Differential Equations with Linear Algebra: *SAGE* Help

Matthew R. Boelkins
Grand Valley State University

J. L. Goldberg
University of Michigan

Merle C. Potter
Michigan State University

October 13, 2009

Preface to SAGE Help

The purpose of this supplement to *Differential Equations with Linear Algebra* is to provide some basic support in the use of *SAGE*, analogous to the subsections of the text itself that offer similar guidance in the use of *Maple*.

SAGE is, as of this writing in 2009, the newest computer algebra system available. It differs from *Maple*, *Mathematica*, and *MATLAB* in two remarkable ways: (1) it can be run in any internet browser, and (2) it is free (including free to download). While it does not rival the three proprietary programs in its interface nor in the completeness of mathematics that the program can do, it certainly rivals them in cost, access, and functionality. For our purposes, SAGE can quite nicely do nearly all of the things that we use other CASs for. We recommend it heartily. For details, see `www.sagemath.org`, including the helpful wiki page at `http://wiki.sagemath.org/` and the quick reference sheets at `http://wiki.sagemath.org/quickref`.

In the following pages, the user will find parallel sections to those in the text titled "Using *Maple* to . . .". In particular, the following sections can be found here:

1.2.1 Row Reduction using *SAGE*

Obviously one of the problems with the process of row reducing a matrix is the potential for human arithmetic errors. Soon we will learn how to use computer software to execute all of these computations quickly; first, though, we can deepen our understanding of how the process works, and simultaneously eliminate arithmetic mistakes, by using a computer algebra system in a step-by-step fashion. In this supplement to the text, we use the software *SAGE*. For now, we only assume that the user is familiar with *SAGE*'s interface, and will introduce relevant commands with examples as we go.

To demonstrate various commands, we will revisit the system from Example 1.2.1. The reader should explore this code actively by entering and experimenting on his or her own. Recall that we were interested in row reducing the augmented matrix

$$\begin{bmatrix} 3 & 2 & -1 & 8 \\ 1 & -4 & 2 & -9 \\ -2 & 1 & 1 & -1 \end{bmatrix}$$

We enter the augmented matrix, say **A**, row-wise in *SAGE* with the command

```
A = matrix(QQ, [[3, 2, -1, 8], [1, -4, 2, -9],[ -2, 1, 1, -1]] )
```

to which the program makes no display. To see the matrix displayed, simply enter `A` on a subsequent command line. The "`QQ`" entry in the code is necessary to tell *SAGE* to do its matrix calculations using rational arithmetic; without this, it will think we're trying to work solely in the integers, and thus it would never be willing to work with numbers that are not whole numbers. For our purposes, it will normally suffice to use rational arithmetic; remembering the "`QQ`" is essential.

To execute row reduction in the desired example, we first want to swap rows 1 and 2.

Note well: *SAGE* enumerates the entries in vectors and rows in matrices *beginning with 0*. As such, if we want what we normally call the first row, we must refer to this as the zeroth row. In general, what we think of as row k is referenced in *SAGE* as row $k-1$. In written text, we'll refer to the row numbers in the usual way. These will be adjusted to one number lower in the commands we enter.

So, to swap rows 1 and 2, we use the syntax

```
A1 = A
A1.swap_rows(0,1)
```

The built-in row operations commands of *SAGE* actually overwrite the original matrix. Thus, before swapping rows, we store `A` in `A1`, and then swap the rows of `A1`. To see where we are in the row-reduction process, the user can now enter `A1`. Doing so reveals the following output:

$$\begin{matrix} 1 & -4 & 2 & -9 \\ 3 & 2 & -1 & 8 \\ -2 & 1 & 1 & -1 \end{matrix}$$

To perform row replacement, our next step is to add $(-3) \cdot R_1$ to R_2 (where rows 1 and 2 are denoted R_1 and R_2) to generate a new second row; similarly, we will add $2 \cdot R_1$ to R_3 for an updated row 3. Using the `M.add_multiple_of_row(i,j,a)`, which adds aR_j to R_i, we thus proceed to enter

```
A2 = A1
A2.add_multiple_of_row(1,0,-3)
A2.add_multiple_of_row(2,0,2)
```

The most updated matrix is now stored in `A2`, and if we display that matrix by entering its name, we see the following output:

$$\begin{pmatrix} 1 & -4 & 2 & -9 \\ 0 & 14 & -7 & 35 \\ 0 & -7 & 5 & -19 \end{pmatrix}$$

Next we will scale row 2 by a factor of $1/14$ and row 3 by $-1/7$ using the commands

```
A3 = A2;
A3.rescale_row(1,1/14)
A3.rescale_row(2,-1/7)
```

to find that $\mathbf{A}3$ has the entries

$$\begin{pmatrix} 1 & -4 & 2 & -9 \\ 0 & 1 & -\frac{1}{2} & \frac{5}{2} \\ 0 & -7 & 5 & -19 \end{pmatrix}$$

The remainder of the computations in this example involve slightly modified versions of the three versions of the commands demonstrated above, and are left as an exercise for the reader. Recall that the unique solution to the original system is $(1, 2, -1)$.

SAGE is certainly capable of performing all of these steps at once. After completing each step-by-step command above in the row-reduction process, the result can be checked by executing the commands

```
A = matrix(QQ, [[3, 2, -1, 8], [1, -4, 2, -9],[ -2, 1, 1, -1]] )
A.echelon_form()
```

The corresponding output should be

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & -1 \end{pmatrix}$$

which clearly reveals the unique solution to the system, $(1, 2, -1)$.

