Plotting and Dynamical Systems How−to

W. Garrett Mitchener

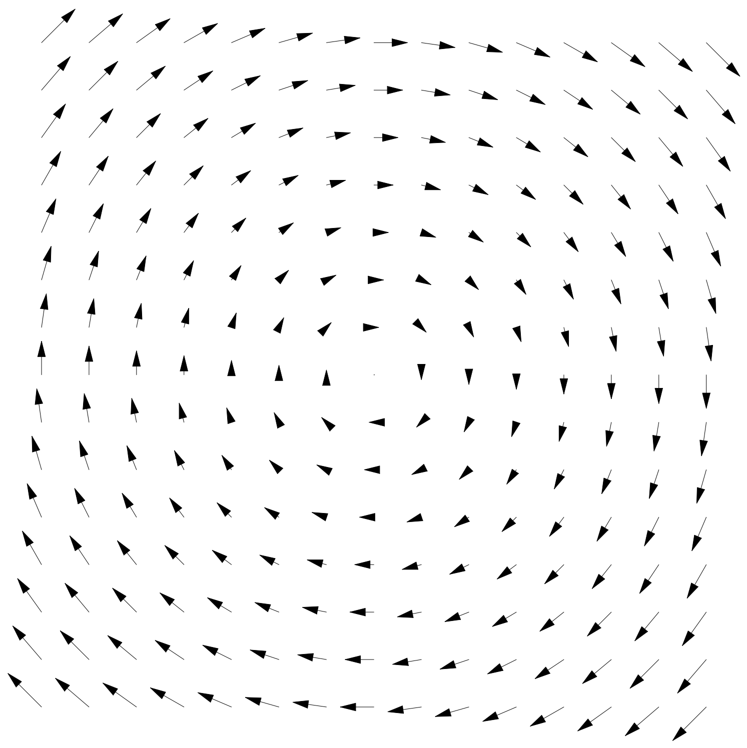
```
Date[]
```

```
{2005, 2, 23, 11, 32, 9.529104}
```

The Graphics package contains a function called PlotVectorField:

```
<< Graphics`
```

```
PlotVectorField[{y, -x}, {x, -5, 5}, {y, -5, 5}]
```



```
- Graphics -
```

Unfortunately, *Mathematica* doesn’t include much in the line of nice point−and−click interfaces for drawing solutions to the dynamical system defined by a vector field. Here’s how to do it through function calls.

Let’s say your dynamical system is:

```
x′ == x (3 - x - 2 y)
y′ == y (2 - x - y)
```

Let’s define a rule table that encodes this system:

```
dynamicalSystem = {x’[t_] → x[t] (3 - x[t] - 2 y[t]),
   y’[t_] → y[t] (2 - x[t] - y[t])}
```

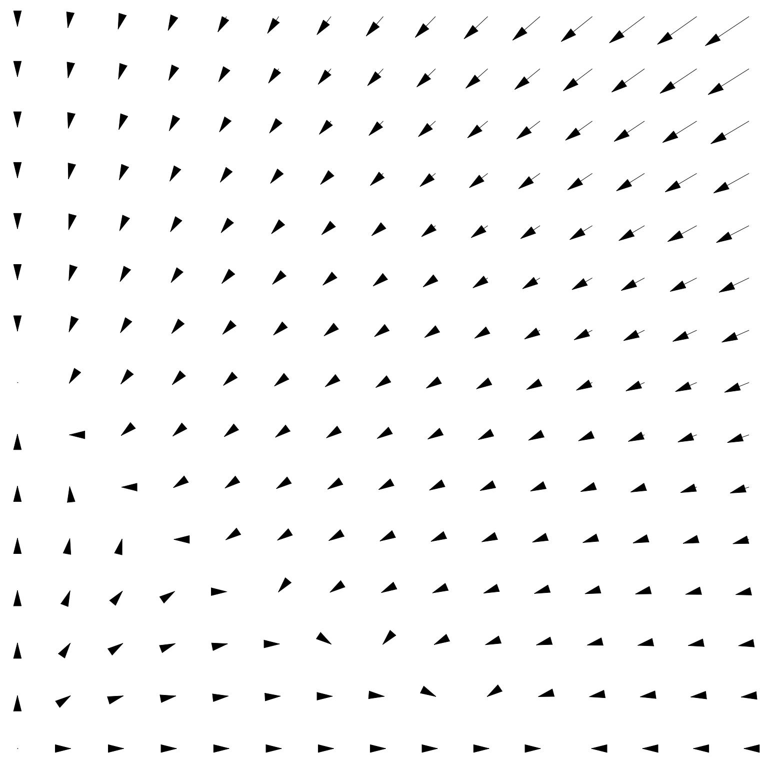
```
{x′[t_] → x[t] (3 - x[t] - 2 y[t]), y′[t_] → (2 - x[t] - y[t]) y[t]}
```

Note that I’ve defined rules that replace *x*’[t_] where the *t* is a pattern. That means we can apply this rule and it will work on *x*’[anything], not just *x*’[*t*]. By using a rule table of this form, we can do lots of interesting things. For example, let’s say we want to plot the vector field:

```
vectorField1 = {x’[t], y’[t]} /. dynamicalSystem
```

```
{x[t] (3 - x[t] - 2 y[t]), (2 - x[t] - y[t]) y[t]}
```

```
PlotVectorField[vectorField1, {x[t], 0, 4}, {y[t], 0, 4}]
```



```
- Graphics -
```

If having expressions like *x*[*t*] and such in your vector field bothers you, you can also get rid of them with this rule:

```
vectorField2 = vectorField1 /. f_[t_] → f
```

```
{x (3 - x - 2 y), (2 - x - y) y}
```

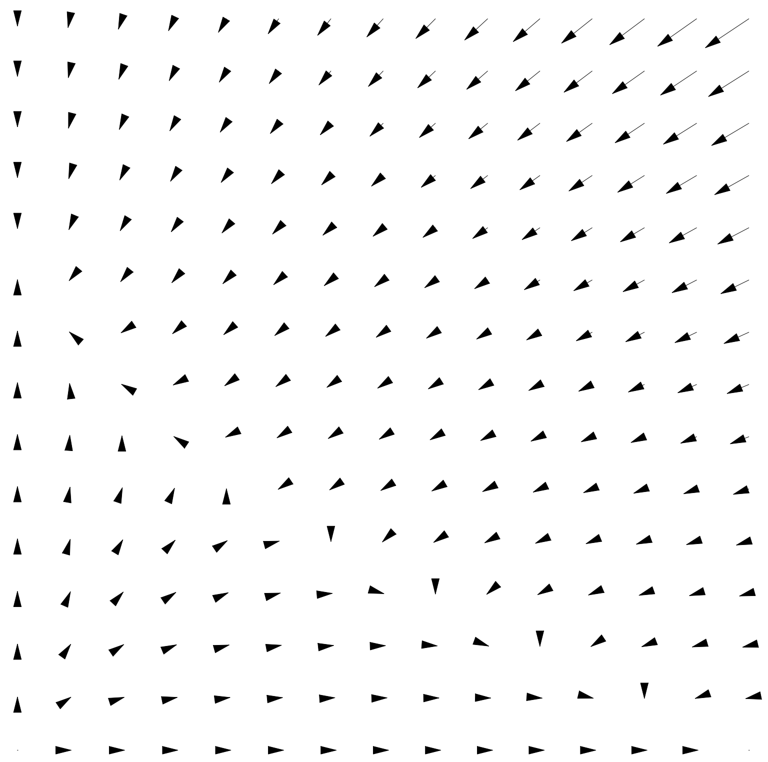
Yes, you can make the name of the function a pattern to. Just so you can see what that is doing, consider:

```
f[x] + g[y] + h[x] + r[y] /. u_[x] → q[x]
```

```
g[y] + 2 q[x] + r[y]
```

So I can now give this command, which is pretty much the same as my earlier command but it doesn't look quite so odd, and this time I'll save the result to be used in later pictures:

```
vectorFieldPicture = PlotVectorField[vectorField2, {x, 0, 3}, {y, 0, 3}]
```



```
- Graphics -
```

Back to our dynamical system. Now suppose you want to plot solutions on top of the vector field. To do that, we should use *Mathematica*'s built−in numerical solving functions. Here's how to turn our dynamical system rule table into a system of equations that NDSolve can handle:

```
ode = {x'[t] == (x'[t] /. dynamicalSystem),
  y'[t] == (y'[t] /. dynamicalSystem)}
```

```
{x'[t] == x[t] (3 - x[t] - 2 y[t]), y'[t] == (2 - x[t] - y[t]) y[t]}
```

We also have to add in initial conditions. Those must be added to the ODE, so we use the Join function:

```
Join[{a, b, c}, {x, y, z}]

{a, b, c, x, y, z}
```

```
Join[ode, {x[0] == 2, y[0] == 2}]

{x′[t] == x[t] (3 - x[t] - 2 y[t]),
 y′[t] == (2 - x[t] - y[t]) y[t], x[0] == 2, y[0] == 2}
```

Here’s how to use NDSolve:

```
numSol1 = NDSolve[Join[ode, {x[0] == 2, y[0] == 2}],
   {x, y}, {t, 0, 5}]

{{x → InterpolatingFunction[{{0., 5.}}, <>],
  y → InterpolatingFunction[{{0., 5.}}, <>]}}
```

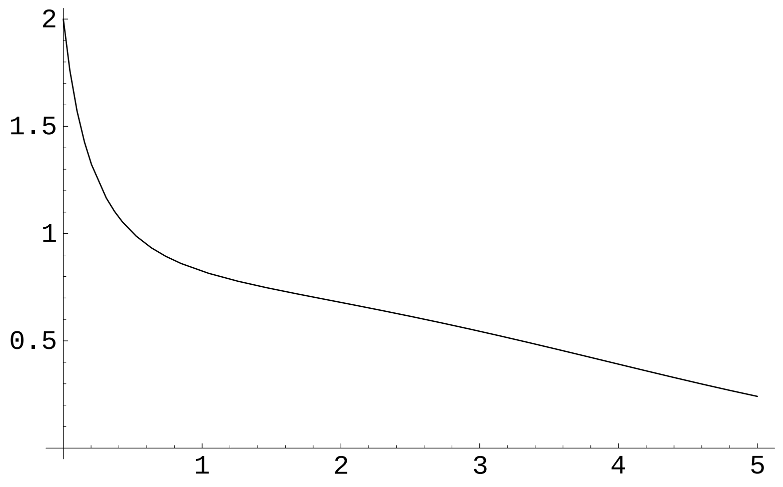
The return value is a list of rule tables that encode numerical solutions. You can use them pretty much like you’d use any regular function. It returns a list of rule tables in case it happens to find multiple solutions, which is rare. Since we usually just want the first solution, it’s best to use the command

```
numSol1[[1]]

{x → InterpolatingFunction[{{0., 5.}}, <>],
 y → InterpolatingFunction[{{0., 5.}}, <>]}
```

Or you can use the First function:

```
First[numSol1]

{x → InterpolatingFunction[{{0., 5.}}, <>],
 y → InterpolatingFunction[{{0., 5.}}, <>]}
```

```
Plot[x[t] /. First[numSol1], {t, 0, 5}]
```



```
- Graphics -
```

Note: Notice that I've used double equal signs == in the definition of ode, because that represents an equation in *Mathematica*, as in something that might be true or false, or something to be solved. That's different from the single equal = or the colon equal := that are used to make definitions. For various reasons, you can use single equal inside an expression, because that makes a definition and returns the value of the definition. That leads to bugs where you use = instead of == by mistake and can't know that anything is wrong until you get a goofy error message. Here's what can go wrong if you use = by mistake:

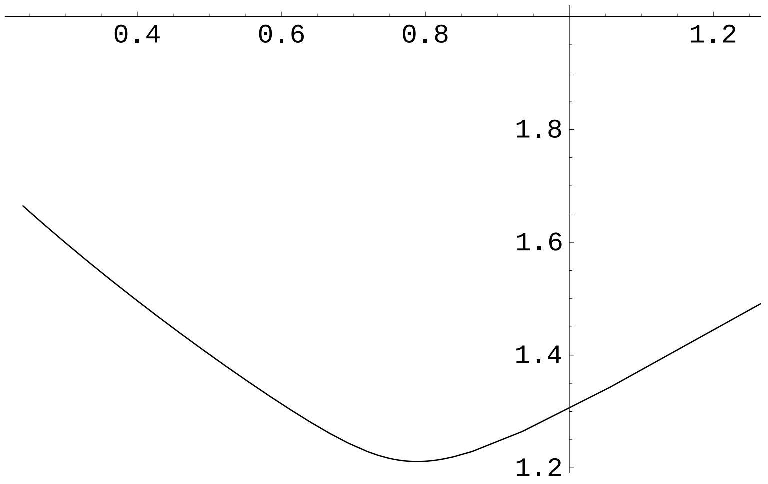
```
odeWrong = {u'[t] = v[t], v'[t] = -u'[t]}

{v[t], -v[t]}


numSolWrong = NDSolve[Join[odeWrong, {u[0] = 2, v[0] = 2}],
   {u, v}, {t, 0, 5}]
– NDSolve::deqn : Equation or list of equations expected
     instead of v[t] in the first argument {v[t], -v[t], 2, 2}. More…
– NDSolve::deqn : Equation or list of equations expected
     instead of v[t] in the first argument {v[t], -v[t], 2, 2}. More…

NDSolve[{v[t], -v[t], 2, 2}, {u, v}, {t, 0, 5}]
```

And what this has done is to define stuff about *u* and *v* rather than create equations. You can see from the output that the actual NDSolve command has turned out all wrong. Here's how to view the definitions we incorrectly created:

```
?u

Global`u

u[0] = 2

?v

Global`v

v[0] = 2
```

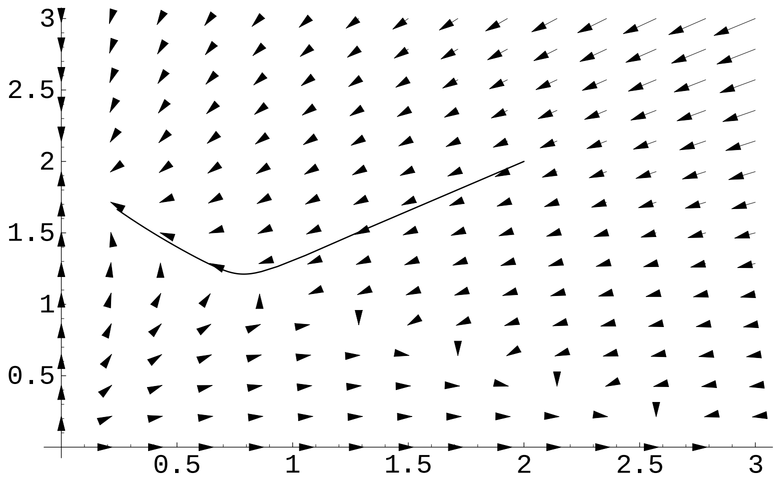
Back to our vector field problem. Now suppose you wanted to overlay solutions on the vector field. Since solution curves are defined parametrically, we have to use the ParametricPlot function:

```
solutionPicture =
 ParametricPlot[{x[t], y[t]} /. First[numSol1], {t, 0, 5}]
```



```
- Graphics -
```

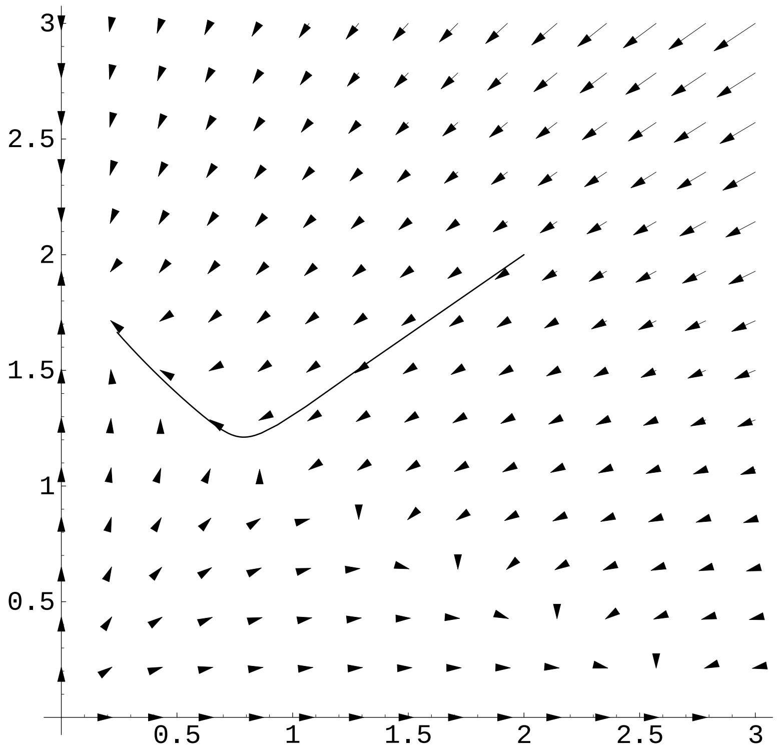
```
Show[{solutionPicture, vectorFieldPicture}]
```



```
- Graphics -
```

Hmm, let’s make it square:

```
Show[{solutionPicture, vectorFieldPicture},
  AspectRatio → 1]
```



```
- Graphics -
```

Now if you wanted to plot a lot of solution curves, the procedure I just described would get tedious. So, we should automate it. First, let's define a function that creates a numerical solution starting at a given initial condition:

```
makeOneSolution[{x0_, y0_}] :=
  First[NDSolve[Join[ode, {x[0] == x0, y[0] == y0}],
    {x, y}, {t, 0, 10}]]
```

I used := rather than = to define makeOneSolution because I don't want *Mathematica* to attempt to evaluate the NDSolve command right now. I want it to wait until I've acutally specified the initial conditions. Here's what happens if you forget and use = instead:

```
makeOneSolutionOops[{x0_, y0_}] =
  First[NDSolve[Join[ode, {x[0] == x0, y[0] == y0}],
    {x, y}, {t, 0, 10}]]
```

```
– NDSolve::ndinnt :
   Initial condition x0 is not a number or a rectangular array of numbers. More…

{x′[t] == x[t] (3 - x[t] - 2 y[t]),
  y′[t] == (2 - x[t] - y[t]) y[t], x[0] == x0, y[0] == y0}
```

```
makeOneSolutionOops[{2, 2}]

{x′[t] == x[t] (3 - x[t] - 2 y[t]),
 y′[t] == (2 - x[t] - y[t]) y[t], x[0] == 2, y[0] == 2}
```

Which is all wrong. So be sure to use := to define it.

Now is a good time to introduce the Map function. It takes a function and a list, and applies the function to each item in the list, returning the result. (The name "map" is traditional for this kind of operation, even though something like "apply to each" would be more descriptive.)

```
Map[f, {1, 2, 3}]

{f[1], f[2], f[3]}
```

```
Map[f, {{a, x}, {b, y}, {c, z}}]

{f[{a, x}], f[{b, y}], f[{c, z}]}
```

So this makes a bunch of solution curves:

```
makeManySolutions[initialConditions_] :=
 Map[makeOneSolution, initialConditions]
```

Again, I use := to tell *Mathematica* to wait and not evaluate the right hand side right away. Here’s how all this works:

```
makeOneSolution[{2, 2}]

{x → InterpolatingFunction[{{0., 10.}}, <>],
 y → InterpolatingFunction[{{0., 10.}}, <>]}
```

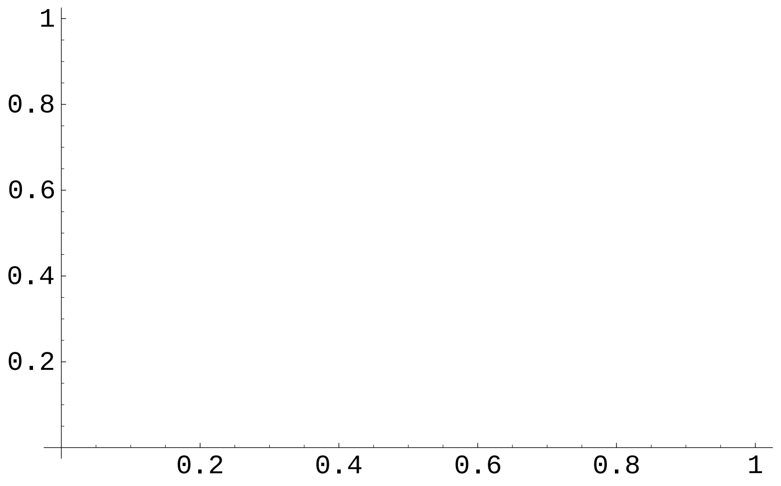
```
makeManySolutions[{{2, 2}, {1, 1}, {2, 1}}]

{{x → InterpolatingFunction[{{0., 10.}}, <>],
  y → InterpolatingFunction[{{0., 10.}}, <>]},
 {x → InterpolatingFunction[{{0., 10.}}, <>],
  y → InterpolatingFunction[{{0., 10.}}, <>]},
 {x → InterpolatingFunction[{{0., 10.}}, <>],
  y → InterpolatingFunction[{{0., 10.}}, <>]}}
```

So now I can make a list of rule tables, each representing a different numerical solution. And I can plot them all at once like this:

```
{x[t], y[t]} /. makeManySolutions[{{2, 2}, {1, 1}, {2, 1}}]

{{InterpolatingFunction[{{0., 10.}}, <>][t],
  InterpolatingFunction[{{0., 10.}}, <>][t]},
 {InterpolatingFunction[{{0., 10.}}, <>][t],
  InterpolatingFunction[{{0., 10.}}, <>][t]},
 {InterpolatingFunction[{{0., 10.}}, <>][t],
  InterpolatingFunction[{{0., 10.}}, <>][t]}}
```

```
ParametricPlot[
 {x[t], y[t]} /. makeManySolutions[{{2, 2}, {1, 1}, {2, 1}}],
 {t, 0, 10}]
```

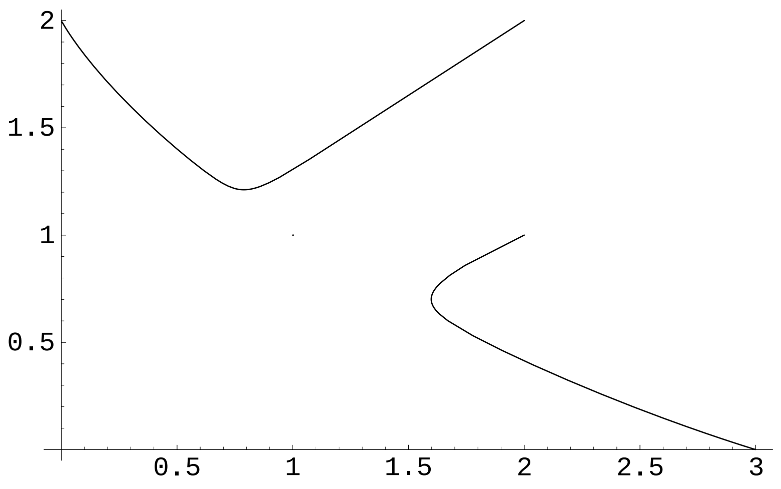
```
— NDSolve::dsvar : 4.166666666666667`*^-7 cannot be used as a variable. More…
— NDSolve::dsvar : 4.166666666666667`*^-7 cannot be used as a variable. More…
— NDSolve::dsvar : 4.166666666666667`*^-7 cannot be used as a variable. More…
— General::stop : Further output of
     NDSolve::dsvar will be suppressed during this calculation. More…
— ReplaceAll::reps :
   {<<1>>} is neither a list of replacement rules nor a valid dispatch
     table, and so cannot be used for replacing. More…
— ReplaceAll::reps :
   {<<1>>} is neither a list of replacement rules nor a valid dispatch
     table, and so cannot be used for replacing. More…
— ReplaceAll::reps :
   {<<1>>} is neither a list of replacement rules nor a valid dispatch
     table, and so cannot be used for replacing. More…
— General::stop : Further output of
     ReplaceAll::reps will be suppressed during this calculation. More…
— ParametricPlot::pptr :
   {x[t], y[t]} /. <<17>>[{<<1>>}] does not evaluate to a pair of
     real numbers at t = 4.166666666666667`*^-7. More…
— ParametricPlot::pptr :
   {x[t], y[t]} /. <<17>>[{<<1>>}] does not evaluate to a pair of
     real numbers at t = 0.40566991572915795`. More…
— ParametricPlot::pptr :
   {x[t], y[t]} /. <<17>>[{<<1>>}] does not evaluate to a pair of
     real numbers at t = 0.8480879985937368`. More…
— General::stop : Further output of
     ParametricPlot::pptr will be suppressed during this calculation. More…
```



```
- Graphics -
```

Ooh, that was unpleasant. This often happens when you try to do fancy plotting commands. The problem is that plotting functions often "evaluate their arguments in a non−standard way," which means they do confusing things like that. You can usually get it to work by using the Evaluate function to force *Mathematica* to create the solutions before attempting to plot anything:

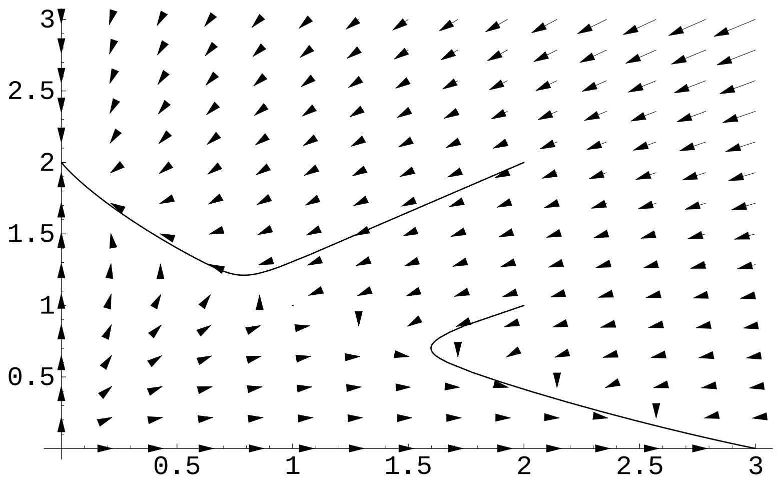
```
manySolutionPicture = ParametricPlot[Evaluate[
    {x[t], y[t]} /. makeManySolutions[{{2, 2}, {1, 1}, {2, 1}}]],
  {t, 0, 10}]
```



```
- Graphics -
```

And you can show the solutions on top of the vector field as follows:

```
Show[{manySolutionPicture, vectorFieldPicture}]
```



```
- Graphics -
```