Using MATLAB to Solve ODEs – Matrix, Symbolic, and Numeric Approaches

**Introduction**

So far in BME 210 we have been focusing on analytical solutions to solve ordinary differential equations (ODEs). We have discussed techniques for solving both homogeneous and nonhomogeneous first order, second order, and higher order ODEs. However, the exception of separable differential equations all of these techniques have been limited to linear ODEs. While linear ODEs are a vast minority among all possible equations, it turns out that a many engineering systems can be approximated with equations of this type.

We also showed that systems of linear ODEs can be solved with a linear-algebra-based approach using *eigenvalues* and *eigenvectors*, which came as a result of the assumed solution form $y = e^{At}$. Furthermore, we showed that higher-order linear ODEs could also be solved with linear algebra by converting them into systems of ODEs (Lecture 3). Again, this technique equips us with a very valuable tool, because now we can use computational tools, such as MATLAB, that are great at manipulating matrices to solve complex single and systems of differential equations.

But what about non-separable *nonlinear* ODEs? Are they solvable? Yes... but not always analytically. Luckily our predecessors noticed that over small enough changes in a dependent variable (e.g. $x$ or $t$), ANY equation can be approximated as a linear equation. So if we break down a function into lots of tiny incremental steps, we can approximate its overall behavior by estimating how it behaves locally (for small $\Delta x$ or $\Delta t$). We call this type of approach a “numerical solution” because we can approximate the solution to the ODE at each increment, but do not ever develop an analytical solution. The smaller each $\Delta x$ or $\Delta t$, the more accurate our overall solution can be... but the more calculations we have to perform. Luckily, we have computers that can handle many repeated calculations with ease.

The goal of today’s activities is twofold:

1. Give you experience solving systems of nonhomogeneous ODEs using two different approaches in MATLAB:
   a. Linear Algebra
   b. MATLAB’s Symbolic Toolbox and *dsolve* function
2. Show how you can use MATLAB’s numeric ODE solvers (ode45, ode23, etc.) to find numerical solutions to nonlinear ODEs.

**Solving Systems of Nonhomogeneous ODEs Using MATLAB**

In our first lecture, we looked at how we could represent a “drug-delivery” problem with a system of first-order ODEs with initial conditions (i.e. an “initial value problem” (IVP)). The model we considered is shown below in Figure 1. We assumed that the drug travels between the stomach and bloodstream at a rate ($k_d$) proportional to the amount of drug in the stomach, which we called $D(t)$. Similarly, the drug is metabolized by the body at a rate ($k_m$), proportional to the amount of drug in the body ($B(t)$), thereby producing a waste product $W(t)$. The waste product is also formed by way of drug in the stomach being absorbed by food and digested at rate $k_d$ proportional to $D(t)$.
We solved this problem as a homogeneous IVP by assuming that we started with an initial dose in the stomach of $D(0) = D_0$ at time $t = 0$, and that there was no other drug in the body at that time ($B(0) = 0$). Using conservation of mass, we know that for some quantity $m$

$$\frac{dm}{dt} = \frac{dm_{in}}{dt} + \frac{dm_{out}}{dt}$$

Which allowed us to develop a system of equations describing the amount of drug in each state as a function of time:

$$D'(t) = -(k_a + k_d)D$$
$$B'(t) = k_a D - k_m B$$
$$W'(t) = k_d D + k_m B$$

With Initial Conditions:

$$D(0) = D_0$$
$$B(0) = W(0) = 0$$

Assuming that the solution will have the form $y = e^{\lambda t}$, we are able to represent this system as

$$\bar{y}' = A\bar{y}; \quad \bar{y}_0 = \begin{bmatrix} D_0 \\ 0 \\ 0 \end{bmatrix}$$

Now we can use our Eigenvalue and Eigenvector methods to produce the solution:

$$\begin{bmatrix} D \\ B \\ W \end{bmatrix} = c_1 \bar{x}_1 e^{\lambda_1 t} + c_2 \bar{x}_2 e^{\lambda_2 t} + c_3 \bar{x}_3 e^{\lambda_3 t}$$

Were $\lambda_1, \lambda_2,$ and $\lambda_3$ are the eigenvalues of matrix $A$, and $\bar{x}_1, \bar{x}_2,$ and $\bar{x}_3$ are their associated eigenvectors. We then can use our initial conditions to find our constants:

$$\begin{bmatrix} D_0 \\ 0 \\ 0 \end{bmatrix} = c_1 \bar{x}_1 + c_2 \bar{x}_2 + c_3 \bar{x}_3 = M \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Where

$$M = [\bar{x}_1, \bar{x}_2, \bar{x}_3]$$
Is a matrix where each column is an eigenvector of $A$. Solving

$$
\begin{bmatrix}
D_0 \\
0 \\
0
\end{bmatrix} = M \begin{bmatrix}
c_1 \\
c_2 \\
c_3
\end{bmatrix}
$$

For $[c_1, c_2, c_3]^T$ gives us our coefficients.

