phaseR: v2.1

Michael J Grayling (michael.grayling@newcastle.ac.uk)

2019-10-11

phaseR is an R package for the qualitative analysis of one- and two-dimensional autonomous ODE systems, using phase plane methods. Programs are available to identify and classify equilibrium points, plot the direction field, and plot trajectories for multiple initial conditions. In the one-dimensional case, a program is also available to plot the phase portrait. Whilst in the two-dimensional case, additionally programs are available to plot nullclines and the stable/unstable manifolds of saddle points. Many example systems are provided for the user.

Introduction

Contrary to what may seem to be the case when you first encounter ordinary differential equations (ODEs), the majority of ODE systems cannot be solved analytically. In such a scenario, there is usually no option but to resort to numerical solution. However, for certain classes of ODE systems it is possible to undertake a qualitative examination using phase plane methods, as introduced by Henry Poincare, amongst others, in the 19th century. These methods allow the analyser to circumvent the need for explicit solutions, via a highly graphical approach. And in fact, this qualitative analysis is often useful even when the system can be solved analytically. Specifically, it is usually possible to plot trajectories for various initial conditions, before obtaining information about stability and other motion patterns of the system.

This package, **phaseR**, allows the user to perform such analyses for one- and two-dimensional autonomous ODE systems. Programs are available to determine and classify equilibrium points, plot the flow or vector field, and plot trajectories for multiple initial conditions. In the one-dimensional case, a program is also available to plot the phase portrait. Whilst in the two-dimensional case, additionally programs are available to plot nullclines and the stable/unstable manifolds of saddle points. This accompanying guide has been written not only to provide further information on how to use **phaseR**, but also as a teaching utility for phase plane methods. In this way, **phaseR** can hopefully serve as both a package for both independent learning, and for group based teaching; assisting lecturers in explaining the supported techniques.

Thus, since it is an important skill to be able to perform phase plane analysis by hand, and also as a background to the package, this guide will proceed by introducing mathematically the systems that it can examine as well as of the techniques for analysis that it supports. The complexity is most similar to a first year undergraduate mathematics course, however it should hopefully also be useful to those from natural science and engineering backgrounds as well. Following the mathematical descriptions, an explanation of the usage of the programs in **phaseR** will be given. For this, good knowledge of R is useful, however the programs are not difficult to use. Worked examples will then be provided for both one- and two-dimensional systems. Further example systems available in the package will next be described, before finally, exercises are provided for the user to undertake should they wish. Throughout, to keep things simple, we will stick to using the letters x, y and t only as variables, as these are the variable names used by the programs. In practice however, it is obviously not difficult to deal with cases where alternative notation is used.

Acknowledgment goes to Professors Daniel Kaplan and Daniel Flath at Macalester College who established the initial code for phase plane analysis of two dimensional systems (http://www.macalester.edu/~kaplan/math135/pplane.pdf) upon which this package was based. Additionally, I would like to thank Stephen Ellner and John Guckenheimer who graciously provided code which allowed the creation of the `findEquilibrium()`, `drawManifolds()`, and `phasePlaneAnalyser()` functions, as well as

Gerhard Burger who helped to substantially advance the package's inner workings. I welcome any corrections or comments on both the programs and these notes.

First order one-dimensional dynamical systems

Autonomous one-dimensional ordinary differential equations

A first order dynamical system of one variable, $y(t)$ say, can be written in the following form:

$$\frac{dy}{dt} = f(y, t).$$

In many cases (usually the ones found in introductory calculus texts) this ODE can be solved analytically; with several techniques, such as integrating factors and separation of variables, at hand to help. However, more often than not, when a differential equation is written down to describe a real life system, it cannot be solved analytically. This is particularly true of non-linear ODEs, for which numerical solution would frequently have to be utilised. As a result, many computer packages are today available to support such calculations.

However, an alternative approach to numerical integration is sometimes possible. This approach is usually termed the phase plane method, or phase plane analysis. This methodology is concerned with determining qualitative features of the solutions to ODEs, without the need for explicit solution. Whilst such analysis may be more germane to systems we cannot solve analytically, the methods are just as valid to systems we can.

Although this qualitative analysis is indeed possible for ODEs of the type above, in this package we restrict ourselves slightly to the case of 'autonomous ODEs', for reasons that should hopefully become clear later. This class of first order ODEs can be written in the following form:

$$\frac{dy}{dt} = f(y),$$

i.e., it is the case where this is no dependence upon the independent variable t in the functional form of f. Moreover, formally, we also assume that f is a continuous, differentiable function. Whilst this may seem a strong restriction, many real life models can be written in this form.

Now, within this framework of qualitative analysis there are several important concepts that we will proceed to discuss. Namely, the flow field, equilibrium points, and the phase portrait.

The flow field

We begin with a discussion of the flow, or direction, field. Consider again the autonomous ODE above, and imagine making a sketch in the t-y plane by drawing at every point (t, y) a small line of slope $f(y)$. This resulting picture of line segments is the flow field. The use of such a picture lies in that solution curves to the ODE must be tangent to the directions of the line segments. Thus we can construct approximate graphical solutions to this ODE system by beginning at any point $(0, y_0)$ (i.e., $y(0) = y_0$) and sketching a curve that proceeds through the plane in the direction of the flow field. In this way, if we start at many different initial points, we can generate a family of solution curves that qualitatively describe the behaviour specified by the ODE.

It is important to note, however, that whilst the flow field method is incredibly useful for plotting trajectories by hand, it is an approximate method: since we can only plot a finite number of line segments some approximation will always be introduced. Usually however, solutions accurate enough to gain a reasonable understanding of the ODE can be achieved, and in general, the more line segments we plot the more accurate our sketches will be. By hand this can be time consuming, utilising a computer however, it is not so difficult.

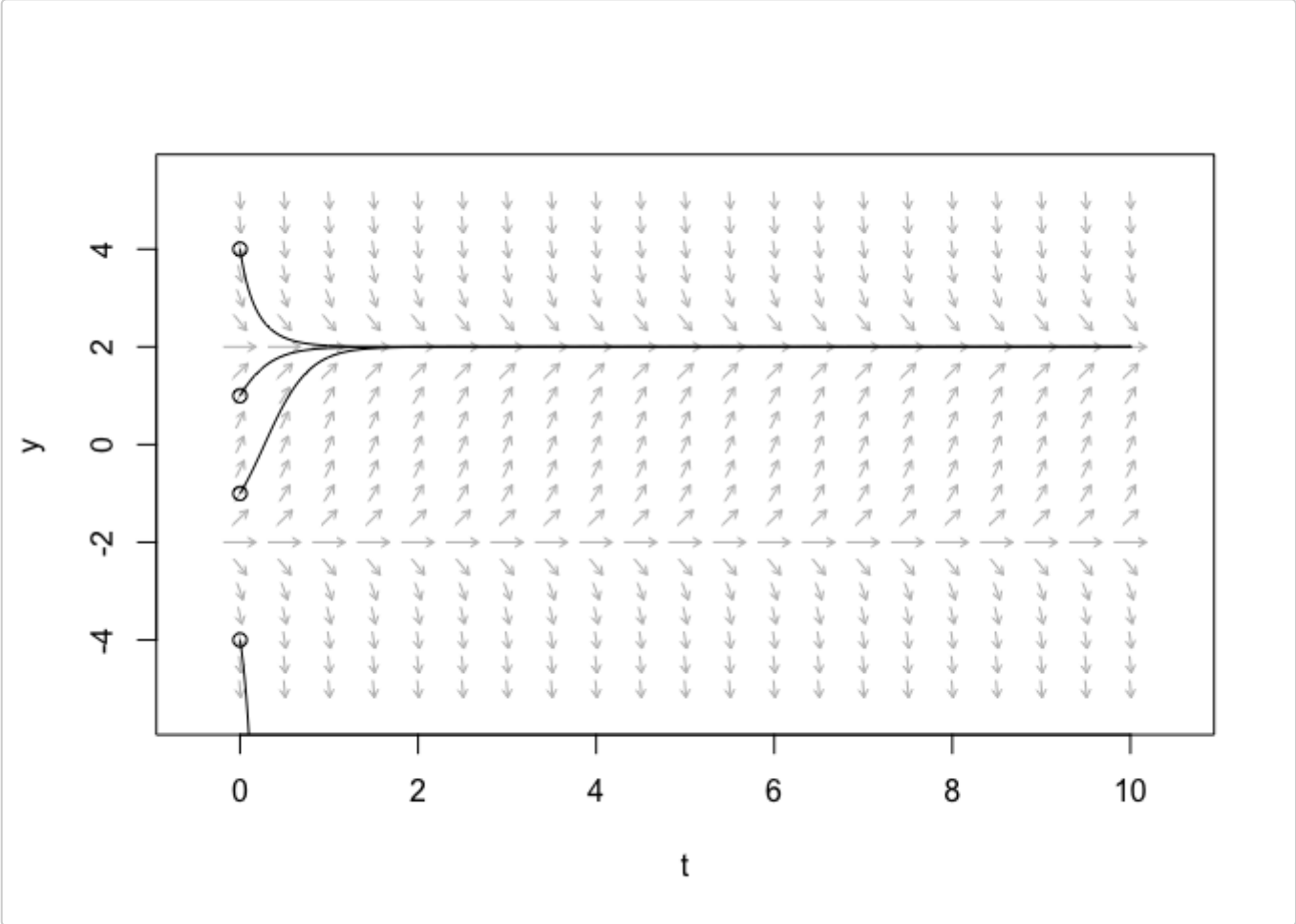
Note also that some texts on phase plane methods would here discuss the concept of isoclines, defined as lines across which dy/dt is constant, i.e., where $f(y) = \alpha$ for different values of α. These lines are used in the same manner as the small line segments of the flow field, since we know the angle at which solution curves should cut them. They however are more useful in the setting of non-autonomous ODEs, and thus we will make little further mention of them.

Additionally, some texts advise to plot the line segments at lengths reflecting the rate of change of y. However, by hand this will almost always be a very laborious task, whilst even with a computer if $f(y)$ takes a large range of values the resulting plot can become somewhat uninformative with obscuring arrows of great length and other arrows of length too short to be useful. Thus, it is usually best to plot all line segments at some small arbitrary length.

As is often the case in mathematics, concepts can be more easily understood through an example. As such, consider now the ODE:

$$\frac{dy}{dt} = 4 - y^2,$$

provided in the package as `example1()`. More information will be provided later on how to utilise the programs in **phaseR**, as well as how to specify your own systems. For now though, simply note the flow field produced below, and the multiple trajectories that follow it:



Equilibrium points and stability

We now turn our attention to the so-called equilibrium points of our autonomous ODE. These points are defined by the locations where:

$$f(y) = 0.$$

It is easy to understand why they are termed equilibrium points. Beginning at a point y_* where $f(y_*) = 0$, the system if unperturbed will remain at y_* throughout its evolution. The great importance of these points lies in determining the long-term behaviour of the ODE.

Considering our example ODE $dy/dt = 4 - y^2$ again, it is a simple matter to find its equilibrium points:

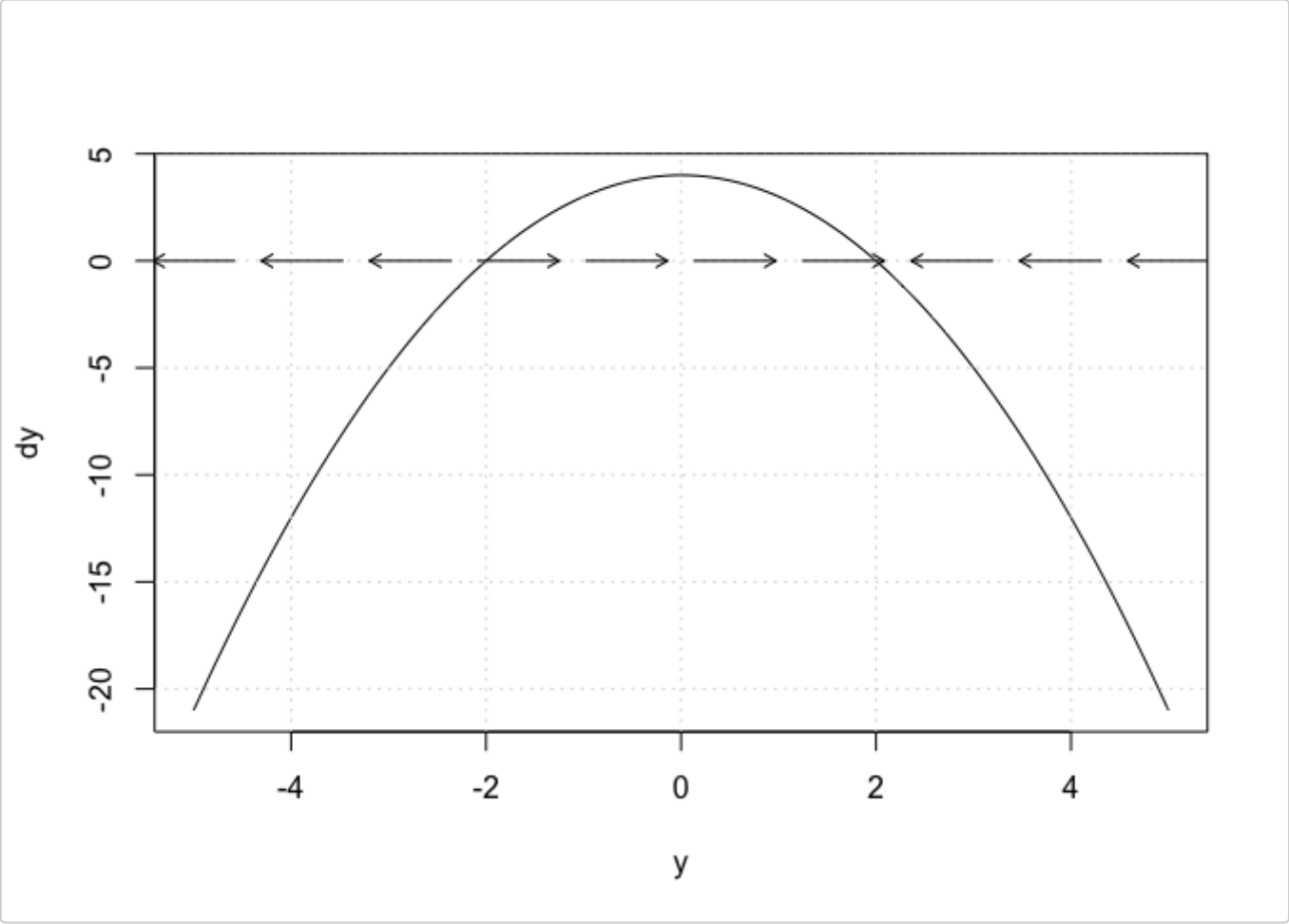
$$f(y_*) = 0 \Rightarrow 4 - y_*^2 = 0 \Rightarrow (2 - y_*)(2 + y_*) = 0 \Rightarrow y_* = -2, 2.$$

For the equilibrium points however, just as much as we are interested in their location, we are interested in whether they are stable or unstable. Here, informally, being stable means that if the system is placed a small distance away from the equilibrium point, it will remain close to this equilibrium point. Whilst being unstable means a small perturbation away from the equilibrium point causes the solution to diverge large distances away. More precisely, the definition of stability can be stated as: if for every $\epsilon > 0$, there exists $\delta > 0$ such that whenever $|y(0) - y_*| < \delta$ then $|y(t) - y_*| < \epsilon$ for all t.

Classically, to determine the stability of any located equilibrium points, we have two options. The first method is the phase portrait. Indeed, our earlier decision to restrict our attention to autonomous systems was motivated by the condition required for phase portrait analysis: when we remove time dependence from our systems evolution, it allows us to collapse our qualitative analysis from the t-y plane to simply considering how $f(y)$ varies with y.

So, in phase portrait analysis, we first plot $f(y)$ against y. From the definition of our autonomous ODE it should be easy to see that whenever $f(y) > 0$, y will increase. Whilst whenever $f(y) < 0$, y will decrease. Moreover, the locations where $f(y)$ cross the y-axis are exactly the equilibrium points (thus this plot can be useful for locating equilibrium points). Therefore, we can represent the evolution of y in this plot by simply placing arrows along the y-axis indicating whether y would be increasing or decreasing. Then, the cases where arrows either side of an equilibrium point towards each other denote stability, whilst when they point away they denote instability.

Again, as an example we consider the system $dy/dt = 4 - y^2$. Plotting $f(y) = 4 - y^2$ against y and adding arrows as described we acquire the following graph:



Thus, we can see that the equilibrium point $y_* = 2$ is stable, whilst $y_* = -2$ is unstable. Indeed, looking back at the trajectories we plotted earlier, we can observe that solutions do converge towards $y = 2$, but away from $y = -2$.

Moreover, we now note an important consequence of requiring f to be continuous and differentiable: that the solution curves cannot touch each other (except to converge at equilibrium points). This is because these conditions on f guarantee solutions to autonmous ODEs are unique. We can observe in our earlier plot of several trajectories of the system $dy/dt = 4 - y^2$ that this is indeed the case.

Our second option to perform such stability analyses comes from utilising the Taylor Series expansion of f. We begin by supposing we are a small distance $\delta(0)$ away from our fixed point y_*, i.e., $y(0) = y_* + \delta(0)$, and in general that $y(t) = y_* + \delta(t)$. Then we can write the Taylor Series of f as:

$$f(y_* + \delta) = f(y_*) + \delta \frac{\partial f}{\partial y}(y_*) + o(\delta),$$

assuming higher order terms can be neglected. Recalling $f(y_*) = 0$, our autonomous ODE becomes:

$$\frac{d}{dt}(y_* + \delta) = \delta \frac{\partial f}{\partial y}(y_*) \Rightarrow \frac{d\delta}{dt} = \delta \frac{\partial f}{\partial y}(y_*) = k\delta.$$

This ODE for δ can be solved easily to give $\delta(t) = \delta(0)e^{kt}$. Then stability can be found based upon whether $\delta(t)$ grows or decays as t increases, i.e., we have:

$$k = \frac{\partial f}{\partial y}(y_*) \begin{cases} > 0 & : \text{ Stable,} \\ < 0 & : \text{ Unstable.} \end{cases}$$

Here, k is sometimes referred to as the discriminant, whilst this approach is often referred to as perturbation analysis.

Returning to our example ODE again, we can perform such analysis easily:

$$\frac{\partial f}{\partial y}(y_*) = -2y_* = \begin{cases} -4 & : \ y_* = 2, \\ +4 & : \ y_* = -2. \end{cases}$$

Thus we draw the same conclusion as before; $y_* = 2$ is stable, and $y_* = -2$ is unstable. We will see later how one of the programs in **phaseR** can perform this stability analysis for us.

It should now be clear that we can clearly state if $y(0) > 2$ or $0 < y(0) < 2$ then the solution will eventually approach $y = 2$. However, if $y(0) < 0$, $y \to -\infty$ as $t \to \infty$. Such general statements can often be made as a result of the phase plane analyses.

It is worthwhile noting here that if we find:

$$\frac{\partial f}{\partial y}(y_*) = 0,$$

then to this order of the Taylor Series no conclusion can be drawn about stability.

So, we have now observed all of the key components required to perform a qualitative analysis upon a one-dimensional autonomous ODE. We begin by plotting the flow field, and from this several trajectories. We then identify the equilibrium points and choose a method to determine their stability. All such techniques are available in this package, and we will later discuss how to implement them. First, however, we will discuss how these methods can be generalised to coupled ODEs.

First order two-dimensional dynamical systems

Autonomous two-dimensional ordinary differential equations

As may well be expected, things get substantially more complex in the world of coupled ODEs; very rarely can such systems be solved analytically. Unfortunately, the analysis of many real life systems does involve interacting variables, and so these systems are not uncommon. Here, the first restriction we make is to the case of two-dimensional (or two variable) systems; a necessity for the following techniques to be implementable (this is often considered a disadvantage of phase plane methods; that they cannot be generalised to more than two dimensions. Fortunately however, many systems can be approximated to two dimensions). These systems can be written in their most general form as:

$$\frac{dx}{dt} = f(x, y, t), \ \frac{dy}{dt} = g(x, y, t),$$

for $x = x(t)$ and $y = y(t)$. In this most general case numerical solution would almost certainly be the only way forward. However, if we again make the restriction to autonomous systems, the phase plane methods from one dimension can be generalised to avoid the need for numerical integration. Following the same route as in the one-dimensional case, an autonomous system can be written for two coupled ODEs as:

$$\frac{dx}{dt} = f(x, y), \ \frac{dy}{dt} = g(x, y).$$

As before, the definition of the flow field (more commonly, and from here on out, referred to as the velocity field) and equilibrium points, as well as their stability, will be important. Here however, we also meet the

concept of nullclines and stable/unstable manifolds. Again, we formally require that f and g are continuous, (and now) partially differentiable functions.

Before we proceed to discuss the generalisation of our earlier techniques to such two-dimensional systems, it is useful to note that certain second order ODEs can indeed be re-cast by variable substitution into a system of the type above. Indeed, consider the second order ODE given by:

$$a(y)\frac{d^2y}{dt} + b(y)\frac{dy}{dt} + c(y) = 0.$$

We make the substitution $x = dy/dt$ and re-write our system as:

$$\frac{dy}{dt} = x,\ \frac{dx}{dt} = \frac{1}{a(y)}\{-b(y)x - c(y)\}.$$

In this way, it is actually possible to analyse the behaviour of certain second order ODEs using the methods for coupled first order ODEs.

The velocity field

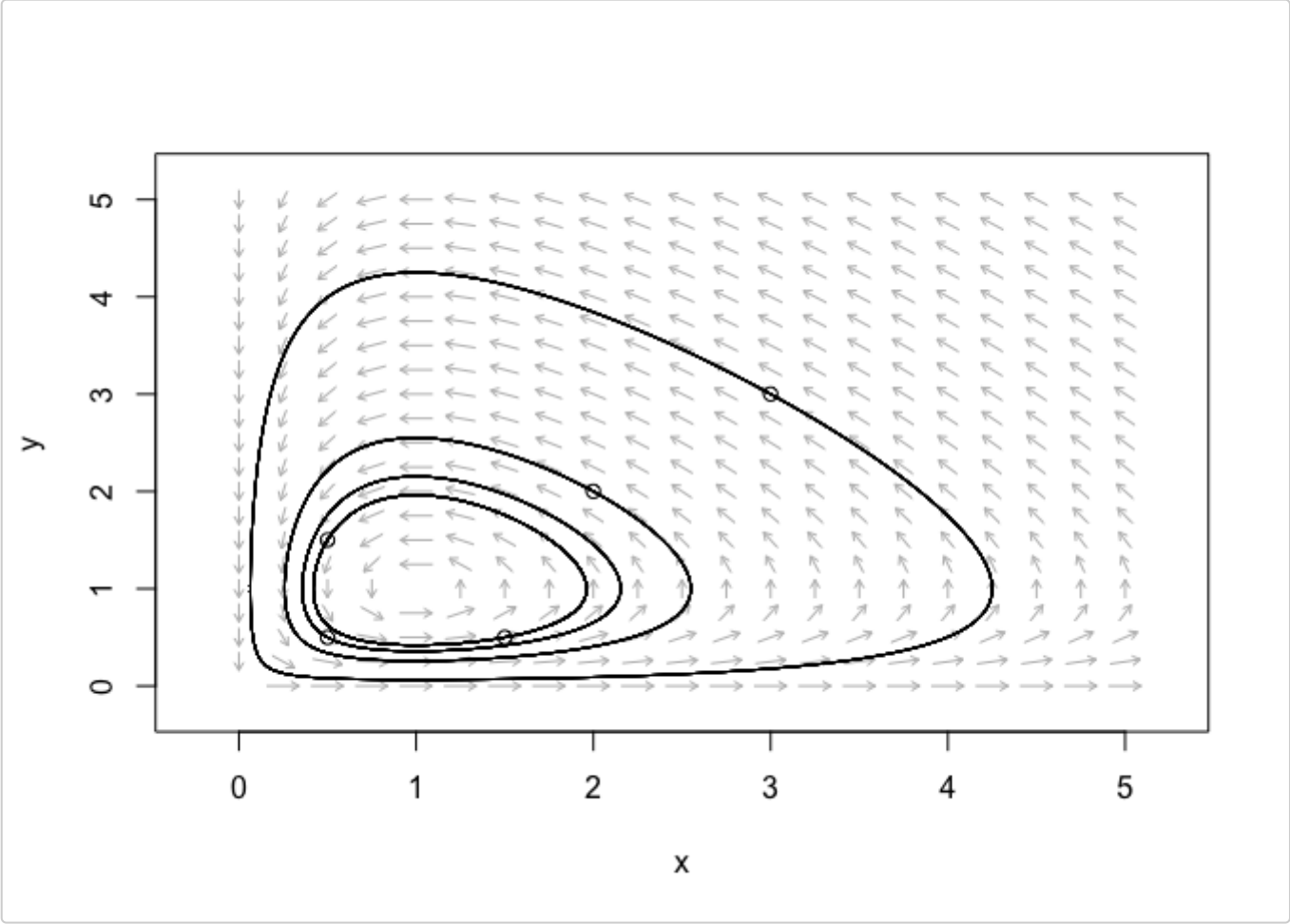
We observed earlier that the restriction to autonomous ODEs in the one-dimensional case allowed us to restrict attention to the phase portrait; the plot of $f(y)$ against y. In the two-dimensional case, this restriction allows us to limit our attention to the plane produced by the two dependent variables. Using our notation from the two-dimensional autonomous system prescribed above, this is the x-y plane, and is often referred to in this context as the phase plane. Representation in this manner proves to be the most convenient way to visualise the system.

In this plane, we can produce a plot analogous to the flow field discussed earlier for one-dimensional autonomous systems: at many points (x, y) we plot a small line segment (a vector) in the direction given by the rates of change of x and y, which are provided by $f(x, y)$ and $g(x, y)$. This plot is usually referred to as the velocity field, or sometimes the direction field, and perhaps confusingly, the phase portrait. We can then again for any point trace out the trajectory of a solution by using the fact that it must pass through our line segments in a tangential manner. Repeating this procedure for several starting points, we can again build up a family of solutions and a good picture of the behaviour of solutions to our two-dimensional autonomous ODE system. As before, however, it is important to understand that using this method is only an approximation to performing numerical integration, and things can here become very ambiguous around certain points (the equilibria).

To illustrate the concept of the velocity field, we again turn to an example. This time consider the system given by (this is an example of a Lotka-Volterra model, which will be discussed more later):

$$\frac{dx}{dt} = x - xy,\ \frac{dy}{dt} = xy - y.$$

Using **phaseR** we can produce the following plot of the velocity field along with several trajectories:



Analogous to the one-dimensional analysis performed earlier, we observe how our restriction to continuous partially differentiable f and g ensures that trajectories cannot cross (though they can again converge at equilibria).

What is more, as before, some texts advise to plot the vectors at lengths reflecting the magnitudes of the rates of change of x and y. However, some small arbitrary length usually still remains the best option.

Finally, as was the case in the one-dimensional analysis, some texts here again refer to the method of isoclines for tracing out trajectories. Isoclines are now defined as curves of constant gradient in the x-y plane, i.e.:

$$\frac{dy}{dx} = \frac{g(x,y)}{f(x,y)} = \alpha,$$

for different values of α. Once more, trajectories would be produced by using the fact that we know the angle they should cut each isocline. We will make no further reference to isoclines in these notes; hopefully for reasons discussed below it should become clear why certain tricks make the need for to plot isoclines very unusual.

Nullclines

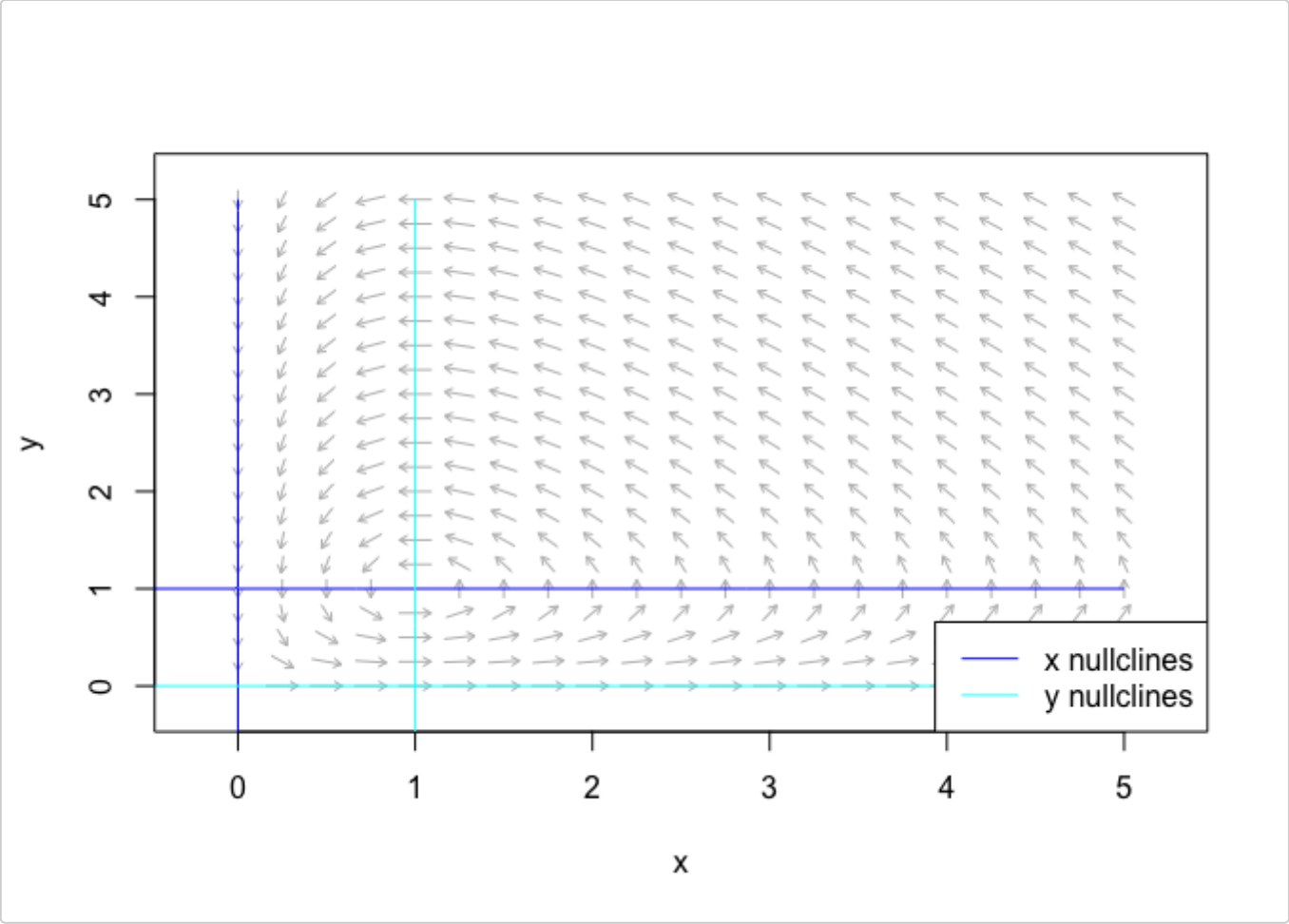
A new important concept in the case of two-dimensional systems is that of nullclines. Here, x-nullclines are defined by the locations where $f(x,y) = 0$, whilst the y-nullclines are defined by the locations where $g(x,y) = 0$. Thus, the x- and y-nullclines define the locations where x and y, respectively, do not change

with t. As a consequence, when plotting a vector field by hand it is usually wise to plot the nullclines first, as the line segments (or vectors) along them move parallel to the x- and y-axes.

Returning to our example given above, we can find its nullclines as follows:

$$x : x - xy = 0 \Rightarrow x(1 - y) = 0 \Rightarrow x = 0 \text{ or } y = 1,$$
$$y : xy - y = 0 \Rightarrow y(x - 1) = 0 \Rightarrow x = 1 \text{ or } y = 0.$$

We can then plot these nullclines along with the velocity field:



Equilibrium points and stability

Equilibrium points maintain their importance in two dimensions. Here, generalisation defines them to be the locations (x_*, y_*) where:

$$f(x_*, y_*) = g(x_*, y_*) = 0.$$

Thus, another utility of nullclines immediately becomes apparent; the locations where x- and y-nullclines cross are the equilibria. However, it is important to note that locations where x-nullclines or y-nullclines cross each other, are not equilibria. For this reason it is usually useful to plot x- and y-nullclines in different colours.

Revisiting our example system from earlier, it is easy to find either analytically, or from the nullcline plot, that two equilibria are present; the points $(0, 0)$ and $(1, 1)$.

We now note a useful fact about equilibria and nullclines, which can be observed in the plot above. On opposite sides of an equilibria, along a nullcline, the orientation of the velocity arrows is reversed. This is a property shared by the majority of systems (with the exception being certain singular cases where the Jacobian that we meet later is zero). Because trajectories must be continuous, the direction vectors must vary

continuously from one point to another everywhere else. So in most cases when seeking to plot the velocity field, it suffices to determine direction vectors at a few select locations and deduce the rest by preserving continuity and switching orientation when an equilibrium point is crossed. It is this trick that makes plotting numerous isoclines often unnecessary.

Again, we must now turn our attention to the stability of the equilibrium points. In two dimensions the definition of stability remains the same, but as well as determining whether a point is stable or unstable, we can additionally classify the nature in which trajectories move away or towards it. Ultimately, as in the one-dimensional case, we aim to identify the long term behaviour of solutions in different regions of the plane.

In this case, we must make use of a mathematical argument; to gain a full understanding use of a graph is not enough (though it can be useful in certain singular cases). Here, many texts distinguish between the case of linear and non-linear systems, and so we will also first make such a distinction, though ultimately we will treat these two types of system equally.

The linear version of a two-dimensional autonomous ODE system is given by:

$$\frac{dx}{dt} = ax + by, \ \frac{dy}{dt} = cx + dy,$$

or in matrix form:

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \frac{d}{dt}\boldsymbol{x} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = A\boldsymbol{x}.$$

Now, provided $\det(A) \neq 0$, a unique solution exists to this system (we will return to discuss the case $\det(A) = 0$ later), and it can be written in the form:

$$\boldsymbol{x} = C_1 e^{\lambda_1 t}\boldsymbol{e}_1 + C_2 e^{\lambda_2 t}\boldsymbol{e}_2,$$

where λ_1 and λ_2 are the eigenvalues of A, $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are their corresponding eigenvectors, and C_1 and C_2 are arbitrary constants. From here we can determine stability based on the values of the eigenvalues. However, the procedure from here on out is the same as that for non-linear systems and so we will move to the analysis required for the more complex case of non-linearity.

So, from the above it should be obvious that provided $\det(A) \neq 0$, linear systems have only one equilibrium point; $(0, 0)$. Non-linear systems, however, are much more complicated; they can have multiple equilibria and even display limit cycle behaviour (as defined later). However, close to an equilibrium point, behaviour can be usually understood by linearising the model about the equilibria.

To do this we proceed in a similar fashion to the Taylor Series method for one-dimensional autonomous ODEs. We suppose we have an equilibrium point given by (x_*, y_*) and that our system lies initially slightly away from this point at $(x_* + \delta(0), y_* + \epsilon(0))$, and in general at $(x_* + \delta(t), y_* + \epsilon(t))$. Then, using the Taylor expansion for f, our differential equation for $\boldsymbol{x}$ becomes:

$$\begin{aligned}
\frac{d\delta}{dt} &= f(x_* + \delta, y_* + \epsilon), \\
&= f(x_*, y_*) + \delta\frac{\partial f}{\partial x}(x_*, y_*) + \epsilon\frac{\partial f}{\partial y}(x_*, y_*) + o(\delta) + o(\epsilon), \\
&= \delta\frac{\partial f}{\partial x}(x_*, y_*) + \epsilon\frac{\partial f}{\partial y}(x_*, y_*) + o(\delta) + o(\epsilon).
\end{aligned}$$

Similarly, our differential equation for y becomes:

$$\frac{d\epsilon}{dt} = \delta\frac{\partial g}{\partial x}(x_*, y_*) + \epsilon\frac{\partial g}{\partial y}(x_*, y_*) + o(\delta) + o(\epsilon).$$

Here we have again assumed terms of second order and higher are negligible.

If we write this system in matrix form we acquire:

$$\frac{d}{dt}\boldsymbol{\delta} = \left.\begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}\right|_{(x_*, y_*)} \boldsymbol{\delta} = \left.\begin{pmatrix} f_x & f_y \\ g_x & g_y \end{pmatrix}\right|_{(x_*, y_*)} \boldsymbol{\delta} = J\boldsymbol{\delta},$$

where J is called the Jacobian of the system, and:

$$\boldsymbol{\delta} = \begin{pmatrix} \delta \\ \epsilon \end{pmatrix}.$$

If we let the eigenvalues of J be denoted by λ_1 and λ_2, with corresponding eigenvectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$, then the general solution to the above is:

$$\boldsymbol{\delta} = C_1 e^{\lambda_1 t}\boldsymbol{e}_1 + C_2 e^{\lambda_2 t}\boldsymbol{e}_2,$$

where C_1 and C_2 are arbitrary constants.

Considering the linear system from earlier we find that $J = A$. Thus, stability of $(0, 0)$ in the linear case can be determined by the same classification rules as below for the non-linear case. Specifically, we have:

- If λ_1 and λ_2 are both real and positive, the solution for δ moves outwards in both the $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ directions. Thus $|\boldsymbol{\delta}|$ will increase exponentially with t and so trajectories move away from the equilibrium point. This is the definition of an unstable node.
- If λ_1 and λ_2 are both real and negative, $|\boldsymbol{\delta}|$ will decrease exponentially and trajectories move towards the equilibrium point. This is the definition of a stable node.
- If λ_1 and λ_2 are both real but have opposite sign, e.g., $\lambda_1 < 0$ with $\lambda_2 > 0$, trajectories move outwards along $\boldsymbol{e}_2$, but inwards along $\boldsymbol{e}_1$. Unless $\boldsymbol{\delta}$ initially lies exactly parallel to $\boldsymbol{e}_1$, the solution will eventually move away from the equilibrium point; and thus it is unstable. This is the definition of a saddle point.
- If λ_1 and λ_2 are complex, e.g., they are $a \pm ib$ for some a and b, then the solution for $\boldsymbol{\delta}$ can be rewritten as:

$$\begin{aligned} \boldsymbol{\delta} &= e^{at}[C_1(\cos bt + i \sin bt)\boldsymbol{e}_1 + C_2(\cos bt - i \sin bt)\boldsymbol{e}_2], \\ &= e^{at}(\boldsymbol{A} \cos bt + \boldsymbol{B} \sin bt), \end{aligned}$$

  where $\boldsymbol{A} = C_1\boldsymbol{e}_1 + C_2\boldsymbol{e}_2$ and $\boldsymbol{B} = i(C_1\boldsymbol{e}_1 - C_2\boldsymbol{e}_2)$. Thus, from this form we can see that the solution will spiral around the equilibrium point. If $a > 0$ then with each loop $|\boldsymbol{\delta}|$ increases; this is the definition of an unstable focus. If $a < 0$ then with each loop $|\boldsymbol{\delta}|$ decreases; this is the definition of a stable focus. If $a = 0$ then the solution continues in a closed loop; this is the definition of a centre.

Fortunately, it is not actually necessary to find the exact values of the eigenvalues (though computationally this is not a difficult task, by hand it can be time consuming). We only require the signs of the eigenvalues, or of their real parts, to perform the classification. To this end, consider the characteristic equation of J:

$$(f_x - \lambda)(g_y - \lambda) - f_y g_x = 0.$$

Observing that $\mathrm{tr}(J) = T = f_x + g_y$ and $\det(J) = \Delta = f_x g_y - f_y g_x$, we can write the characteristic equation of J as:

$$\begin{aligned} &\lambda^2 - T\lambda + \Delta = 0, \\ &\implies \lambda = \frac{T \pm \sqrt{T^2 - 4\Delta}}{2}. \end{aligned}$$

From this we can draw up the following table that allows us to classify the equilibria using the signs of T, Δ, and $T^2 - 4\Delta$:

Δ	$T^2 - 4\Delta$	**Eigenvalues of J**	T	**Classification**
< 0	> 0	Real, opposite signs	N/A	Saddle
> 0	> 0	Real, same signs	< 0	Stable node
> 0	> 0	Real, same signs	> 0	Unstable node
> 0	< 0	Complex conjugate pair	< 0	Stable focus
> 0	< 0	Complex conjugate pair	$= 0$	Centre
> 0	< 0	Complex conjugate pair	> 0	Unstable focus
$= 0$	N/A		N/A	Indeterminate
N/A	$= 0$	Real, equal	< 0	Stable node
N/A	$= 0$	Real, equal	> 0	Unstable node

Note: Focus' are often referred to as spirals.

From here, we will refer to $T^2 - 4\Delta$ as the discriminant.

To be more precise, for the case of $\Delta = 0$; we would have to consider second-order terms in the Taylor Series approximation made earlier in order to determine stability. Alternatively, in this case, use of the velocity field and traced trajectories can allow us to identify if the point is stable or not.

Returning to our example system from earlier, taking partial derivatives we can compute the Jacobian at any equilibrium point (x_*, y_*):

$$J = \begin{pmatrix} 1 - y_* & -x_* \\ y_* & x_* - 1 \end{pmatrix}.$$

Thus, at $(0, 0)$, we have:

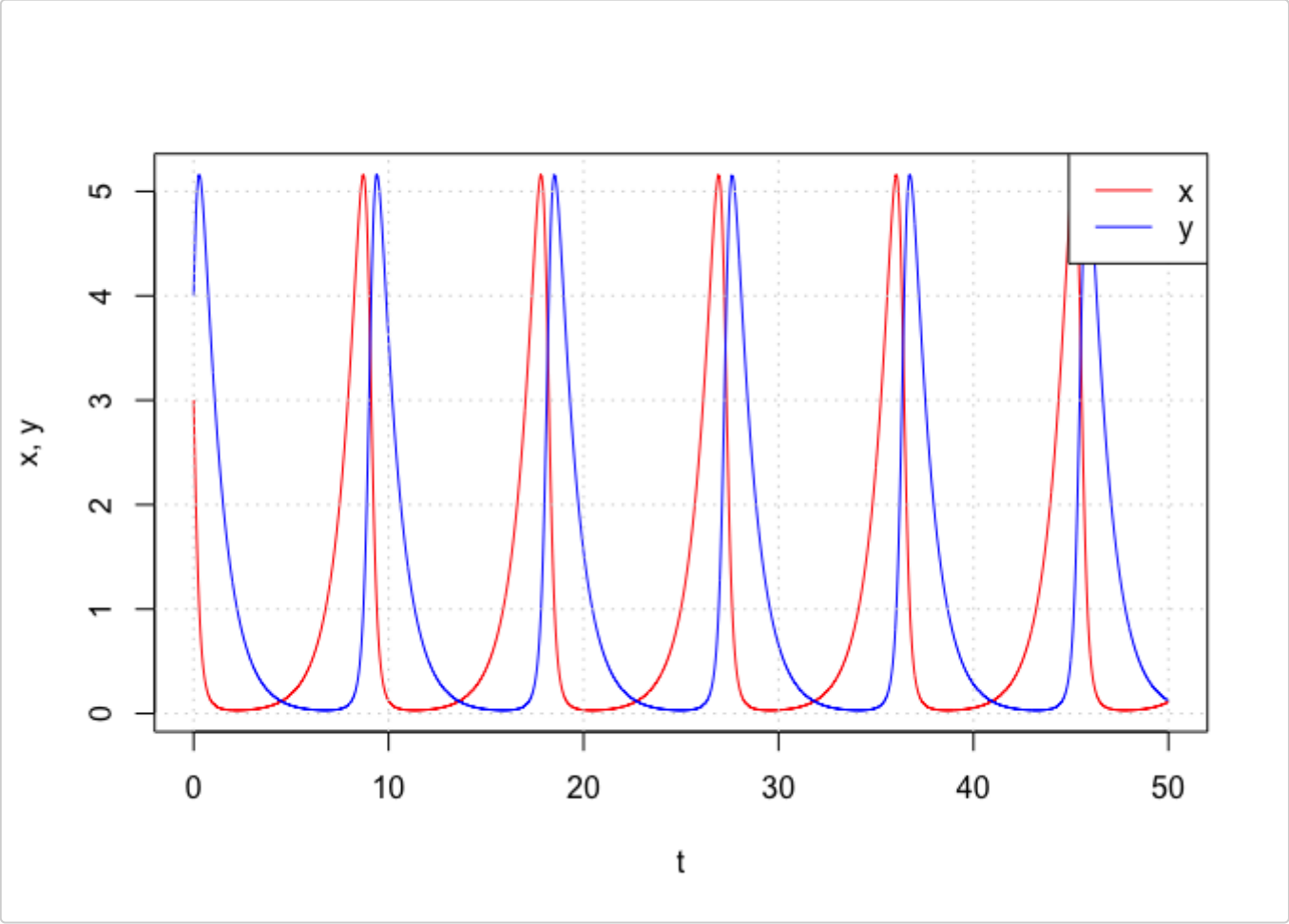
$$J = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\Bigg|_{(0,0)}.$$

So $\text{tr}(J) = T = 0$ and $\det(J) = \Delta = -1$; which from our table above makes $(0, 0)$ a saddle point. For $(1, 1)$, however, we have:

$$J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\Bigg|_{(1,1)}.$$

Therefore, $\text{tr}(J) = T = 0$ and $\det(J) = \Delta = 1$; which from our table above makes $(1, 1)$ a centre. Indeed, if we look back at our earlier plot, we can observe trajectories diverging away from $(0, 0)$, but traversing around $(1, 1)$. Again, we will see later how this analysis can be performed for us in **phaseR**.

As a last point of interest, note that it is sometimes interesting to plot x and y trajectories against t. For the case of $(x_0, y_0) = (3, 4)$ in our example system this results in the following plot where we can witness the oscillating nature of x and y:



The utility of such plots becomes more apparent in cases where trajectories can be seen to converge upon an equilibrium point; indicating its stability and often whether it is a node or focus.

So, we have now discussed all of the techniques required to a perform phase plane analysis of a two-dimensional autonomous ODE system. We begin by locating and plotting nullclines, using these to create the velocity field. From this we can plot numerous trajectories. We then identify any equilibria and classify them according to the table above. Before we move to discuss how to use **phaseR** however, we will give brief attention to two additional considerations of interest in the analysis of two-dimensional autonomous ODE systems.

Stable and unstable manifolds

The stable and unstable manifolds of an equilibrium point give a formal mathematical definition to the general notions embodied in the idea of a point being an attractor or repellor. Precisely, the stable and unstable manifolds are defined as the set of all points in the plane which tend to the equilibrium point as time goes to positive, and respectively negative, infinity. For example, a system with a single stable node will have the whole plane as its stable manifold and just the node itself as its unstable manifold.

Things are slightly more interesting for saddle points, however. Recall that saddle points have a positive eigenvalue and a negative eigenvalue. The stable (unstable) manifold of a saddle point is the eigenvector corresponding to the positive (negative) eigenvalue. Moreover, the stable and unstable manifolds of saddle points are very important to understanding the global behaviour of the orbits of the whole planar system. They form separatrices; partitioning the plane into invariant regions of differing dynamics. We will see evidence of such separatrices in an example later when we observe how **phaseR** can be used to plot the stable and unstable manifolds of any saddle point.

Limit cycles

Non-linear systems can also exhibit a type of behaviour known as a limit cycle. In the phase plane, a limit cycle is defined as an isolated closed orbit. Closed here denotes the periodic nature of the motion and isolated denotes the limiting nature of the cycle; with nearby trajectories converging to, or diverging away from, it. Limit cycles have a complex mathematical theory behind them, which we will not go into here. We will however observe an example of limit cycle behaviour later on.

phaseR usage

To perform all of the techniques discussed above, the package contains nine key functions. Below is a description of each ones utility, as well as the user specifiable input variables, and the output you can expect. The description of the inputs and outputs is repetitive on purpose to reflect how many are common across programs. For the most part, they are also equal to the descriptions seen in the function help files in **R**. In addition, we will continue to use x, y, and t as our variables.

`drawManifolds()`

```
drawManifolds(deriv, y0 = NULL, parameters = NULL, tstep = 0.1, tend = 1000,
              col = c("green", "red"), add.legend = TRUE,
              state.names = c("x", "y"), ...)
```

This function allows the user to plot the stable and unstable manifolds of a saddle point. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `y0`: The initial point from which a saddle will be searched for. This can either be a numeric vector of length two, reflecting the location of the two dependent variables, or alternatively this can be specified as `NULL`, and then `graphics::locator()` can be used to specify the initial point on a plot. Defaults to `NULL`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `tstep`: The step length of the independent variable, used in numerical integration. Decreasing the absolute magnitude of `tstep` theoretically makes the numerical integration more accurate, but increases computation time. Defaults to `0.01`.
- `tend`: The final time of the numerical integration performed to identify the manifolds.
- `col`: Sets the colours used for the stable and unstable manifolds. Should be a character vector of length two. Will be reset accordingly if it is of the wrong length. Defaults to `c("green", "red")`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.
- `add.legend`: Logical. If `TRUE`, a legend is added to the plot. Defaults to `TRUE`.
- `...`: Additional arguments to be passed to `graphics::plot()`.

Returned by `drawManifolds()` is a list containing all of the input variables as well as following components:

- `stable.1`: A numeric matrix whose columns are the numerically computed values of the dependent variables for part of the stable manifold.
- `stable.2`: A numeric matrix whose columns are the numerically computed values of the dependent variables for part of the stable manifold.

- `unstable.1`: A numeric matrix whose columns are the numerically computed values of the dependent variables for part of the unstable manifold.
- `unstable.2`: A numeric matrix whose columns are the numerically computed values of the dependent variables for part of the unstable manifold.
- `ystar`: Location of the identified equilibrium point.

`findEquilibrium()`

```
findEquilibrium(deriv, y0 = NULL, parameters = NULL, system = "two.dim",
                tol = 1e-16, max.iter = 50, h = 1e-6, plot.it = FALSE,
                summary = TRUE,
                state.names = if (system == "two.dim") c("x", "y") else "y")
```

This function allows the user to search for an equilibrium point. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `y0`: The starting point of the search. In the case of a one-dimensional system, this should be a numeric vector of length one indicating the location of the dependent variable initially. In the case of a two-dimensional system, this should be a numeric vector of length two reflecting the location of the two dependent variables initially. Alternatively this can be specified as `NULL`, and then `graphics::locator()` can be used to specify the initial point on a plot. Defaults to `NULL`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `tol`: The tolerance for the convergence of the search algorithm. Defaults to `1e-16`.
- `max.iter`: The maximum allowed number of iterations of the search algorithm. Defaults to `50`.
- `h`: Step length used to approximate the derivative(s). Defaults to `1e-6`.
- `plot.it`: Logical. If `TRUE`, a point is plotted at the identified equilibrium point, with shape corresponding to its classification.
- `summary`: Logical. If `TRUE`, a summary of the classification of the equilibrium point is printed to the console. Defaults to `TRUE`.

Returned by `findEquilibrium()` is a list object containing all of the input variables as well as following components (the exact make up is dependent on the value of `system`):

- `Delta`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `discriminant`: In the one-dimensional system case, the value of the discriminant used in perturbation analysis to assess stability. In the two-dimensional system case, the value of `tr^2 - 4*Delta` ($T^2 - 4\Delta$).
- `eigenvalues`: In the two-dimensional system case, the value of the Jacobian's eigenvalues at `ystar`.
- `eigenvectors`: In the two-dimensional system case, the value of the Jacobian's eigenvectors at `ystar`.
- `jacobian`: In the two-dimensional system case, the Jacobian at `ystar`.
- `tr`: In the two-dimensional system case, the value of the Jacobian's trace at `ystar`.

`flowField()`

```
flowField(deriv, xlim, ylim, parameters = NULL, system = "two.dim", points = 21,
          col = "gray", arrow.type = "equal", arrow.head = 0.05, frac = 1,
          add = TRUE,
          state.names = if (system == "two.dim") c("x", "y") else "y",
          xlab = if (system == "two.dim") state.names[1] else "t",
          ylab = if (system == "two.dim") state.names[2] else state.names[1],
          ...)
```

This function allows the user to plot the flow or velocity field for a one- or two-dimensional autonomous ODE system. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `xlim`: In the case of a two-dimensional system, this sets the limits of the first dependent variable in which gradient reflecting line segments should be plotted. In the case of a one-dimensional system, this sets the limits of the independent variable in which these line segments should be plotted. Should be a numeric vector of length two.
- `ylim`: In the case of a two-dimensional system this sets the limits of the second dependent variable in which gradient reflecting line segments should be plotted. In the case of a one-dimensional system, this sets the limits of the dependent variable in which these line segments should be plotted. Should be a numeric vector of length two.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `points`: Sets the density of the line segments to be plotted; `points` segments will be plotted in the x and y directions. Fine tuning here, by shifting `points` up and down, allows for the creation of more aesthetically pleasing plots. Defaults to `11`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `col`: Sets the colour of the plotted line segments. Should be a character vector of length one. Will be reset accordingly if it is of the wrong length. Defaults to `"gray"`.
- `arrow.type`: Sets the type of line segments plotted. If set to `"proportional"` the length of the line segments reflects the magnitude of the derivative. If set to `"equal"` the line segments take equal lengths, simply reflecting the gradient of the derivative(s). Defaults to `"equal"`.
- `arrow.head`: Sets the length of the arrow heads. Passed to `graphics::arrows()`. Defaults to `0.05`.
- `frac`: Sets the fraction of the theoretical maximum length line segments can take without overlapping, that they can actually attain. In practice, `frac` can be set to greater than 1 without line segments overlapping. Fine tuning here assists the creation of aesthetically pleasing plots. Defaults to `1`.
- `add`: Logical. If `TRUE`, the flow field is added to an existing plot. If `FALSE`, a new plot is created. Defaults to `TRUE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.
- `xlab` Label for the x-axis of the resulting plot.
- `ylab` Label for the y-axis of the resulting plot.
- `...`: Additional arguments to be passed to either `graphics::plot()` or `graphics::arrows()`.

Returned by `flowField()` is a list object containing all of the input variables as well as following components (the exact the exact make up is dependent on the value of `system`):

- `dx`: A numeric matrix. In the case of a two-dimensional system, the values of the derivative of the first dependent derivative at all evaluated points.
- `dy`: A numeric matrix. In the case of a two-dimensional system, the values of the derivative of the second dependent variable at all evaluated points. In the case of a one-dimensional system, the values

of the derivative of the dependent variable at all evaluated points.
- `x`: A numeric vector. In the case of a two-dimensional system, the values of the first dependent variable at which the derivatives were computed. In the case of a one-dimensional system, the values of the independent variable at which the derivatives were computed.
- `y`: A numeric vector. In the case of a two-dimensional system, the values of the second dependent variable at which the derivatives were computed. In the case of a one-dimensional system, the values of the dependent variable at which the derivatives were computed.

`nullclines()`

```
nullclines(deriv, xlim, ylim, parameters = NULL, system = "two.dim",
           points = 101, col = c("blue", "cyan"), add = TRUE, add.legend = TRUE,
           state.names = if(system == "two.dim") c("x", "y") else "y", ...)
```

This function allows the user to plot nullclines for two-dimensional autonomous ODE systems. Or it can be used to plot horizontal lines at equilibrium points for one dimensional autonomous ODE systems. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `xlim`: In the case of a two-dimensional system, this sets the limits of the first dependent variable in which gradient reflecting line segments should be plotted. In the case of a one-dimensional system, this sets the limits of the independent variable in which these line segments should be plotted. Should be a numeric vector length two.
- `ylim`: In the case of a two-dimensional system this sets the limits of the second dependent variable in which gradient reflecting line segments should be plotted. In the case of a one-dimensional system, this sets the limits of the dependent variable in which these line segments should be plotted. Should be a numeric vector of length two.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `points`: Sets the density at which derivatives are computed; `points` `points` derivatives will be computed. Levels of zero gradient are identified using these computations and the function `graphics::contour()`. Increasing the value of points improves identification of nullclines, but increases computation time. Defaults to `101`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `col`: In the case of a two-dimensional system, sets the colours used for the x- and y-nullclines. In the case of a one-dimensional system, sets the colour of the lines plotted horizontally along the equilibria. Should be character vector of length two. Will be reset accordingly if it is of the length. Defaults to `c("blue", "cyan")`.
- `add`: Logical. If `TRUE`, the nullclines are added to an existing plot. If `FALSE`, a new plot is created. Defaults to `TRUE`.
- `add.legend`: Logical. If `TRUE`, a `graphics::legend()` is added to the plots. Defaults to `TRUE`.
- `...`: Additional arguments to be passed to either `graphics::plot()` or `graphics::contour()`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.

Returned by `nullclines()` is a list object containing all of the input variables as well as following components (the exact the exact make up is dependent upon the value of system):

- `dx`: A numeric matrix. In the case of a two-dimensional system, the values of the derivative of the first dependent derivative at all evaluated points.

- `dy`: A numeric matrix. In the case of a two dimensional system, the values of the derivative of the second dependent variable at all evaluated points. In the case of a one dimensional system, the values of the derivative of the dependent variable at all evaluated points.
- `x`: A numeric vector. In the case of a two dimensional system, the values of the first dependent variable at which the derivatives were computed. In the case of a one dimensional system, the values of the independent variable at which the derivatives were computed.
- `y`: A numeric vector. In the case of a two dimensional system, the values of the second dependent variable at which the derivatives were computed. In the case of a one dimensional system, the values of the dependent variable at which the derivatives were computed.

numericalSolution()

```
numericalSolution(deriv, y0 = NULL, tlim, tstep = 0.01, parameters = NULL,
                  type = "one", col = c("red", "blue"), add.grid = TRUE,
                  add.legend = TRUE, state.names = c("x", "y"), xlab = "t",
                  ylab = state.names)
```

Used for two-dimensional systems, this function numerically solves the autonomous ODE system for a given initial condition. It then plots the dependent variables against the independent variable. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `y0`: The initial condition. Should be a numeric vector of length two reflecting the location of the two dependent variables initially.
- `tlim`: Sets the limits of the independent variable for which the solution should be plotted. Should be a numeric vector of length two. If `tlim[2] > tlim[1]`, then `tstep` should be negative to indicate a backwards trajectory.
- `tstep`: The step length of the independent variable, used in numerical integration. Decreasing the absolute magnitude of `tstep` theoretically makes the numerical integration more accurate, but increases computation time. Defaults to `0.01`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `type`: If set to `"one"` the trajectories are plotted on the same graph. If set to `"two"` they are plotted on separate graphs. Defaults to `"one"`.
- `col`: Sets the colours of the trajectories of the two dependent variables. Should be a character vector of length two. Will be reset accordingly if it is of the wrong length. Defaults to `c("red", "blue")`.
- `add.grid`: Logical. If `TRUE`, grids are added to the plots. Defaults to `TRUE`.
- `add.legend`: Logical. If `TRUE`, a `graphics::legend()` is added to the plots. Defaults to `TRUE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.
- `xlab`: Label for the x-axis of the resulting plot.
- `ylab`: Label for the y-axis of the resulting plot.
- `...`: Additional arguments to be passed to `graphics::plot()`.

Here, the numerical integration is performed by the function `deSolve::ode()`.

Returned by `numericalSolution()` is a list object containing all of the input variables as well as following:

- `t`: A numeric vector containing the values of the independent variable at each integration step.
- `x`: A numeric vector containing the numerically computed values of the first dependent variable at each integration step.

- `y`: A numeric vector containing the numerically computed values of the second dependent variable at each integration step.

phasePlaneAnalysis()

```
phasePlaneAnalysis(deriv, xlim, ylim, tend = 100, parameters = NULL,
                   system = "two.dim", add = FALSE,
                   state.names = if (system == "two.dim") c("x", "y") else "y")
```

This function allows the user to perform a basic phase plane analysis and produce a simple plot without the need to use the other functions directly. Specifically, a list is provided and the user inputs a value to the console to decide what is added to the plot. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `xlim`: In the case of a two-dimensional system, this sets the limits of the first dependent variable in any subsequent plot. In the case of a one-dimensional system, this sets the limits of the independent variable. Should be a numeric vector of length two.
- `ylim`: In the case of a two-dimensional system this sets the limits of the second dependent variable in any subsequent plot. In the case of a one-dimensional system, this sets the limits of the dependent variable. Should be a numeric vector of length two.
- `tend`: The value of the independent variable to end any subsequent numerical integrations at.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `add`: Logical. If `TRUE`, the chosen features are added to an existing plot. If `FALSE`, a new plot is created. Defaults to `FALSE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.

phasePortrait()

```
phasePortrait(deriv, ylim, ystep = 0.01, parameters = NULL, points = 10,
              frac = 0.75, arrow.head = 0.075, col = "black", add.grid = TRUE,
              state.names = "y", xlab = state.names,
              ylab = paste0("d", state.names), ...)
```

For a one-dimensional autonomous ODE, it plots the phase portrait, i.e., the derivative against the dependent variable. In addition, along the dependent variable axis it plots arrows pointing in the direction of dependent variable change with increasing value of the independent variable. From this stability of equilibrium points (i.e., locations where the horizontal axis is crossed) can be determined:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `ylim`: Sets the limits of the dependent variable for which the derivative should be computed and plotted. Should be a numeric vector of length two.
- `ystep`: Sets the step length of the dependent variable vector for which derivatives are computed and plotted. Decreasing `ystep` makes the resulting plot more accurate, but comes at a small cost to

computation time. Defaults to `0.01`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `points`: Sets the density at which arrows are plotted along the horizontal axis; `points` arrows will be plotted. Fine tuning here, by shifting `points` up and down, allows for the creation of more aesthetically pleasing plots. Defaults to `10`.
- `frac`: Sets the fraction of the theoretical maximum length line segments can take without overlapping, that they actually attain. Fine tuning here assists the creation of aesthetically pleasing plots. Defaults to `0.75`.
- `arrow.head`: Sets the length of the arrow heads. Passed to `graphics::arrows()`. Defaults to `0.075`.
- `col`: Sets the colour of the line in the plot, as well as the arrows. Should be a character vector of length one. Will be reset accordingly if it is of the wrong length. Defaults to `"black"`.
- `add.grid`: Logical. If `TRUE`, a grid is added to the plot. Defaults to `TRUE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.
- `xlab`: Label for the x-axis of the resulting plot.
- `ylab`: Label for the y-axis of the resulting plot.
- `...`: Additional arguments to be passed to either `graphics::plot()` or `graphics::arrows()`.

Returned by `phasePortrait()` is a list object containing all of the input variables as well as the following:

- `dy`: A numeric vector containing the value of the derivative at each evaluated point.
- `y`: A numeric vector containing the values of the dependent variable for which the derivative was evaluated.

stability()

```
stability(deriv, ystar = NULL, parameters = NULL, system = "two.dim", h = 1e-07,
          summary = TRUE,
          state.names = if (system == "two.dim") c("x", "y") else "y")
```

Uses stability analysis to classify equilibrium points. Uses the Taylor Series approach (also known as Perturbation Analysis) to classify equilibrium points of a one dimensional autonomous ODE system, or the Jacobian approach to classify equilibrium points of a two dimensional autonomous ODE system. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `ystar`: The point at which to perform stability analysis. For a one-dimensional system this should be a numeric vector of length one, for a two-dimensional system this should be a numeric vector of length two (i.e., presently only one equilibrium point's stability can be evaluated at a time). Alternatively this can be specified as `NULL`, and then `graphics::locator()` can be used to choose a point to perform the analysis for. However, given you are unlikely to locate exactly the equilibrium point, if possible enter `ystar` yourself. Defaults to `NULL`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `h`: Step length used to approximate the derivative(s). Defaults to `1e-7`.
- `summary`: Set to either `TRUE` or `FALSE` to determine whether a summary of the stability analysis is returned. Defaults to `TRUE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.

Returned by `stability()` is a list object containing all of the input variables as well as following components (the exact the exact make up is dependent upon the value of system):

- `Delta`: In the two-dimensional system case, the value of the Jacobian's determinant at `ystar`.
- `discriminant`: In the one-dimensional system case, the value of the discriminant used in perturbation analysis to assess stability. In the two-dimensional system case, the value of `tr^2 - 4*Delta` ($T^2 - 4\Delta$).
- `eigenvalues`: In the two-dimensional system case, the value of the Jacobian's eigenvalues at `ystar`.
- `eigenvectors`: In the two-dimensional system case, the value of the Jacobians eigenvectors at `ystar`.
- `Jacobian`: In the two-dimensional system case, the Jacobian at `ystar`.
- `tr`: In the two-dimensional system case, the value of the Jacobian's trace at `ystar`.

`trajectory()`

```
trajectory(deriv, y0 = NULL, n = NULL, tlim, tstep = 0.01, parameters = NULL,
           system = "two.dim", col = "black", add = TRUE,
           state.names = if (system == "two.dim") c("x", "y") else "y", ...)
```

This function allows the user to plot trajectories by performing numerical integration of the chosen ODE system, for a user specifiable range of initial conditions. The following inputs can be set:

- `deriv`: A function computing the derivative at a point for the ODE system to be analysed. Discussion of the required structure of these functions can be found at the end of this section, or in the help file for `deSolve::ode()`.
- `y0`: The initial condition(s). In the case of a one-dimensional system, this can either be a numeric vector of length one, indicating the location of the dependent variable initially, or a numeric vector indicating multiple initial locations of the independent variable. In the case of a two-dimensional system, this can either be a numeric vector of length two, reflecting the location of the two dependent variables initially, or it can be numeric matrix where each row reflects a different initial condition. Alternatively this can be left blank and the user can use `graphics::locator()` to specify initial condition(s) on a plot. In this case, for one-dimensional systems, all initial conditions are taken at `tlim[1]`, even if not selected so on the graph. Defaults to `NULL`.
- `n`: If `y0` is left `NULL`, such initial conditions can be specified using `graphics::locator()`, `n` sets the number of initial conditions to be chosen. Defaults to `NULL`.
- `tlim`: Sets the limits of the independent variable for which the solution should be plotted. Should be a numeric vector of length two. If `tlim[2] > tlim[1]`, then `tstep` should be negative to indicate a backwards trajectory.
- `tstep`: The step length of the independent variable, used in numerical integration. Decreasing the absolute magnitude of `tstep` theoretically makes the numerical integration more accurate, but increases computation time. Defaults to `0.01`.
- `parameters`: Parameters of the ODE system, to be passed to `deriv`. Supplied as a numeric vector; the order of the parameters can be found from the `deriv` file. Defaults to `NULL`.
- `system`: Set to either `"one.dim"` or `"two.dim"` to indicate the type of system being analysed. Defaults to `"two.dim"`.
- `col`: The colour(s) to plot the trajectories in. Should be a character vector. Will be reset accordingly if it is not of the length of the number of initial conditions. Defaults to `"black"`.
- `add`: Logical. If `TRUE`, the trajectories added to an existing plot. If `FALSE`, a new plot is created. Defaults to `TRUE`.
- `state.names`: The state names for `deSolve::ode()` functions that do not use positional states.
- `...`: Additional arguments to be passed to `graphics::plot()`.

Here, the numerical integration is performed by the function `deSolve::ode()`.

Returned by `trajectory()` is a list object containing all of the input variables as well as following components (the exact the exact make up is dependent on the value of `system`):

- `t`: A numeric vector containing the values of the independent variable at each integration step.
- `x`: In the two-dimensional system case, a numeric matrix whose columns are the numerically computed values of the first dependent variable for each initial condition.
- `y`: In the two-dimensional system case, a numeric matrix whose columns are the numerically computed values of the second dependent variable for each initial condition. In the one-dimensional system case, a numeric matrix whose columns are the numerically computed values of the dependent variable for each initial condition.

Derivative specification

In addition to the above functions, **phaseR** contains multiple example one- and two-dimensional autonomous ODE systems; these are the focus of later sections. Here, we discuss how the user can create their own system.

In order to be compatible with **phaseR**, system functions need to be compatible with `deSolve::ode()`, which is used internally to perform numerical integration. Thus, detailed guidance on how to write such functions is available in the **deSolve** documentation. A brief overview is provided here.

Functions that detail a system need to be coded to return a list, and must take three inputs: `t`, `y`, and `parameters`. Thus the basic skeleton for a one- or two-dimensional system (with the function named `derivative`) is as follows:

```
derivative <- function(t, y, parameters) {
  # Enter derivative computation here
  list(dy)
}
```

All that needs to be done is to set the value of `dy`, using `t`, `y`, and `parameters` (note here that `t` should not generally be used explicitly as we work only with autonomous systems). The approach should change slightly depending on whether you are setting up a one- or two-dimensional system, as `dy` should be a vector that has length equal to the number of dimensions.

Thus, for a system such as:

$$\frac{dx}{dt} = 3y, \qquad \frac{dy}{dt} = 2x,$$

we could use the following code:

```
derivative <- function(t, y, parameters) {
  x  <- y[1]
  y  <- y[2]
  dy <- c(3*y, 2*x)
  list(dy)
}
```

As a more complex example, consider instead changing the system above to:

$$\frac{dx}{dt} = \alpha y, \qquad \frac{dy}{dt} = \beta x,$$

with α and β parameters. The code would then proceed as follows:

```
derivative <- function(t, y, parameters) {
  alpha <- parameters[1]
  beta  <- parameters[2]
  x     <- y[1]
  y     <- y[2]
  dy    <- c(alpha*y, beta*x)
  list(dy)
}
```

Things are slightly simpler for one-dimensional systems. We could for example create a derivative function for the system:

$$\frac{dy}{dt} = a(b - 3 - y)^2,$$

using the following code:

```
derivative <- function(t, y, parameters) {
  a  <- parameters[1]
  b  <- parameters[2]
  dy <- a*(b - 3 - y)^2
  list(dy)
}
```

Examples

Within **phaseR**, numerous example systems are available. Here we will analyse some of them in order to indicate how to perform phase plane analyses by hand or with the help of **phaseR**. The language is again at times deliberatively repetitive; here to indicate how a general procedure can be used when performing analyses. It is not a useful exercise for me to provide inferior hand drawn plots, only ones produced by **phaseR** are shown. It is in addition useful to note that the small circles on the trajectory plots indicate initial conditions specified by the user.

`example2()`

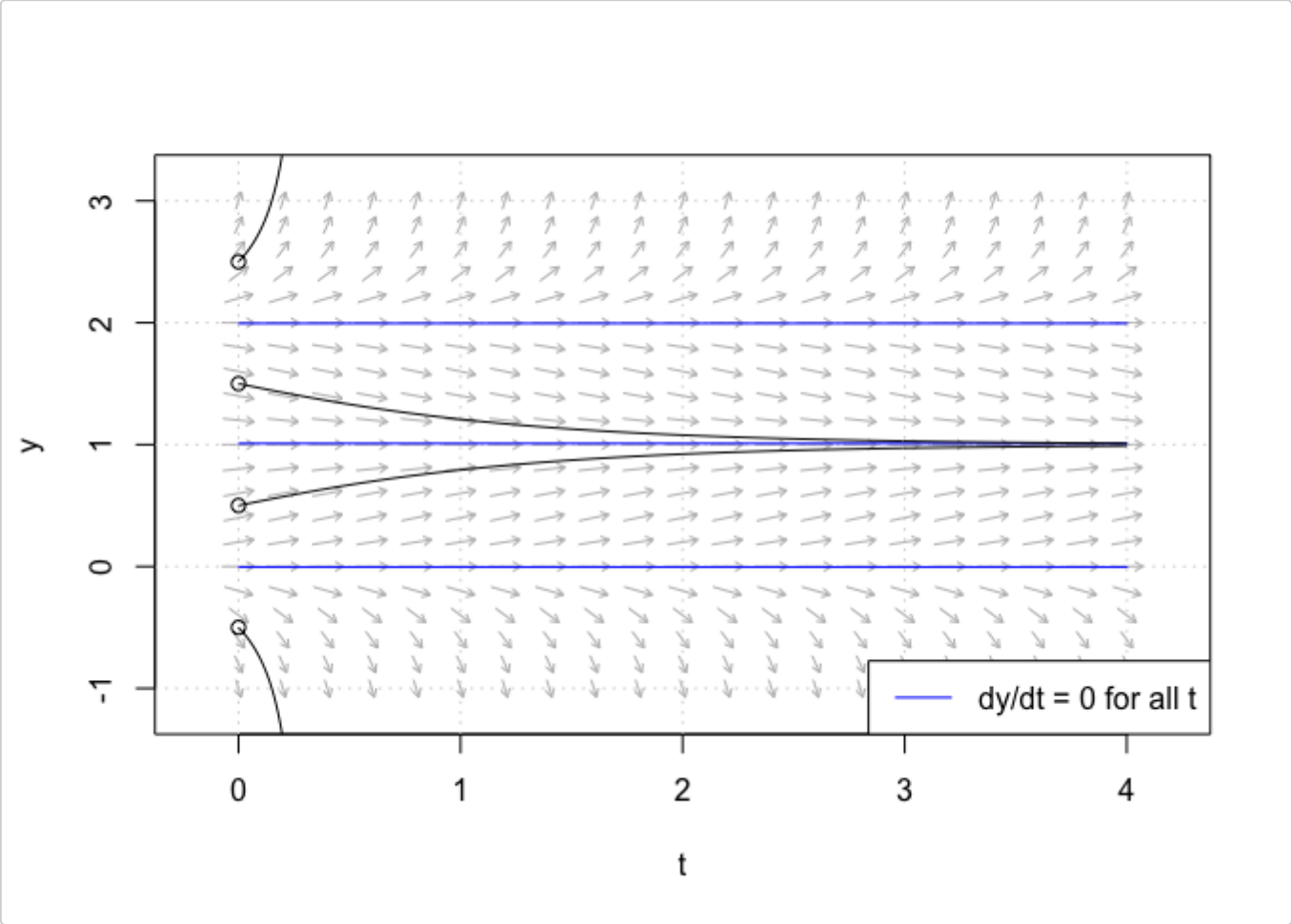
We begin with the one-dimensional autonomous ODE:

$$\frac{dy}{dt} = y(1 - y)(2 - y),$$

provided in the package as `example2()`. We begin by plotting the flow field and several trajectories using the following code, adding horizontal lines at any equilibrium points to indicate their presence as well:

```
example2_flowField  <- flowField(example2,
                                 xlim    = c(0, 4),
```

```
                                ylim    = c(-1, 3),
                                system  = "one.dim",
                                add     = FALSE)
grid()
example2_nullclines <- nullclines(example2,
                                 xlim   = c(0, 4),
                                 ylim   = c(-1, 3),
                                 system = "one.dim")
example2_trajectory <- trajectory(example2,
                                 y0     = c(-0.5, 0.5, 1.5, 2.5),
                                 tlim   = c(0, 4),
                                 system = "one.dim")
#> Note: col has been reset as required
```

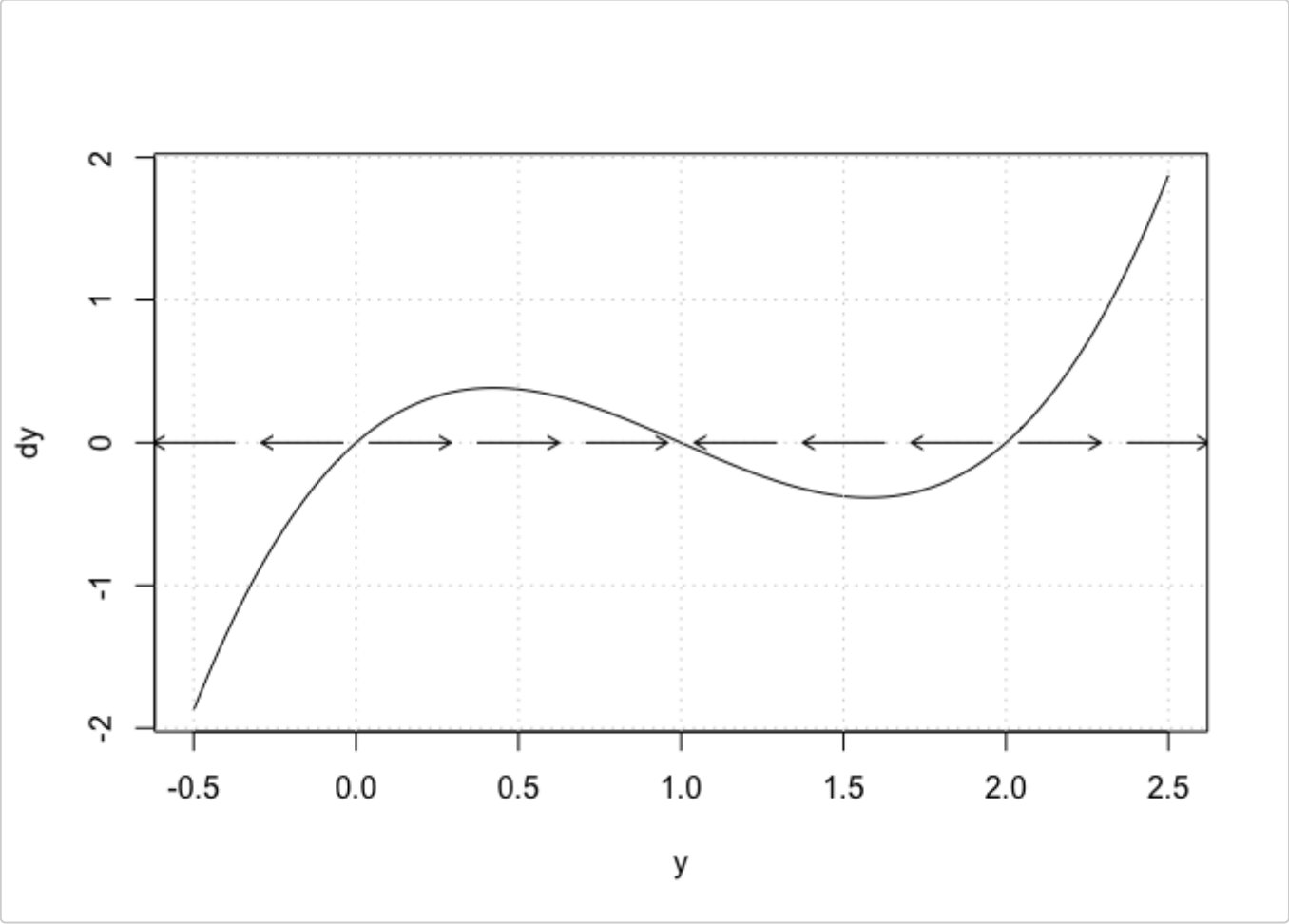


Thus, three equilibrium points have been identified; appearing to be $y_* = 0, 1, 2$ Indeed if we set the RHS of our ODE to zero we can identify these three points as the equilibrium points analytically:

$$y_*(1 - y_*)(2 - y_*) = 0 \Longrightarrow y_* = 0, 1, 2.$$

Plotting the phase portrait, we find that $y_* = 0$ and $y_* = 2$ are unstable, whilst $y_* = 1$ is stable (as is also apparent from the flow field and trajectories above):

```
example2_phasePortrait <- phasePortrait(example2,
                                        ylim = c(-0.5, 2.5))
```



Alternatively, using the Taylor Series approach we have:

$$\frac{d}{dy}\left(\frac{dy}{dt}\right)\Big|_{y=y_*} = 3y_*^2 - 6y_* + 2 = \begin{cases} 2 & : \ y_* = 0, \\ -1 & : \ y_* = 1, \\ 2 & : \ y_* = 2. \end{cases}$$

Thus we draw the same conclusion as from the phase portrait.

Finally, we can confirm our Taylor analysis using `stability()` and the following code:

```
example2_stability_1 <- stability(example2,
                                  ystar  = 0,
                                  system = "one.dim")
#> discriminant = 2, classification = Unstable
example2_stability_2 <- stability(example2,
                                  ystar  = 1,
                                  system = "one.dim")
#> discriminant = -1, classification = Stable
example2_stability_3 <- stability(example2,
                                  ystar  = 2,
                                  system = "one.dim")
#> discriminant = 2, classification = Unstable
```

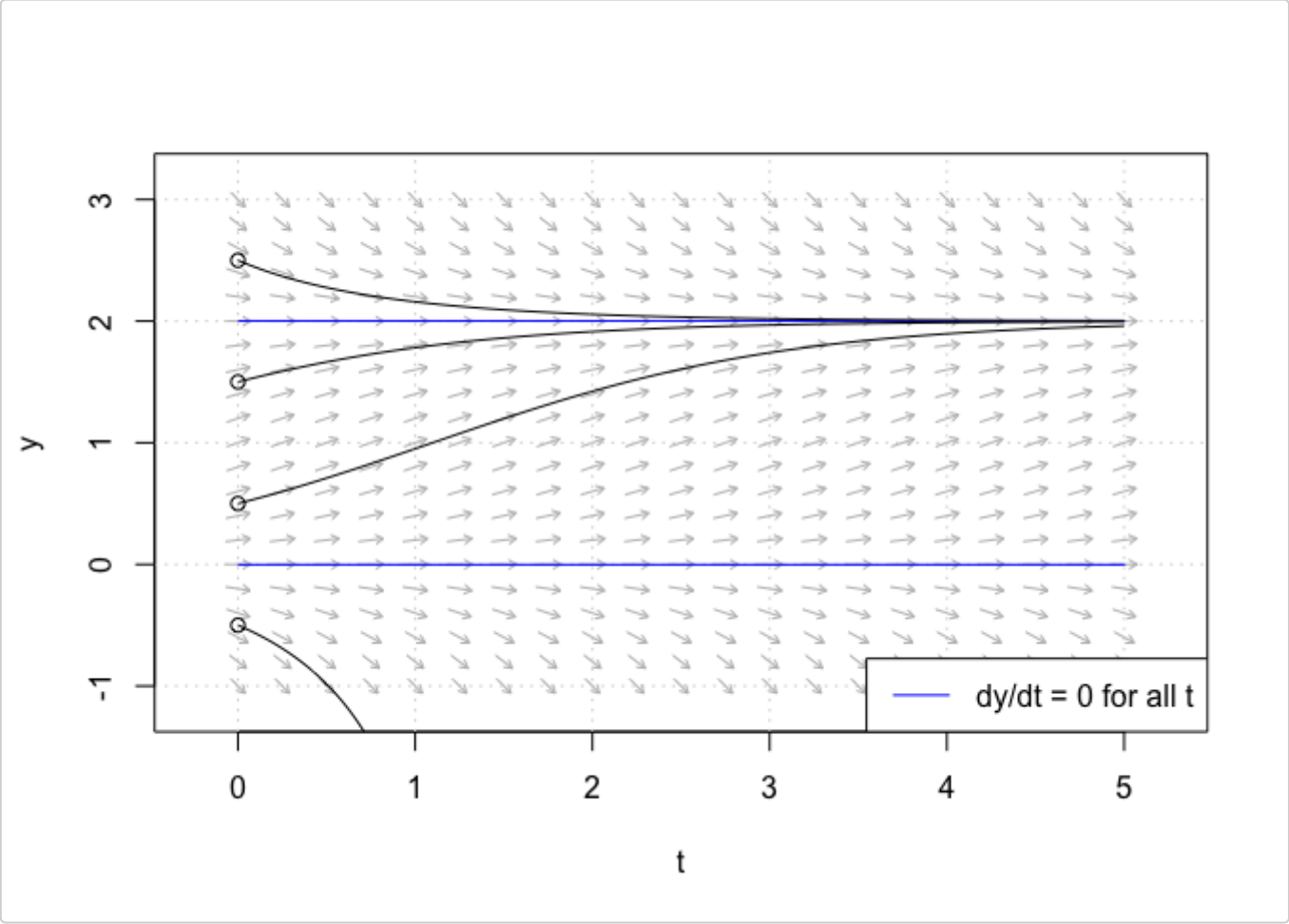
`logistic()`

The logistic growth model is frequently used in biology to model the growth of a population under density dependence. It is given by:

$$\frac{dy}{dt} = \beta y \left(1 - \frac{y}{K}\right).$$

With the following code, we can plot the flow field and several trajectories (for the case $\beta = 1$ and $K = 2$), as well as adding horizontal lines at any equilibrium points to indicate their presence:

```
logistic_flowField  <- flowField(logistic,
                                 xlim       = c(0, 5),
                                 ylim       = c(-1, 3),
                                 parameters = c(1, 2),
                                 system     = "one.dim",
                                 add        = FALSE)
grid()
logistic_nullclines <- nullclines(logistic,
                                  xlim       = c(0, 5),
                                  ylim       = c(-1, 3),
                                  parameters = c(1, 2),
                                  system     = "one.dim")
logistic_trajectory <- trajectory(logistic,
                                  y0         = c(-0.5, 0.5, 1.5, 2.5),
                                  tlim       = c(0, 5),
                                  parameters = c(1, 2),
                                  system     = "one.dim")
#> Note: col has been reset as required
```

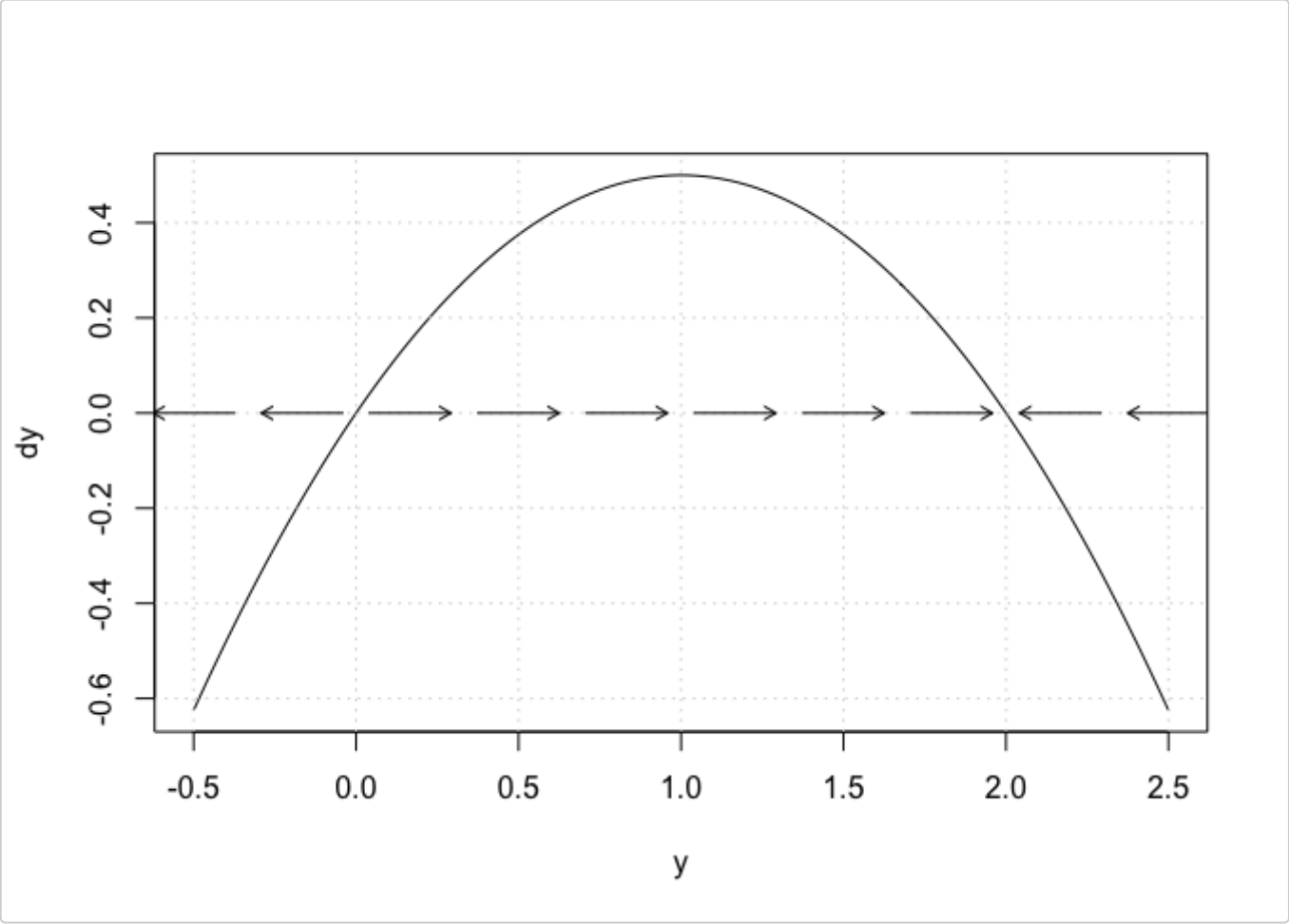


Again, two equilibrium points have been identified. We can confirm their location in the general case analytically by setting the RHS of the ODE to zero:

$$\beta y_* \left(1 - \frac{y_*}{K}\right) = 0 \Longrightarrow y_* = 0, K.$$

Plotting the phase portrait we can observe that $y_* = 0$ is unstable and $y_* = K$ stable, for the case $\beta = 1$ and $K = 2$:

```
logistic_phasePortrait <- phasePortrait(logistic,
                                        ylim       = c(-0.5, 2.5),
                                        parameters = c(1, 2))
```



Finally, if we use our Taylor Series method we can draw this same conclusion:

$$\frac{d}{dy}\left(\frac{dy}{dt}\right)\Big|_{y=y_*} = \beta - \frac{2\beta y_*}{K} = \begin{cases} \beta & : \ y_* = 0, \\ -\beta & : \ y_* = K. \end{cases}$$

So for $\beta = 1$ and $K = 2$, we have a stable point at $y = 2$. Moreover, from this we can see that the point $y = K$ will in general be stable provided $\beta > 0$.

The following code verifies our findings for the specific case studied above:

```
logistic_stability_1 <- stability(logistic,
                                  ystar      = 0,
                                  parameters = c(1, 2),
                                  system     = "one.dim")
#> discriminant = 1, classification = Unstable
logistic_stability_2 <- stability(logistic,
                                  ystar      = 2,
                                  parameters = c(1, 2),
                                  system     = "one.dim")
#> discriminant = -1, classification = Stable
```

example4()

We now turn our attention to linear two-dimensional autonomous ODE systems. Here we consider the coupled system given by:

$$\frac{dx}{dt} = -x, \qquad \frac{dy}{dt} = 4x.$$

This is provided in the package as `example4()`. We can find the x- and y-nullclines by setting the derivatives to zero as follows:

$$x : -x = 0 \Longrightarrow x = 0,$$
$$y : 4x = 0 \Longrightarrow x = 0.$$

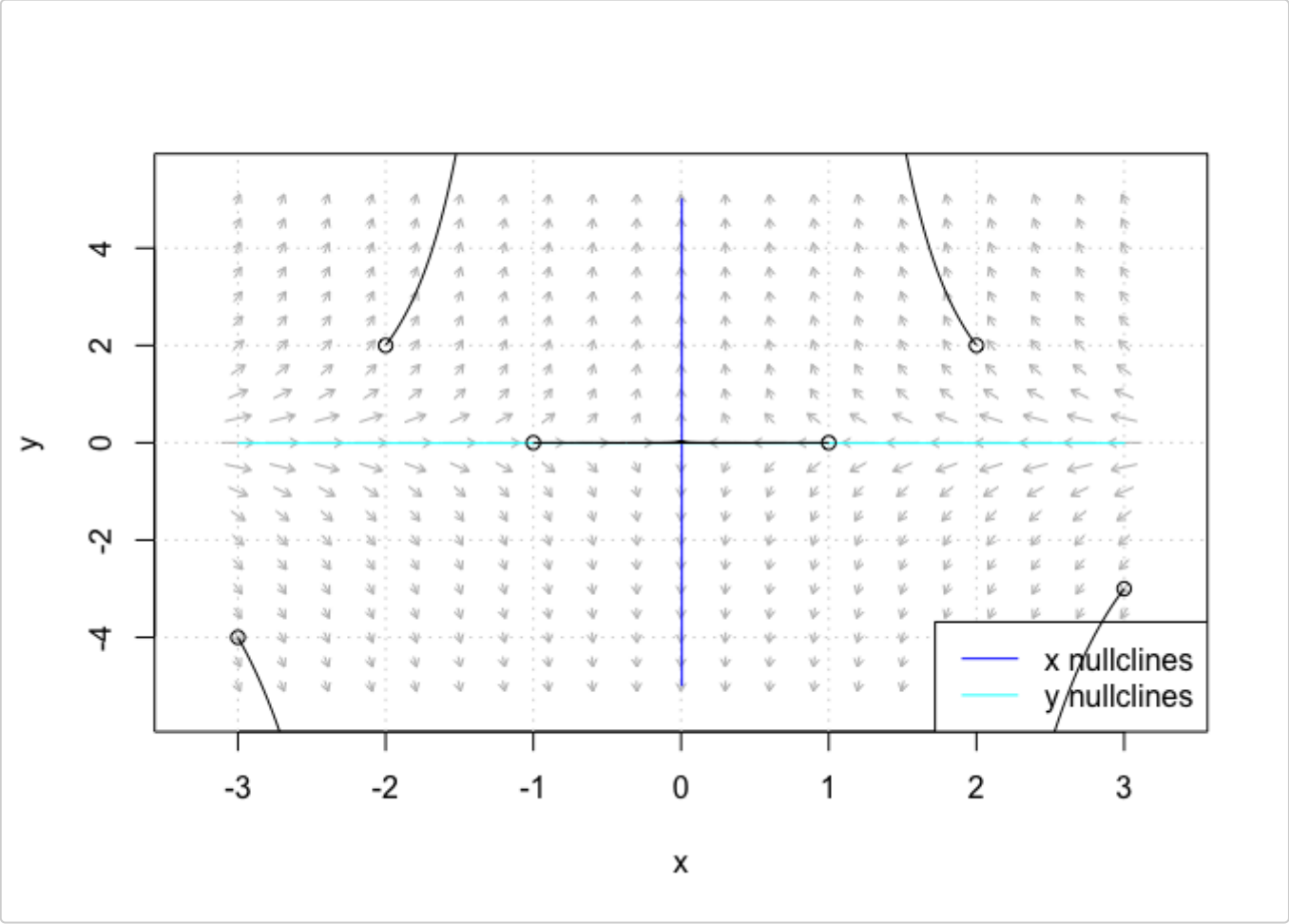
Thus the nullclines are the same. This means we have a line of equilibrium points given by $x = 0$. To see why this is the case, and there is no unique solution, we examine the Jacobian of our system:

$$J = \begin{pmatrix} -1 & 0 \\ 4 & 0 \end{pmatrix}.$$

Thus the determinant of J is zero, and we have a singular case of the general linear two-dimensional system; the Taylor approach cannot be used to classify $(0, 0)$.

Thus, here, in order to determine whether the points along the line $x = 0$ are stable or not, we plot the nullclines, the velocity field, and add several trajectories:

```
example4_flowField  <- flowField(example4,
                                 xlim = c(-3, 3),
                                 ylim = c(-5, 5),
                                 add  = FALSE)
grid()
example4_nullclines <- nullclines(example4,
                                  xlim = c(-3, 3),
                                  ylim = c(-5, 5))
y0                  <- matrix(c(1, 0, -1, 0, 2, 2, -2, 2, 3, -3, -3, -4),
                              6, 2, byrow = TRUE)
example4_trajectory <- trajectory(example4,
                                  y0   = y0,
                                  tlim = c(0, 10))
#> Note: col has been reset as required
```



Thus we observe the trajectories moving towards the line $x = 0$; indicative of stability. This example illustrates that plotting trajectories can be useful when the Taylor approach fails.

example5()

We will now examine a further example of a linear two-dimensional system, given by:

$$\frac{dx}{dt} = 2x + y, \qquad \frac{dy}{dt} = 2x - y.$$

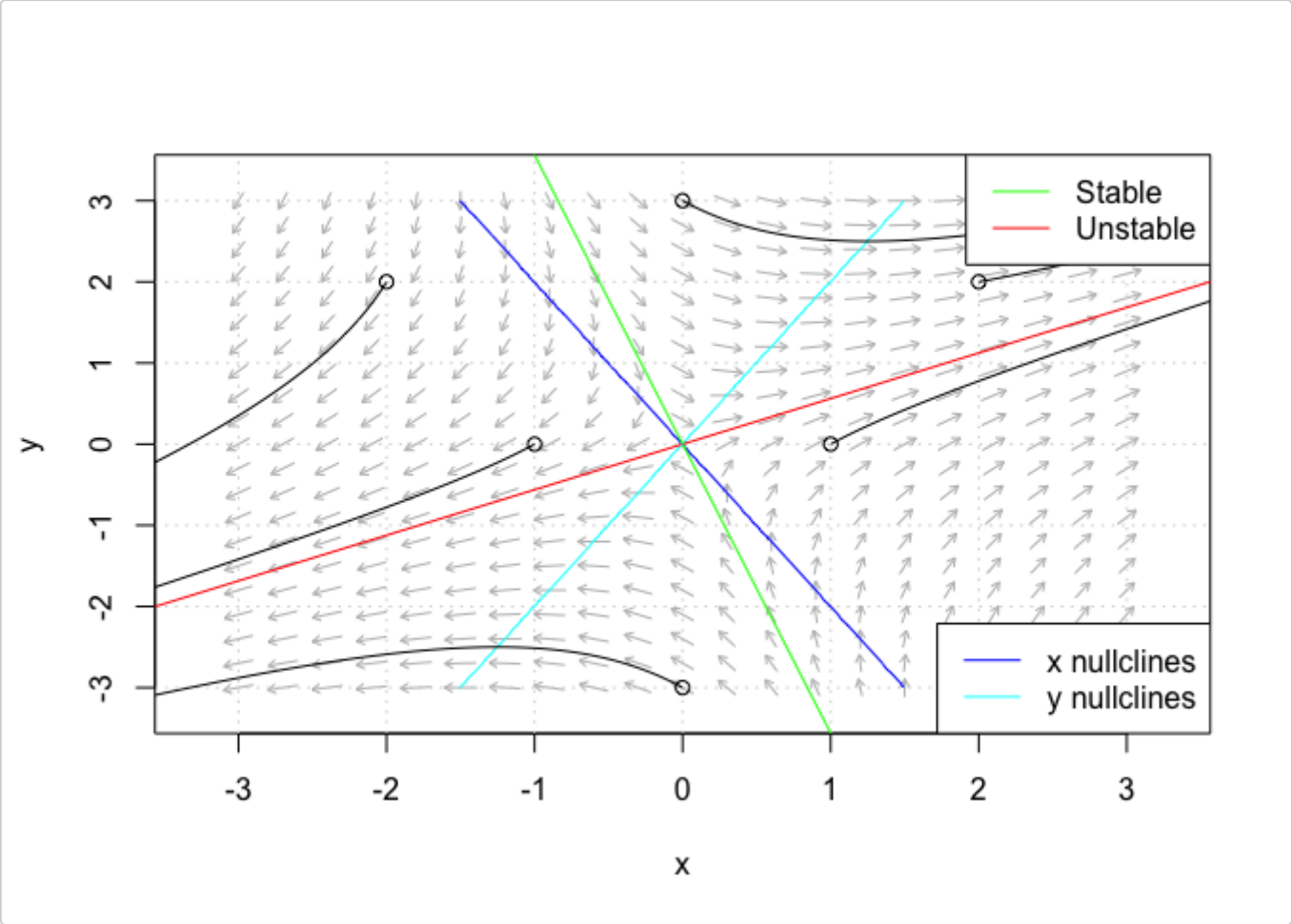
It is provided in the package as `example5()`. Again we begin by setting the derivatives to zero to identify the nullclines:

$$x : 2x + y = 0 \Longrightarrow y = -2x,$$
$$y : 2x - y = 0 \Longrightarrow y = 2x.$$

From these two equations it is easy to see that the only equilibrium point is at $(0, 0)$. We begin by plotting the nullclines, velocity field, and several trajectories (along with the stable and unstable manifolds of the saddle, we discuss below):

```
example5_flowField  <- flowField(example5,
                                 xlim = c(-3, 3),
                                 ylim = c(-3, 3),
                                 add  = FALSE)
grid()
```

```
example5_nullclines <- nullclines(example5,
                                  xlim = c(-3, 3),
                                  ylim = c(-3, 3))
y0                  <- matrix(c(1, 0, -1, 0, 2, 2, -2, 2, 0, 3, 0, -3),
                              6, 2, byrow = TRUE)
example5_trajectory <- trajectory(example5,
                                  y0   = y0,
                                  tlim = c(0, 10))
#> Note: col has been reset as required
example5_manifolds  <- drawManifolds(example5,
                                     y0 = c(0, 0))
```



From the trajectories it appears that $(0, 0)$ is a saddle point. To verify this we compute the Jacobian of the system:

$$J = \begin{pmatrix} 2 & 1 \\ 2 & -1 \end{pmatrix}.$$

Thus we have $T = 1$, $\Delta = -4$ and $T^2 - 4\Delta = 17$. From our classification rules this confirms that $(0, 0)$ is indeed a saddle. Finally, we verify this analysis using stability and the following code:

```
example5_stability <- stability(example5,
                                ystar = c(0, 0))
#> tr = 1, Delta = -4, discriminant = 17, classification = Saddle
```

example8()

As a final example of a linear two-dimensional system, we will examine:

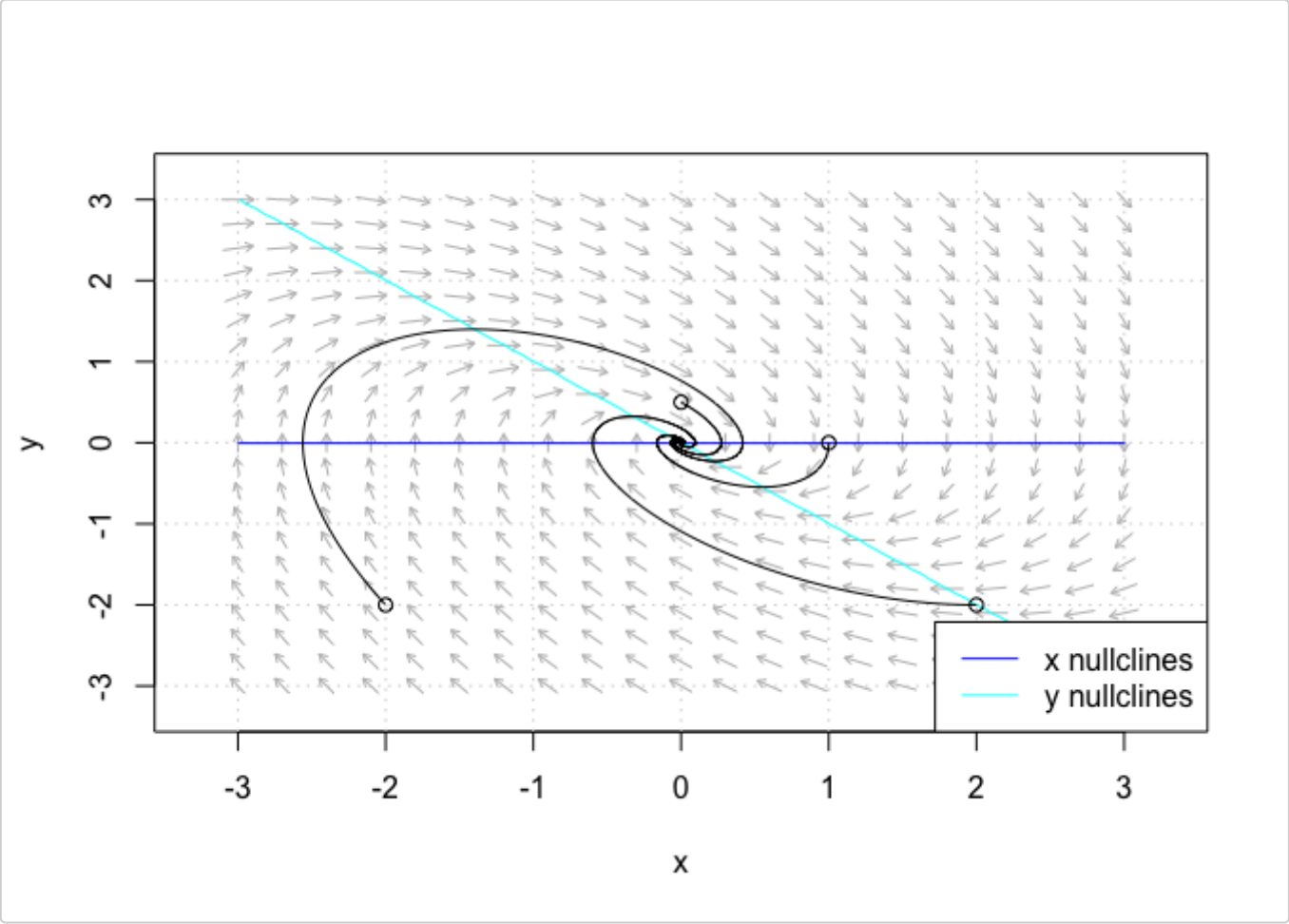
$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = -x - y,$$

available in the package as `example8()`. Setting the derivatives to zero, we first locate the nullclines:

$$x : y = 0,$$
$$y : -x - y = 0 \Longrightarrow y = -x.$$

From this, again we can identify the one equilibrium is at $(0, 0)$. We now plot the nullclines and velocity field, along with several trajectories:

```
example8_flowField  <- flowField(example8,
                                 xlim = c(-3, 3),
                                 ylim = c(-3, 3),
                                 add  = FALSE)
grid()
example8_nullclines <- nullclines(example8,
                                  xlim = c(-3, 3),
                                  ylim = c(-3, 3))
y0                  <- matrix(c(1, 0, 0, 0.5, 2, -2, -2, -2),
                              4, 2, byrow = TRUE)
example8_trajectory <- trajectory(example8,
                                  y0   = y0,
                                  tlim = c(0, 10))
#> Note: col has been reset as required
```



It appears from the plot that $(0, 0)$ is a stable focus, but we can verify that this is the case using the Jacobian:

$$J = \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}.$$

Thus we have $T = -1$, $\Delta = 1$ and $T^2 - 4\Delta = -3$; indeed $(0, 0)$ is a stable focus. Finally, we confirm our stability analysis using **phaseR**:

```
example8_stability <- stability(example8,
                                ystar = c(0, 0))
#> tr = -1, Delta = 1, discriminant = -3, classification = Stable focus
```

example11()

We now advance to a non-linear example of a two-dimensional system, given by:

$$\frac{dx}{dt} = x(3 - x - 2y), \qquad \frac{dy}{dt} = y(2 - x - y),$$

and provided in the package as `example11()`. As always, we begin by setting the derivatives to zero to locate the nullclines. First, for x:

$$x(3 - x - 2y) = 0 \Longrightarrow x = 0 \text{ or } y = \frac{1}{2}(3 - x).$$

Then for y:

$$y(2 - x - y) = 0 \implies x = 0 \text{ or } y = 2 - x.$$

Here, as is often the case for non-linear systems, there are here multiple equilibria, which we find via the intersections of the above nullclines. Easily, we can identify several as $(0, 0)$, $(0, 2)$, and $(3, 0)$. The final equilibrium point comes from the intersection of the two more complex nullclines:

$$\frac{1}{2}(3 - x_*) = 2 - x_* \implies x_* = 1 \implies y_* = 1,$$

i.e., the point $(1, 1)$. We can determine the stability of these four equilibria from the general case Jacobian of the system:

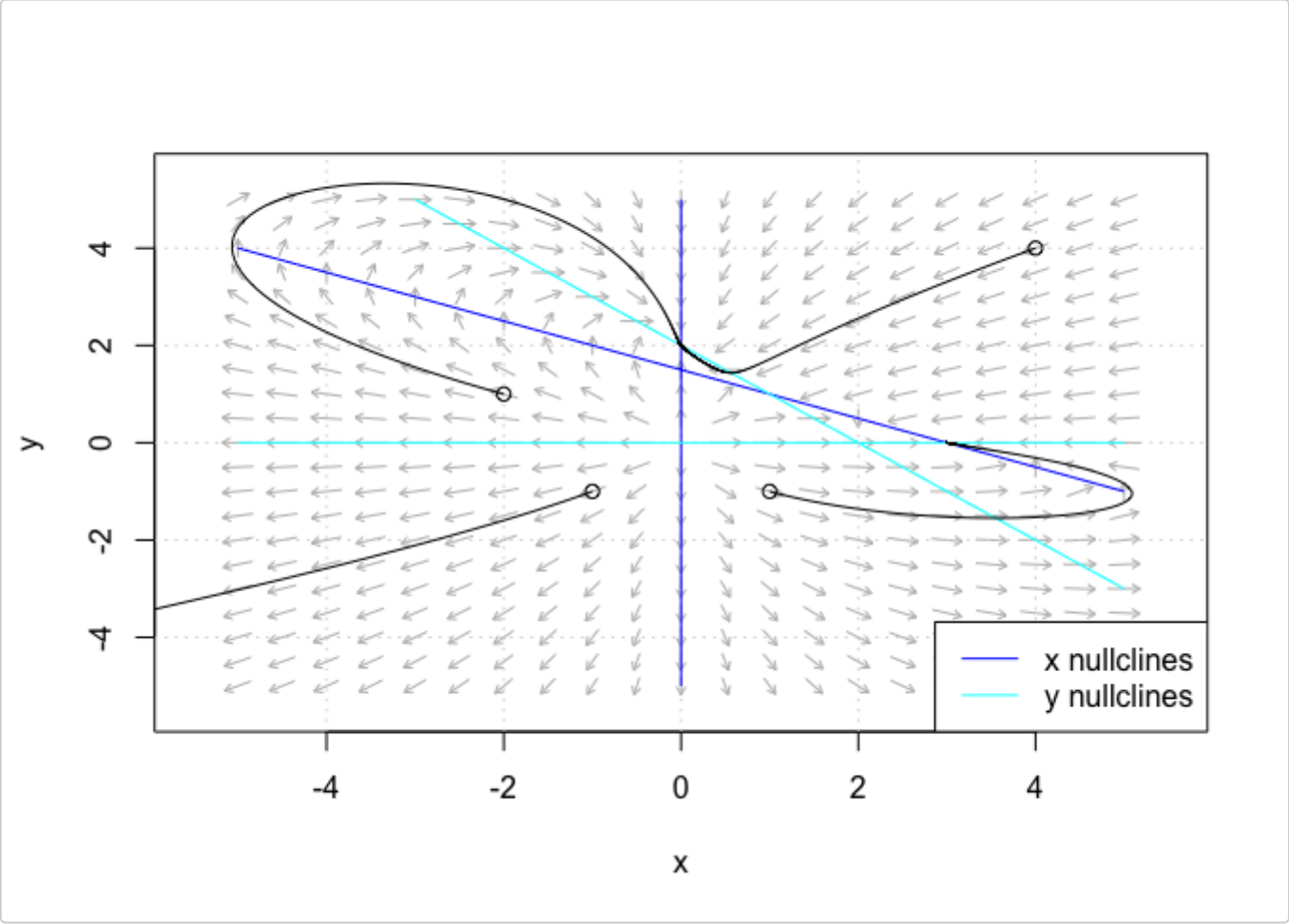
$$J = \begin{pmatrix} 3 - 2x_* - 2y_* & -2x_* \\ -y_* & 2 - x_* - 2y_* \end{pmatrix}.$$

In the case of multiple equilibria, it is then often a good idea to present the classification in a table:

Equilibrium Point	Δ	$T^2 - 4\Delta$	T	Classification
$(0, 0)$	6	1	5	Unstable node
$(0, 2)$	8	4	-6	Stable node
$(1, 1)$	-1	8	-2	Saddle
$(3, 0)$	3	4	-4	Stable node

To summarise all of the above analysis we produce a plot of the nullclines, velocity field, and several trajectories using **phaseR**:

```
example11_flowField  <- flowField(example11,
                                  xlim = c(-5, 5),
                                  ylim = c(-5, 5),
                                  add  = FALSE)
grid()
example11_nullclines <- nullclines(example11,
                                   xlim   = c(-5, 5),
                                   ylim   = c(-5, 5),
                                   points = 500)
y0                   <- matrix(c(4, 4, -1, -1, -2, 1, 1, -1),
                               4, 2, byrow = TRUE)
example11_trajectory <- trajectory(example11,
                                   y0   = y0,
                                   tlim = c(0, 10))
#> Note: col has been reset as required
```



In addition, we verify the stability results using stability:

```
example11_stability_1 <- stability(example11,
                                   ystar = c(0, 0))
#> tr = 5, Delta = 6, discriminant = 1, classification = Unstable node
example11_stability_2 <- stability(example11,
                                   ystar = c(0, 2))
#> tr = -3, Delta = 2, discriminant = 1, classification = Stable node
example11_stability_3 <- stability(example11,
                                   ystar = c(1, 1),
                                   h     = 1e-8)
#> tr = -2, Delta = -1, discriminant = 8, classification = Saddle
example11_stability_4 <- stability(example11,
                                   ystar = c(3, 0))
#> tr = -4, Delta = 3, discriminant = 4, classification = Stable node
```

example12()

Moving on, we now consider a further example of a non-linear two-dimensional system:

$$\frac{dx}{dt} = x - y, \qquad \frac{dy}{dt} = x^2 + y^2 - 2.$$

Provided in **phaseR** as `example12()`. As usual, we first locate the nullclines:

$$x : x - y = 0 \implies y = x,$$
$$y : x^2 + y^2 - 2 = 0 \implies x^2 + y^2 = 2.$$

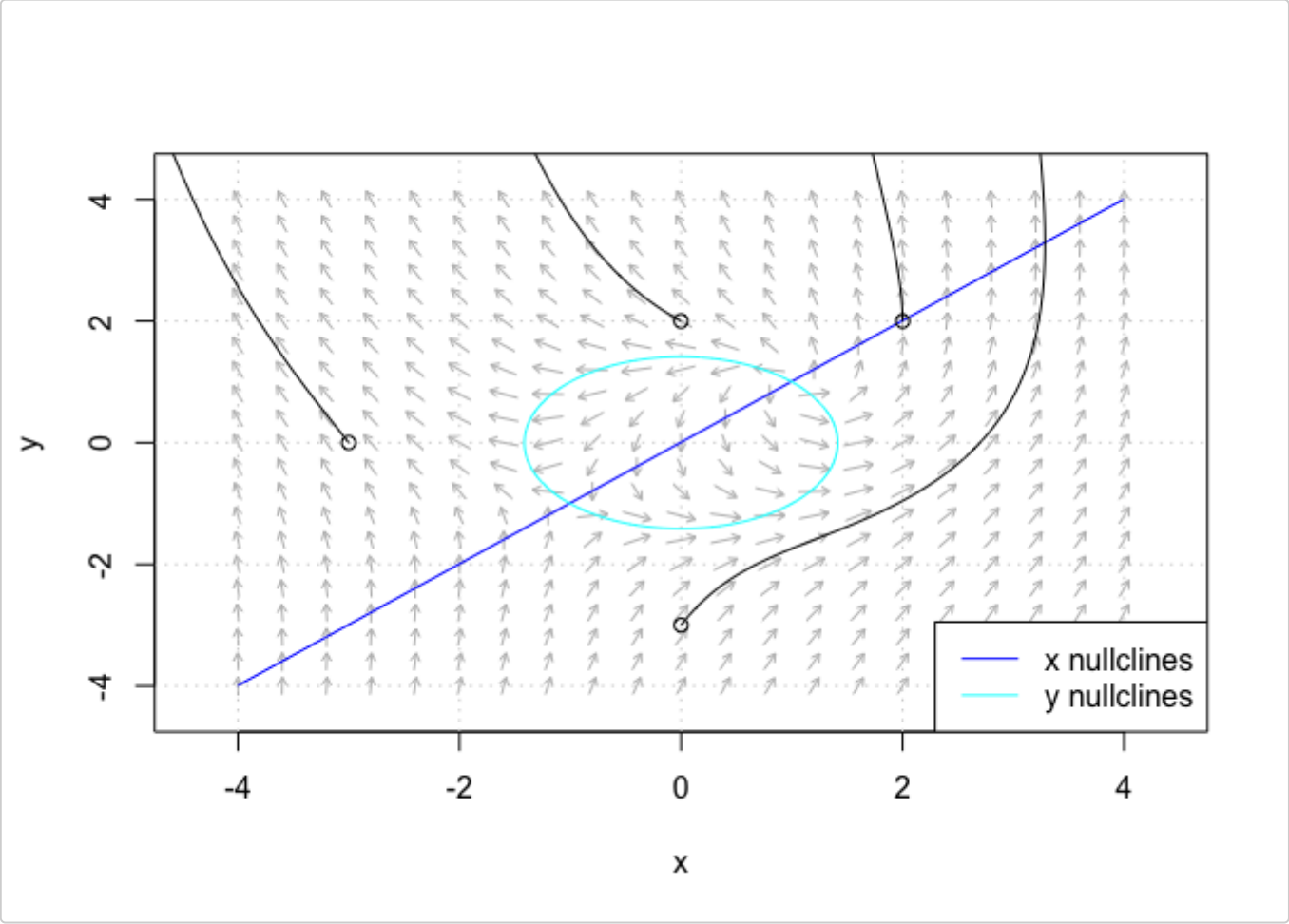
From this, we substitute one condition into the other to locate the equilibria:

$$x_*^2 + x_*^2 = 2 \implies x_*^2 = 1 \implies x_* = \pm 1 = y_*,$$

therefore we have equilibria at $(1, 1)$ and $(-1, -1)$.

We continue by plotting the nullclines, velocity field, and several trajectories:

```
example12_flowField  <- flowField(example12,
                                  xlim = c(-4, 4),
                                  ylim = c(-4, 4),
                                  add  = FALSE)
grid()
example12_nullclines <- nullclines(example12,
                                   xlim   = c(-4, 4),
                                   ylim   = c(-4, 4),
                                   points = 500)
y0                   <- matrix(c(2, 2, -3, 0, 0, 2, 0, -3),
                               4, 2, byrow = TRUE)
example12_trajectory <- trajectory(example12,
                                   y0   = y0,
                                   tlim = c(0, 10))
#> Note: col has been reset as required
```



It appears that both of the equilibria are unstable, but to classify them accurately we will use the Jacobian:

$$J = \begin{pmatrix} 1 & -1 \\ 2x_* & 2y_* \end{pmatrix}.$$

Therefore, we have:

Equilibrium Point	Δ	$T^2 - 4\Delta$	T	Classification
$(1, 1)$	4	-7	3	Unstable Focus
$(-1, -1)$	-3	13	-1	Saddle

Indeed it was the case that both points were unstable. Finally, we verify this analysis using stability:

```
example12_stability_1 <- stability(example12,
                                   ystar = c(1, 1))
#> tr = 3, Delta = 4, discriminant = -7, classification = Unstable focus
example12_stability_2 <- stability(example12,
                                   ystar = c(-1, -1),
                                   h     = 1e-8)
#> tr = -1, Delta = -4, discriminant = 17, classification = Saddle
```

simplePendulum()

This next example comes from a real life modelling scenario; the equation for a simple pendulum (i.e., no damping force) acting under gravity can be written in the form:

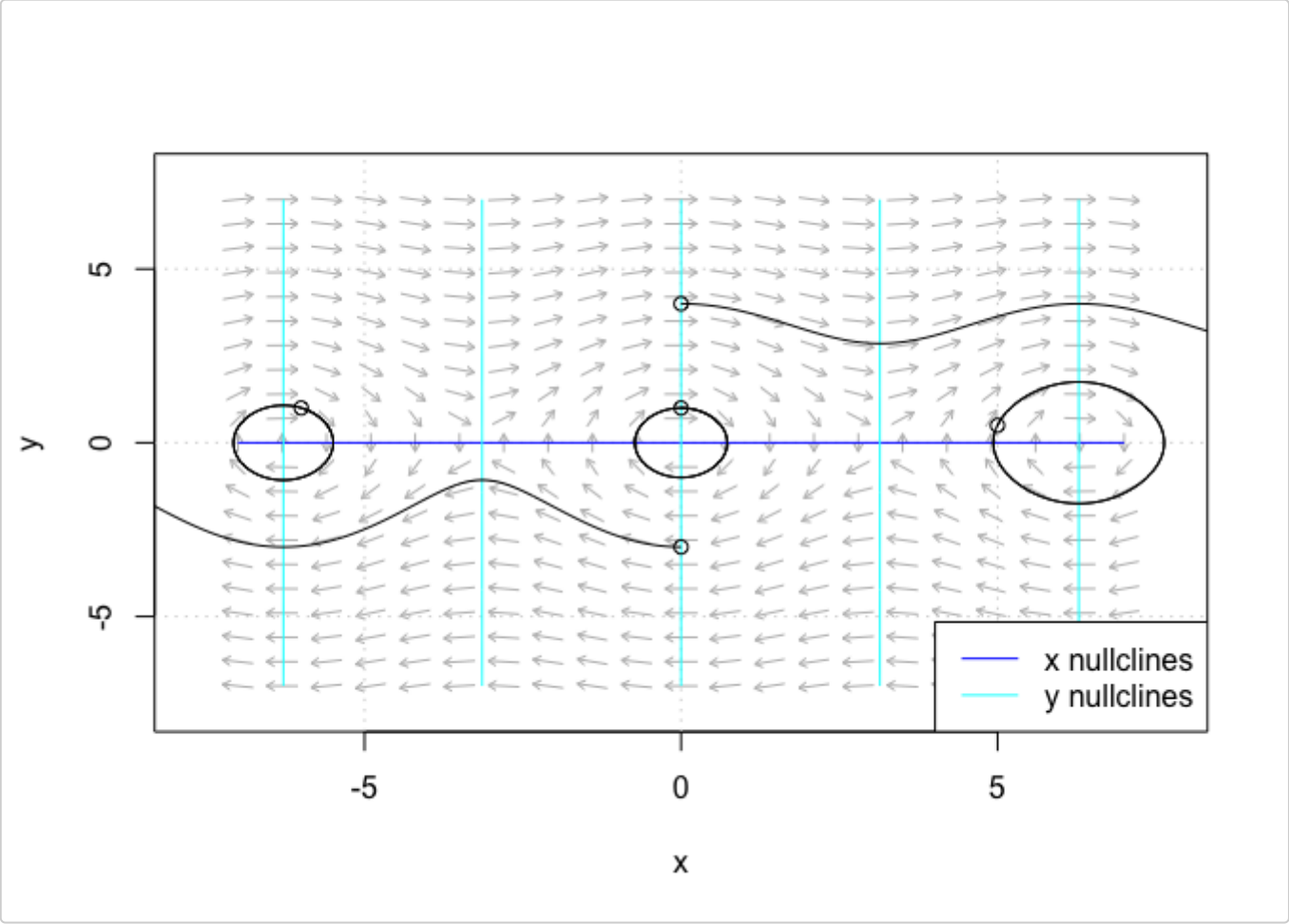
$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = -\frac{g}{l}\sin x.$$

It is provided in the model as `simplePendulum()`. We first set the gradients to zero to locate the nullclines:

$$\begin{aligned} x &: y = 0, \\ y &: -\frac{g}{l}\sin x = 0 \Longrightarrow x = n\pi, \ \forall n \in \mathbb{N}. \end{aligned}$$

From this we can identify that equilibria will be present at all points $(0, n\pi)$, where n is an integer. Using this we produce our familiar plot, choosing the case $l = 5$:

```
simplePendulum_flowField  <- flowField(simplePendulum,
                                       xlim       = c(-7, 7),
                                       ylim       = c(-7, 7),
                                       parameters = 5,
                                       add        = FALSE)
grid()
simplePendulum_nullclines <- nullclines(simplePendulum,
                                        xlim       = c(-7, 7),
                                        ylim       = c(-7, 7),
                                        parameters = 5,
                                        points     = 500)
y0                        <- matrix(c(0, 1, 0, 4, -6, 1, 5, 0.5, 0, -3),
                                    5, 2, byrow = TRUE)
simplePendulum_trajectory <- trajectory(simplePendulum,
                                        y0         = y0,
                                        tlim       = c(0, 10),
                                        parameters = 5)
#> Note: col has been reset as required
```



We then turn to the Jacobian in order to determine the stability of the equilibria:

$$J = \begin{pmatrix} 0 & 1 \\ -\frac{g}{l}\cos x_* & 0 \end{pmatrix},$$

$$\implies T = 0, \Delta = \frac{g}{l}\cos x_*, T^2 - 4\Delta = -\frac{4g}{l}\cos x_*.$$

Therefore, for $x_* = 2n\pi$, $\Delta > 0$ and the equilibria is a centre. However, for $x_* = (2n+1)\pi$, $\Delta < 0$ and the equilibria is a saddle. We can confirm this for the points $(0, 0)$ and $(\pi, 0)$ using **phaseR**:

```
simplePendulum_stability_1 <- stability(simplePendulum,
                                        ystar      = c(0, 0),
                                        parameters = 5)
#> tr = 0, Delta = 1.962, discriminant = -7.848, classification = Centre
simplePendulum_stability_2 <- stability(simplePendulum,
                                        ystar      = c(pi, 0),
                                        parameters = 5)
#> tr = 0, Delta = -1.962, discriminant = 7.848, classification = Saddle
```

vanDerPol()

As a final example, we again turn to a real physical system. The van Der Pol oscillator is a classic example in physics, describing a non-conservative oscillator with non-linear damping. It can be written in the form:

$$\frac{dx}{dt} = y, \qquad \frac{dy}{dt} = \mu(1 - x^2)y - x.$$

It is provided in the package as `vanDerPol()`. We consider only the case of $\mu > 0$, i.e., when the oscillator is damped. The nullclines can then be computed as:

$$\begin{aligned} &x : y = 0, \\ &y : \mu(1 - x^2)y - x = 0 \Longrightarrow y = \frac{x}{\mu(1 - x^2)}. \end{aligned}$$

The form of these nullclines indicates that the only equilibrium point is $(0, 0)$. The stability of this, we again find from the Jacobian:

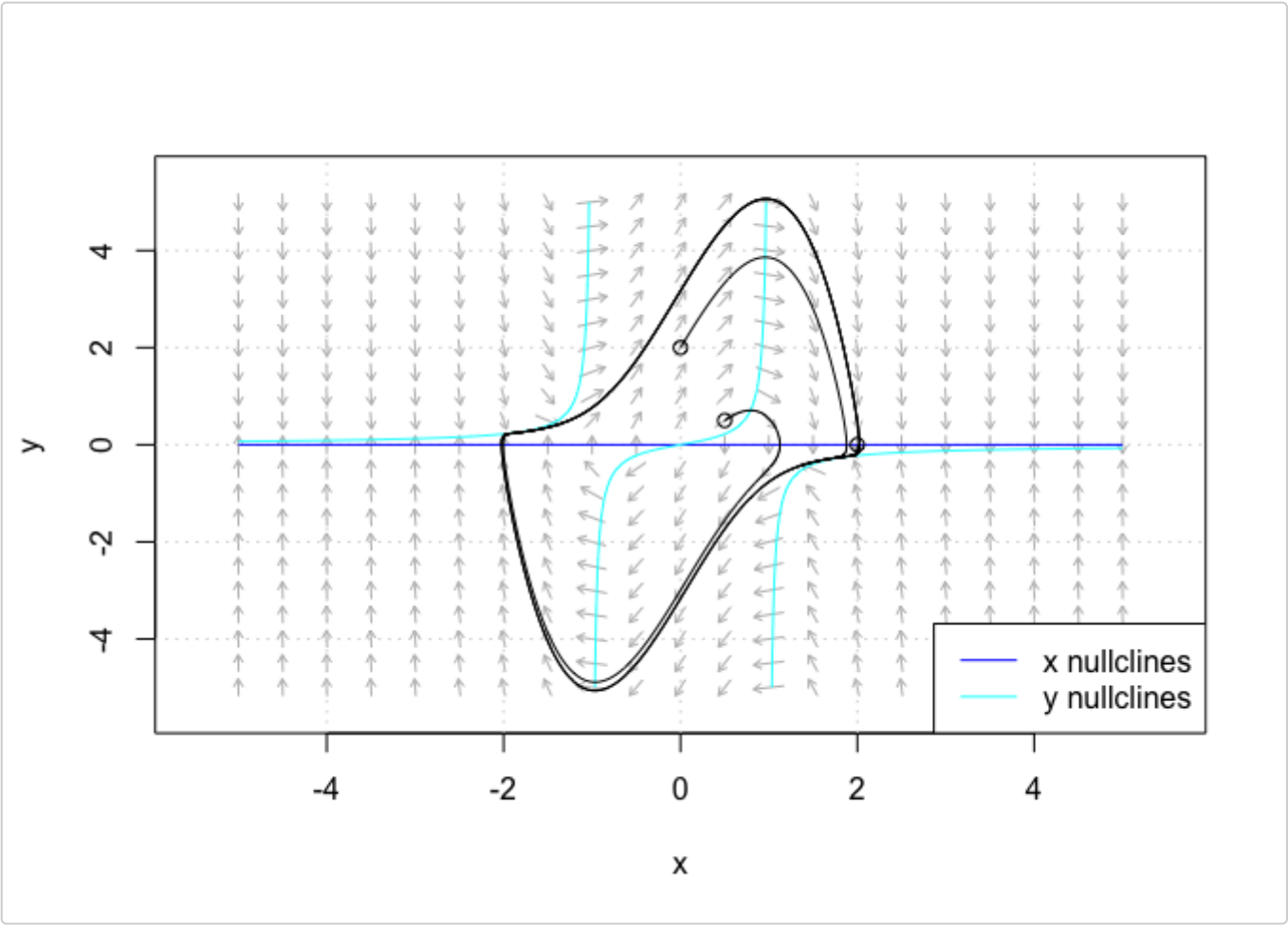
$$\begin{aligned} J &= \begin{pmatrix} 0 & 1 \\ -2\mu x_* y_* - 1 & \mu(1 - x_*^2) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & \mu \end{pmatrix}\Bigg|_{(0,0)}, \\ \Longrightarrow T &= \mu,\ \Delta = 1,\ T^2 - 4\Delta = \mu^2 - 4. \end{aligned}$$

Thus if $\mu > 2$ then we will have an unstable node, whereas for $\mu < 2$ we will have an unstable focus. We take the cases $\mu = 1$ and $\mu = 3$ as examples to indicate this using **phaseR**:

```
vanDerPol_stability_1 <- stability(vanDerPol, ystar = c(0, 0),
                                   parameters = 3)
#> tr = 3, Delta = 1, discriminant = 5, classification = Unstable node
vanDerPol_stability_2 <- stability(vanDerPol, ystar = c(0, 0),
                                   parameters = 1)
#> tr = 1, Delta = 1, discriminant = -3, classification = Unstable focus
```

However, when we plot trajectories along with the nullclines and velocity field we find:

```
vanDerPol_flowField  <- flowField(vanDerPol,
                                  xlim       = c(-5, 5),
                                  ylim       = c(-5, 5),
                                  parameters = 3,
                                  add        = FALSE)
grid()
vanDerPol_nullclines <- nullclines(vanDerPol,
                                   xlim       = c(-5, 5),
                                   ylim       = c(-5, 5),
                                   parameters = 3,
                                   points     = 500)
y0                   <- matrix(c(2, 0, 0, 2, 0.5, 0.5),
                               3, 2, byrow = TRUE)
vanDerPol_trajectory <- trajectory(vanDerPol,
                                   y0         = y0,
                                   tlim       = c(0, 10),
                                   parameters = 3)
#> Note: col has been reset as required
```



It appears that the solutions are bounded, and indeed oscillate as the name suggests. This oscillating behaviour is an example of a limit cycle. This characterisation; where trajectories are pushed away near the equilibria (hence the classification as unstable), but move towards it far away, is typical of limit cycles.

Additional available systems

As well as those studied earlier, numerous other derivative functions for one- and two-dimensional systems are available in **phaseR**. This section provides a list of them. In some instances further explanation of the models is provided in their respective exercises at the end of this vignette. Again, parameters are specified in the order they appear in the model; for ultimate clarify see the help page for the function within R.

`exponential()`

The exponential growth model, often used in biology and chemistry to model growth and decay of biological or chemical species, is given by:

$$\frac{dy}{dt} = \beta y.$$

`monomolecular()`

The monomolecular growth model, often used to model the heating and cooling of objects, or to model physiological processes, is given by:

$$\frac{dy}{dt} = \beta(K - y).$$

vonBertalanffy()

The von Bertalanffy model, often used in Biology to model the growth of organisms, is given by:

$$\frac{dy}{dt} = \alpha y^{2/3} - \beta y.$$

example3()

Function `example3()` is a linear two-dimensional system given by:

$$\frac{dx}{dt} = -x, \ \frac{dy}{dt} = -4x.$$

example6()

Function `example6()` is a linear two-dimensional system given by:

$$\frac{dx}{dt} = x + 2y, \ \frac{dy}{dt} = -2x + y.$$

example7()

Function `example7()` is a linear two-dimensional system given by:

$$\frac{dx}{dt} = -x - y, \ \frac{dy}{dt} = 4x + y.$$

example9()

Function `example9()` is a linear two-dimensional system given by:

$$\frac{dx}{dt} = -2x + 3y, \ \frac{dy}{dt} = 7x + 6y.$$

example10()

Function `example10()` is a non-linear two-dimensional system given by:

$$\frac{dx}{dt} = -x + x^3, \ \frac{dy}{dt} = -2y.$$

example13()

Function `example13()` is a non-linear two-dimensional system given by:

$$\frac{dx}{dt} = 2 - x^2 - y^2, \ \frac{dy}{dt} = x^2 - y^2.$$

example14()

Function `example14()` is a non-linear two-dimensional system given by:

$$\frac{dx}{dt} = x^2 - y - 10, \ \frac{dy}{dt} = -3x^2 + xy.$$

example15()

Function `example15()` is a non-linear two-dimensional system given by:

$$\frac{dx}{dt} = x^2 - 3xy + 2x, \ \frac{dy}{dt} = x + y - 1.$$

lindemannMechanism()

The non-dimensional version of the Lindemann mechanism, used for gas-phase unimolecular reaction modelling, can be written in the form:

$$\frac{dx}{dt} = -x^2 + \alpha xy, \ \frac{dy}{dt} = x^2 - \alpha xy - y.$$

SIR()

The SIR model for the spread of an infectious disease can be written in the form:

$$\frac{dx}{dt} = -\beta xy, \ \frac{dy}{dt} = \beta xy - \nu y.$$

lotkaVolterra()

The Lotka-Volterra model, used to model interacting species of predator and prey in biology, is given by:

$$\frac{dx}{dt} = \lambda x - \epsilon xy, \ \frac{dy}{dt} = -\delta y + \eta xy.$$

competition()

A simple two species competition model, used in ecology, is given by:

$$\frac{dx}{dt} = r_1 x \left(\frac{K_1 - x - \alpha_{12} y}{K_1} \right), \ \frac{dy}{dt} = r_2 y \left(\frac{K_2 - y - \alpha_{21} x}{K_2} \right).$$

toggle()

A simple model for a genetic toggle switch is given by:

$$\frac{dx}{dt} = -x + \frac{\alpha}{1+y^{\beta}}, \ \frac{dy}{dt} = -y + \frac{\alpha}{1+x^{\gamma}}.$$

`morrisLecar()`

The Morris-Lecar model is a biological neuron model developed to reproduce the variety of oscialltory behaviour in relation to C_a^{++} and $K+$ conductance in the giant barnacle muscle fiber. It is given by:

$$\begin{aligned}\frac{dx}{dt} &= \frac{1}{20}\left[90 - 0.5g_{C_a}\left\{1+\tanh\left(\frac{x+1.2}{18}\right)\right\}(x-120) - 8y(x+84) - 2(x+60)\right], \\ \frac{dy}{dt} &= \phi\left[0.5\left\{1+\tanh\left(\frac{x-2}{30}\right)\right\} - y\right]\cosh\left(\frac{x-2}{60}\right).\end{aligned}$$

Exercises

Finally, this section contains numerous exercises that can be undertaken by the user to practice phase plane analysis themselves, both by hand and/or with help from **phaseR**. As such, parts of each exercise can be chosen so as to practice either performing the analysis yourself or computationally.

Exercise 1

Reproduce the plots and stability analysis of `example1()` given in this guide using the programs available in **phaseR**.

Exercise 2

In biology, the exponential growth model is used, for example, to model the growth or decline of a population. Qualitatively analyse it using the function `exponential()`. Restrict attention to the case $y(0) > 0$ and take $\beta = 1$. Where is the equilibrium point? What happens as you change the sign of β? What conclusions can we draw in general about this model? How does the sign of β reflect the biological system we may be modelling? Perform the analysis both yourself, and provide code for checking your results using **phaseR**.

Exercise 3

The monomolecular growth model assumes that the rate of change of a function, is proportional to the difference between its current value and some hypothetical value, K say, i.e.:

$$\frac{dy}{dt} = \beta(K - y).$$

Qualitatively analyse this model using the derivative function `monomolecular()`. Restrict attention to the case $y(0) > 0$, and begin with the values $\beta = 1, K = 3$. Where is the equilibrium point? What happens when you change the sign of β? How does the sign of β reflect the biological system we may be modelling? Perform the analysis both yourself, and provide code for checking your results using **phaseR**.

Exercise 4

von Bertalanffy assumed that an organism would gain material by anabolic processes, proportional to surface area. In addition, material would be lost by catabolic processes, proportional to weight. Since weight is related to surface area by a 2/3 power, his ODE for rate of change of weight y was therefore:

$$\frac{dy}{dt} = \alpha y^{2/3} - \beta y.$$

Qualitatively analyse this model using the derivative function `vonBertanalffy()`. Restrict attention to the case $y(0) > 0$, and begin with the values $\alpha = 2, \beta = 1$. Where are the equilibrium points? Are they stable? Perform the analysis both yourself, and provide code for checking your results using **phaseR**.

Exercise 5

Create your own derivative function, as explained earlier, for the ODE given by:

$$\frac{dy}{dt} = \sin(y).$$

Focusing on the range $y \in [-2\pi, 2\pi]$, perform a qualitative analysis yourself and provide code for checking your results using **phaseR**. Specifically, identify the location of the equilibrium points and classify them.

Exercise 6

Repeat the example analysis of the Lotka-Volterra system from this guide, using the `lotkaVolterra()` function and the parameter values $\lambda = \epsilon = \delta = \eta = 1$, along with the programs available in **phaseR**.

Exercise 7

For each of the following linear two-dimensional systems, perform a phase plane analysis. Ensure to identify and plot the nullclines, and then plot the velocity field. From this add trajectories for several initial conditions. Then locate the equilibrium point(s) and determine their classification. Perform this analysis first by hand and then also provide code to check your results using **phaseR**.

- `example3()`.
- `example6()`.
- `example7()`.
- `example9()`.

Exercise 8

For each of the following non-linear two-dimensional systems, perform a phase plane analysis. Ensure to identify and plot the nullclines, and then plot the velocity field. From this add trajectories for several initial conditions. Then locate the equilibrium point(s) and determine their classification. Perform this analysis first by hand and then also provide code to check your results using **phaseR**.

- `example10()`.
- `example13()`.
- `example14()`.
- `example15()`.
- `example16()`.

Exercise 9

Create your own derivative function, as explained earlier, for the ODE system given by:

$$\frac{dx}{dt} = 6x - 3y, \ \frac{dy}{dt} = 2x - y.$$

Then perform a phase plane analysis; identifying and plotting nullclines, the velocity field, trajectories, and locating the equilibrium point and classifying it. Perform this analysis first by hand and then also provide code to check your results using **phaseR**.

Exercise 10

Create your own derivative function, as explained earlier, for the ODE system given by:

$$\frac{dx}{dt} = x^2 + y^2 - 13, \ \frac{dy}{dt} = xy - 2x - 2y + 4.$$

Then perform a phase plane analysis; identifying and plotting nullclines, the velocity field, trajectories, and locating the equilibria and classifying them. Perform this analysis first by hand and also provide code to check your results using **phaseR**.

Exercise 11

Perform a phase plane analysis of the Lindemann mechanism, first by hand and then using the function `lindemannMechanism()` in **phaseR**. Where is the equilibrium point? Is it stable? How do things depend on the value of the parameter α?

Exercise 12

Perform a phase plane analysis of the SIR epidemic model, first by hand and then using the function `SIR()` in **phaseR**. Where are the equilibrium points and what is their stability? What does this mean biologically?

Exercise 13

Perform a phase plane analysis of the Lotka-Volterra model, first by hand and then using the function `lotkaVolterra()` in **phaseR**. Restrict your attention to the case where all four parameters are positive. Where are the equilibrium points? Are they stable? What does this mean biologically?

Exercise 14

Perform a phase plane analysis of the species competition model, first by hand and then using the function `competition()` in **phaseR**. Restrict your attention to the case where all four parameters are positive. Identify the four possible cases depending upon the parameter values. Where are the equilibrium points? Are they stable? What does this mean biologically?

Exercise 15

Perform a phase plane analysis of the genetic toggle switch model, first by hand and then using the function `toggle()` in **phaseR**. Restrict your attention to the case where all three parameters are positive. Where are the equilibrium points? Are they stable? What does this mean biologically?

Exercise 16

Perform a phase plane analysis of the Morris-Lecar model using the function `morrisLecar()` in **phaseR**. Restrict your attention to the case where the two parameters are positive. Where are the equilibrium points? Are they stable? What does this mean biologically?