in **Workshop 8 (Advanced)**, but the general syntax is to declare:

`def name(arg_1, arg_2, etc.):`

where `name` is your chosen name of the function, and `arg_1`, `arg_2`, *etc*. are the required inputs of your function. Take note of the colon, it is very important!

In the next lines that follow, you must give Python some instructions as to what it should do with the given inputs.

Finally, we end a function with a `return` and an output. If you are plotting graphs (as we are) you can just set this to `None`.

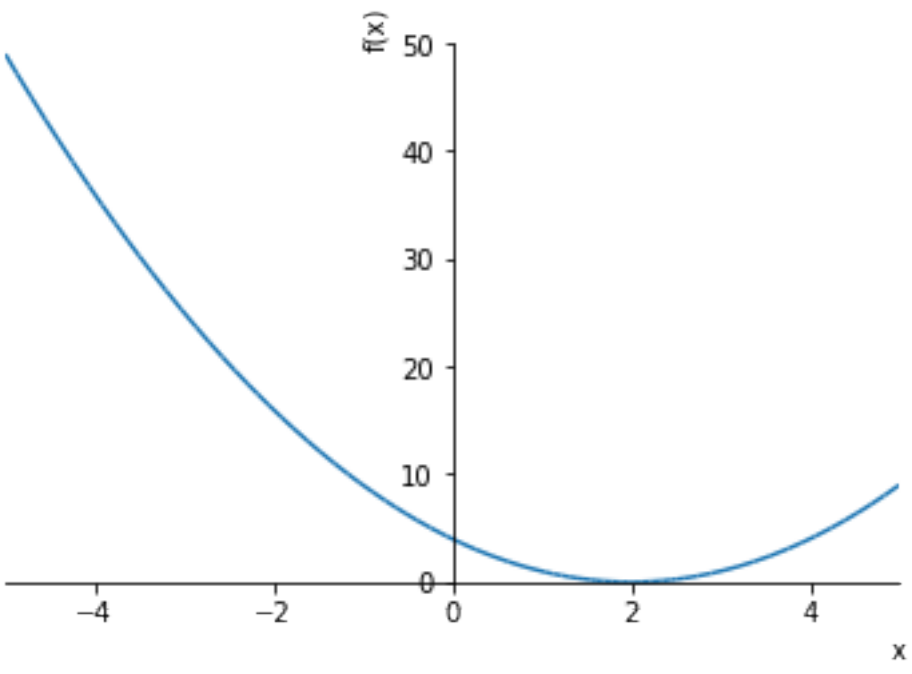
So what is the point of this? Well, it means we can define a plot-valued function which takes a parameter and uses that to plot a graph with that parameter. Here is an example:

In [24]:

```
def valplot(a):
    plot((x-a)**2,(x,-5,5))
    return None
```

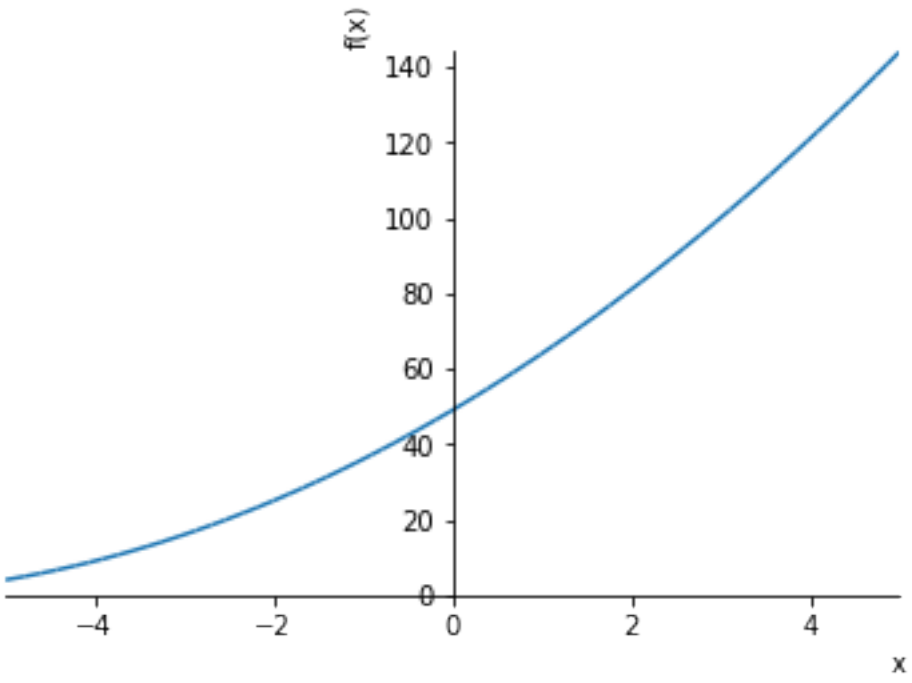
In [25]:

```
valplot(2)
```



In [26]:

```
valplot(-7)
```



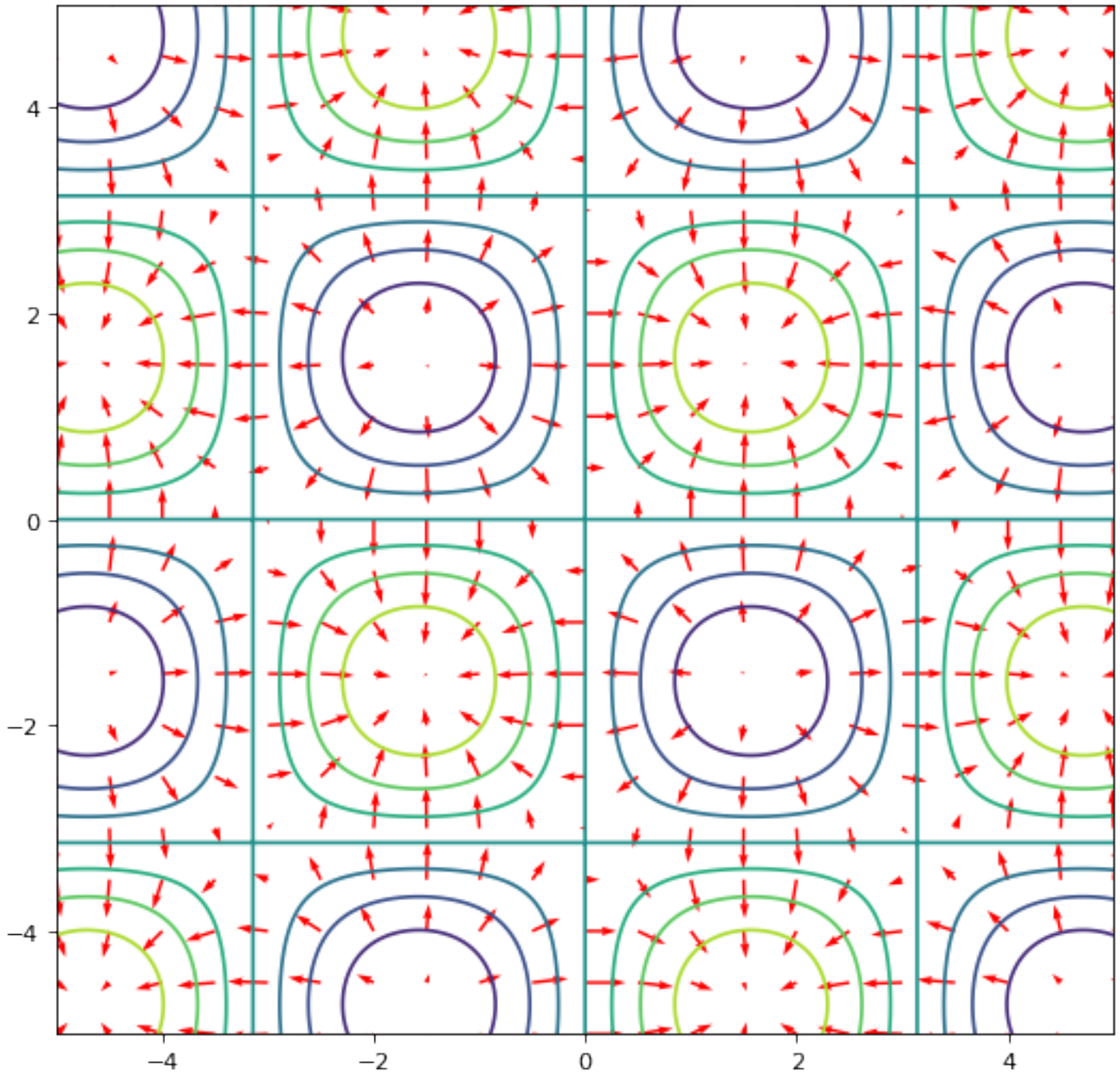
5.6. Multiple plots in one graph

We have already seen how to construct a graph of multiple plots when we are plotting using *SymPy* (recall workshop 1: `plot(f(x),g(x),h(x),(x,-pi,pi))`).

