Mathematica 101

D D Smaranda

Abstract. The following tutorial is meant as an introduction to Mathematica, symbolic computation and computer algebra systems.

0. Some Housekeeping

So let's do some housekeeping. First off the **prerequisites**. Mathematica (as the name very creatively suggests) is a programming language that specialises in symbolic programming that is used widely in mathematical applications. That being said this course assumes that you're familiar with the following concepts from your courses: *linear systems of equations, multi-variable functions , non-linear equations, simple and partial derivatives, ordinary differential equations, matrices, vectors, matrix algebra, Taylor series and power expansions.*

As for the programming side of things this course assumes that you are already familiar with the fundamentals and basics of a programming language. From this point of view this will focus on the inherent particularities of Mathematica.

This being your first intro to Mathematica , we'll introduce a couple of concepts that are specific to the Wolfram style of programming.

- **Mathematica Notebook**. Mathematica notebooks are the programming environment in the Wolfram language. It is a interactive style of environment which is divided into cells (marked by a] at the right hand side of the screen). A notebook can have multiple of these cells which can be used to write and run code on an individual basis. To those of you which might have some experience with programming it is similar to the Python Jupyter notebooks.
- Through this tutorial whenever I'll write a bit of code it will have the `>>` delimiter at the start of the line (don't put these in when you're typing in code!). The output of the code is going to be marked by `Out =`, e.g:

```
>> 1 + 2
Out = 3
```

- To **run code** after you've typed it in, you just need to hit `Shift + Enter`, and the output is sent to a new cell underneath the current one. If you want to **suppress the output**, you just have to add a `;` at the end of the code that you're running.
- Mathematica has a couple of different brackets that have different functions.
  - `[]` is used to call a function with some arguments inside the brackets.
  - `[[ ]]` is used to access a single or multiple elements of a list.
  - `{}` is used to define a list.

  - `{{ }}` is used to define a matrix (or an array).
  - `()` is used to define a specific mathematical ordering.
- Every now and then Mathematica decides it doesn't want to work, you might get some weird error out of the blue even though the syntax seems fine. This is usually to do with the interactive notebook style of programming, and what's most likely happening is that you've got the same variable name being used twice in different cells with different scopes and Mathematica is trying to sort it out. To fix that we're going to use the Mathematica equivalent of turning it on and off, i.e. quitting and starting the kernel, which is done by navigating to *Evaluation* → *Quit Kernel* → *Local*, followed by *Evaluation* → *Start Kernel* → *Local*.
- As is usual with most programming languages `=` is used for variable assignment and `==` is used as a truth equator, with the additional detail that Mathematica also uses it to define equations. For example the following code:

  ```
  myExpr1 = a^2 + b^2 - c^2 ==0
  ```

  stores the mathematical expression $a^2 + b^2 - c^2 = 0$ into the `myExpr1` data structure. In a sense the `==` operator is just a bit more general in Mathematica.
- Mathematica uses a host of special characters from Greek letters to all sorts of mathematical notations. To produce a special character the sequence generally goes as `Esc + (character code) + Esc`. In general when you start typing the character code a drop down list will appear with all the available options for that specific character code start. E.g. type `Esc + a` and see what the list looks like.

  Furthermore Mathematica has an inbuilt formatting for fractions, subscripts and superscripts, which are accessed via `Ctrl + /` and `Ctrl + _` and `Ctrl + ^`. Any syntax produced by it will have the same meaning as it normally would, and one can call variables with subscripts and superscripts just like any other variable.

  Note! Sometimes the special character and the special formatting serve a functional purpose as well. E.g. if we typed in Mathematica ∂_x (achieved by typing `Esc + pd + Esc` followed by `Ctrl + _ + x`) it acts as a partial derivative with respect to x to whatever follows it.
- As per usual with any programming language if you're stuck and there's no one around to help you, google and the documentation are your best friends!

1. Mathematica 101

We'll start of with a couple of simple commands in Mathematica so you get familiarised with it's syntax and behaviour. First off we'll assign two variables `a, b` values `1, 2`, and add them together and store the result in a variable `c`:

```
>> a = 1;
>> b = 2;
>> c = a + b;
```

Note that I've suppressed the output with the `;` operator, and nothing will be outputted when I execute the code. To output the value of a variable we can use the `Print[]` function:

```
>> Print[c]
Out:: 3
```

We'll now square `c` (the power operator is `^`) and multiply it with `a` and store it in a variable `d`. Note that we don't suppress the value of `d`.

```
>> d = (c^2) * a
Out:: 9
```

We'll now define a list `testList = {4, 5, 6}`, to which we'll append `d` by using the `AppendTo[]` function, and print out the new values in `testList`:

```
>> testList = {4, 5, 6};
>> AppendTo[testList, d];
>> Print[testList]
Out:: {4, 5, 6, 9}
```

Now we'll go through the elements in `testList` and print the values on a separate line. To do that we'll use the `For[]` function to loop over the elements in the list and the `Length[]` function to tell Mathematica in what range of indexes to look at:

```
>> For[listIndex = 1, listIndex <= Length[testList], listIndex = listIndex + 1,
        Print[ testList[[listIndex]] ]
      ];

Out:: 4
      5
      6
      9
```

Note! Mathematica indexes list values from 1, and `For[]` loop works as in most other programming languages with the `,`'s specifying where the different bits should be (as in variable to loop over, bounds and increment).

We'll now modify our code to only print out the even values. For that we're going to use the `If[]` function:

```
>> For[listIndex = 1, listIndex <= Length[testList], listIndex = listIndex + 1,
        If[ Mod[ testList[[listIndex]] , 2] == 0,
         Print[testList[[listIndex]]]
         ]
      ];

Out:: 4
      6
```

Note! The tab indenting doesn't matter in Mathematica like it does in other programming languages such as Python. Nonetheless I like to keep the indenting neat since it helps a lot with debugging your code and forming a mental picture of what's going on or going wrong.

Don't worry if the syntax looks a bit wonky, Mathematica has a bit of a learning curve and does things in a slightly unorthodox manner. If you need to look at the documentation of a function, you can either navigate to the *Help* → *Wolfram Documentation* (`F1` as a short cut), and then search the function. Alternatively you can just put a `?` in front of the function name in the notebook.

Look at how the following behave:

```
>> ?Sqrt
>> ?Table
>> ?ListPlot
```

Now that we're a bit used to the Mathematica syntax, we'll move on to the algebraic language side. To get the most of this we'll need to define a function `myF[]` which for our example will be a function of `x`. To this extent let's define the following using the `:=` operator:

```
>> myF[x_] := C1 x^2 + Sin[x] + 2 x* Tan[3 x] + C2
```

The underscore in `x_`, specifies that `x` is the variable of `myF`. Since the other variables `C1, C2` aren't specified in the definition, they are treated as function constants. Note that in Mathematica you don't necessarily need to specify the multiplication by a number or a scalar. If the code is of the form `3 x` it is plainly interpreted as $3 \times x$.

Note! You need a space between the multiplication elements or else Mathematica interprets it as a variable. E.g. `a b` is interpreted as the multiplication of two variables $a \times b$, where `ab` is treated as a single variable named `ab`.

We can now call it like we would in any other programming language as:

```
>> myF[x]
Out:: C2 + C1 x^2 + Sin[x] + 2 x Tan[3 x]
```

and Mathematica produces the expression with `x` as the variable.

While we're looking at defining functions let's look at `Module[]` and the `Return[]` functions. In Mathematica, if you want to define a procedural function, the best way is to use the `Module[]` function, and specify what you want it to return via `Return[]`. For example, say that we want to write a function that produces the following function $V(x)$:

$$V(x) = f_N\left((x/a)^2\right) + f_N\left((x/b)^2\right) + 3\sin\left(\frac{x^2}{ab}\right) \qquad \text{where} \quad f_N(x) = \frac{x^2 - 4x + 3 + \log(x)}{(1-x)^3}$$

This is done by the following implementation:

```
>> myFModule[x_] := Module[{fN},
>>   fN[z_] := (z^2 - 4 z + 3 + 2 Log[z])/(1 - z)^3;
>>   V = fN[(x/a)^2] + fN [(x/b)^2] + 3 Sin[x^2/(a b)];
>>   Return[V]
>>   ]
```

To explain what we're doing, `Module[]` takes two arguments, the `{fPlus}`, is there to tell Mathematica that `fPlus` is an intermediary function to be declared inside the module, and the second argument is the actual definitions. Inside the second argument we are now free to define and write as many things as we please. To make Mathematica export the expression for V, we simply use `Return[V]`.

Note! The variables inside the local functions must have different names from the arguments of the parent function (i.e. the `z_` argument in `fN`, versus the argument `x_` in `myFModule`).

We'll now look at the replacement operator `/.{}`. The replacement operator can be used at the end of any sort of Mathematica expression that contains variables. It has the effect of replacing the quoted variable by the specified rule. For example:

```
>> myF[x] /. {x -> y, C2 -> 0, C1 -> z}
Out:: y^2 z + Sin[y] + 2 y Tan[3 y]
```

After the `/.` operator the `{}` specifies the replacement, rules. I.e. in the above we've replaced `x` with `y`, set the constant `C2` to `0` and replaced the constant `C1` with `z`.

We're not in any way limited by replacing variables with other variables, we can use even full expressions. For example we can call `myF[y]` and replace `y` with `Sin[z]`:

```
>> myF[y] /. {y -> Sin[z]}
Out:: C2 + C1 Sin[z]^2 + Sin[Sin[z]] + 2 Sin[z] Tan[3 Sin[z]]
```

Similarly we can replace the variables with numbers so we can actually evaluate the functions that we define at points of interest, e.g. :

```
>> myF[x] /. {x -> 2}
Out:: 4 C1 + C2 + Sin[2] + 4 Tan[6]
```

So we've replaced `x` with 2, but **Mathematica tends to be as general as possible until you tell it not to be**. In this case we need to **numerically evaluate** the resultant expression, which is done by using the `N[]` function:

```
>> N[myF[x] /. {x -> 2}]
Out:: -0.254727 + 4. C1 + C2
```

Again note that Mathematica will now numerically evaluate whatever it can and leave the rest untouched. In our case since we haven't specified anything for `C1, C2`, it just leaves them as they are.

We're now equipped to deal with a couple of the features of Mathematica, and in the next tutorial we'll start to see why Mathematica can be a very powerful tool. In preparation for that we'll introduce the differentiation function `D[]`. We'll want to find the 2nd derivative of `myF[x]` w.r.t. `x`, i.e. $\partial^2(\texttt{myF}(x))/\partial x^2$:

```
>> D[myF[x], {x, 2}]
Out:: 2 C1 + 12 Sec[3 x]^2 - Sin[x] + 36 x Sec[3 x]^2 Tan[3 x]
```

Note that so far we've only used functions with one variable. Extending it to multiple is trivial , since we need only specify them. I.e. lets define a new function `myF2[x_, y_]` as:

```
>> myF2[x_, y_] := x^2 + y^2 + C3 + C4 x^2 * y^2
```

And lets find $\partial^2(\texttt{myF2}(x,y))/\partial x \partial y$. To that extent:

```
>> D[myF2[x, y], {x, 1}, {y, 1}]
Out:: 4 C4 x y
```

You can check this by doing the maths by hand and comparing them to the output to see that indeed it is the expected result!

In the next tutorial we'll move on to slightly more complicated problems and we'll begin to see where Mathematica's strengths are at.

For each tutorial there'll be a set of exercises to familiarize yourself with the concepts presented. Try and get through as many as possible before the time runs out, but don't worry if you don't get through all of them. Furthermore I'll put in one or two more challenging exercises which will be marked by a dagger $^{(\dagger)}$.

Exercises

(i) Find the first derivative w.r.t. x of the following function $f(x)$:

$$f(x) = 1 + x^2 - ax^3 + bx^4 + cx^4$$

(ii) Evaluate the function $f(x)$ at $x = 0, x = 1, x = \pi$

(iii) $^{(\dagger)}$ Find the points x_i for which $f(x)$ is either a minimum or a maximum.

Hint! Remember that the minimum/maximum points of a function are given by the solutions of $\partial f(x)/\partial x = 0$. To find this you'll need to use the `Solve[]` function. Use Mathematica's documentation to see how the syntax looks like.

(iv) Let $f(x, y)$ be the following function of the variables x, y:

$$f(x,y) = a\sin(x) + b\cos(y) - \sin(xy)\exp(2\pi i(x+y)) + Cx^2$$

Find the following: $\dfrac{\partial f}{\partial x}, \dfrac{\partial f}{\partial y}, \dfrac{\partial^2 f}{\partial x \partial y}, \dfrac{\partial^2 f}{\partial y \partial x}, \dfrac{\partial^2 f}{\partial x^2}, \dfrac{\partial^2 f}{\partial y^2}$. Does this agree with your expectations?

(v) Evaluate the above at the pair of points $\{x = 0, y = 0\}, \{x = \pi, y = \pi\}, \{x = 1.5, y = 1.5\}$, where we assign the constant values $a = 2, b = 3, C = 1$.

(vi) (†) Create a list of the following functions:

$$\{\sin(x), \cos(x), \exp(2\pi i x), \ln(1+x), x^3\}$$

and write a script that randomly evaluates one of the list functions at either $x \in \{0, \pi/2, \pi\}$. *Hint!* Look at the `RandomInteger[]` function, and think about lists and list indices.

(vii) (†) Create a function that takes as an argument a number n. The function should then proceed to verify if n is an integer, and if it isn't it should display the message `'n must be an integer'`. If n is an integer the function should check if n is even or odd. If:

- n is even, then the function should return the value of $\sin(2\pi n) + F(n)$
- n is odd, then the function should return the value of $\cos(2\pi n) - F(n)$,

where $F(n) = \sum_{i=1}^{n} i^2$.

2. Mathematica 102

In this tutorial we'll look at some of Mathmatica's more specialized functions and get a bit of insight into the strengths of Mathematica.

We'll start off by looking at the `Solve[]` function. As per usual if you find that you need a bit more insight into how the function works just use the Mathematica `Help` section or just type in the notebook `?Solve`.

To that extent `Solve[]` is used to solve a system of equations, specified for the user. It takes an equation/s and the variable/s as arguments. Let's start of with a simple example. Suppose we want to solve for x the quadratic equation:

$$x^2 + ax - 1 = 0$$

This is quite simple in the fact that the code is just :

```
>> Solve[ x^2 + a x - 1 ==0, x]
```

Out:: $\left\{\left\{x \to \frac{1}{2}\left(-\sqrt{a^2+4} - a\right)\right\}, \left\{x \to \frac{1}{2}\left(\sqrt{a^2+4} - a\right)\right\}\right\}$

Let's move on to a slightly more 'complicated' example. To that extent we'll try and solve the following system of equations for x, y:

$$\begin{cases} x + y = 0 \\ x - y = 2 \end{cases}$$

The simplest way to do this would be to again just call `Solve[]` with both the equations and the variables as arguments.

```
>> Solve[x + y == 0 && x - y == 2 , {x, y}]
Out:: {{x -> 1, y -> -1}}
```

Note that in this case the `&&` operator specifies that `Solve[]` needs to take both the equations into account simultaneously, and `{x, y}`, tells it for which variables to solve for.

To highlight Mathematica's flexibility we'll try and implement this is a slightly different way, by defining two functions, that will act as equations, putting them into an intermediary structure and then calling `Solve[]` to solve the system. I.e. :

```
>> Eqn1[x_, y_] := x + y == 0
>> Eqn2[x_, y_] := x - y == 2
>> sysEqn = Eqn1[x, y] && Eqn2[x, y];
>> Solve[sysEqn, {x, y}]
Out:: {{x -> 1, y -> -1}}
```

`Solve[]` is really powerful tool. The user can specify all sorts of constraints, such as the solutions that Mathematica finds should be Real ($\in \mathbb{R}$). To highlight this lets look at the following quartic equation:

$$(x^2 + 2)(x^2 - 2) = 0$$

The above has 4 solutions, two real $x_0 = \sqrt{2}, x_1 = -\sqrt{2} \in \mathbb{R}$, and two purely imaginary $x_2 = i\sqrt{2}, x_3 = -i\sqrt{2} \in \mathbb{C}/\mathbb{R}$. So if we want to only get the Real solutions we can tell Mathematica what to do by specifying the solution type via the $\in$ operator (you can type it in your Notebook by using the `Esc` key and typing `el` as `Esc + el + Esc`):

```
>> Solve[(x^2 + 2)(x^2 - 2 ) == 0, x ∈ Reals]
Out:: {{x -> - √2 }, {x -> √2 }}
```

Sticking with the solving theme, we'll now look at solving a simple ordinary differential equation (ODE). For this we'll look at a simple ODE describing a population's growth over time. The problem itself is quite simple in the sense that it asks the question : Given a population $f(t)$ of people/ants/bacteria, etc. that changes with time t, we know it grows at a rate proportional to itself, i.e. $rf(t)$. We then ask, how will the population look like in time?

The above problem is described by the following ODE:

$$\frac{df(t)}{dt} = rf(t)$$

So, to solve this we need just code the function and use `DSolve[]`. `DSolve[]`, takes as arguments a differential equation/s, the function/s to solve for and the variable/s. To this extent:

```
>> diffEqn1[f_, t_] := f'[t] == r f[t]
>> DSolve[diffEqn1[f, t], f[t], t]
Out:: {{f[t] -> e^(r t) C[1]}}
```

Note that I've used the `'` character instead of the usual `D[]` function for the derivative (only works for single variable functions). This is a recurring theme in Mathematica, there's usually a couple of ways to do the same thing. For example the same effect as the general `D[]` can be obtained by using $\partial_x, \partial_y, \ldots$, depending on the variables to which we take the derivative.

This is indeed what we expect, where we see that Mathematica has even defined a constant for us corresponding to the initial condition, i.e. `C[1]`. To this extent we can even tell Mathematica to use a provided initial condition (generally called a boundary condition). In our case we'll specify the following initial condition $f(0) = F$:

```
>> solDiff = DSolve[ {diffEqn1[f, t], f[0] == F}, f[t], t]
Out:: {{f[t] -> e^(r t) F}}
```

Note that I've stored the solution into the `solDiff` structure. If we now substitute the solution into the original equation we see that indeed we get the expected result:

```
>> diffEqn1[f, t] /. solDiff[[1]]
Out:: f'[t] == e^(r t) F r
```

Note! The `[[1]]` is there since `DSolve[]` returns the solutions as being part of a nested list and the `[[1]]` just specifies the position in the list (we can tell that `DSolve[]` returns a list by looking at the output and noting the `{{}}` bracket structure).

Another thing to note is that depending on how complicated the equations are, Mathematica can struggle with finding the solution. If that is the case we often use a numerical solving function `NDSolve[]` instead. We'll go over this in the Exercises.

Moving on from solving differential and systems of equations, we'll now look at the `Series[]` function. The `Series[]` function is used whenever we want a power series expansion of a function around a point, up to a certain order. `Series[]` takes as arguments the function itself, the variable for which we expand, the value to expand around and the order.

For example , let's say we want to find the power series for $f(x) = \exp(x)$ around $x = 0$, and we only want terms up to x^{10} (in mathematical notation up to $\mathcal{O}(x^{10})$):

```
>> expRes = Series[Exp[x], x, 0, 10]
```

$$\texttt{Out::} 1 + \mathtt{x} + \frac{\mathtt{x}^2}{2} + \frac{\mathtt{x}^3}{6} + \frac{\mathtt{x}^4}{24} + \frac{\mathtt{x}^5}{120} + \frac{\mathtt{x}^6}{720} + \frac{\mathtt{x}^7}{5040} + \frac{\mathtt{x}^8}{40320} + \frac{\mathtt{x}^9}{362880} + \frac{\mathtt{x}^{10}}{3628800} + O\left(\mathtt{x}^{11}\right)$$

If we want to get rid of the $\mathcal{O}(x^{10})$, we can just call `Normal[expRes]`. Note that we don't necessarily need an explicit function, `Series[]` works for a general function $f(x)$, and some arbitrary point to expand around a:

```
>> Series[f[x], x, a, 3]
```

$$\texttt{Out::}\quad \mathtt{f(a)} + (\mathtt{x} - \mathtt{a})\ \mathtt{f'(a)} + \frac{1}{2}\ (\mathtt{x} - \mathtt{a})^2\ \mathtt{f''(a)} + \frac{1}{6}\ (\mathtt{x} - \mathtt{a})^3\ \mathtt{f}^{(3)}\mathtt{(a)} + O\left((\mathtt{x} - \mathtt{a})^4\right)$$

Now that we're equipped with some of the basic tools that we need in general in sciences, we need some way of visualizing the results, que in **plotting**.

As long as you need to work with 2 dimensions, `Plot[]`, and `ListPlot[]` will in general suffice. `Plot[]` is used to plot a number of analytic functions $f_i(x)$ in a range $x \in \{x_{min}, x_{max}\}$. On the other hand `ListPlot[]` is used to plot a list of points $\{x_i, y_i\}$, or more general some discrete data set. In this category Mathematica is quite versatile and they're a lot of different functionalities depending on your needs. For example you can look at `Plot3D[], ContourPlot[]`.

Let's now use the two in some simple examples. First of we'll use `Plot[]` to plot some simple functions, namely $\sin(x)$ and $e^x \sin(x)$, both on the same plot between $x \in \{0, \pi + 1\}$. The code is quite simple, where the `PlotLegends -> "Expressions"` handle automatically assigns a legends to the plots based on the functions :

```
>> Plot[{Sin[x], E^x * Sin[x]}, {x, 0, Pi + 1}, PlotLegends -> "Expressions"]
```

While we're at it let's also use `ListPlot[]`. We're going to try and plot the squares of all the even integers between 0 and 50. To do this we'll use the `Table[]` function to generate a table of even integers , where the rows will be the integers, and their squares. The code for this will be:

```
dataStruct = Table[{i, i^2} , {i, 0, 50, 2}];
ListPlot[dataStruct]
```

The plots will look something like the ones in Figures 1, 2.

So all in all we're kind of equipped to tackle more complicated problems. We'll now do a couple of exercises so you can get used to some of the concepts we've just introduced, along with some extra bits and pieces. In the next tutorial we'll look a bit at matrix algebra.

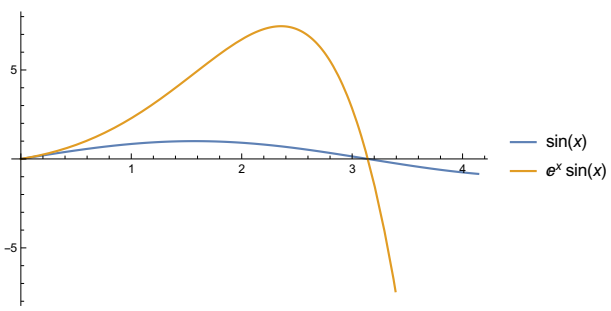

Figure 1: $\sin(x)$ and $e^x \sin(x)$

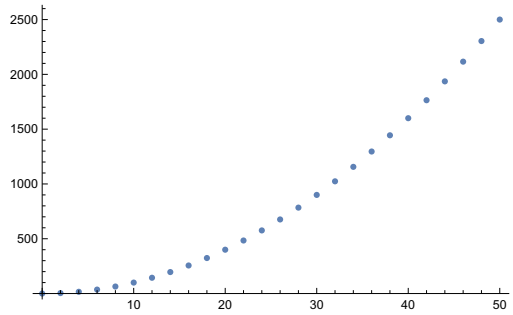

Figure 2: Square of even integers

Exercises

(i) Solve the following system of equations for x, y, z:

$$\begin{cases} x + by - cz = 0 \\ x + cz = 5 \\ ax - by = 4 \end{cases}$$

(ii) Find the positive, integer solutions for x, y (i.e. $x > 0, y > 0$ where $x, y \in \mathbb{Z}$) of the following equation:

$$x^2 + 2y^2 = 3681$$

Now try finding the solutions where both x, y are positive, but only x is an integer solution, and y is just real (again in mathspeak $x > 0, y > 0$ where $x \in \mathbb{Z}, y \in \mathbb{R}$).

(iii) (†) Solve the following system of differential equations [1] for the following functions $g_1(\mu), g_2(\mu), g_3(\mu), y(\mu), \lambda(\mu)$, where μ is the variable:

$$\begin{aligned}
\mu\frac{dg_1(\mu)}{d\mu} &= \frac{41}{6}\frac{(g_1(\mu))^3}{16\pi^2} \\
\mu\frac{dg_2(\mu)}{d\mu} &= -\frac{19}{6}\frac{(g_2(\mu))^3}{16\pi^2} \\
\mu\frac{dg_3(\mu)}{d\mu} &= -7\frac{(g_3(\mu))^3}{16\pi^2} \\
\mu\frac{dy(\mu)}{d\mu} &= \left(\frac{9y(\mu)}{2} - \frac{17(g_1(\mu))^2}{12} - \frac{9(g_2(\mu))^2}{4} - 8(g_3(\mu))^2\right)\frac{y(\mu)}{16\pi^2} \\
\mu\frac{d\lambda(\mu)}{d\mu} &= \frac{1}{16\pi^2}\left\{\frac{3(g_1(\mu))^4}{8} + \frac{3(g_1(\mu))^2(g_2(\mu))^2}{4} + \frac{9(g_2(\mu))^2}{8} - 6(y(\mu))^4\right. \\
&\quad \left. - [3(g_1(\mu))^2 + 9(g_2(\mu))^2 - 12y(\mu)^2]\lambda(\mu) + 24(\lambda(\mu))^2\right\}
\end{aligned}$$

For the above use the following boundary conditions:

$$g_1(80) = 0.35 \qquad g_2(80) = 0.62 \qquad g_3(80) = 1.22 \qquad y(80) = 1 \qquad \lambda(80) = 0.13$$

[1] These are some of the differential equations describing the Standard Model of particle physics, more specifically how different particle and force couplings behave at different energies. They're usually referred to as the renormalization group flow equations (RGEs).

where the system of equations should be solved for μ between 80 and 10^5 (i.e. $\mu \in \{80, 10^5\}$). After you've found the solutions for the system of ODEs, find the value of μ at which $\lambda(\mu)$ becomes negative. Plot $\lambda(\mu)$ in that region to confirm the solution.

Hint! Use `NDSolve[]` to solve the system. This will give you an output in the form of `g1->InterpolatingFunction[]`, which you can use as a substitution rule due to the arrow `->` operator! Furthermore the `FindRoot[]` function to make your life easier for the 2nd part.

(iv) (†) Find the Taylor series for $\sin(x)$ around $x = 0$ up to $\mathcal{O}(x^1), \mathcal{O}(x^3)$, and $\mathcal{O}(x^5)$. Using the series that you just found evaluate the percentage difference between $\sin(x)$ and each of the approximations at the following x's, $x \in \{1.0, 1.5, 2.0, 2.5\}$. To this extent make Mathematica print out a statement of the form:

```
Sin(x) to O(x^i) is accurate to: [percentageDifference] % at x=[...]
```

Lastly plot all three of the series and $\sin(x)$ on the same plot to confirm your numerical evaluation. On the same plot, plot the value of $\sin(x)$ and the three series at $x = 1.5$, and observe if they roughly correspond to your numerical analysis.

Hint! You can store the different plots that you produce just like any other variable, e.g. `plt1 = Plot[...]; , plt2 = ListPlot[...];, ...` and then using the `Show[plt1, plt2, ...]` function you can plot all of them in the same figure.

3. Mathematica 103

We'll now look at a couple of matrix algebra topics. Starting with the basics, let's say we want to work with a matrix A, a column vector $\vec{x}$ and a row vector $\vec{y}$:

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 10 & 12 \\ 15 & 20 & 25 \end{pmatrix} \qquad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \vec{y} = \begin{pmatrix} y_1 & y_2 & y_3 \end{pmatrix}$$

It's quite simple to define the above in Mathematica. The declaration is similar to a list:

```
>> A1 = {
>>      {1, 3, 5},
>>      {2, 10, 12},
>>      {15, 20, 25}
>>       };
>> xvec = {
>>        {x1},
>>        {x2},
>>        {x3}
>>        };
>> yvec = { {y1, y2, y3} };
```

Some useful tools when working with matrices and vectors, are `Dimensions[]` and `MatrixForm[]`. Calling `Dimensions[A1]` for example will return the dimensions of the matrix, in this case `{3, 3}`. This is especially useful when working with matrix equations and something goes wrong and you need to try and work out where it went wrong. Similarly `MatrixForm[A1]` will output the matrix you've just defined in a human readable fashion as in Equation 3

While we're at it let's look at the usual matrix algebra functions, i.e. `Tr[], Det[], Inverse[], IdentityMatrix[]`. Let's look at them one of a time.

`Tr[A1]` will return the trace of the `A1` matrix (remember the trace is just the sum of the elements on the main diagonal of the matrix). In our case `Out = 36`.

`Det[A1]` will return the determinant of the `A1` matrix. In our case `Out = -150`.

`Inverse[A1]` will return the inverse matrix of `A1` (recall the inverse A^{-1} is the matrix such that $A^{-1}A = \mathbb{1}$). In our case this will output:

$$\begin{pmatrix} -\frac{1}{15} & -\frac{1}{6} & \frac{7}{75} \\ -\frac{13}{15} & \frac{1}{3} & \frac{1}{75} \\ \frac{11}{15} & -\frac{1}{6} & -\frac{2}{75} \end{pmatrix}$$

Lastly `IdentityMatrix[n]` creates a $(n \times n)$ identity matrix (1's on the diagonal 0's in the rest of it). Using these let's look at a bit of algebra, specifically matrix multiplication denoted by `.` (e.g. `A.B`), and scalar multiplication denoted by `*` (e.g. `a*A`).

Let's say we want to compute the following expression:

$$aA_1 + b\mathbb{1} + c(A_1)^2 + A_1^{-1}A_1$$

To this extent:

```
>> a A1 + b * IdentityMatrix[3] + c*A1.A1 + Inverse[A1].A1 // MatrixForm
```

Note! In the same way as before if we just write `a A1` Mathematica interprets it as `a*A1`, and by using `//MatrixForm` at the end of the expression, the output will be in human readable form:

$$\begin{pmatrix} a+b+82c+1 & 3a+133c & 5a+166c \\ 2a+202c & 10a+b+346c+1 & 12a+430c \\ 15a+430c & 20a+745c & 25a+b+940c+1 \end{pmatrix}$$

While we're at it, let's check the Cayley-Hamilton equation for 2×2 matrices, which says that for any 2×2 matrix A, it obeys the equation:

$$A^2 - \mathrm{Tr}(A)A + \det(A)\mathbb{1}_2 = 0$$

So let's try it out, for a general matrix $A = ((a, b), (c, d))$:

```
>> A = {
>>       {a, b},
>>       {c, d}
>>     };
>> A.A - Tr[A] A + Det[A] IdentityMatrix[2] // MatrixForm
```

Note that this gives the following output:

$$\begin{pmatrix} a^2+da-(a+d)a & ab+db-(a+d)b \\ ac+dc-(a+d)c & d^2+ad-(a+d)d \end{pmatrix}$$

If we actually expand out the brackets in the matrix we see that indeed it will be 0 as expected. This is the thing we mentioned at the start that Mathematica is as general as possible until we tell it otherwise. Whenever we want it to reduce an expression to its most minimal form we can use `Simplify[]`. Therefore if we modify the above as:

```
>> Simplify[A.A - Tr[A] A + Det[A] IdentityMatrix[2]] // MatrixForm
```

which will now produce the expected output:

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

Let's do one more example to familiarize ourself with `Simplify[]`. We will want to check if the following matrices are **Hermitean**:

$$A_2 = \begin{pmatrix} c & 3-ai \\ 3+ai & d \end{pmatrix} \qquad A_3 = \begin{pmatrix} c+ai & 3-ai \\ 3+ai & d-ai \end{pmatrix}$$

$\forall a, b, c, d \in \mathbb{R}$.

