Python For Lecture 1

December 4, 2020

1 Plotting Direction Fields

Our primary computational tool will be Python using Jupyter notebooks. I'll point out at the beginning that this is not a programming class and I will often sacrifice good programming principles to make the things more clear.

1.1 Direction Fields

A **direction field** is a particular type of **vector field**. Recall that a two dimensional vector field on R^2 is a function that assigns to each point (t, x) in $\mathbb{R}^2$ a vector denoted, for example, by $\langle u(t,x), v(t,x)\rangle$. In the context of ODEs, the vector field is constructed in the following way: If $(t, \varphi(t))$ is a solution to the ODE $x' = f(t, x)$, then the vector field is :

$$\langle u(t,x), v(t,x)\rangle = \langle 1, \varphi'(t)\rangle = \langle t, f(t,x)\rangle.$$

Of course, when drawing a vector field using python (or any software) we're not going to draw a vector at every point of $\mathbb{R}^2$. Instead, we'll do it at a regurlarly spaced set of points over a rectangular grid $[t_0, t_1] \times [x_0, x_1]$. Decisions like what rectangular grid to choose and how many points to sample the direction field at are domain specific. For example, if $x(t)$ represents the population of field mice, we wouldn't want to include any negative values of x in the sampling.

The way numpy does this is using meshgrids. First, we define two arrays:

[1]:
```
import numpy as np
t = np.arange(0,10,1)
x = np.arange(0,2,.5)
```

To run a block of code, press Shift+Enter. The `import numpy` allows us to use the functionality of numpy. If this is all we did, then we would have to write things like `numpy.arange(0,10,1)`. The `as np` allows us to refer to `numpy` as `np`. These import statements usually go at the top of the Jupyter notebook, and in the future this is what we will do. We will have other import statements below that would usually go at the top as well.

To see the value of a variable you can use the print command:

[2]:
```
print (f"t={t}")
print(f"x={x}")
```

```
t=[0 1 2 3 4 5 6 7 8 9]
x=[0.  0.5 1.  1.5]
```

The expression `np.arange(0,10,1)` gives an array from 0 to 10 (not including 10) that is evenly spaced with step size 1. In general `np.arange(a,b,h)` gives an evenly spaced array from `a` to `b` (but not including `b`) with step size equal to `h`.

The array `t` has length 10 and the array `x` has length 4. The two arrays together encode $4 \times 10 = 40$ ordered pairs in the (t, x) plane at which to draw a vector field. The way that numpy organizes this data is via a "Meshgrid":

[3]:
```
T, X = np.meshgrid(t, x)
```

This assigns to `T` and `X` a 4×10 matrix. Each **row** of the `T` matrix is the vector `t` and each **column** of the `X` matrix is the vector `x`. That is:

[4]:
```
print(f"T = \n{T}")
print(f"X = \n{X}")
```

```
T =
[[0 1 2 3 4 5 6 7 8 9]
 [0 1 2 3 4 5 6 7 8 9]
 [0 1 2 3 4 5 6 7 8 9]
 [0 1 2 3 4 5 6 7 8 9]]
X =
[[0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5]]
```

Observe that we have used lower caseletters - `t` and `x` - for the arrays and uppercase letters - `T` and `X` - for the corresponding vectors. This is a good convention to use.

If T_{kj} represents the the (k, j) entry of T and similarly for X_{kj}, then as k runs through 0 to 3 and j runs through 0 to 9, the ordered pairs (T_{kj}, X_{kj}) run through all 40 sample points.

At this point we have an efficient way to store the coordinates in the (t, x) plane at which we will draw our direction field. We now need to tell numpy which vectors are the ones to be drawn and then we need to get python to draw them.

We'll assume that we are dealing with ODEs of the form:

$$\frac{dx}{dt} = f(t, x).$$

And we will use the specific example:

$$\frac{dv}{dt} = -v - 9.8.$$

We first define a function that we can use (this isn't necssary but it makes things much easier). The `#...` is a comment. The Python interpreter doesn't look at this when running the code. It is just for us to give a short description of what the code is doing.

[5]:
```
# This represents the right hand side of the ODE
def f(t,x):
    return -9.8-x/5
```

Notice that we are using the variable `x` instead of `v`. This is because I prefer to keep things standardized throughout different problems (as a form of abstraction.) But this is mostly a personal choice and in your own code, you could have done:

```
def f(t,v):
    return -v-9.8.
```

And instead of using `x` and `X` you can use `v` and `V`. However, `u`, `U`, `v`, `V` are used for the vector field as you will see below.

[6]:
```
dXdT = f(T,X)
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
```

There is a lot going on here. Let's look at `dXdT`:

[7]:
```
print(f"dXdT = \n{dXdT}")
```

```
dXdT =
[[ -9.8  -9.8  -9.8  -9.8  -9.8  -9.8  -9.8  -9.8  -9.8  -9.8]
 [ -9.9  -9.9  -9.9  -9.9  -9.9  -9.9  -9.9  -9.9  -9.9  -9.9]
 [-10.  -10.  -10.  -10.  -10.  -10.  -10.  -10.  -10.  -10. ]
 [-10.1 -10.1 -10.1 -10.1 -10.1 -10.1 -10.1 -10.1 -10.1 -10.1]]
```

This is a 4×10 matrix. The (k, j) entry is:

$$f(T_{kj}, X_{kj}).$$

This is why the rows are constant (because $f(t, x)$ really only depends on x in the sense that $\frac{\partial f}{\partial t} \equiv 0$). The fact that this works is a minor miracle and is both liberating and somewhat alarming for those who are familiar with a language like C.

To explain the lines:

```
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
```

recall that our vector field is $\langle 1, f(t, x) \rangle$. We actually don't care about the lengths too much, and especially for display purposes we want to pick lengths that make things easier to see. So instead of plotting this vector field, we will plot a normalized version:

$$\frac{1}{\|\langle 1, f(t,x) \rangle\|} \langle 1, f(t,x) \rangle = \left\langle \frac{1}{\sqrt{1+f(t,x)^2}}, \frac{f(t,x)}{\sqrt{1+f(t,x)^2}} \right\rangle.$$

`U` and `V` will both be 4×10 matrices and the vector drawn at the point (T_{kj}, X_{kj}) will be $\langle U(k, j), V(k, j) \rangle$. Note that this means dimensions need to be consistent among `T`, `X`, `dXdT`, `U`, and `V`.

The expression `T.shape` returns a 2×1 array containg the dimensions of `T`:

```
[8]: print(T.shape)
```

```
(4, 10)
```

The expression `np.ones(T.shape)` creates a matrix of dimension equal to `T.shape`:

```
[9]: print(np.ones(T.shape))
```

```
[[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 python, the expression `a**b` raises `a` to the power of `b`:

```
[10]: print(2**3)
```

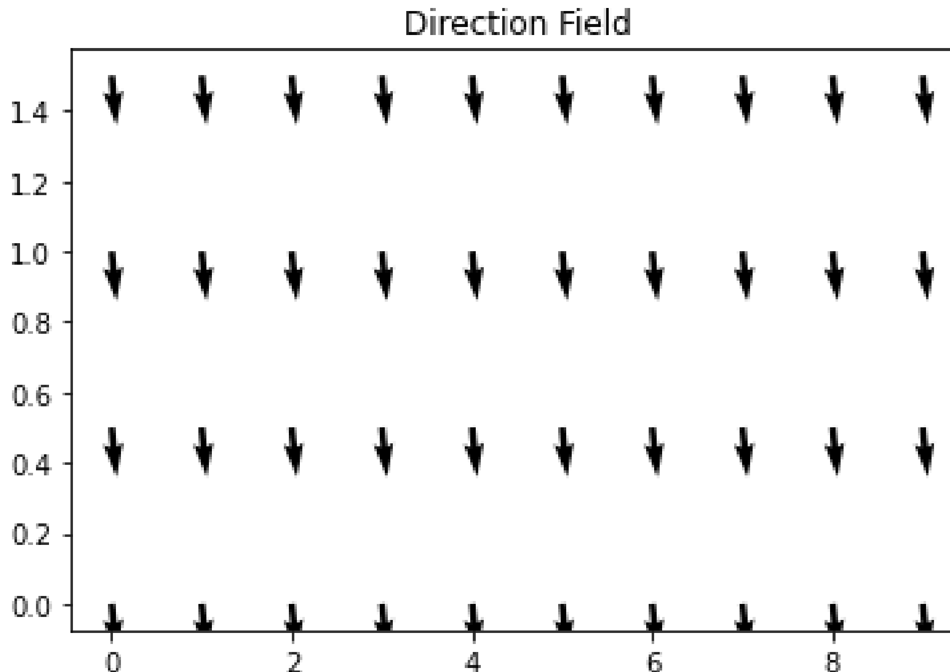
```
8
```

So multplying `np.ones(T.shape)` by `(1 / (1 + dXdT**2)**.5)` has the effect of dividing `np.ones(T.shape)` by $\|\langle 1, f(t,x)\rangle\|$. Similarly, the expression `V = (1 / (1 + dXdT**2)**.5)*dXdT` divides the matrix representing $f(t,x)$ by $\|\langle 1, f(t,x)\rangle\|$. Overall, as discussed above, (U, V) represents the vector field:

$$\left\langle \frac{1}{\sqrt{1+f(t,x)^2}}, \frac{f(t,x)}{\sqrt{1+f(t,x)^2}} \right\rangle.$$

Next we need to get python to plot the vector field $\langle U, V\rangle$ ad the points $\langle T, X\rangle$. The `matplotlib` library is a library of plotting functions.

```
[11]: import matplotlib.pyplot as plt
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V) # draws the arrows at (X,Y) with slope dYdX
```

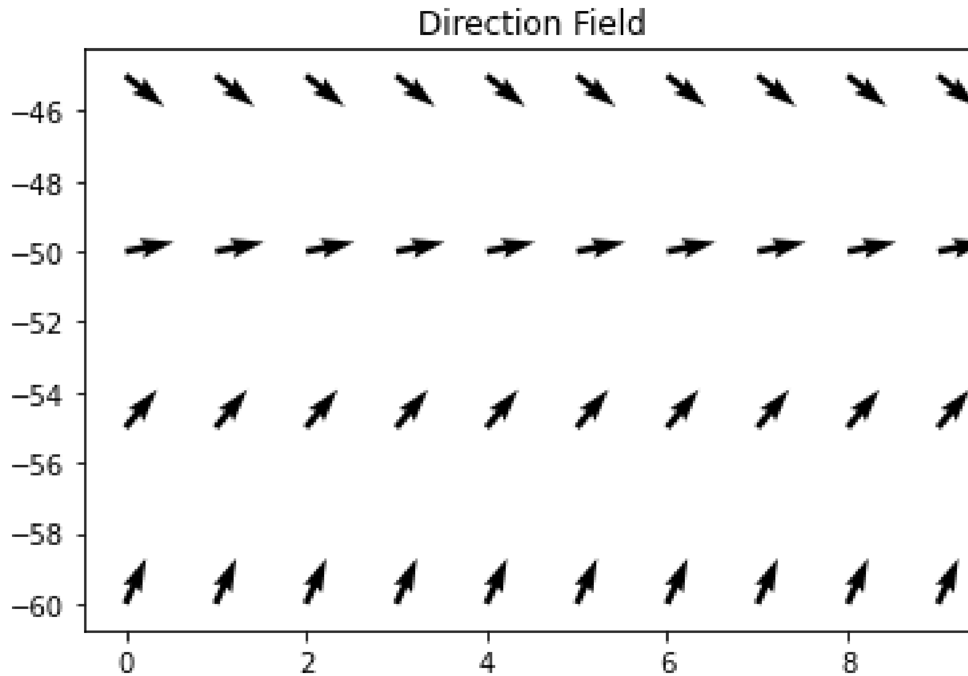


The expression `plt.figure()` creates a figure on which to draw. The expression `plt.title('Direction Field')` sets the title of the plot. Finally, the expression `Q = plt.quiver(T,X,U,V)` is the one that draws the vectors.

At this point, we should try to determine whether we have chosen a good range of t and v values (or using the variable names in the code, `t` and `x` values). The problem domain suggests that all velocities should be negative *for this model* if we are dropping something from a height (and we have a coordinate system in which downward velocities are negative.) Also, there are certain features that we want to see in the direction field and these features can be deduced from the model itself (more on this below.)

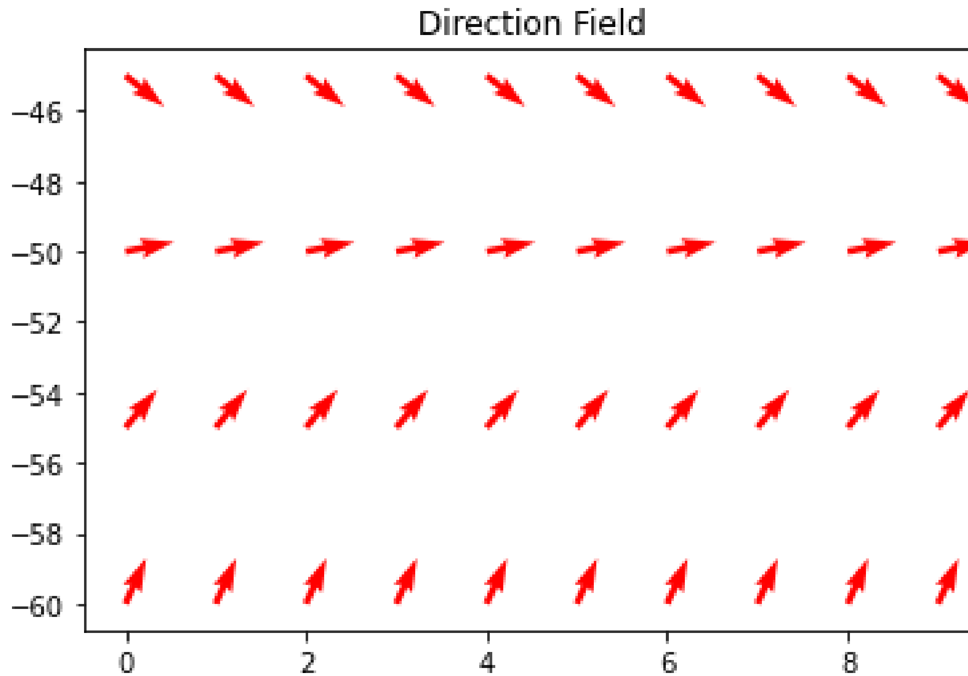
Sol, let's change the v range to be from -40 to -60 (or, using the variables in the code, we will change the `x` vector to be `x = np.arange(-60,-40,5)`).

[12]:
```
t = np.arange(0,10,1)
x = np.arange(-60,-40,5)
T, X = np.meshgrid(t, x)
dXdT = f(T,X)
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V) # draws the arrows at (X,Y) with slope dYdX
```



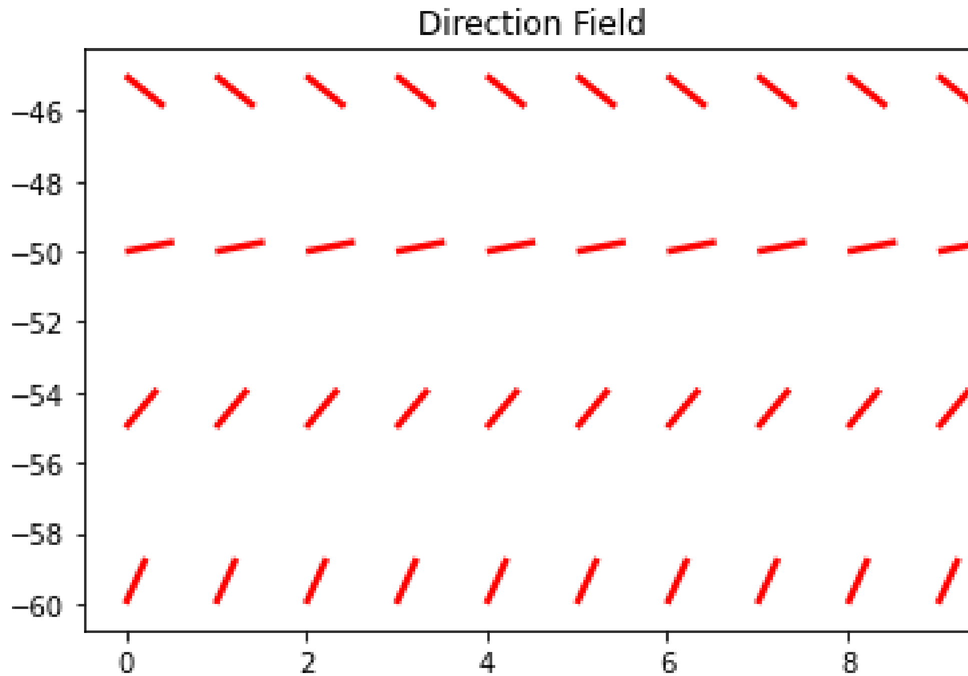
We will analyze this below, but first I want to describe ways to change the apperance. Some of the changes in appearance is purely esthetic I prefer for the vectors to be red (this is inline with what is in the textbook as well). To do this, we can add color in the following way:

[13]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V, color='red') # draws the arrows at (X,Y) with slope␣
 ↪dYdX
```



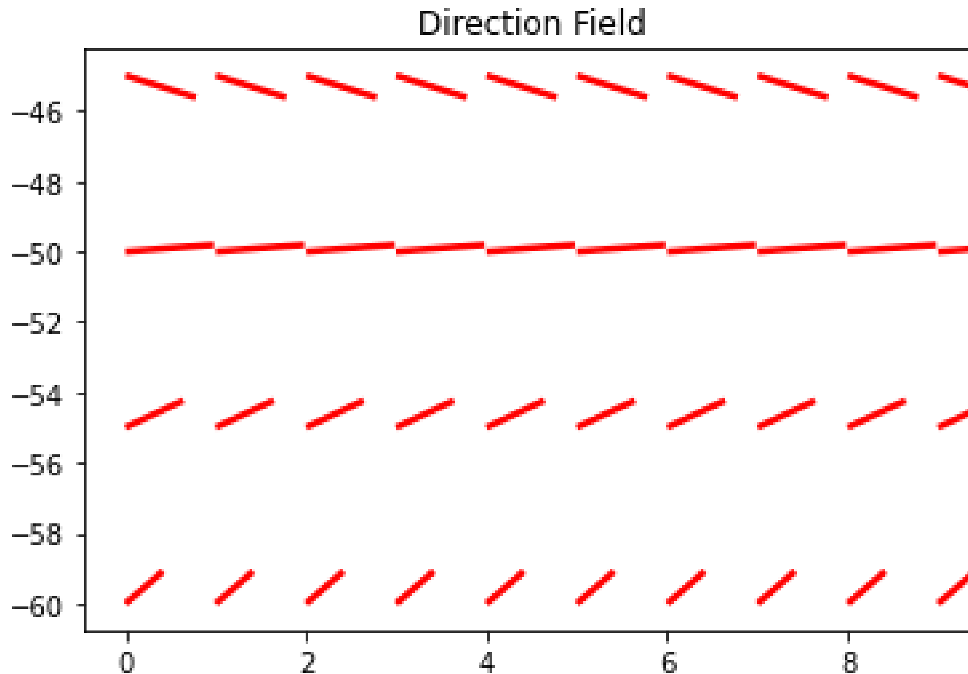
In addition, the arrowheads aren't really needed. We really only need short line segments that represent the tangent lines to the solution curves:

[14]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V, headlength=0, headwidth=1, color='red') # draws the␣
↪arrows at (X,Y) with slope dYdX
```



The `plt.quiver` function doesn't know what scale to draw the lines/arrows at and it basically makes an educated guess. However, we want to tell it to use the scale in the units of the plot. We do this by adding `angles='xy', scale_units='xy', scale=1` to the `plt.quiver` call:

[15]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V, headlength=0, headwidth=1, color='red', angles='xy',␣
 ↪scale_units='xy', scale=1)
```

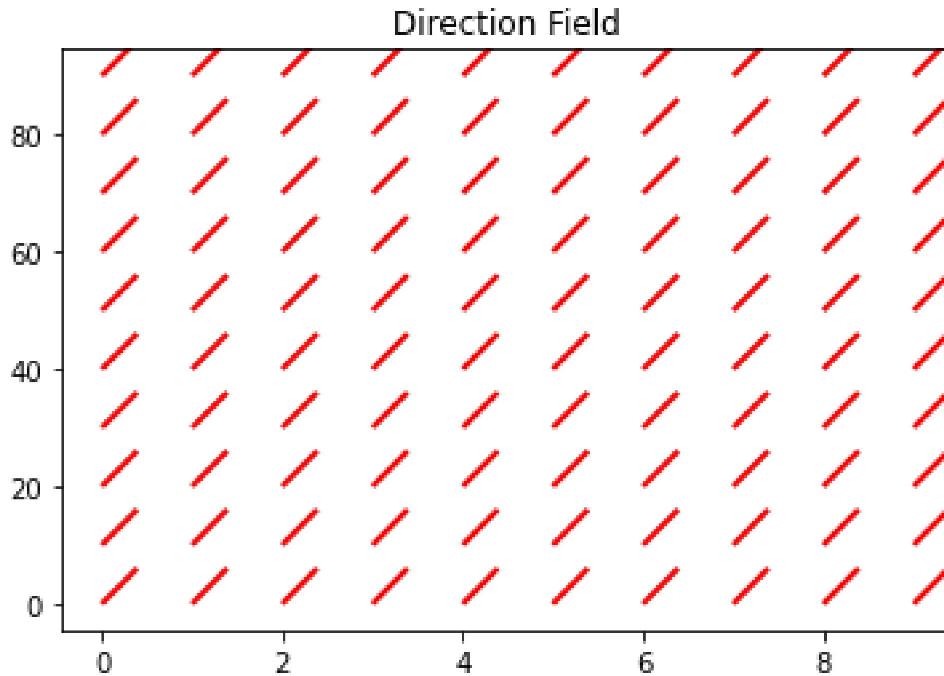


The `angles='xy'` tells `np.quiver` to draw the vectors with respect to the scale in the plot. (They use `x` and `y` instead of `t` and `x` respectivley; this is kind of confusing and annoying, but the conventions in ODE don't match with the conventions used in python; it's just something one has to get used to.)

(For those more interested, the default choice `np.quiver` makes is bascially independent of the scale of the figure. For example, in the code below, I make a figure that goes from 0 to 100 in the vertical axis and 0 to 10 in the horizontal axis. The vector field is the constant field $\langle 1, 1 \rangle$. Observe that these are drawn at actual 45 degree angles, i.e. the $\tan\theta = 1$. However, if using the scale of figure, they should be drawn so that $\tan\theta = 10$):

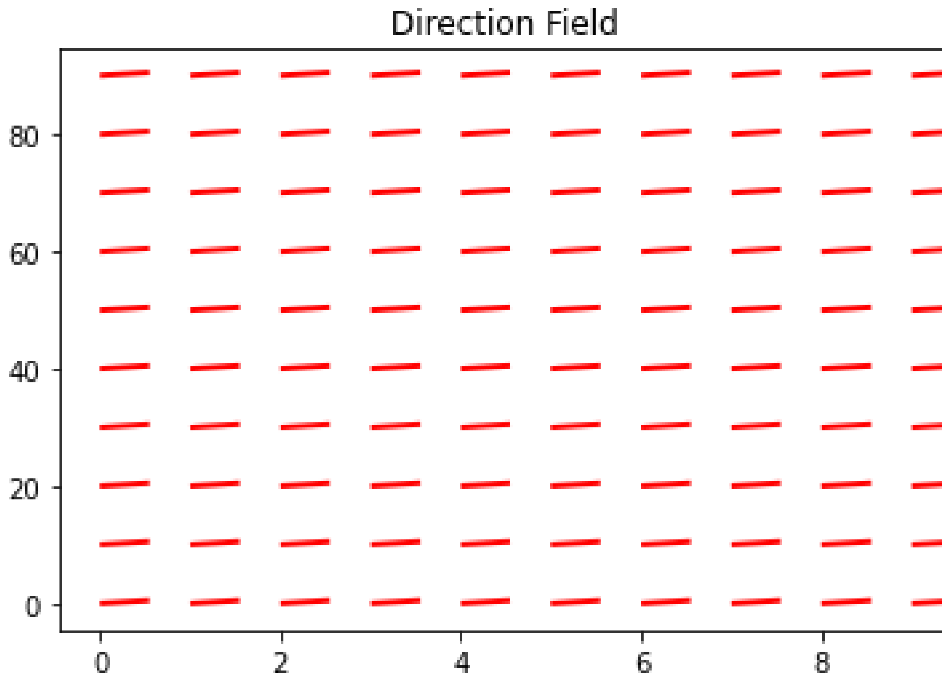
[16]:
```
## This code goes along with the explanation of axis='xy' directly above and is␣
 ↪not part of the ODE direction
## field!
a = np.arange(0,10,1)
b = np.arange(0,100, 10)
A, B = np.meshgrid(a, b)
C = D = np.ones(A.shape)
```

[17]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(A, B, C, D, headlength=0, headwidth=1, color='red')
```



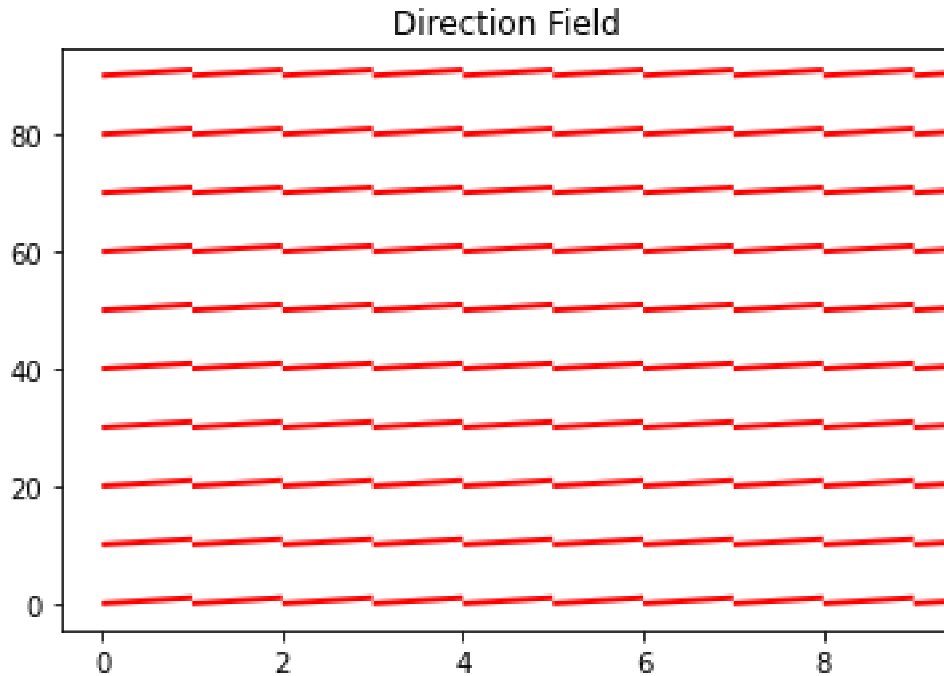
If we tell `np.quiver` to determine the angle at which it draws the vector from the figure's scale, we get:

[18]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(A, B, C, D, headlength=0, headwidth=1, color='red', angles='xy')
```



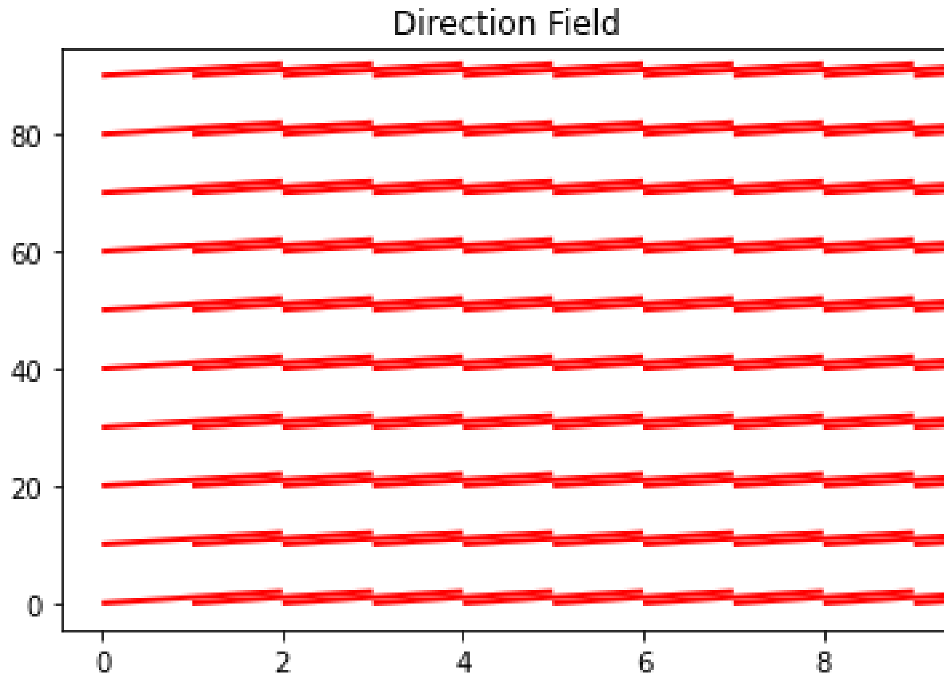
The `scale_units='xy', scale=1` tells `np.quiver` by how much to scale the line segments (that's the `scale=1` part) and which units to use (that's the `scale_units='xy'` part). If you don't give it `scale_units` then `np.quiver` first chooses to make all arrows unit length, and then picks a `scale` factor based on the data you give it (i.e. the vector field) to make it look the best. By giving it `scale_units` and a `scale`, you are telling `np.quiver` to scale the vectors using the units of the figure. So, in our little side example above this gives:

[19]:
```
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(A, B, C, D, headlength=0, headwidth=1, color='red', angles='xy',␣
 ↪scale_units='xy', scale=1)
```

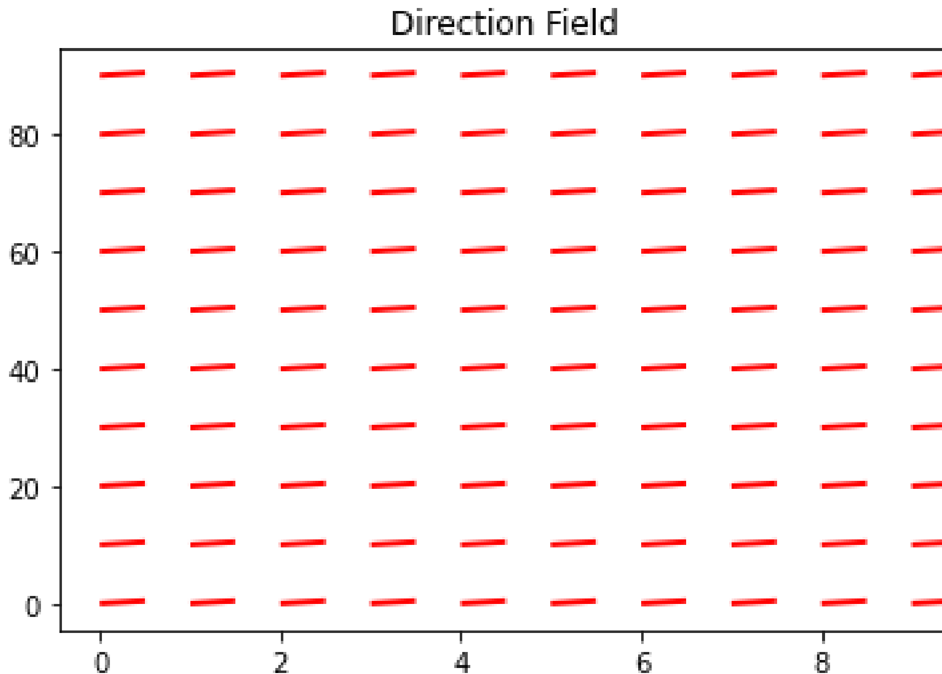


The bigger `scale` is, the shorter the line segments, and the smaller `scale` is, the shorter:

[20]:
```
# Scale = .5
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(A, B, C, D, headlength=0, headwidth=1, color='red', angles='xy',␣
↪scale_units='xy', scale=.5)
```



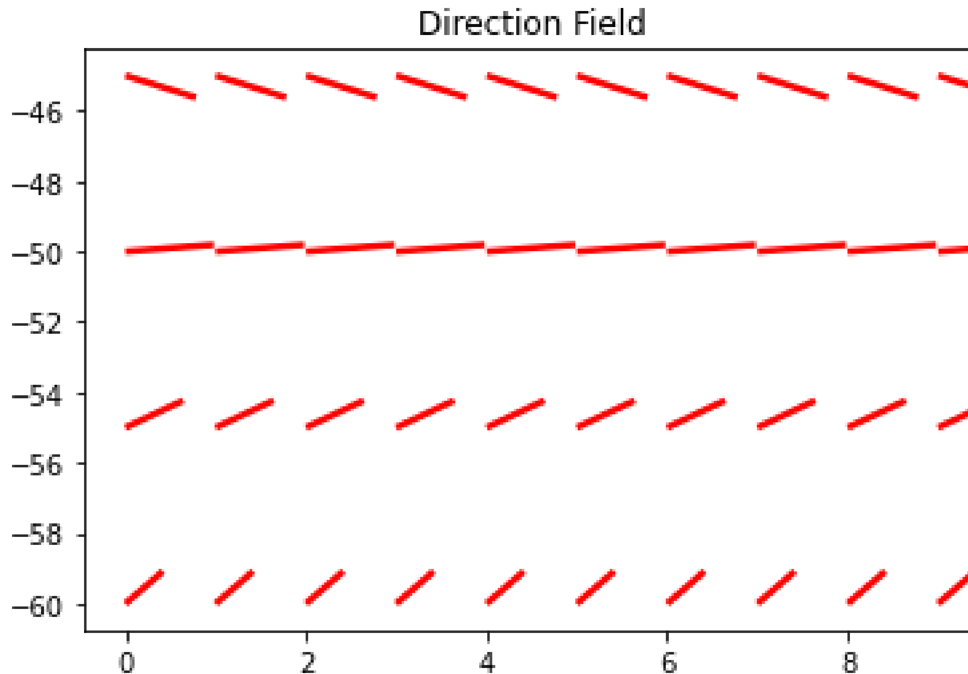
[21]:
```
# scale = 2
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(A, B, C, D, headlength=0, headwidth=1, color='red', angles='xy',␣
↪scale_units='xy', scale=2)
```



Since we are (mainly) intersted in the *direction* of the line segments and not their lengths we pick the `scale` parameter to give us the "best" picture. The fact that our primary concern is with direction and not lengths is manifested in the fact that we normalize the vectors prior to drawing them. Then the `angles='xy', scale_units='xy', scale=1` part let's us scale all the vectors simulteanously.

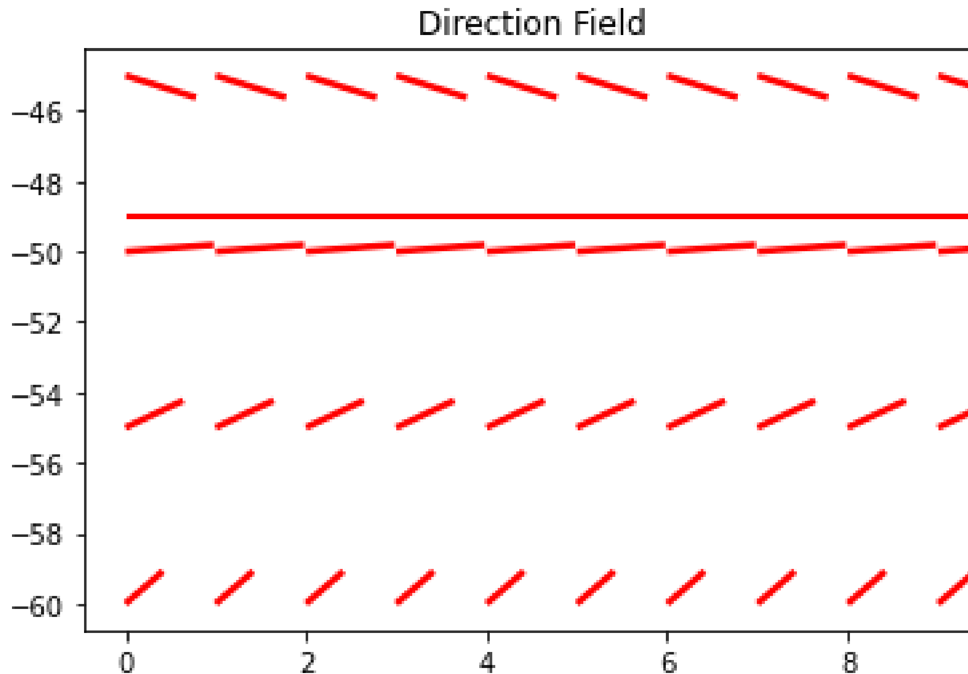
Now let's get back to the direction field; all the code is copied again below for convinience:

[22]:
```
t = np.arange(0,10,1)
x = np.arange(-60,-40,5)
T, X = np.meshgrid(t, x)
dXdT = f(T,X)
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V, headlength=0, headwidth=1, color='red', angles='xy',␣
 ↪scale_units='xy', scale=1)
```



Observe the qualitative difference between the line segments along the $v = -50$ line and the ones along the $v = -46$ line: their slopes have opposite sign. Given a continuous vector field, there should be some c for which the line segments on $v = c$ have zero slope. We know from class this happens when $f(t, v) = 0$, that is $v = -49$. We want to add this to our plot. The following code does that:

[23]:
```
t = np.arange(0,10,1)
x = np.arange(-60,-40,5)
x = np.append(x,-49)
x.sort()
T, X = np.meshgrid(t, x)
dXdT = f(T,X)
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
plt.figure()
plt.title('Direction Field')
Q = plt.quiver(T, X, U, V, headlength=0, headwidth=1, color='red', angles='xy',␣
 ↪scale_units='xy', scale=1)
```

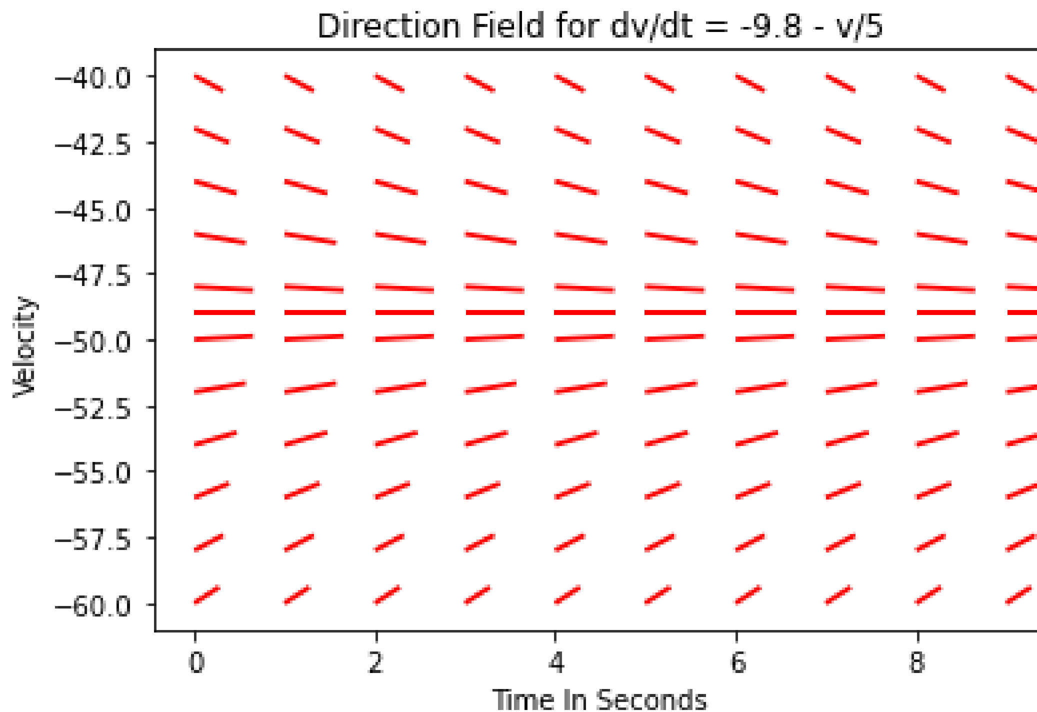


The `np.append(x,49)` expression creates a new array that is just `x` with `-49` appeneded to it. It doesn't change `x` so we have to assign this back to `x` which is why I wrote `x = np.arange(-60,-40,5)`. This just adds `-49` to the end of `x`. It isn't necessary to get the plot we want, but the `x.sort()` line sorts the `x` array. This is good thing to do because we are going to be thinking of `x` as being sorted, and it's possible that it will cause problems later (though, not actually for today's example) if it isn't. In general, it's best that your variables have the properties that you think they have.

Observe that our scale makes the line segments at $v = -49$ look like a line. Also, an optical illusion makes it look slanted. We will change the scale below. We will also add some to the `x` array to sample a few more v values. We also recenter the vertical axis so that the $v = -49$ line is (more or less) in the middle. We also add some labels to the axes.

[24]:
```
t = np.arange(0,10,1)
x = np.arange(-60,-39,2)
x = np.append(x,-49)
x.sort()
T, X = np.meshgrid(t, x)
dXdT = f(T,X)
U = (1 / (1 + dXdT**2)**.5)*np.ones(T.shape)
V = (1 / (1 + dXdT**2)**.5)*dXdT
plt.figure()
plt.title('Direction Field for dv/dt = -9.8 - v/5')
plt.xlabel('Time In Seconds')
plt.ylabel('Velocity')
```

```
Q = plt.quiver(T, X, U, V,  angles='xy', headlength=0, headwidth=1,␣
↪scale_units='xy', scale=1.5, color='red')
```



This is all we will do with the plot. I made several "style" decisions that you might like to do differently (e.g., maybe you like blue vector fields or maybe you wanted to keep the arrowheads.) The "correct" choices to make are the choices that convey the information in the best way.

1.2 Field Mice

Let's do a direction field using the field mouse example. First, let's define the $f(t, p)$ for the ODE. Recall that the equation is:

$$\frac{dp}{dt} = \frac{p}{2} - 450.$$

So we define our `f(t,x)` by:

```
[25]: def f(t,x):
          return x/2 - 450
```

To set the range of t and p values (in python this will be `t` and `x` values) note that we want to model things going forward in time, so we want `t` to be positive. Also, since the units of time is months, we probably want `t` to go through a year or so. We know that $p \equiv 900$ is the equilibrium solution, so we want to include that in the `x` range.

[26]:

```
## t is time measured in months
## x is field mouse population
t = np.arange(0,13,1)
x = np.arange(800,1000,20)


T,X = np.meshgrid(t,x)
dXdT = f(T,X)
#U = 1/(1+dXdT**2)**0.5*np.ones(T.shape) # Normalizes the arrows to see␣
↪near-zero slopes.
#V = 1/(1+dXdT**2)**0.5*dXdT
U = np.ones(T.shape) # Normalizes the arrows to see near-zero slopes.
V = dXdT
plt.figure()
plt.title('Direction Field for dp/dt = p/2 - 450')
Q = plt.quiver(T, X, U, V, angles='xy', scale_units='xy', scale=3, headlength=0,␣
↪headwidth=1, color='red')
plt.xlabel('Time In Months')
plt.ylabel('Field Mouse Population')
```

[26]:

```
Text(0, 0.5, 'Field Mouse Population')
```