To plot *matplotlib* graphs together, we run desired commands in the same code block, and set the `overlay` argument to `True` for all plot commands but the first. So for example:

In [27]:

```
gradplot(sin(x)*sin(y), (-5,5), (-5,5))
contourplot(sin(x)*sin(y), (-5,5), (-5,5), overlay=True)
```



(Note that, as you have learnt in lectures, lines of gradient are perpendicular to contour lines. However, whether this can be seen in a specific graph depends on the aspect ratio of the image! The `gradplot` function has been coded to produce a square image, so you just have to make the ranges the same and call `gradplot` first in order to get an appropriate image.)

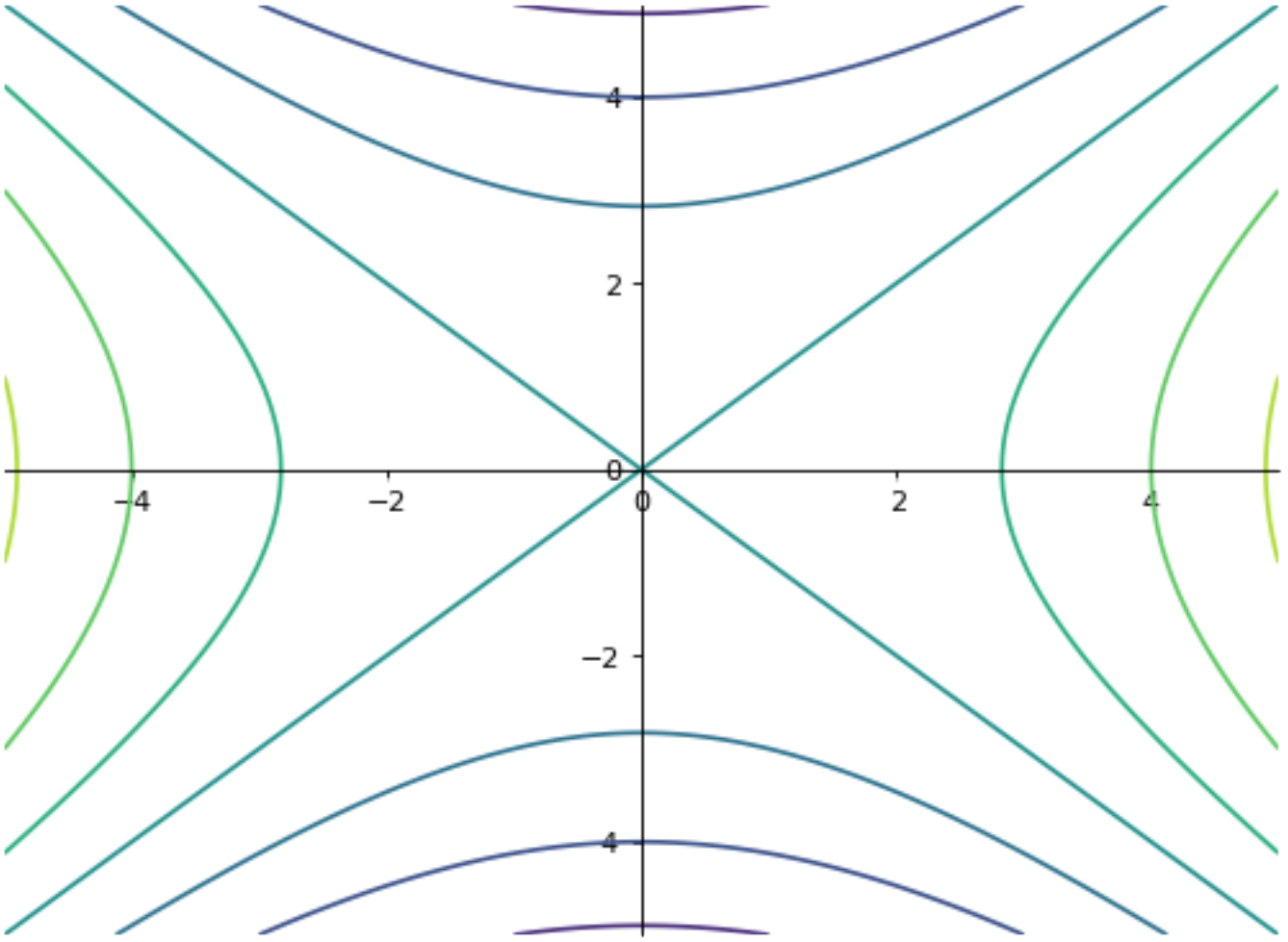
5.7. Changing the axes of a graph

For the commands: `implicitplot`, `contourplot` and `gradplot` we can center the axes of the graphs by setting the additional argument: `center_axes = True`.