Note! A matrix is Hermitean if it is equal to it's complex conjugate transpose, i.e. $\overline{A^T} = A$, where the operation of conjugation and transposition is called hermitean conjugation, and is denoted by a dagger † (i.e. $\overline{A^T} \equiv A^\dagger$) . In Mathematica the transpose of a matrix can be found using `Transpose[]`. To check the hermiticity we'll take the difference

$$A^\dagger - A$$

and check if it is equal to 0.

Again we'll have to use `Simplify[]`, where this time we'll have to specify that a, b, c, d are real.

```
>> Simplify[Conjugate[Transpose[A2]],{a ∈ Reals,c ∈ Reals,d ∈ Reals}]] - A2
>> Simplify[Conjugate[Transpose[A3]],{a ∈ Reals,c ∈ Reals,d ∈ Reals}]] - A3
```

The above then give us the output:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} -2ia & 0 \\ 0 & 2ia \end{pmatrix}$$

So, we can see that A_2 is a hermitean matrix, whereas A_3 is hermitean iff $a = 0$.

Note that vectors behave in the same way as matrices. Let's say we want to find the magnitude of $\vec{x}$, i.e. $|\vec{x}|^2$. To do that we note that for the usual real component vectors in flat 3D space the magnitude is given by:

$$|\vec{x}|^2 = (\vec{x})^T \vec{x}$$

where $(\vec{x})^T$ is now a row vector with the same components as $\vec{x}$.

Therefore in Mathematica we'll have:

```
>> (Transpose[xvec].xvec)[[1, 1]]
Out = x1^2 + x2^2 + x3^2
```

Note! `Transpose[xvec].xvec` gives us a dimension 1×1 matrix which is in fact a scalar, but we still need to access the scalar inside the 1×1 structure via `[[1, 1]]` (I know this doesn't make a lot of sense but Mathematica is a bit alien at times).

Exercises

(i) Given a column vector $\vec{x}$, with three real components x_1, x_2, x_3, check that rotating the vector around the z axis by an angle θ, followed by a rotation around the y axis by an angle ϕ and finally a rotation around the x axis by an angle Ω, the vector has the same length (or magnitude $|\vec{x}|$).

Hint! The rotation matrices around the x, y, z axis are given by:

$$R_x(\Omega) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\Omega) & -\sin(\Omega) \\ 0 & \sin(\Omega) & \cos(\Omega) \end{pmatrix} \qquad R_y(\phi) = \begin{pmatrix} \cos(\phi) & 0 & -\sin(\phi) \\ 0 & 1 & 0 \\ \sin(\phi) & 0 & \cos(\phi) \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(ii) (†) Write a script that produces the following $n \times n$ matrices (where n is provided by the user):

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1(n-2)} & a_{1(n-1)} & a_{1n} \\ a_{21} & 0 & 0 & \dots & 0 & 0 & a_{2n} \\ a_{31} & 0 & 0 & \dots & 0 & 0 & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{(n-1)1} & 0 & 0 & \dots & 0 & 0 & a_{(n-1)n} \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{n(n-2)} & a_{n(n-1)} & a_{nn} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & 0 & 0 & \dots & 0 & 0 & b_{1n} \\ 0 & b_{22} & 0 & \dots & 0 & b_{2(n-1)} & 0 \\ 0 & 0 & b_{33} & \dots & b_{3(n-2)} & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & b_{(n-2)3} & \dots & b_{(n-2)(n-2)} & 0 & 0 \\ 0 & b_{(n-1)2} & 0 & \dots & 0 & b_{(n-1)(n-1)} & 0 \\ b_{(n-1)1} & 0 & 0 & \dots & 0 & 0 & b_{nn} \end{pmatrix}$$

where the non zero elements are given by $a_{ij} = i + j, b_{ij} = i$. Afterwards use `MatrixPlot[]` to plot the two matrices as a grid, where the numbers in the matrix act as a temperature map (look at the `ColorFunction ->` option). Finally add and then plot $A + B$ in the same way but this time the colormap should be Monochrome.

Hint! Look at the different elements and try and figure out a pattern relating to the different matrix indexes. Some functions that might be helpfull are `ConstantArray[]`, and `If[]`. This exercise more than ever you'll have to use the Mathematica `Help` to figure out all the options.