**Side Note:**
Conservation of mass also tells us that at every time $t$, the total amount of drug in the system has to remain constant. Therefore, $D + B + W = D_0$ must be true for all $t$. Therefore, we actually only have two independent equations, and the third can always be calculated from the other two. As a result, we can simply solve for $D(t)$ and $B(t)$, then calculate $W(t)$ at the end. Our system now becomes:

$$
\begin{align*}
D'(t) &= -(k_a + k_d)D \\
B'(t) &= k_aD - k_mB \\
W(t) &= D_0 - (D + B)
\end{align*}
$$

$$
\begin{align*}
D(0) &= D_0 \\
B(0) &= 0
\end{align*}
$$

And produces the solution:

$$
\begin{bmatrix}
D \\
B
\end{bmatrix} = c_1 \tilde{x}_1 e^{\lambda_1 t} + c_2 \tilde{x}_2 e^{\lambda_2 t} + \cdots + c_n \tilde{x}_n e^{\lambda_n t} \\
W = D_0 - (D + B)
$$

**Optional Exercise:**
Show that solving the system of three equations produces the an eigenvalue $\lambda_3 = 0$ and eigenvector $\tilde{x}_3 = [0 \ 0 \ 1]^T$. What does this result imply?

**Steps to Solve a Homogeneous System of ODEs:**
1. Set up our governing equation: $y'' = A \tilde{y}$
2. Find eigenvalues and eigenvectors of $A$
3. Find general equation: $y = c_1 \tilde{x}_1 e^{\lambda_1 t} + c_2 \tilde{x}_2 e^{\lambda_2 t} + \cdots + c_n \tilde{x}_n e^{\lambda_n t}$
4. Define initial conditions vector $\tilde{y}_0$
5. Solve for coefficients $c_1$ and $c_2$

**Nonhomogeneous Systems of ODEs – Example: Time-Release Capsules & Other “Smart Drugs”**
A growing area of biomedical engineering and pharmaceuticals focuses on the development of controlled drug delivery, such as time-release capsules, biodegradable capsules, osmosis pumps, etc. Rather than making all of the drug available at once and relying on the body’s natural absorption to regulate the amount of drug present in the bloodstream, drug casings (capsules) can be designed to break-down at a specific rates. This regulated degradation allows the drug manufacturers to exert more control on how much drug is available to the body at any given time.

Let’s update our problem to include a time-released capsule. Rather than setting our initial condition of the stomach to $D(0) = D_0$, we can define an input function

$$
f(t) = D_0 (1 - e^{-rt})
$$
That represents the amount of drug made available for absorption by the time-released capsule as a function of time (Figure 2). Our conservation equation for the amount of drug in the stomach now becomes

\[
\frac{dD}{dt} = \frac{dD_{in}}{dt} - \frac{dD_{out}}{dt} = \frac{df}{dt} - (k_a + k_d)D
\]

Therefore, we can rewrite our system of equations as:

\[
\ddot{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + A \begin{bmatrix} D \\ B \end{bmatrix} + \begin{bmatrix} D' \\ B' \end{bmatrix} = A \ddot{y} + \ddot{g}
\]

\[
W = f(t) - (D + B)
\]

Where \(\frac{df}{dt} = g(t)\). We can recognize \(\ddot{y} = A \ddot{y} + \ddot{g}\) as our standard form for a nonhomogeneous system of equations.

We know that to solve such a system by hand, we would follow the steps below:

**Steps to Solve a Nonhomogeneous System of ODEs**

1. Solve homogeneous system of equations (as shown above): \(\ddot{y} = A \ddot{y}\)
2. Define input function \(g(t)\) to establish: \(\ddot{y} = A \ddot{y} + \ddot{g}\)
3. Pick a form for \(\ddot{y}_p\) and solve for undetermined coefficients
4. Combine homogeneous and particular solutions: \(\ddot{y} = \ddot{y}_h + \ddot{y}_p\)
5. Define initial conditions vector \(\ddot{y}_0\)
6. Solve for coefficients \(c_1\) and \(c_2\) using initial conditions vector

**Using MATLAB to Solve a System of Nonhomogeneous ODEs – Linear Algebra Approach**

(This section follows along with the demonstration script ‘DrugProblem_SystemApproach.m’. You should attempt to develop your own script to solve the problem, but can refer to the demo script if necessary. We have already seen an example of how to solve systems of linear homogeneous ODEs in MATLAB.)

Since we know how we would go about solving this system by hand, let’s use our steps to develop a MATLAB procedure to do the heavy lifting for us.
Looking at Step 1 in our procedure, we know that we are going to have to find the eigenvalues and eigenvectors of our matrix \( \mathbf{A} \). Luckily, MATLAB has a function, called \texttt{eig} \(^1\), that makes these calculations very simple. Once we have defined a matrix, \( \mathbf{A} \), we can use the following syntax:

\[
>> [V,D] = \text{eig}(\mathbf{A});
\]

To assign the eigenvectors and eigenvalues of input matrix \( \mathbf{A} \) to variables \( \mathbf{V} \) and \( \mathbf{D} \), respectively. The output variable \( \mathbf{V} \) will be a matrix with the same dimensions of \( \mathbf{A} \), with each column in \( \mathbf{V} \) corresponding to an eigenvector of \( \mathbf{A} \). Similarly, the output variable \( \mathbf{D} \) will be a diagonal matrix, with the same dimensions of \( \mathbf{A} \). The \( i^{th} \) diagonal value of \( \mathbf{D} \) is the eigenvalue corresponding to the \( i^{th} \) column (eigenvector) of \( \mathbf{V} \).

### Eigenvalue/vector Activities

1. Use the \texttt{eig} function to calculate the eigenvectors and eigenvalues of the matrix

\[
\mathbf{A} = \begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

2. Remember that the definition of an eigenvector, \( \mathbf{x} \), satisfies the equation \( \mathbf{A} \mathbf{x} = \lambda \mathbf{x} \), \( \lambda \) is the associated eigenvalue. Use MATLAB to prove that this equation holds for the second (middle) eigenvector and associated eigenvalue calculated in Question 1 above.

3. Sometimes it is inconvenient to have our eigenvalues stored as a diagonal matrix (all those extra zeros!). Use the \texttt{diag} function to convert the matrix of eigenvalues into a \( [3 \times 1] \) vector of eigenvalues.

4. If our matrix \( \mathbf{A} \) represents a system of linearly dependent equations, then we will only get as many independent and nonzero eigenvalues and eigenvectors as linearly independent equations. Try finding the eigenvalues and eigenvectors of

\[
\mathbf{A} = \begin{bmatrix}
-5 & 0 & 0 \\
3 & -4 & 1 \\
2 & 4 & -1
\end{bmatrix}
\]

How does this property relate to the first Optional Exercise?

Once we have found the eigenvectors and eigenvalues, we have all that we need to develop a general homogeneous solution:

\[
\hat{y}_h = c_1 \mathbf{x}_1 e^{\lambda_1 t} + c_2 \mathbf{x}_2 e^{\lambda_2 t}
\]

So now it’s time to find our particular solution, \( \hat{y}_p \). Noting that we have \( \mathbf{g} = \begin{bmatrix} rD_0 \\ 0 \end{bmatrix} e^{-rt} \), we can use the Method of Undetermined Coefficients and guess a form for \( \hat{y}_p \):

\[
\hat{y}_p = \bar{u} e^{-rt}
\]

Where \( \bar{u} \) is an unknown vector of coefficients \( \bar{u} = [u_1, u_2]^T \). Plugging \( \hat{y}_p \) into our equation \( \hat{y}' = A\hat{y} + \hat{g} \) produces:

\[
\hat{y}'_p = A\hat{y}_p + \hat{g} \\
-r\bar{u} e^{-rt} = A\bar{u} e^{-rt} + \hat{g}
\]

---

\(^1\) https://www.mathworks.com/help/matlab/ref/eig.htmls
\[-r \ddot{u} = A \ddot{u} + \dddot{g} \]
\[(A + rl) \dddot{u} = -\dddot{g} \]

So now we see that our coefficient vector, \( \ddot{u} \), is the solution to the equation \((A + rl) \dddot{u} = -\dddot{g} \). Recall that we can use MATLAB’s \texttt{mldivide} \(^3\) (backslash) function to easily solve for \( \ddot{u} \) (since we know \( A \), \( r \), and \( \dddot{g} \)).

**Activity:**

1. The identity matrix \( I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \) has ones on the diagonals and zeros on all off-diagonal terms. In MATLAB, we can define the identity matrix with the \texttt{eye(n)} \(^4\) function, where ‘n’ defines the dimensions of the square matrix (e.g. \( n = 3 \) creates a \( 3 \times 3 \) matrix). Use \texttt{eye} to define a \( 3 \times 3 \) identity matrix.

2. The \texttt{mldivide} (backslash) function is great for solving matrix equations of the form \( Ax = b \). To find the vector \( \ddot{x} \), we use the syntax:
   \[
   \gg \; \; x = A \backslash b 
   \]
   Where \( A \) is a \( m \times n \) matrix and \( b \) is a \( m \times p \) matrix (meaning \( x \) must be \( n \times p \). Use the backslash to solve the system:
   \[
   \begin{bmatrix}
   -5 \\
   3 \\
   2
   \end{bmatrix} 
   \begin{bmatrix}
   0 \\
   -4 \\
   4
   \end{bmatrix} 
   \begin{bmatrix}
   0 \\
   -1 \\
   2
   \end{bmatrix} = \begin{bmatrix}
   -5 \\
   -1 \\
   6
   \end{bmatrix}
   \]

Once we have solved the unknown coefficients \( \ddot{u} \), we have a general solution:

\[
\ddot{y} = c_1 \ddot{x}_1 e^{\lambda_1 t} + c_2 \ddot{x}_2 e^{\lambda_2 t} + \dddot{u} e^{-rt}
\]

And can use our initial conditions to solve for \( c_1 \) and \( c_2 \). The process is identical to solving a for the equations in a homogeneous ODE, except we have the addition of the \( \ddot{u} \) term, which produces:

\[
\begin{align*}
\ddot{y}_0 &= c_1 \ddot{x}_1 + c_2 \ddot{x}_2 + \dddot{u} \\
M \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} &= (\ddot{y}_0 - \dddot{u})
\end{align*}
\]

Which we (again) can easily solve for \([c_1, c_2]^T\) using the \texttt{mldivide} function. And that’s it. All that is left to do is define a time vector and use it to define our equation for \( \ddot{y} \).

**Activity**

Use \texttt{linspace} \(^5\) to define a vector for time, \( t \), where \( 0 \leq t \leq 60 \) containing 100 points. Now define a new vector \( \ddot{x} = [1, 2]^T \). Using \( t \) and \( \ddot{x} \), define a set of two equations, \( \ddot{y} = \begin{bmatrix} \ddot{y}_1 \\ \ddot{y}_2 \end{bmatrix} = 2 \ddot{x} \cos \frac{\pi}{10} t \). Plot your solutions as a function of time.

---

\(^3\) https://www.mathworks.com/help/matlab/ref/mldivide.html?searchHighlight=mldivide&s_tid=doc_srchtitle

\(^4\) https://www.mathworks.com/help/matlab/ref/eye.html?searchHighlight=eye&s_tid=doc_srchtitle

\(^5\) https://www.mathworks.com/help/matlab/ref/linspace.html?searchHighlight=linspace&s_tid=doc_srchtitle

M. Leineweber

San Jose State University
In the previous section, we saw how we can use MATLAB to solve a nonhomogeneous system of linear ODEs by converting it into a matrix equation. However, this approach, requires a fair amount of “pencil-and-paper” work to properly set up our matrices. To avoid this extra work, we can take advantage of MATLAB's symbolic toolbox to directly create an analytic solution.

Note: The basic student MATLAB package does not include a symbolic solver. As an alternative, try using https://octave-online.net/, which can run and process MATLAB syntax. To be able to create and run entire scripts offline, you will need to create an account with Octave. There are a few small differences in syntax between MATLAB and Octave that I will try and identify when applicable. Note that Octave’s symbolic solver is more limited in its scope that MATLAB’s.

Let’s take a look at our same nonhomogeneous system of ODEs. Rather than defining our rate constants immediately, we can use the syms command to identify specific variables and functions as symbolic.

\[
\begin{align*}
\text{>> syms } & \quad a \quad b \quad c \quad d \\
\text{Defines variables} & \quad a, b, c, \text{ and } d \text{ as symbolic in MATLAB, meaning it will treat these letters as variables when processing. Similarly syms can be used to define functions, such as } \\
\text{>> syms } & \quad F(a) \\
\text{That creates a function variable, } F, \text{ that operates on the symbol } a.
\end{align*}
\]

Activity

Create symbolic variables \( x \) and \( y \). Define a function \( f(x,y) = x^2 + y \) using the syms command.

Once we have created a set of symbolic variables and functions, we can use the diff command to calculate symbolic derivatives. For example

\[
\begin{align*}
\text{>> d2F } & \quad = \quad \text{diff}(F,a,2) \\
\text{Assigns the symbolic second derivative of } F, \text{ with respect to symbolic variable } a, \text{ to the output symbolic function } d2F. \text{ The first argument (i.e. input) of the diff function is the symbolic function to be differentiated. The second argument specifies the variable with which the function should be differentiated, and the third argument specifies the order of derivative. Later, we will see how the diff command can also be used to find the difference between two sequential entries in a vector, which is very useful when computing numerical derivatives!}
\end{align*}
\]

Activity

---

Use the `diff` function to create a new function \( dF \) that computes the first derivative of your function \( F \) (from the previous activity) with respect to \( x \). What expression does \( dF \) represent?

Within the symbolic toolbox, we can also define entire equations and assign them to single variables. For example, consider the code:

```
>> odeF = dF == a*F
```

The first “single equals” sign (=) means, in MATLAB speak, “is assigned to”, and refers to the expression on the right of the single equals sign being assigned to the variable on the left. In this case, the entire expression ‘\( dF = aF \)’ is assigned to the variable `odeF`. When including an equality within an expression, we have to use the “double equals” (==) sign.

We can also assign initial conditions using the symbolic toolbox. Remember that we defined \( F(\alpha) \) and \( d2F(\alpha) \) as functions. Therefore they can receive inputs. To define an initial condition for \( F \), we use the syntax:

```
>> cond1 = F(0) == 0
```

Note that once again we use the (==) within our expression for the initial condition \( F(0) = 0 \), and the (==) to assign that expression to the variable `cond1`.

Activity

Define a new function \( g(x) \), and its derivative \( \frac{dg}{dt} \). Define the differential equation \( \frac{dg}{dt} = -2g(x) + x \) and the initial condition \( g(0) = 10 \).

So now that we’ve defined our symbolic ODE, it’s time to solve using MATLAB’s `dsolve` command. `dsolve` takes either a single ODE or a vector of ODEs, as well as their corresponding initial conditions as inputs, and provides the symbolic solution.

```
>> F_Sol = dsolve(ode,[cond1,cond2])
```

The above expression uses `dsolve` to solve the ODE defined by the symbolic equation `ode`, using the initial conditions `cond1` and `cond2` concatenated together into a single row vector. The result is output to the variable `F_Sol` as a symbolic equation containing the expression for the solution to the ODE.

**Note:** Octave does not like its initial conditions contained in a single row vector. Instead, separate each initial condition by a comma:

```
>> F_Sol = dsolve(ode,cond1,cond2)
```

Activity

Use `dsolve` to find the solution to the ODE established in the previous activity.

---

7 https://www.mathworks.com/help/symbolic/dsolve.html?searchHighlight=dsolve&s_tid=doc_srchtitle
Define new ODE: $my'' + by' + ky = \sin(2t)$, $y(0) = 0, y'(0) = 0$. Use `dsolve` to find $y(t)$. Store the solution $y(t)$ to the variable `Y_Sol`.

Once we have a symbolic expression for our solution, we still want to be able to evaluate it at a specific point or range of points. Creating a numeric output requires that we replace our symbolic variables with their numeric equivalents. In other words, we need to redefine our symbolic constants and independent variables with scalars and vectors. Note that changing these variables does not cause our original symbolic solution to change, unless we specifically overwrite it. To see this in action, complete the activity below.

**Activity**

Let's give some values to the symbolic variables defined in the previous example.

```matlab
>> m = 2;
>> b = 1;
>> k = 2;
>> t = linspace(0,10,100);
```

Now let's take a look at our expression for `Y_Sol`.

```matlab
>> disp(Y_Sol);
```

We see that it is still a symbolic expression containing the variables $m, b, k$. Since we have not explicitly assigned a new expression to the variable `Y_Sol`, it retains the value it had before we assigned numbers to $m, b, k$.

Now that we have assigned numbers to all of our symbolic variables, we want to evaluate the expression contained within `Y_Sol` using our newly assigned values. For this task, we use the `eval` function, which executes the MATLAB commands contained within a string (i.e. text expression) used as an input to the function. For example,

```matlab
>> stringCommand = 'A = magic(3)';
>> eval('stringCommand')
```

In this case, the `eval` function executes the MATLAB expression `A=magic(3)`, which creates a $3 \times 3$ “magic” square where the sum of every row and every column is identical. Note that a new variable $A$ has been created in the workspace.

**Activity**

Now that we have created a time vector $t$, and defined our coefficients $m, b, k$, we can evaluate our expression `Y_Sol` for those values by using the expression:

```matlab
>> Y_Num = eval(Y_Sol);
```

The `eval` function created a new vector `Y_Num` that contains the values of `Y_Sol` evaluated at every point in the vector $t$. Note that the symbolic expression `Y_Sol` still remains unchanged. Let's plot the solution as a function of time:

```matlab
>> plot(t,Y_Num);
```

---

8 https://www.mathworks.com/help/matlab/ref/eval.html

*M. Leineweber*

*San Jose State University*
Now let’s see what happens when we update our coefficients again. This time let’s set $m = 2$, $b = 4$, and $k = 1$. Re-evaluate $Y_{Sol}$ with these coefficients and assign them to the variable $Y_{Num2}$. Let’s plot the new solution on top of its predecessor:

```matlab
>> hold on;
>> plot(t,Y_Num2);
```

We see that we get a new solution that produces different behavior, but the original symbolic expression $Y_{Sol}$ still remains unchanged.

**Modeling Discontinuous Input Functions**

Taking multiple doses of a drug can also be modeled as a nonhomogeneous input to the system. However, rather than having the amount of drug available vary continuously, we have discrete times where the amount of drug available changes abruptly. We might recognize from lecture that discontinuous input functions can be modeled using the Unit Step (Heaviside) and Dirac Delta (Impulse) functions, which are denoted as $u(t - a)$ and $\delta(t - a)$, respectively.

MATLAB has its own functions for representing the Heaviside and Dirac Delta functions that can be used with both numeric and symbolic input:

- $y = \text{heaviside}(x)$ returns either $y = 1$ or $y = 0$ depending whether its input $x$ is greater than zero ($x > 0$) or less than zero ($x < 0$), respectively. At $x = 0$, MATLAB defines $y = \frac{1}{2}$, but has options to change this definition if desired. Just like the analytical Unit Step function, MATLAB’s `heaviside` function can be shifted by a distance $a$ by using the subtracting a scalar value $a$ from the input. Similarly, the output can be multiplied by both constants and functions by using the ‘.*’ element-by-element multiplication. Refer to the documentation to see more details on how $y = \text{heaviside}^9(x)$ is used.

- $y = \text{dirac}(x)$ Unlike the `heaviside` function, `dirac(x)^{10}$ only works with symbolic input. It can be used with the symbolic toolbox and `dsolve` to represent the Dirac Delta function in solving ODE responses to impulse input.

**Activity**

1. Define a variable $t$ as a vector from zero to 10, and a variable, $y_1$, containing the function $y_1 = \sin(\pi t)$ evaluated at each time point in $t$. Now use the `heaviside` function to define a new variable, $y_2$, that contains the function:

   $$
   y_2(t) = \begin{cases} 
   0 & t < 2 \\ 
   5 \sin(\pi (t - 2)) & t \geq 2 
   \end{cases}
   $$

2. Use the symbolic toolbox to define the shifted unit step function: $u(x - 5)$. Compute the derivative of $u(x - 5)$ using the `diff` function. What is MATLAB’s output? Is it what you would expect?

3. Using your functions from (1), now create a new variable, $y_3$, that is defined:

   $$
   y_3(t) = \begin{cases} 
   0 & t < 2 \\ 
   5 \sin(\pi t - 2) & 2 \leq t \leq 8 \\ 
   0 & t > 8 
   \end{cases}
   $$

Plot $y_1$, $y_2$, and $y_3$ vs. $t$ all on the same graph.

---

9 https://www.mathworks.com/help/symbolic/heaviside.html?searchHighlight=heaviside&s_tid=doc_srchtitle
10 https://www.mathworks.com/help/symbolic/dirac.html

M. Leineweber  
San Jose State University
**END OF SECTION ASSIGNMENT**

1. Write a script to use the systems (i.e. eigenvalue/eigenvector) approach to solve the system of nonhomogeneous ODEs describing the time-released drug delivery system where the amount of drug released by the capsule at time \( t \) is given by the function \( f(t) = D_0(1 - e^{-rt}) \). \( D_0 = 50 \) mg. Assume that there is no drug present in the patient prior to taking the first dose. Plot the amount of drug at each stage \((D(t), B(t), W(t))\) over time.

2. Write a script to solve the same problem, this time using the symbolic solver. Plot your results and compare to the plot from Question 1 above.

3. Now consider a patient who takes multiple doses of the same capsule. Doses are taken at times \( t = 0, 120, \) and \( t = 240 \) minutes. Update your symbolic solver script to find \( D(t), B(t) \) and \( W(t) \) as functions of time for this new input. Plot your results as functions of time.

---

**Numerical Methods for Solving ODEs in MATLAB – Runge-Kutta Algorithms**

As previously discussed, many nonlinear ODEs cannot be solved analytically, but we can use numerical analysis techniques to get very good approximations. MATLAB has several functions, called solvers, that can be used to approximate the behavior of ODEs using, among others, the Runge-Kutta method. The most common of these algorithms are \texttt{ode45}, \texttt{ode23}, and \texttt{ode23t}. The most widely applicable of these is \texttt{ode45}.

**Anonymous Functions**

Before we delve into using the ODE solvers, it may be useful to do a brief recap of functions and anonymous functions. MATLAB defines functions as:

> Functions are files that can accept input arguments and return output arguments. The names of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

All of the MATLAB commands that we have used in this lab (\texttt{eval}, \texttt{diff}, \texttt{dsolve}, etc.) have been functions because they operate on some input (enclosed in () parentheses), and return an output. While, as stated above, we generally define functions with their own dedicated m-file, it is sometimes convenient to define functions only in the workspace. These functions, called Anonymous Functions\(^{11}\), usually only contain a single line, and are designated with the ‘@’ symbol. For example:

\[
\begin{align*}
\text{>> fun} &= @(x,y) \text{sqrt}(x.^2 + y.^2); \\
\text{>> z} &= \text{fun}(3,4)
\end{align*}
\]

Creates a function called \( \text{fun} \) that receives two variables as input: \((x, y)\), and then returns the square root of the sum of squares of each element. We know that it is an anonymous function because of the

\(^{11}\) https://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html

---

\( M. \) Leineweber  
San Jose State University
\( \theta (x, y) \). The next line calls the function \( \text{fun} \) the same as any other function, and then the variable \( z \) is assigned the value of the output when the input is \((3,4)\): 
\[ z = \sqrt{3^2 + 4^2} = 5. \]

Anonymous functions can also take and return vectors as input and output. Consider the example below:

\[
\begin{align*}
>> \text{fun2} &= @(x) [\cos(x); \sin(x)]; \\
>> t &= \text{linspace}(0, 2*\pi, 1000); \\
>> y &= \text{fun2}(t);
\end{align*}
\]

The input to the function is a vector \( t \) with 1000 elements. Looking at the definition of \( \text{fun2} \), we see that it computes \( \cos(t) \) and \( \sin(t) \) and then combines them each as row in a \( 2 \times n \) vector. Since \( n = 1000 \) in the problem, the resulting output \( y \) is a \( 2 \times 1000 \) matrix. Take this opportunity to plot \( t \) vs. \( y \). What do you see? How many plots are there? What will we get if we plot the first row of \( y \) vs. the second row of \( y \)?

**Activity**

(1) Use anonymous functions to compute the following expressions for \( 0 \leq x \leq 10 \):

\[
\begin{align*}
y &= \sin^2 x - 3 \\
y &= e^{-2.5x} - e^{-1x} \\
y &= \begin{bmatrix} x^2 - 2x + 1 \\ 2x - 2 \end{bmatrix}
\end{align*}
\]

Plot \( x \) vs. \( y \) for each case in its own plot in the same figure window.

**Representing ODEs in Canonical Form**

Recall from Lecture 4 that we can represent any higher order ODE as an \( n^{th} \) order system of first-order ODEs, where \( n \) is the order of the original ODE:

\[
y^{(n)} = F(t, y, y', y'', ..., y^{(n-1)}) \rightarrow \begin{bmatrix} y_1' = y_2 \\ y_2' = y_3 \\ \vdots \\ y_{n-1}' = y_n \end{bmatrix}
\]

For example, consider the ODE:

\[ y'' + ay' + by = 0 \]

By defining a new variable \( y_1 = y \), we can then introduce a second new variable:

\[ y_2 = y_1' = y' \]

And a third

\[ y_3 = y_2' = y'' \]

Noting that our original ODE can be rearranged to give an expression for \( y'' \), we can set up a system of equations:

\[
\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{pmatrix} 0 & 1 \\ -b & a \end{pmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
\]

Which we recognize is our standard form for a system of first-order ODEs: \( \ddot{y} = Ay \). Using our definitions for \( y_1' = y_2 \) and \( y_2' = y'' = -by_1 - ay_2 \) we can complete our equation:

\[
\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{pmatrix} 0 & 1 \\ -b & a \end{pmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
\]
For a linear system of equations like this one, we can use our “systems approach” from the last section to solve for y₁ and y₂ (corresponding to y and y' in the original ODE). But what if the original ODE was nonlinear?

\[ y'' + 2yy' + 4y = 0 \]

Our system would now become:

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= -2y_1 - 4y_1y_2
\end{align*}
\]

Since we do not have the tools to solve this equation analytically, we will have to try a numerical approach.

**Numerical Solvers in MATLAB**

MATLAB’s most commonly used solver, `ode45`, employs a 4th-order Runge-Kutta algorithm to solve a single or system of first-order ODEs. `ode45` is a versatile solver and will be used for most applications in this course. Similar to `ode45` is the second-order solver `ode23`, which uses a faster, but not as accurate algorithm. Finally “stiff” solvers, such as `ode23t` should be used to solve unstable, or “stiff” systems where even very small changes in the independent variable (i.e. x or t) can produce very large changes in the ODE output. Typically, `ode45` offers the best balance between accuracy and computational time required to solve.

Let’s take a look at how we can use `ode45`.

```matlab
>> [t,y] = ode45(odefun,tspan,y0);
```

As we can see, `ode45` takes three inputs:

1. `odefun` – is the MATLAB function (or anonymous function) containing our first-order ODE or system of first-order ODEs to be solved.
2. `tspan` – identifies the range of values over which `ode45` should approximate a solution. The format for the `tspan` input should be \([t_0, t_f]\), where \(t_0\) is the initial time and \(t_f\) is the final time.

**NOTE:** if the solution to the ODE contains a vertical asymptote or other discontinuity over the selected range, the Runge-Kutta algorithm may not converge on a solution.

3. `y0` – specifies the vector of initial conditions containing the initial condition of each ODE identified in `odefun`.

Let’s consider our last nonlinear ODE

\[ y'' + 2yy' + 4y = 0 \]

Which we represented as:

\[
\begin{align*}
y_1' &= y_2 \\
y_2' &= -2y_1 - 4y_1y_2
\end{align*}
\]

To solve with MATLAB we first need to define a function to act as our “odefun” input. Let’s use an anonymous function:

```matlab
>> fun = @(t,y) [y(2);-2*y(1) - 4*y(1)*y(2)];
```

---

12 https://www.mathworks.com/help/matlab/ref/ode45.html
We see that our anonymous function takes the (unused) variable \( t \) and vector \( \vec{y} \) as input, where \( \vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \), and then returns the vector of first-derivatives \( \begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ -4y_1 - 2y_1y_2 \end{bmatrix} \) as output. Note that even though the vector \( t \) is unused in our ODE, since we are solving for \( y(t) \) we need to include it in our function.

Once we have have specified our odefun, we can define our initial conditions and solution interval. Let’s try \( y_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \) from \( 0 \leq t \leq 20 \).

\[
>> [t,Y] = \text{ode}45(\text{fun},[0,10],[1;0]);
\]

The resulting variable \( Y \) contains two rows of data. We should expect this because we know that we are looking for the solution \( \vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} \). So using \( \text{ode}45 \) to solve our original ODE produces solutions for both \( y \) and \( y' \). For an ODE with order \( n \), \( \text{ode}45 \) will produce \( n \) solutions corresponding to each \( (n - 1)^{th} \) derivative.

The output \( t \) contains the time-points that the function at which the solutions were evaluated. If you only supply \( \text{ode}45 \) with the start and end points, it will automatically select the increments of \( t \). If you would prefer to evaluate the ODE at specific intervals, you can use a full vector (e.g. \( 0:0.01:t_{\text{max}} \)) as input to the \( \text{ode}45 \). For example:

\[
>> [t,Y] = \text{ode}45(\text{fun},\text{linspace}(0,20,1000),[1;0]);
\]

Will return a vector \( t \) that contains the values in \( \text{linspace}(0,20,1000) \), and \( Y \) will contain the solutions evaluated at those points.

Activity

Try plotting this solution \( t \) vs. \( Y \), and then finding solutions for initial conditions: \( y_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \) and \( y_0 = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \). Plot all solutions on the same plot.

Activity

Rewrite the following ODE as a system of first-order ODEs:

\[ y^{(3)} + 2y'' + 3y' + y^2 = 4 \sin \pi t \]

Write a function called “myODE” that takes variables \( t \) and \( y \) as an input and returns the vector of derivatives as an output.

Use \( \text{ode}45 \) to solve the ODE over the interval \( 0 \leq t \leq 10 \) using the initial conditions: \( y(0) = 0, y'(0) = 2, y''(0) = 0 \). Evaluate the function at evenly-spaced \( t \), with \( \Delta t = 0.005 \).

Plot your solution, and use the legend command to label each line with its proper name (e.g. \( y'(t) \)).
END OF LAB ASSIGNMENT

1. Write a MATLAB script to solve our nonhomogeneous drug-delivery problem using `ode45`. Solve for the Timed-Release Capsule input function: \( f(t) = D_0(1 - e^{-rt}) \)

   Plot the results. How well does the numerical solution compare to the analytical solutions?

2. MATLABs (and Octave's) `dirac(x)` function does not work with numerical input. As a result, attempting to solve a multiple-dose problem with the systems approach or `ode45` will result in NaN output. Is there a way to use these approaches to implement a multiple-dose input? (Hint: can you use multiple sets of “initial” conditions?)

   Write a MATLAB script using `ode45` to simulate three doses of a 50 mg time-release capsule, taken 120 minutes apart.