For example:

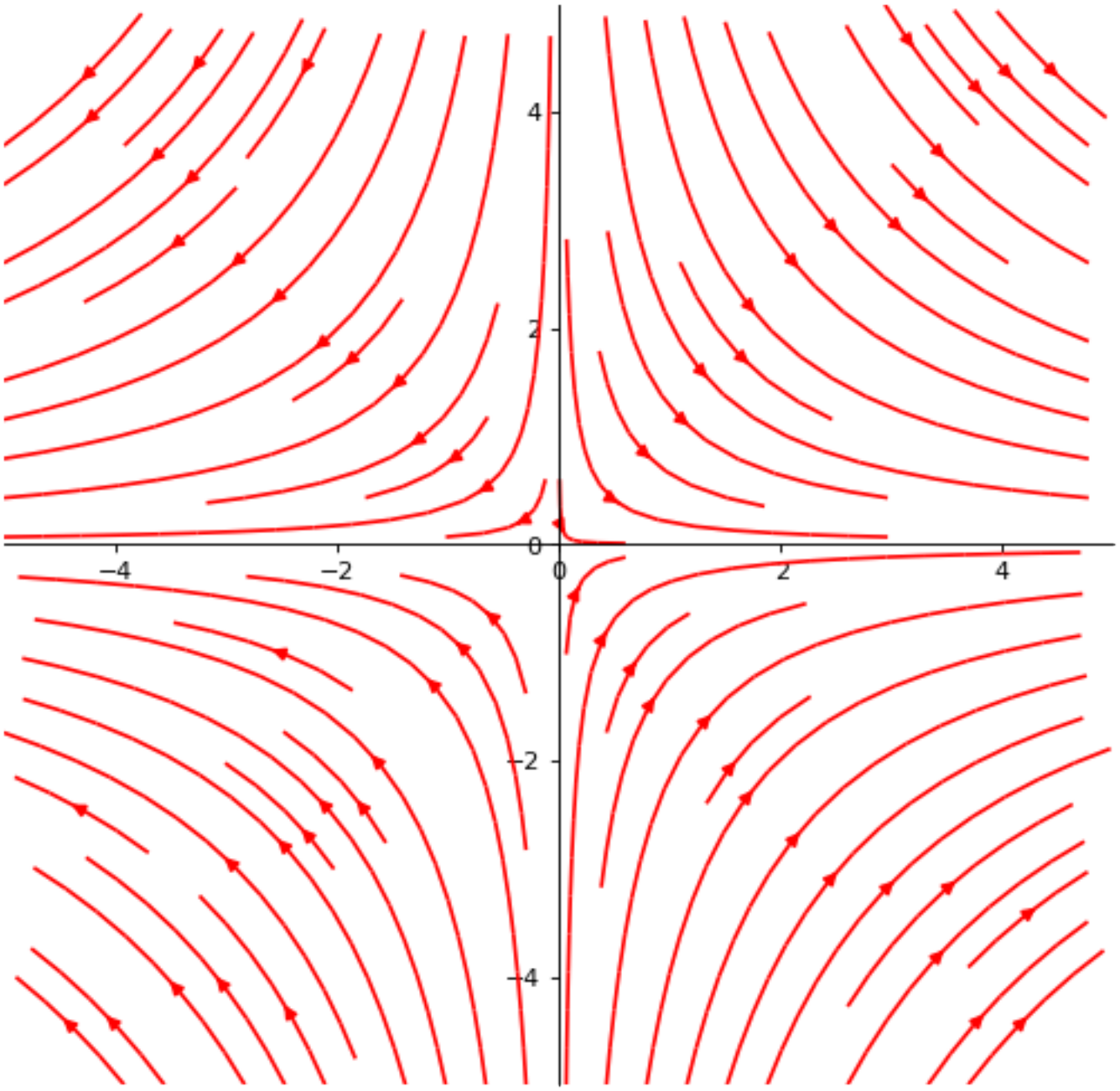
In [28]:

```
contourplot(10*x**2-10*y**2, [-5.0,5.0], [-5.0,5.0], center_axes=True)
```



In [29]:

```
gradplot(x**2 - y**2, [-5.0,5.0], [-5.0,5.0], center_axes=True, stream=True)
```



5.8. Further practice

If you would like to know how to alter your graphs further, or would like to understand more about what it is going on in the sections marked as *background code*, then you may want to look at the links below.

The official matplotlib website: http://matplotlib.org/ (http://matplotlib.org/)

A nice tutorial in matplotlib: http://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html (http://www.labri.fr/perso/nrougier/teaching/matplotlib/matplotlib.html)

In [ ]: