Sheet 2

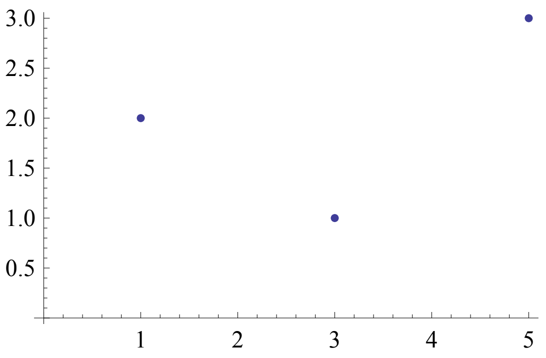
Interpolation, numerical calculus, and polar coordinates

2.1 Plotting Points

The command **`ListPlot[{{a,b},{c,d},{e,f}}]`** will plot the points (a,b), (c,d) and (e,f), provided they are numerical. Note that each pair of points is a list (the inner curly brackets), and then all the points are collected together into a single list with another curly bracket around them.

```
In[40]:= ListPlot[{{1, 2}, {3, 1}, {5, 3}}, AxesOrigin -> {0, 0}]
```

Out[40]=



Adding the option **`PlotStyle->{PointSize[Large],Red}`** would give large red dots. The points can be joined up by either using **`ListLinePlot`** instead of **`ListPlot`**, or by adding **`Joined->True`** as an option at the end of the command (before the closing square bracket).

Exercise

Select two points, e.g. $(2,12)$ and $(9,-23)$ (or choose your own) and plot them using **`ListPlot`**. Keep these two points in mind for the next section.

2.2 Linear Interpolation

Recall that the linear interpolation formula between two known points (x_0, f_0) and (x_1, f_1) is given by:

$$f(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0)$$

To use this, we will define a function,

```
In[41]:= f[x_]:=f0 + ((f1 - f0)/(x1 - x0))*(x - x0)
```

where we have used **`f0,f1,x0,x1`** as the values for f_0 etc. You can also use the palette in Mathematica to format this nicely as a fraction.

There are multiple different ways to pass the particular local points to the function. One way is to give the points global values, e.g. **`{x0,f0}={2,12}`** (or individually as **`x0=2;f0=12`**). Be aware these will be remembered and affect all your subsequent evaluations until you **`Clear`** the values or quit the Kernel.

```
In[42]:= {x0,f0} = {2,12};
         {x1,f1} = {9,-23};
         f[x] // Simplify

Out[42]= 22 - 5 x
```

After setting different values of the points, the function will use the new points due to the colon in the definition of **`f`**:

```
In[43]:= {x0,f0} = {5,1};
         {x1,f1} = {4,-1};
         f[x] // Simplify

Out[43]= -9 + 2 x
```

After we use **`Clear`** on the values, the general definition will be returned:

```
In[44]:= Clear[x0,x1,f0,f1]
         f[x]
```

Out[44]= $f0 + \frac{(-f0 + f1)\ (x - x0)}{-x0 + x1}$

Another way to insert the values into the above function is to use replacement rules, this can be quite clunky but keeps everything locally defined.

```
In[45]:= f[x] /. {x0 -> 2, f0 -> 12, x1 -> 9, f1 -> -23} //Simplify

Out[45]= 22 - 5 x
```

A third way is to define a different function with all the values as arguments, e.g.

```
In[46]:= f[x_,x0_,f0_,x1_,f1_] := f0 + (f1 - f0)/(x1 - x0) (x - x0)
         f[x,2,12,9,-23] //Simplify

Out[46]= 22 - 5 x
```

We can have multiple definitions of **`f`** defined at the same time, with different numbers of arguments, and the appropriate definition will be used depending on how many parameters are used when calling the function.

Finally, a fourth way is to use the **`Block`** environment, where you define all the local variables as a list as the first element and the command as the second (separating intermediate commands with a semicolon if necessary):

```
In[47]:= Block[{x0 = 2, f0 = 1, x1 = 9, f1 = -23}, f[x]//Simplify ]

Out[47]=  22 - 5 x
```

`Block` localises the variables only within that specific **`Block`**, so to plot the function you would write

```
In[48]:= Block[{x0 = 2, f0 = 1, x1 = 9, f1 = -23}, Plot[f[x],{x, 0, 3}]]
```

There are a number of other ways of passing the parameters to the interpolation function, such as using wrapping the code in a **`Module`** environment, but four is enough!

Exercises

1. Use one of the above methods to find $f(x)$ at various x values between your two points using linear interpolation, and plot them all.

2. Plot both the interpolated function and the individual points on the same graph using **Show**.

3. Repeat the above process with two different points.

4. Now choose three points, for example $(4,-3),(7,-6),(11,5)$, and repeat the same exercise with quadratic interpolation instead of linear interpolation. Recall that the function for quadratic interpolation is given by

$$f(x)=\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f_0+\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f_1+\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f_2$$

2.3 Numerical Differentiation

Consider the function $g = e^{\sin x}$. The two-point approximation for estimating the derivative at $x = x_0$ with step size h is

$$g'(x_0) \approx \frac{g(x_0+h)-g(x_0-h)}{2h}.$$

We can implement this by

```
In[49]:= g[x_] = Exp[Sin[x]];
         dg[x0_, h_]:= (g[x0 + h] - g[x0 - h])/(2 h)
```

dg is now a function which requires two variables, x_0 and h (in that order). At $x_0 = 1$ say, we can find the value when $h = 0.1$ as:

```
In[50]:= dg[1, 0.1]

Out[50]= 1.24665
```

We can then compare with the exact derivative, **g'[1]** (which is the same as **D[g[x],x]/.x->1**):

```
In[51]:= dg[1, 0.1] - g'[1.]
         dg[1, 0.01] - g'[1.]
         dg[1, 10^-5] - g'[1.]
         dg[1, 10^-10] - g'[1.]

Out[51]= -0.0067328
         -0.0000675236
         -7.47922*10^-11
          1.07527*10^-6
```

We can see the relative error reduces with h (the smaller the value of h, the better the approximation), up until a point where the error starts becoming larger again as h gets very small. As described in the lectures, this is due to 'roundoff' error, when two very similar numbers $g(x_0+h)$ and $g(x_0-h)$ are subtracted. We avoid this by having Mathematica's evaluate the expression exactly, for which we need to specify exact numbers (**g'[1]** not **g'[1.]**) and then using **N** to evaluate the answer numerically.

```
In[52]:= N[dg[1, 10^-10] - g'[1], 5]

Out[52]= .7526*10^-21
```

Exercises

1. The four-point approximation is given by

$$f'(x_0) \approx \frac{-f(x_0+2h)+8f(x_0+h)-8f(x_0-h)+f(x_0-2h)}{12h} \tag{2.1}$$

   Implement (with a different function name!) and use the four-point approximation to estimate the derivative of g at $x = 2$ with $h = 0.01$.

2. Use the two-point formula to estimate the first derivative of $y = x^3 \sin x$ at the point $x = 1$ with $h = 0.01$. Compare with the four-point formula, and see which is more accurate.

2.4 Numerical Integration

As before, take the function $g(x) = e^{\sin x}$. The integral

$$\int_0^4 g(x)dx$$

cannot be done exactly (check by using **Integrate**), but we can evaluate the integral numerically with **NIntegrate**:

```
In[53]:= nint = NIntegrate[g[x], {x, 0, 4}]

Out[53]= 6.79647
```

Here we have assigned the numerical result to the variable **nint**.

To use the Trapezium rule with $n = 4$ to approximate this, we could write

```
In[54]:= h = (4-0)/4;
         h/2 (g[0]+2 g[1]+ 2 g[2] + 2 g[3] + g[4]) //N

Out[54]= 6.6885
```

which is a slight overestimate of the exact value. We can repeat this manual calculation for other values of n, but a better way would be to define a function that calculates the trapezium rule for n intervals as,

```
In[55]:= Clear[h]
         trapRule[g_, n_, a_, b_] :=
         Block[{h = (b - a)/n}, Sum[h/2 (g[a + (i-1)*h] + g[a + i*h]),
         {i, 1, n}]]
```

This code uses **Block** to introduce a temporary local variable $h = (b - a)/n$, and then **Sum** to add pairs of terms such as $g(a) + f(a + h)$. See if you can understand how this code is equivalent to the definition of the trapezium rule given in lectures. For the Trapezium rule with $n = 4$ between $a = 0$ and $b = 4$, we find

```
In[56]:= trapRule[g, 4, 0, 4] // N

Out[56]= 6.6885
```

fairly close to the exact value, even for only $n = 4$. We can use the command **Table** to repeatedly evaluate the function at n from 1 to 10 to see how quickly this converges to a stable result:

```
In[57]:=  trapValues = Table[trapRule[g, n, 0, 4], {n, 1, 10}] //N

Out[57]=  {2.93833, 6.43432, 6.60987, 6.6885, 6.72716, 6.74825,
            6.761, 6.7693, 6.77499, 6.77907}
```

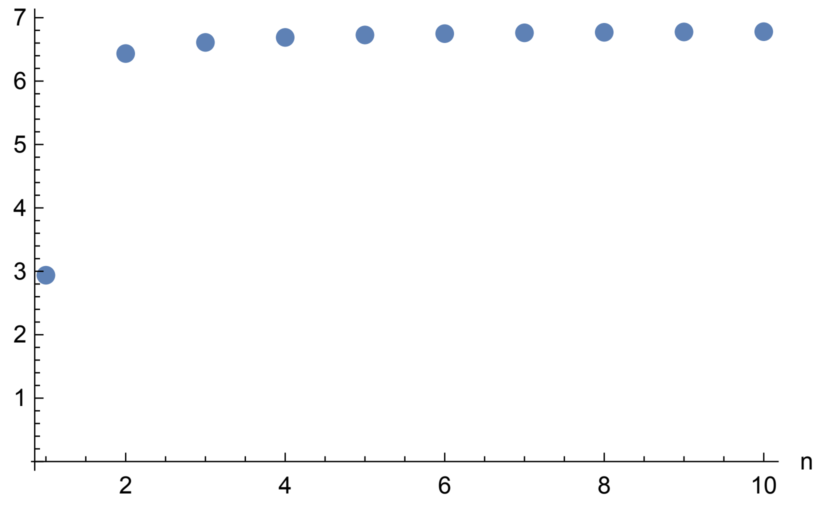
Each number in the output is the trapezium rule evaluated with $n = 1, 2, 3, \ldots, 10$ intervals. The function **`Table`** evaluates its first argument repeatedly, increasing the variable in its second argument within the given range. You can change the increment size by including an extra value in the second argument:

```
In[58]:=  Table[n, {n, 0, 1, 0.1}]
Out[58]=  {0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
In[59]:=  Table[n, {n, 0.1, 1, 0.2}]
Out[59]=  {0.1, 0.3, 0.5, 0.7, 0.9}
```

We can plot these values using **ListPlot**:

```
In[60]:= ListPlot[trapValues, PlotRange -> All,
           PlotStyle -> PointSize[Large]]
```

Out[60]=



A similar but more convoluted function for Simpson's rule would be:

```
In[61]:= simpRule[g_, n_?EvenQ, a_, b_] :=
         Block[{h = (b - a)/n},
          Sum[h/3 (g[a + (i-1) h] + 4 g[a + i h] + g[a + (i+1) h]),
         {i, 1, n-1, 2}]]
```

here **?EvenQ** restricts the definition of **simpRule** to only be applied for even values of n, and the **Sum** (which works like Table for numerical values but adds the values together) is incrementing in steps of 2.

If you try and give an odd value of n, Mathematica will return the input unevaluated:

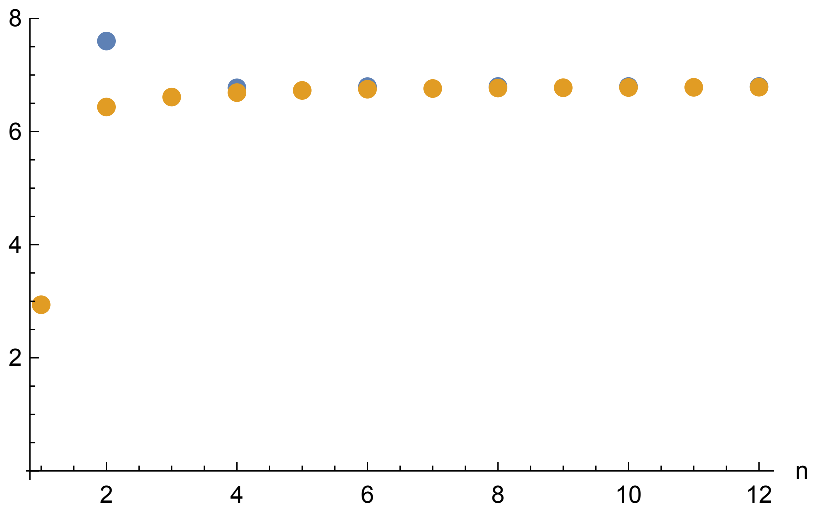
```
In[62]:=  simpValues = Table[simpRule[g, n, 0, 4], {n, 1, 10}] // N

Out[62]=  {simpRule[g, 1., 0., 4.], 7.59965, simpRule[g, 3., 0., 4.], 6.7732
           simpRule[g, 5., 0., 4.], 6.79437, simpRule[g, 7., 0., 4.], 6.7962
           simpRule[g, 9., 0., 4.], 6.79637}
```

We can plot the two sets of data together:

```
In[63]:= ListPlot[{trapValues, simpValues}, PlotRange -> All]
```

Out[63]=



Note that the points where **simpValues** does not evaluate to a number are (silently) ignored. If you try to plot a **ListPlot** and nothing appears, then it is likely that the data you have supplied is not being evaluated to a number.

Exercises

Now consider the following integral

$$\int_0^3 \cos(x^3 + x)dx$$

1. Define a function **q[x_]** as the integrand.
2. For this integral, compare the Trapezium Rule and Simpson's Rule for $n = 4$ with the good numerical approximation from **NIntegrate**. A plot of the integrand $g(x)$ may help explain why $n = 4$ points gives such a poor approximation?
3. Plot how the two approximations converge to the exact value as n is increased (you may need to try up to $n = 100$, or more).

2.5 Polar Coordinates

The function **ToPolarCoordinates** may be used to take a point (or points) in Cartesian coordinates (x, y) and translate them to Polar coordinates (r, θ).

```
In[64]:= ToPolarCoordinates[{1,1}]
```

Out[64]= $\{\sqrt{2}, \frac{\text{Pi}}{4}\}$

As with most functions in Mathematica, you can provide symbolic variables rather than explicit numbers:

```
In[65]:= ToPolarCoordinates[{x,y}]
```

Out[65]= $\{\sqrt{x^2 + y^2}, \text{ArcTan}[x,y]\}$

(the two-argument form of $\tan^{-1}$, **ArcTan[x,y]**, returns an angle between $-\pi$ and π that is correct for (x, y) coordinates in all four quadrants of the plane). **FromPolarCoordinates** will do the reverse, transform a point from polar to Cartesian coordinates.

```
In[66]:= FromPolarCoordinates[{1,1}]

Out[66]= {Cos[1], Sin[1]}
```

Again, algebraic expressions can be used:

```
In[67]:= FromPolarCoordinates[{r, th}]

Out[67]= {r Cos[th], r Sin[th]}
```

Exercises

Convert the following points between coordinate systems:

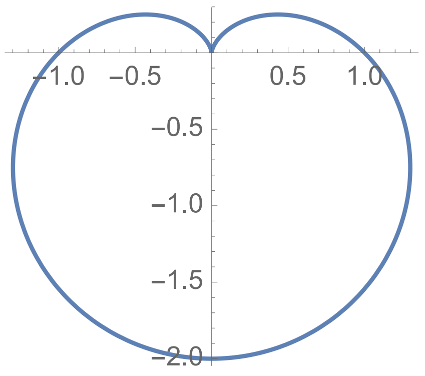
1. $x = 1, y = 2$
2. $x = -1, y = -3$
3. $x = 5, y = 0$
4. $r = 1, \theta = \pi$
5. $r = 3, \theta = -2$
6. $r = 2, \theta = 0$

2.6 Polar Curves

The command **`PolarPlot[f,{th,a,b}]`** draws the polar curve $r = f$ as a function of θ for $a \leq \theta \leq b$:

```
In[68]:= PolarPlot[1 - Sin[th], {th, -Pi, Pi}]

Out[68]=
```



Here **`th`** has been used for θ but you can use **`t`**, the whole word **`theta`**, the Greek letter θ (entered via the palette or Esc q Esc) or anything else as appropriate.

Exercises

Using **`PolarPlot`**, draw a few curves in polar coordinates. Remember that you just need to give the value of r to be used for each angle θ.

1. A circle with radius 2.
2. A straight line passing through $(4, 0)$ and $(2, 1)$.
3. A spiral with constant distance between the arms. You will need to allow π to go above 2π to show multiple arms of the spiral.

End

Make sure that you save your file somewhere that you can retrieve it later. You now have all the material needed to answer Q3 and Q4 on the coursework.