The reader is cautioned regarding three issues: the first is again that *SAGE* makes assumptions about the type of data the user inputs unless instructed otherwise. If we enter a matrix whose components are all integers, *SAGE* will try to do integer arithmetic unless we say to use rational numbers. Thus, the `QQ` in the definition of the matrix is essential. Further, even though we introduced new matrices `A1`, `A2`, and so on, *SAGE* still ends up altering the original matrix `A` in the step by step row operations. As such, if we want to check the individual commands we used in row operations with the `echelon_form` command, it may be necessary to re-execute the original matrix `A`, as shown above. And finally, we must always recall that *SAGE* numbers the rows in a matrix or entries in a vector starting with the index 0 for the first row.

1.3.2 Matrix Products using *SAGE*

After becoming comfortable with computing elementary matrix products by hand, it is useful to see how *SAGE* can assist us with more complicated computations. Here we demonstrate the relevant command.

Revisiting Example 1.3.2, to compute the product $\mathbf{Ax}$, we first enter $\mathbf{A}$ and $\mathbf{x}$ using the commands

```
A = matrix(QQ, [[1, -3],[-4, 1],[2, 5]])
x = vector(QQ, [-5, 2])
```

Next, we use the `*` symbol to inform *SAGE* that we want to multiply. Entering

```
b = A*x
```

and then asking *SAGE* to display `b` yields the expected output

$$(-11, 22, 0)$$

Note that *SAGE* displays this result as an ordered triple, which we interpret as a column vector, but the vector does not appear as we might expect on the screen. This can be addressed by instead using the syntax `x = matrix(QQ, [[-5],[2]])` for the vector $\mathbf{x}$.

Note: *SAGE* will obviously only perform the multiplication when it is defined. If, say, we were to attempt to multiply $\mathbf{A}$ and $\mathbf{x}$ where

```
A = matrix(QQ, [[1, -4, 2],[-3, 1, 5]]
x = vector(QQ,[-5, 2]),
```

then *SAGE* would report

```
Traceback (click to the left for traceback)
...
TypeError:  unsupported operand parent(s) for '*':  'Full MatrixSpace
of 2 by 3 dense matrices over Rational Field' and 'Vector space of dimension
2 over Rational Field'
```

which is *SAGE*'s complicated way of saying "you can't multiply a 2×3 matrix and a 2×1 vector."

1.7.1 Matrix Algebra using *SAGE*

While it is important that we first learn to add and multiply matrices by hand to understand how these processes work, just like with row reduction it is reasonable to expect that we'll often use available technology to perform tedious computations like multiplying a 4×5 and 5×7 matrix. Moreover, in real world applications, it is not uncommon to have to deal with matrices that have thousands of rows and columns, or more. Here we introduce a few *SAGE* commands that are useful in performing some of the algebraic manipulations we have studied in this section.

Let's consider some of the matrices defined in earlier examples:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -7 & 2 \end{bmatrix}, \ \mathbf{B} = \begin{bmatrix} -6 & 10 \\ 3 & 2 \end{bmatrix}, \ \mathbf{C} = \begin{bmatrix} -6 & 10 & -1 \\ 3 & 2 & 11 \end{bmatrix}$$

After defining each of these three matrices with the usual commands in *SAGE*, such as

```
A = matrix(QQ, [[1, 3, -4],[0, -7, 2]])
```

we can execute the sum of $\mathbf{A}$ and $\mathbf{C}$ and the scalar multiple $-3\mathbf{B}$ and see the results with the respective commands

```
A + C
-3*B
```

for which *SAGE* will report the outputs

$$\begin{pmatrix} -5 & 13 & -5 \\ 3 & -5 & 13 \end{pmatrix} \text{ and } \begin{pmatrix} 18 & -30 \\ -9 & -6 \end{pmatrix}$$

We have previously seen that to compute a matrix-vector product, the "star" is used to indicate multiplication, as in `A*x;`. The same syntax holds for matrix multiplication, where defined. For example, if we wish to compute the product $\mathbf{BA}$, we enter

```
B*A
```

which yields the output

$$\begin{pmatrix} -6 & -88 & 44 \\ 3 & -5 & -8 \end{pmatrix}$$

If we try to have *SAGE* compute an undefined product, such as $\mathbf{AB}$ through the command `A*B`, we get the error message

```
Traceback (click to the left for traceback)
...
TypeError:  unsupported operand parent(s) for '*':  'Full MatrixSpace
of 2 by 3 dense matrices over Rational Field' and 'Full MatrixSpace
of 2 by 2 dense matrices over Rational Field'
```

In the event that we need to execute computations involving an identity matrix, rather than tediously enter all the 1's and 0's, we can use the built-in *SAGE* command `identity_matrix(n)` where n is the number of rows and columns in the matrix. For example, entering

```
Id := identity_matrix(4)
```

so that displaying `Id` results in the output

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, if we desire to compute the transpose of a matrix **A**, such as

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -7 & 2 \end{bmatrix}$$

the relevant command is

```
    A.transpose()
```

which generates the output

$$\begin{pmatrix} 1 & 0 \\ 3 & -7 \\ -4 & 2 \end{pmatrix}$$

1.8.2 Matrix Inverses using *SAGE*

Certainly we can use *SAGE*'s row reduction commands to find inverses of matrices. However, an even simpler command exists that enables us to avoid having to enter the corresponding identity matrix. Let us consider the two matrices from Examples 1.8.2 and 1.8.3. Let

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & -2 \\ 1 & 1 & -1 \\ -2 & -1 & 3 \end{bmatrix}$$

be defined in *SAGE*. If we enter the command

```
A.inverse()
```

we see the resulting output which is indeed $\mathbf{A}^{-1}$,

$$\begin{pmatrix} 2 & -1 & 1 \\ -1 & 2 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

For the matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ -6 & -3 \end{pmatrix}$$

executing the command `Inverse[A]` produces the output

```
Traceback (click to the left for traceback)
...
ZeroDivisionError
```

which is *SAGE*'s way of saying "**A** is not invertible."

1.9.1 Determinants using *SAGE*

Obviously for most square matrices of size greater than 3×3, the computations necessary to find determinants are tedious and present potential for error. As with other concepts that require large numbers of arithmetic operations, *SAGE* offers a single command that enables us to take advantage of the program's computational powers. Given a square matrix $\mathbf{A}$ of any size, we simply enter

```
    A.det()
```

As we explore properties of determinants in the exercises of this section, it will prove useful to be able to generate random matrices. In *SAGE*, one accomplishes this for a 3×3 matrix with integer entries between -99 and 99 by the command

```
    A=random_matrix(ZZ,3,3,x=-99,y=99)
```

From here, we can compute the determinant of this random matrix simply by entering

```
    A.det()
```

See Exercise 11 in Section 1.9 for a particular instance where this code will be useful.

1.10.2 Using *SAGE* to Find Eigenvalues and Eigenvectors

Due to its reliance upon determinants and the solution of polynomial equations, the eigenvalue problem is computationally difficult for any case larger than 3×3. Sophisticated algorithms have been developed to compute eigenvalues and eigenvectors efficiently and accurately. One of these is the so-called QR algorithm, which through an iterative technique produces excellent approximations to eigenvalues and eigenvectors simultaneously.

While *SAGE* implements these algorithms and can find both eigenvalues and eigenvectors, it is essential that we not only understand what the program is attempting to compute, but also how to interpret the resulting output.

Given an $n \times n$ matrix $\mathbf{A}$, we can compute the eigenvalues of $\mathbf{A}$ with the command

```
A.eigenvalues()
```

Doing so for the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

from Example 1.10.2 yields the *SAGE* output

$$[3, 1]$$

and thus the two eigenvalues of the matrix $\mathbf{A}$ are 3 and 1. If we desire the eigenvectors, we can use the command

```
A.eigenvectors_right()
```

which leads to the output

```
[(3, [
(1, 1)
], 1), (1, [
(1, -1)
], 1)]
```

which tells us the eigenvector corresponding to $\lambda = 3$ is $[1 \;\; 1]^T$ and that the eigenvalue has algebraic multiplicity 1, the eigenvector corresponding to $\lambda = 1$ is $[-1 \;\; 1]^T$, again with algebraic multiplicity 1. The algebraic multiplicity of an eigenvalue λ is the number of times that λ appears as a root of the characteristic polynomial.

As a brief aside, we note that it is possible to consider both the so-called "left eigenvectors" and "right eigenvectors" of a matrix $\mathbf{A}$. The traditional eigenvectors encountered in linear algebra are the "right" ones; *SAGE* is programmed to be able to find both, and this is why we use the command `A.eigenvectors_right()`.

SAGE is extremely powerful. It is not at all bothered by complex numbers. So, if we enter a matrix similar[1] to the one in Example 1.10.3 that has no real eigenvalues, *SAGE* will find complex eigenvalues

[1] *SAGE* prefers to work exactly with the entries in matrices, specifically working in a particular *ring* to do computations. Because the matrix in Example 1.10.3 as to be considered over the ring of real numbers, which *SAGE* cannot represent exactly, the program actually balks at computing its eigenvalues. Here we present a different matrix whose entries are rational numbers, in which the program can work the arithmetic exactly, though it still reports to the user numerical estimates to the eigenvalues, as shown above. There are ways to get around this issue in *SAGE*, but that discussion lies beyond the scope of this course.

and eigenvectors. To see how this appears, we enter the matrix

$$\mathbf{R} = \begin{bmatrix} \frac{3}{5} & -\frac{4}{5} \\ \frac{4}{5} & \frac{3}{5} \end{bmatrix}$$

and execute the command

```
R.eigenvectors_right()
```

The resulting output is

```
[(0.6000000000000000?  - 0.8000000000000000?*I, [(1, 1*I)], 1),
(0.6000000000000000?  + 0.8000000000000000?*I, [(1, -1*I)], 1)]
```

Note that here *SAGE* is using "`I`" to denote $\sqrt{-1}$. This rotation matrix $\mathbf{R}$ does not have any real eigenvalues. We can use familiar properties of complex numbers (most importantly, $i^2 = 1$) to actually check that the equation $\mathbf{Rx} = \lambda\mathbf{x}$ holds for the listed complex eigenvalues and complex eigenvectors above. However, at this point in our study, these complex eigenvectors are of less importance, so we defer further details on them until later work with systems of differential equations. We also mention the presence of the "`?`" in the output above. This notation indicates that (a) *SAGE* knows the exact value of the eigenvalue, and (b) that in its numerical approximation shown on the screen, it is promising that every digit up until the last one is correct.

One final example is relevant here to see how *SAGE* deals with repeated eigenvalues and "missing" eigenvectors. If we enter the 3×3 matrix $\mathbf{A}$ from Example 1.10.4 and then execute the `Eigensystem` command, we receive the output

```
[(-4, [
(1, -4/3, -1/2)
], 1), (3, [
(1, -2/5, 1/5)
], 2)]
```

The first portion of the output, `[(-4, [(1, -4/3, -1/2)], 1)`, shows that $\lambda = -4$ is an eigenvalue of algebraic multiplicity 1 that corresponds to the eigenvector $[1 \ -4/3 \ -1/2]^T$. The algebraic multiplicity is the number of times the eigenvalue appears as a root of the characteristic equation.

The second portion of the output, `(3, [(1, -2/5, 1/5)], 2)]`, reveals that 3 is a repeated eigenvalue of $\mathbf{A}$ with multiplicity 2 with corresponding eigenvector $[1 \ -2/5 \ 1/5]^T$. It is especially notable that although this eigenvalue is a repeated root of the characteristic equation, there is only one corresponding linearly independent eigenvector. If there were another linearly independent eigenvector corresponding to $\lambda = 3$, *SAGE* would list this vector as well.

Because *SAGE* reports just two linearly independent eigenvectors for this 3×3 matrix, this demonstrates that $\mathbb{R}^3$ does not have a linearly independent spanning set that consists of eigenvectors of $\mathbf{A}$.

2.2.1 Plotting Slope Fields using *SAGE*

In its current internet interface, *SAGE* does not have a single command that we can use to plot the slope field for a differential equation; however, with a bit of effort, we can manipulate its ability to plot a general vector field to generate the visual representation of a slope field that we week. Doing so uses the program's ability to plot vector fields.

To plot a vector field, we need a function of two variables x and y with two components u and v that will generate the direction of a vector in the field at any point. Said differently, given a point (x, y), we want to generate a vector $[u \;\; v]^T$ to plot emanating from the point (x, y). This is precisely what the differential equation

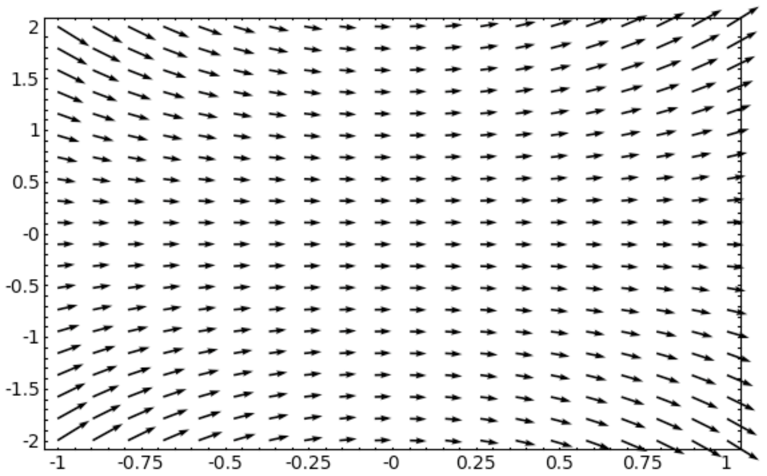
$$\frac{dy}{dx} = f(x, y)$$

accomplishes. With $f(x, y)$ being the slope of the tangent line at point (x, y), we can realize a vector at that point by considering the vector $[1 \;\; f(x, y)]^T$, whose slope will be exactly $f(x, y)$.

Thus, to plot the direction field associated with a given differential equation, say $\frac{dy}{dx} = xy$, on the window $[-1, 1] \times [-1, 1]$, we use the syntax

```
plot_vector_field((1, x*y), (x,-1,1), (y,-1,1))
```

which generates the output



It is often ideal to have all vectors in the slope field of the same length. If we normalize the vector field by making the function that generates it one whose output is a unit vector, *SAGE* will take care of the rest. We do this in the command below, plus name the plot "`SlopeField`" for use in a moment:

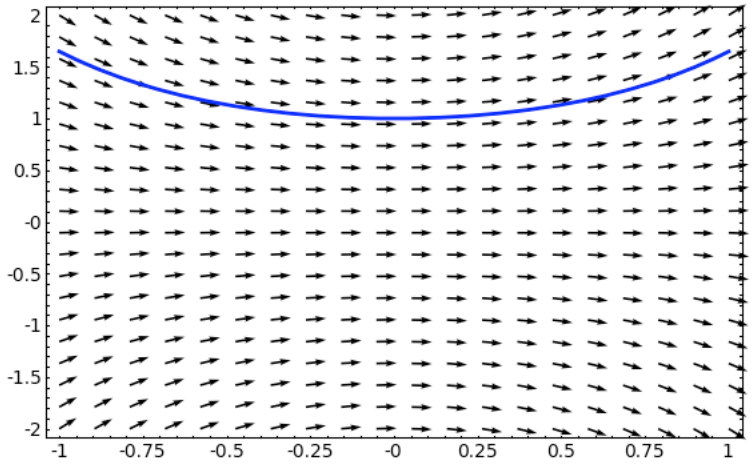
```
SlopeField = plot_vector_field( (1/(sqrt(1^2 + (x*y)^2)), x*y/(sqrt(1^2
+ (x*y)^2)) ), (x,-1,1), (y,-1,1))
```

Without a well-chosen window selected by the user, the plot *SAGE* generates may not be very insightful. For example, if the above command were changed so that the range of y values is `-10 .. 10`, almost no information can be gained from the slope field. As such, we will strive to learn to analyze the expected behavior of a differential equation from its form so that we can choose windows well in related plots; we may often have to experiment and explore to find graphs that are useful.

Finally, if we are interested in one or more related initial-value problems, some additional effort enables us to sketch the graph of each corresponding solution. Say we are interested in the IVP solution that satisfies the given DE along with the initial condition $y(0) = 1$. We may use *SAGE* to solve the IVP and then plot this result using the following syntax:

```
x = var('x')
y = function('y', x)
DE = diff(y,x) - x*y
f = desolve(DE, [y,x], (0,1))
SolnPlot = plot(f, (x, -1,1), color=[0,0,1], thickness=2))
(SolnPlot+SlopeField).show()
```

This results in the image shown below that demonstrates the slope field and the curve that represents the solution to the IVP.



It is often easier for the user to simply plot an IVP's solution by hand, just by using the initial condition and following the "directions" provided by the slope field.

3.4.1 Plotting Direction Fields for Systems using *SAGE*

As we noted in section 2.2.1, *SAGE* does not have a single command that we can use to plot the slope field for a differential equation; however, with a bit of effort, we can manipulate its ability to plot a general vector field to generate the visual representation of a slope field that we week.

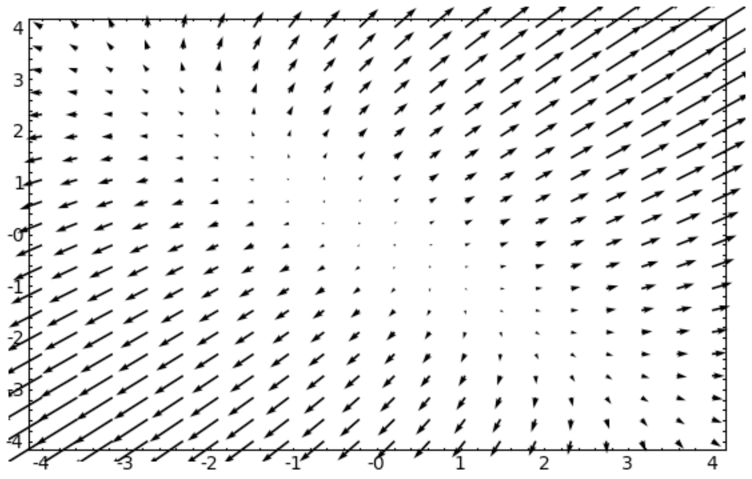
To plot a vector field, we need a function $f(x, y)$ with two components u and v that will generate the direction of a vector in the field at any point. Said differently, given a point (x, y), we want to generate a vector $[u \;\; v]^T$ to plot emanating from the point (x, y).

In the context of plotting the direction field for the system of DEs given by $\mathbf{x}' = \mathbf{A}\mathbf{x}$, the vector we wish to plot at (x, y) is $\mathbf{x}'$. Viewing $\mathbf{x} = [x(t) \;\; y(t)]^T$, it follows $\mathbf{x}' = [x'(t) \;\; y'(t)]^T$. Thus, the vector field we desire is given by $u = x'$ and $v = y'$. This precisely is the vector given by taking the product $\mathbf{A}\mathbf{x}$.

Therefore, to plot the direction field associated with the system of DEs $\mathbf{x}' = \mathbf{A}\mathbf{x}$ given by the matrix $\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$ from Example 3.4.1, we use the following syntax:

```
plot_vector_field( (3*x+2*y, 2*x+3*y), (x,-4,4), (y,-4,4) )
```

This command produces the output shown below.



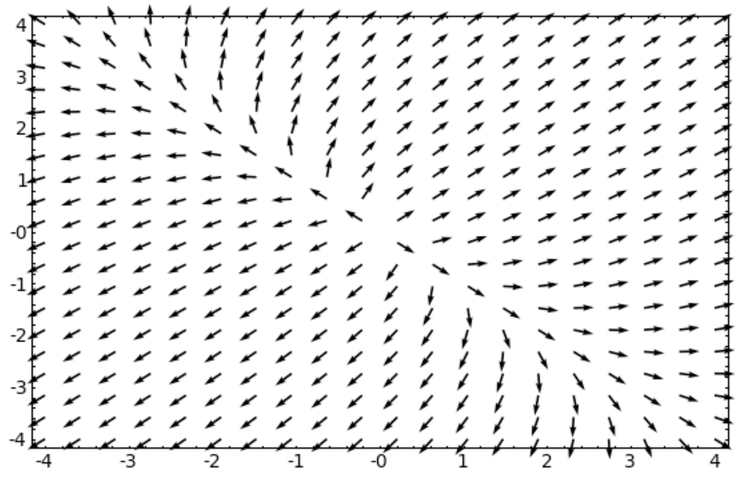
Because *SAGE* does not automatically scale the vectors in the vector field to be of consistent length, it is more helpful to make each vector in the field of the same length. This is accomplished by scaling the vector to be plotted by its norm, making it a unit vector. To that end, we use the commands

```
n = sqrt( (3*x+2*y)^2 + (2*x + 3*y)^2 )
```

and

```
plot_vector_field( ((3*x+2*y)/n, (2*x+3*y)/n), (x,-4,4), (y,-4,4) )
```

which produces the output



From here, it is a straightforward exercise to sketch trajectories by hand; while one can also generate these trajectories in *SAGE*, doing so is relatively complicated. The user is instead encouraged to think about how the direction field aligns with the eigenvalues and eigenvectors of the matrix of the system, to plot the straight-line solutions by hand, and then think about how the nonlinear trajectories should appear, especially as $t \to \infty$. As shown in section 2.2.1, it is also possible to superimpose the coordinate axes on the plot, and the directions stated in 2.2.1 apply in the exact same way in this context.

As always, the user can experiment some with the window in which the plot is displayed: the range of x- and y-values will affect how clearly the direction field is revealed.

3.7.1 Applying Variation of Parameters Using *SAGE*

Here we address how *SAGE* can be used to execute the computations in a problem such as the one posed in Example 3.7.2, where we are interested in solving the nonhomogeneous linear system of equations given by

$$\mathbf{x}' = \begin{bmatrix} 2 & -1 \\ 3 & -2 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1/(e^t+1) \\ 1 \end{bmatrix}$$

Because we already know how to find the complementary solution, we focus on determining $\mathbf{x}_p$ by variation of parameters. First, we use the complementary solution,

$$\mathbf{x}_h = c_1 e^{-t} \begin{bmatrix} 1 \\ 3 \end{bmatrix} + c_2 e^t \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

to define the fundamental matrix $\mathbf{\Phi}(t)$. To define and view this matrix in *SAGE*, we first declare the variable t with the command `var('t')`, followed by

```
    Phi = matrix([[exp(-t), exp(t)],[3*exp(-t), exp(t)]])
```

Remember, to actually see the matrix `Phi` displayed on the screen, we simply enter its name as a command. We next use the `Inverse` command to find $\mathbf{\Phi}^{-1}$ by entering

```
    PhiInv = simplify(Phi.inverse())
```

The `simplify` command will be used repeatedly in this context to help ensure that the program doesn't unnecessarily complicate the algebraic expressions. At this stage, when we display resulting output ($\mathbf{\Phi}^{-1}$), we see the matrix

$$\begin{pmatrix} -\frac{e^t}{2} & \frac{e^t}{2} \\ \frac{3e^{-t}}{2} & -\frac{e^{-t}}{2} \end{pmatrix}$$

Next, in order to compute $\mathbf{\Phi}^{-1}(t)\mathbf{b}(t)$, we must enter the function $\mathbf{b}(t)$. We enter

```
    b = matrix([[1/(exp(t) + 1)], [1]])
```

and then

```
    y = simplify(PhiInv*b)
```

At this point, y is a 2×1 array that holds the vector function $\mathbf{\Phi}^{-1}(t)\mathbf{b}(t)$. Specifically, the output for y displayed by *SAGE* is

$$\begin{pmatrix} -\frac{e^t}{2e^t+2} + \frac{e^t}{2} \\ \frac{2e^{-t}}{2e^t+2} - \frac{e^{-t}}{2} \end{pmatrix}$$

To access the components in y, we reference them with the commands `y[0,0]` and `y[1,0]`, remembering that *SAGE* starts its indexing of rows and columns of matrices with 0, rather than 1. Since we have to integrate $\mathbf{\Phi}^{-1}(t)\mathbf{b}(t)$ component-wise, we enter

```
    Y = matrix([[integrate( y[0,0], t)], [integrate(y[1,0], t)]])
```

This last command, together with asking *SAGE* to display Y, produces the output

$$\begin{pmatrix} \frac{e^t}{2} - \frac{1}{2}\ln(1+e^t) \\ -\frac{3}{2}t - e^{-t} + \frac{3}{2}\ln(1+e^t) \end{pmatrix}$$

and obviously stores $\mathbf{\Phi}^{-1}(t)\mathbf{b}(t)$ in $\mathbf{Y}$. We add that we write "ln" where *SAGE* writes "Log."

Finally, in order to compute $\mathbf{\Phi}(t)\int \mathbf{\Phi}^{-1}(t)\mathbf{b}(t)\,dt$, we need to enter `Phi*Y`. Of course, we again want to simplify, so we first expand and use the syntax

```
    simplify(expand(Phi*Y))
```

which produces the output

$$\begin{pmatrix} -\frac{3}{2}te^t - \frac{1}{2}e^{-t}\ln(e^t+1) + \frac{3}{2}e^t\ln(e^t+1) - \frac{1}{2} \\ \frac{3}{2}te^t - \frac{3}{2}e^{-t}\ln(e^t+1) + \frac{3}{2}e^t\ln(e^t+1) + \frac{1}{2} \end{pmatrix}$$

This last result is precisely the particular solution $\mathbf{x}_p$ to the original system of nonhomogeneous equations given in Example 3.7.2.

4.6.1 Solving Characteristic Equations using *SAGE*

While solving linear differential equations of order n requires nearly identical methods to DEs of order 2, there is one added challenge from the outset: solving the characteristic equation. The characteristic equation is a polynomial equation of degree n; while every such equation of degree 2 can be solved using the quadratic formula, equations of higher order can be much more difficult, and (for equations of degree 5 and higher) often impossible, to solve by algebraic means.

Computer algebra systems like *SAGE* provide useful assistance in this matter with commands for solving equations exactly and approximately. For example, say we have the characteristic equation

$$r^4 - r^3 - 7r^2 + r + 6 = 0$$

To solve this exactly in *SAGE*, we first enter `var('r')`, followed by

```
solve(r^4 - r^3 - 7 * r^2 + r + 6 == 0, r)
```

SAGE produces the output

```
[r == -2, r == -1, r == 1, r == 3]
```

showing that these are the four roots of the characteristic equation.

Of course, not all polynomial equations will have all integer solutions, much less all real solutions. For example, if we consider the equation

$$r^4 + r^3 + r^2 + r + 1 = 0$$

and use the `solve` command, we see that

```
solve(r^4 + r^3 + r^2 + r + 1 == 0, r)
```

results in the output

```
[r == -1/4*I*sqrt(sqrt(5) - 1)*sqrt(2)*5^(1/4) - 1/4*sqrt(5) - 1/4,
r == 1/4*I*sqrt(sqrt(5) - 1)*sqrt(2)*5^(1/4) - 1/4*sqrt(5) - 1/4, r
== -1/4*I*sqrt(sqrt(5) + 1)*sqrt(2)*5^(1/4) + 1/4*sqrt(5) - 1/4, r ==
1/4*I*sqrt(sqrt(5) + 1)*sqrt(2)*5^(1/4) + 1/4*sqrt(5) - 1/4]
```

which is not very helpful. The same result comes from the `root` command, which would be entered as

```
(r^4 + r^3 + r^2 + r + 1 == 0, r).roots(r)
```

In this case, rather than having exact results in terms of roots of negative numbers, we might prefer a decimal approximation to the zeros of the equation. Because *SAGE* always strives to produce exact solutions to equations, especially ones with integer coefficients, we have to provide direct instruction to work around this. *SAGE* works by default in certain *rings* (a mathematical number structure beyond the scope of this course), and which ring is being used affects to form of the output. One way to achieve complex roots in numerical format is to first apply the command `r = polygen[ZZ]`, and then to apply the roots command with a particular option set. Specifically,

```
(r^4 + r^3 + r^2 + r + 1).roots(ring=CIF)
```

which generates the result

```
[(-0.809016994374948? - 0.587785252292473?*I, 1), (-0.809016994374948?
+ 0.587785252292473?*I, 1), (0.309016994374948? - 0.951056516295154?*I,
1), (0.309016994374948? + 0.951056516295154?*I, 1)]
```

Note that the `?` again indicates that the preceding decimal place is in question.

For polynomial equations of degree 5 or more, the `roots(ring=CIF)` command is usually the appropriate tool to use to determine accurate approximations of the equation's solutions.

5.4.3 The Heaviside and Dirac Functions in *SAGE*

As of this writing in early fall 2009, neither the Heaviside nor Dirac functions belong to *SAGE*'s library of basic functions. Discussion of a patch that can be downloaded to include them can be found at `http://groups.google.com/group/sage-devel/browse_thread/thread/5f7de72642b8e568/f226acf9a6331d4d?pli=1` and the patch itself may be obtained at `http://trac.sagemath.org/sage_trac/ticket/2452`.

Short of implementing the patch to run on a locally-installed version of *SAGE*, or writing one's own patch, there does not seem to be an elementary way to handle the Heaviside and Dirac functions at this time. We will update this file if or when the standard *SAGE* package includes such capability.

5.6.1 Laplace Transforms and Inverse Transforms using *SAGE*

As we have noted, while we have computed Laplace transforms for a range of functions, there are many more examples we have not considered. Moreover, even for familiar functions, certain combinations of them can lead to tedious, involved calculations. Perhaps the most involved computations arise from IVPs that have a unit step function or impulse function involved. As of this writing in fall 2009, *SAGE* does not yet have the built-in capacity to work with these functions easily. See the discussion in section for more information on this topic. We note that both *Maple* and *Mathematica* do have this capability and that we discuss this in the respective help sections for those programs.

Other than problems involving step and impulse functions, *SAGE* is fully capable of computing Laplace transforms of functions in a straightforward way, as well as inverse transforms. Here we demonstrate the syntax required in the solution of an initial-value problem similar to Example 5.5.4:

$$y'' + 4y' + 13y = 2\sin 3t, \quad y(0) = 1,\ y'(0) = 0 \tag{1}$$

To use *SAGE* to compute the Laplace transform of $2\sin 3t$, we use the syntax

```
(2*sin(3*t)).laplace(t,s)
```

The above command results in the output

$$\frac{6}{s^2+9}$$

which is precisely the transform we expect.

After computing by hand the transform of the left-hand side of (1) and solving for $Y(s)$, we have

$$Y(s) = \frac{s+4}{s^2+4s+13} - 2e^{-\pi s}\frac{3}{(s^2+9)(s^2+4s+13)}$$

Here we may use *SAGE*'s `inverse_laplace` command to determine $\mathcal{L}^{-1}[Y(s)]$. While we could choose to do so all at once, for simplicity of display we do so in two steps. First, we declare the variables `t = var('t')` and `s = var('s')`, and then let `F = (s+4)/(s^2 + 4*s + 13)`. To find the inverse transform, we enter

```
y1 = F.inverse_laplace(s, t)
```

which results in the output

$$\frac{1}{3}e^{-2t}(2\sin(3t) + 3\cos(3t)) \tag{2}$$

Similarly, for the second term in $Y(s)$, we let `G = 6/( (s^2 + 9)*(s^2 + 4*s + 13) )` and then enter

```
y2 = F.inverse_laplace(s, t)
```

Here, *SAGE* produces the output

$$\frac{1}{20}e^{-2t}(\sin(3t) + 3\cos(3t)) + \frac{1}{20}\sin(3t) - \frac{3}{20}\cos(3t) \tag{3}$$

The sum of the two functions of t that have resulted from inverse transforms in (2) and (3) is precisely the solution to the IVP.

Note that in computing the inverse transform (3), *SAGE* has implicitly executed the partial fraction decomposition of the expression

$$G = \frac{6}{(s^2+9)(s^2+4s+13)}$$

If we wish to find this explicitly, we can use the command

```
    G.partial_fraction()
```

which produces the output

$$-\frac{3}{20}\frac{s-1}{s^2+9} + \frac{3}{20}\frac{s+3}{s^2+4s+13}$$

In general, we see that to compute the Laplace transform of $f(t)$ in *SAGE* we use the syntax

```
    f.laplace(t,s)
```

whereas to compute the inverse transform of $F(s)$, we enter

```
    F.inverse_laplace(s, t)
```

6.2.1 Plotting Direction Fields of Nonlinear Systems using *SAGE*

The *SAGE* syntax used to generate the plots in this section is essentially identical to that discussed for direction fields for linear systems in section 3.4.1.

Consider the system of differential equations from Example 6.2.1 given by

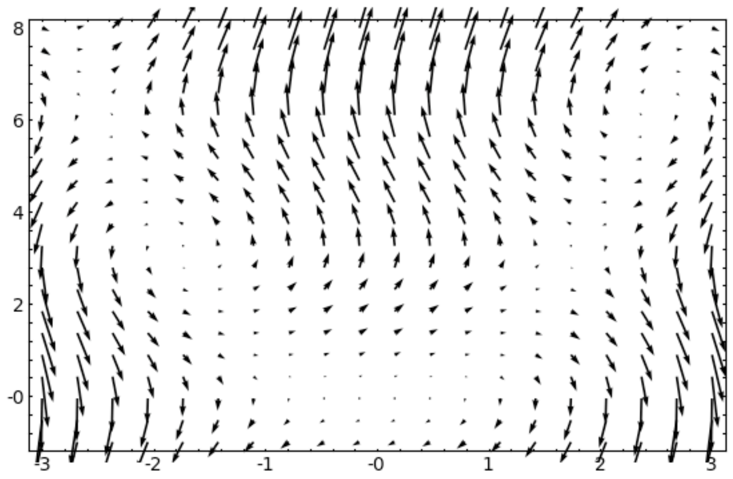
$$\begin{aligned} x' &= \sin(y) \\ y' &= y - x^2 \end{aligned}$$

We use x and y in place of x_1 and x_2 to simplify the syntax in *SAGE*.

As in 3.4.1, we use the `plot_vector_field` command to plot the vector $[x', y']^T$, and thus use the syntax

```
plot_vector_field( (sin(y), y-x^2), (x,-3,3), (y,-1,8) )
```

which produces the somewhat output



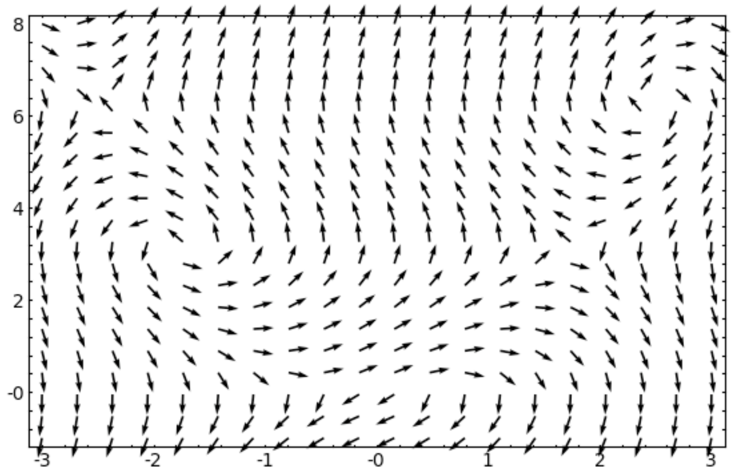
We thus again scale the vectors being plotted to be of consistent length, first letting

```
n = sqrt( sin(y)^2 + (y-x^2)^2 )
```

so that `n` is the norm of the vector being plotted, and then using

```
plot_vector_field( (sin(y)/n, (y-x^2)/n), (x,-3,3), (y,-1,8) )
```

which results in the more helpful plot



As we can see, while the plot provides some useful information, it principally shows that there is something interesting happening at some points along the parabola $y = x^2$, specifically at the equilibrium points discussed in Example 6.2.1. Further analysis to accompany the work of the computer is merited; some possibilities for this exploration are considered in Sections 6.3 and 6.4 of the text, after which it will be easier to plot trajectories by hand.

It is worth noting that while *SAGE* has many excellent traits and capabilities, some other computer algebra systems are easier to work with for plotting meaningful direction fields and trajectories. For example, see the discussion in the text regarding *Maple*.