

Solving ODEs and DAEs with NDSolve

The Wolfram Language™ has extensive capabilities for solving differential equations both symbolically and numerically. This section covers:

- Introduction to solving differential equations
- Solving differential equations numerically with **NDSolve**
- Specifying boundary conditions
- Features and options of NDSolve

Introduction to Solving Differential Equations

Symbolic Solutions

With **DSolve**, the symbolic solution for many types of differential equations can be found.

Nonlinear Differential Equations:

```
In[ ]:= DSolve[y''[x] == 3 y[x] * y'[x] + (3 y[x]^2 + 4 y[x] + 1), y[x], x] [[1]] // Column // FullSimplify
```

$$\text{Out[]}= y[x] \rightarrow \frac{\left(i \sqrt{6} \sqrt{e^{-2x}} \sqrt{c_1} - 6 c_2\right) \cosh\left[\sqrt{\frac{3}{2}} \sqrt{e^{-2x}} \sqrt{c_1}\right] + \left(-3 i + 2 \sqrt{6} \sqrt{e^{-2x}} \sqrt{c_1} c_2\right) \sinh\left[\sqrt{\frac{3}{2}} \sqrt{e^{-2x}} \sqrt{c_1}\right]}{6 c_2 \cosh\left[\sqrt{\frac{3}{2}} \sqrt{e^{-2x}} \sqrt{c_1}\right] + 3 i \sinh\left[\sqrt{\frac{3}{2}} \sqrt{e^{-2x}} \sqrt{c_1}\right]}$$

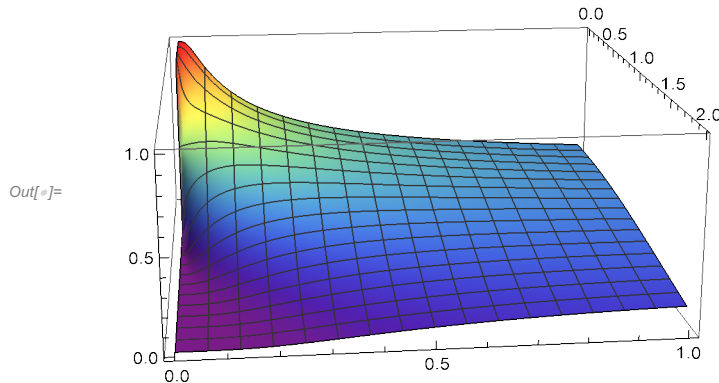
Partial Differential Equations (PDEs):

```
In[ ]:= BurgersEqn = D[u[x, t], t] + u[x, t] * D[u[x, t], x] == v D[u[x, t], {x, 2}];
```

```
In[ ]:= sol = DSolve[{BurgersEqn, u[x, 0] == UnitBox[x]}, u[x, t], {x, t}] [[1]] // FullSimplify
```

$$\text{Out[]}= \left\{ u[x, t] \rightarrow \frac{e^{\frac{1+t}{4v}} \left(-\operatorname{Erf}\left[\frac{-1+2t-2x}{4\sqrt{tv}}\right] + \operatorname{Erf}\left[\frac{1+2t-2x}{4\sqrt{tv}}\right] \right)}{e^{\frac{1+t}{4v}} \left(\operatorname{Erfc}\left[\frac{-1+2t-2x}{4\sqrt{tv}}\right] - \operatorname{Erfc}\left[\frac{1+2t-2x}{4\sqrt{tv}}\right] \right) + e^{\frac{x}{2v}} \left(\operatorname{Erfc}\left[\frac{1-2x}{4\sqrt{tv}}\right] + e^{\frac{1}{2}/v} \operatorname{Erfc}\left[\frac{1+2x}{4\sqrt{tv}}\right] \right)} \right\}$$

```
In[ ]:= Plot3D[Evaluate[u[x, t] /. sol /. v -> 1], {x, 0, 2}, {t, 0, 1},
  ColorFunction -> "Rainbow", PlotPoints -> 50, PlotRange -> All]
```



System of Equations:

```
In[ ]:= A = {{4, 6}, {1, -1}};
```

```
In[ ]:= X[t_] = {x[t], y[t]};
```

```
In[ ]:= DSolve[X'[t] == A.X[t], X[t], t] // Column
```

```
Out[ ]:= {x[t] -> 1/7 e^{-2t} (1 + 6 e^{7t}) c_1 + 6/7 e^{-2t} (-1 + e^{7t}) c_2, y[t] -> 1/7 e^{-2t} (-1 + e^{7t}) c_1 + 1/7 e^{-2t} (6 + e^{7t}) c_2}
```

Differential Algebraic Equations (DAEs):

```
In[ ]:= DSolve[{x'[t] == x[t] + 2 y[t], y[t] == x[t] + 2^t, x[t] + z[t] == 0}, {x[t], y[t], z[t]}, t][[1]]
```

```
Out[ ]:= {x[t] -> 1/8 (-e^{3t} c_1 + 16 e^{3t+t} (-3+Log[2]) / (-3+Log[2])),
  y[t] -> 2^t + 1/8 (-e^{3t} c_1 + 16 e^{3t+t} (-3+Log[2]) / (-3+Log[2])), z[t] -> 1/8 (e^{3t} c_1 - 16 e^{3t+t} (-3+Log[2]) / (-3+Log[2]))}
```

Asymptotic Solutions

Beginning in Version 11.3, asymptotic methods have been implemented. This provides a third method for solving differential equations (aside from exact symbolic solutions with `DSolve` and numeric solutions with `NDSolve`).

With **`AsymptoticDSolveValue`**, the asymptotic solution at infinity of the nonlinear equation presented in the previous subsection can be found:

```
In[ ]:= AsymptoticDSolveValue[y''[x] == 3 y[x] x y'[x] + (3 y[x]^2 + 4 y[x] + 1),
  y[x], {x, Infinity, 2}] // FullSimplify
```

```
Out[ ]:= -1 + (sqrt(2/3) e^{-x} sqrt(c_1) (-2 c_2 + Cot[sqrt(3/2) e^{-x} sqrt(c_1)])) / (1 + 2 c_2 Cot[sqrt(3/2) e^{-x} sqrt(c_1)])
```

Similar to `AsymptoticDSolveValue`, **`AsymptoticIntegrate`** provides asymptotic solutions for integrals.

Similar functions like **`AsymptoticSolve`**, **`AsymptoticRSolveValue`** and **`AsymptoticSum`** provide a third

method to solve a particular problem, complementing the analogous symbolic and numeric functions.

Numeric Solutions

Not all differential equations have symbolic solutions. In such a case, numeric solutions should be sought with `NDSolve`.

Finding numeric solutions to differential equations with `NDSolve` will be the focus of the rest of this course.

The types of differential equations that can be solved depend on the region over which a solution is sought.

Rectangular Regions

Rectangular regions are:

- Lines in 1D
- **Rectangle** boxes in 2D
- **Cuboid** boxes in 3D
- n -dimensional boxes in n dimensions

`NDSolve` can solve both linear and nonlinear PDEs up to any order when solving over rectangular regions using the method of lines and tensor product grid.

Arbitrarily Shaped Regions

When solving over arbitrarily shaped regions, `NDSolve` will use the finite element method (FEM). The types of equations that can be solved with FEM fall into a general form:

$$m \frac{\partial^2}{\partial t^2} u + d \frac{\partial}{\partial t} u + \nabla \cdot (-c \nabla u - \vec{\alpha} u + \vec{\gamma}) + \vec{\beta} \cdot \nabla u + a u - f = 0.$$

```
In[ ]:= Clear["Global`*"]
```

Basics of Using NDSolve

When using `NDSolve`, the initial or boundary conditions must be sufficient to determine a unique solution. For ODEs of a single dependent variable, the number of conditions must equal the highest-order derivative:

```
In[ ]:= eqn = y'[x] + y[x] + Sin[x] == 0;
cond = y[0] == 0;
```

```
In[ ]:= sol = NDSolveValue[{eqn, cond}, y[x], {x, -Pi, 3 Pi}]
```

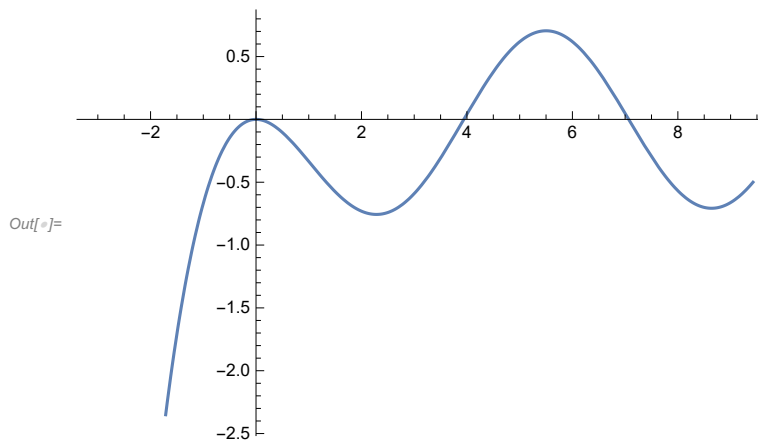
```
Out[ ]:= InterpolatingFunction[ Domain: {{-3.14, 9.42}}  
Output: scalar ] [x]
```

With arguments:

1. equation and function conditions
2. function to solve for
3. range of the independent variable.

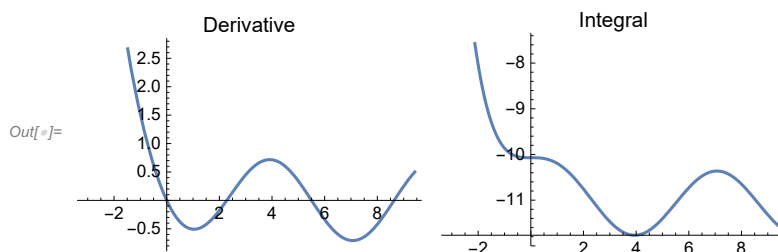
Solutions are given in terms of **InterpolatingFunction**. They can be visualized using **Plot** and other similar functions:

```
In[ ]:= Plot[sol, {x, -Pi, 3 Pi}]
```



Further processing and operations can be performed on the InterpolatingFunction. For example, derivatives and integrals can be computed:

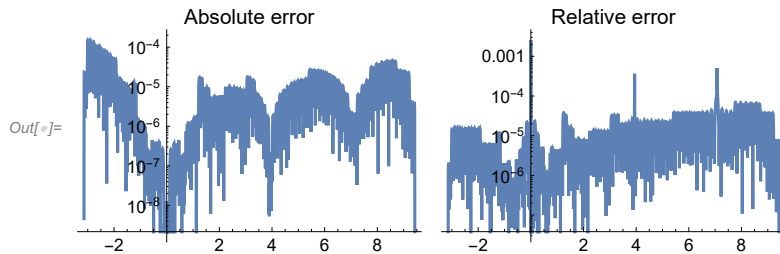
```
In[ ]:= Grid[{{  
  Plot[Evaluate[D[sol, x]], {x, -Pi, 3 Pi}, PlotLabel -> "Derivative",  
  Plot[Evaluate[ $\int$  sol dx], {x, -Pi, 3 Pi}, PlotLabel -> "Integral"]  
}}]
```



NDSolve is very reliable. Nonetheless, it is important to verify results. With the preceding example, you can put the solution into the left-hand side of the equation and plot the result to show that it approximately equals zero everywhere:

```
In[ ]:= verify = eqn[[1]] /. {y[x] → sol, y'[x] → D[sol, x]};
```

```
In[ ]:= Grid[{
  LogPlot[Abs[verify], {x, -Pi, 3 Pi}, PlotLabel → "Absolute error",
  LogPlot[Abs[verify/sol], {x, -Pi, 3 Pi}, PlotLabel → "Relative error"]
}]
```



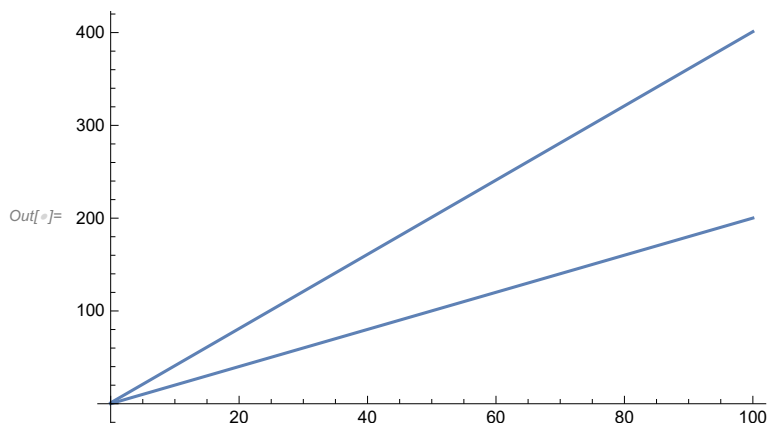
It is also possible to specify vectorial variables in NDSolve:

```
In[ ]:= sol1 = NDSolve[
  {n[0] == {.1, 1}, r[0] == 1, r'[t] == 0, n'[t] == {2 r[t], 4 r[t]}}, {n[t], r[t]}, {t, 0, 100}]
```

```
Out[ ]:= { {n[t] → InterpolatingFunction[
  Domain: {{0., 100.}}
  Output dimensions: {2} ] [t],
```

```
  r[t] → InterpolatingFunction[
  Domain: {{0., 100.}}
  Output: scalar ] [t] } }
```

```
In[ ]:= Plot[n[t] /. sol1, {t, 0, 100}, PlotStyle → Small]
```



```
In[ ]:= Clear["Global`*"]
```

NDSolve versus NDSolveValue

NDSolve and NDSolveValue both do the exact same calculation. The only difference between the two functions is how the output is presented:

```

In[ ]:= eqn = y' [x] + y[x] + Sin[x] == 0;
cond = y[0] == 0;
solNDSolve = NDSolve[{eqn, cond}, y[x], {x, -π, 3 π}];
solNDSolveValue = NDSolveValue[{eqn, cond}, y[x], {x, -π, 3 π}];

```

Both ways to present the solution have their own advantages and disadvantages. Which function to use depends on the task you are trying to accomplish, as well as personal taste.

Any task one function can accomplish, the other function can as well, just with a little more code in some cases.

Dependent Variable: $y[x]$

To illustrate the difference between these two functions, consider the symbolic equivalent of these functions: `DSolve` and **`DSolveValue`**. This is simply so you can view the functional form of the solution:

```

In[ ]:= solDSolveValue = DSolveValue[{eqn, cond}, y[x], {x, -π, 3 π}]
solDSolve = DSolve[{eqn, cond}, y[x], {x, -π, 3 π}]

```

$$\text{Out[]} = \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right)$$

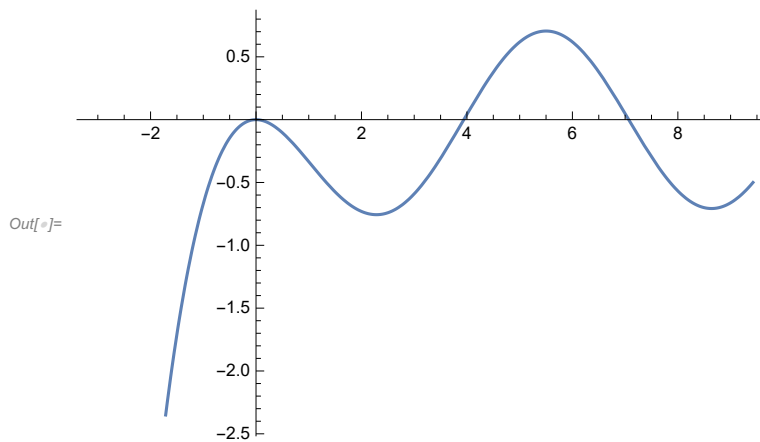
$$\text{Out[]} = \left\{ \left\{ y[x] \rightarrow \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) \right\} \right\}$$

Using `DSolveValue` makes plotting the solution extremely easy:

```

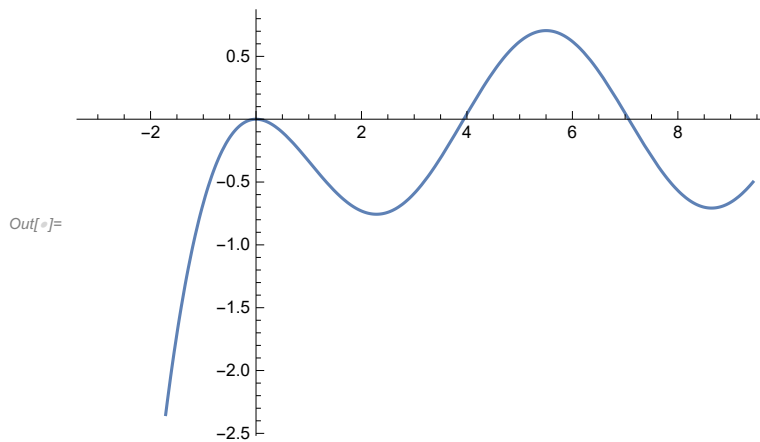
In[ ]:= Plot[solDSolveValue, {x, -π, 3 π}]

```



Whereas the `DSolve` solution requires the use of a replacement rule to extract the function to plot:

```
In[ ]:= Plot[Evaluate[y[x] /. solDSolve], {x, -π, 3 π}]
```



If you want to plug the solution into some other equation, the DSolveValue solution requires a replacement rule, whereas the DSolve solution does not, as the rule is built into the solution:

```
In[ ]:= otherEqn = (y[x]^2 - y[x])^2;
```

```
In[ ]:= otherEqn /. solDSolve[[1]]
```

```
otherEqn /. y[x] → solDSolveValue
```

$$\text{Out[]} = \left(\frac{1}{4} \left(-e^{-x} + \cos[x] - \sin[x] \right)^2 + \frac{1}{2} \left(e^{-x} - \cos[x] + \sin[x] \right) \right)^2$$

$$\text{Out[]} = \left(\frac{1}{4} \left(-e^{-x} + \cos[x] - \sin[x] \right)^2 + \frac{1}{2} \left(e^{-x} - \cos[x] + \sin[x] \right) \right)^2$$

However, if you are putting the solution into an equation that involves derivatives of the function, then both types of solutions will have problems because of the derivative terms:

```
In[ ]:= eqn /. solDSolve[[1, 1]]
```

$$\text{Out[]} = \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) + \sin[x] + y'[x] == 0$$

```
In[ ]:= eqn /. y[x] → solDSolveValue
```

$$\text{Out[]} = \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) + \sin[x] + y'[x] == 0$$

```
In[ ]:= (* eqn /. {solDSolve[[1, 1]], D[solDSolve[[1, 1]], x]} *)
```

```
In[ ]:= (* eqn /. {y[x] → solDSolveValue, y'[x] → D[solDSolveValue, x]} *)
```

Dependent Variable: y

In the last calculation of the previous subsection, notice that $y'[x]$ was not replaced. This is because the replacement rule just says how to replace $y[x]$, not $y'[x]$. You need to include the derivative explicitly:

```
In[ ]:= eqn /. {solDSolve[[1, 1]], D[solDSolve[[1, 1]], x]}
```

$$\text{Out[]} = \frac{1}{2} \left(e^{-x} - \cos[x] - \sin[x] \right) + \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) + \sin[x] == 0$$

You can get the derivatives to be applied automatically by changing the second argument of DSolve or DSolveValue. This argument defines the dependent variable. Changing $y[x]$ to y will change the form of the output:

```
In[ ]:= solDSolveValue = DSolveValue[{eqn, cond}, y, {x, -π, 3 π}]
solDSolve = DSolve[{eqn, cond}, y, {x, -π, 3 π}]
```

$$\text{Out[]} = \text{Function}[\{x\}, \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right)]$$

$$\text{Out[]} = \left\{ \left\{ y \rightarrow \text{Function}[\{x\}, \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right)] \right\} \right\}$$

Since the head of the function y is defined in the rule, rather than the $y[x]$ function itself, the derivatives are automatically applied. This is very convenient when verifying the solution of a complicated equation:

```
In[ ]:= eqn /. solDSolve[[1]]
```

$$\text{Out[]} = \frac{1}{2} \left(e^{-x} - \cos[x] - \sin[x] \right) + \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) + \sin[x] == 0$$

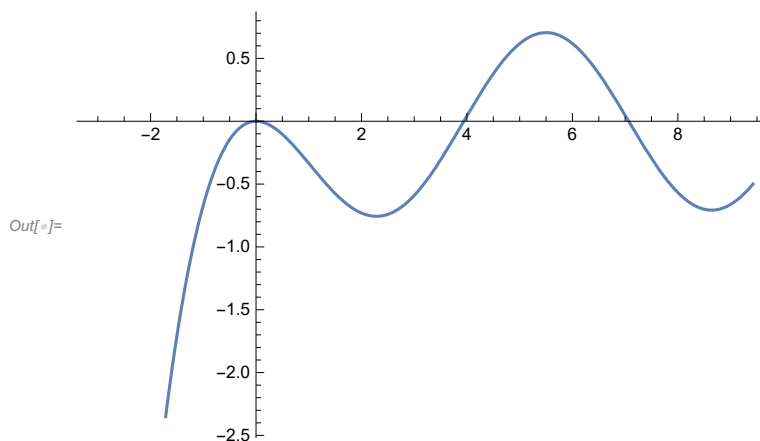
You still need to define the replacement rule with the DSolveValue result:

```
In[ ]:= eqn /. y → solDSolveValue
```

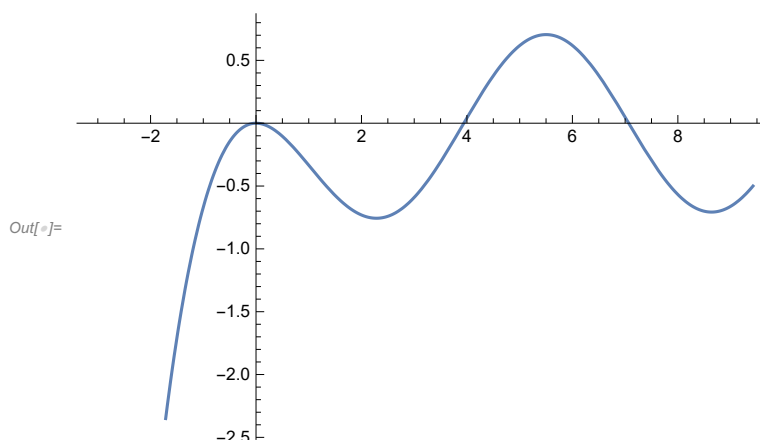
$$\text{Out[]} = \frac{1}{2} \left(e^{-x} - \cos[x] - \sin[x] \right) + \frac{1}{2} \left(-e^{-x} + \cos[x] - \sin[x] \right) + \sin[x] == 0$$

Plotting is done in the same way, only the argument x must be given to the functions explicitly:

```
In[ ]:= Plot[solDSolveValue[x], {x, -π, 3 π}]
```




```
In[ ]:= Plot[y[x] /. solDSolve[[1]], {x, -π, 3π}]
```




Equations with Multiple Solutions


NDSolveValue will only return one solution. If multiple solution branches exist, then NDSolveValue will only return the first solution and issue a warning message about the missing solutions:


```
In[ ]:= NDSolveValue[{y'[x]^2 - y[x]^2 == 0, y[0]^2 == 4}, y[x], {x, 1}]
```


NDSolveValue: There are multiple solution branches for the equations, but NDSolveValue will return only one. Use NDSolve to get all of the solution branches.


```
Out[ ]:= InterpolatingFunction[ Domain: {{0., 1.}}  
Output: scalar][x]
```

```
In[ ]:= NDSolve[{y'[x]^2 - y[x]^2 == 0, y[0]^2 == 4}, y[x], {x, 1}]
```

```
Out[ ]:= {{y[x] → InterpolatingFunction[ Domain: {{0., 1.}}  
Output: scalar][x]},
```

```
{y[x] → InterpolatingFunction[ Domain: {{0., 1.}}  
Output: scalar][x]},
```

```
{y[x] → InterpolatingFunction[ Domain: {{0., 1.}}  
Output: scalar][x]},
```

```
{y[x] → InterpolatingFunction[ Domain: {{0., 1.}}  
Output: scalar][x]} }
```

```
In[ ]:= Clear["Global`*"]
```

Specifying Derivative Boundary Conditions

There are several ways that derivative boundary conditions can be entered. Consider the following equation and condition:

```
In[ ]:= eqn = f''[x] + f[x] == 0;
cond = f[0] == 1;
```

Suppose you would also like to specify the first derivative at $x = 0$ as an additional condition. It may be tempting to use the following syntax, but this does not work:

```
In[ ]:= D[f[0], x]
```

You need to take the derivative first with respect to a general x , then set $x = 0$ afterward:

```
In[ ]:= D[f[x], x] /. x -> 0
```

```
Out[ ]:= f'[0]
```

This should be used with caution, as multivariate functions do not accept the prime notation. The **D** function should be used in the multivariate case:

```
In[ ]:= D[f[x, y], x] /. x -> 0
```

```
Out[ ]:= f(1,0)[0, y]
```

Another alternative is to use the **Derivative** function. This allows you to take the derivative without needing to set $x = 0$ afterward and works with multivariate functions:

```
In[ ]:= Derivative[1][f][0]
Derivative[1, 0][f][0, y]
```

```
Out[ ]:= f'[0]
```

```
Out[ ]:= f(1,0)[0, y]
```

Finally, the partial derivative notation works as well:

```
In[ ]:= ∂t,t ϕ[x, t] /. t -> 0
```

```
Out[ ]:= ϕ(0,2)[x, 0]
```

All four methods are valid for representing derivative boundary conditions in the single-variate case:

```
In[ ]:= NDSolve[{eqn, cond, (D[f[x], x] /. x -> 0) == 0}, f, {x, 0, 10}];
NDSolve[{eqn, cond, f'[0] == 0}, f, {x, 0, 10}];
NDSolve[{eqn, cond, Derivative[1][f][0] == 0}, f, {x, 0, 10}];
NDSolve[{eqn, cond, (∂x f[x] /. x -> 0) == 0}, f, {x, 0, 10}];
```

Notice that parentheses had to be used in $(D[f[x], x] /. x \rightarrow 0) == 0$ to control the order of operations.

The syntax is analogous when working in the multivariate case:

```
In[ ]:= L = 4;
eqns = {
  D[u[t, x, y], {t, 2}] == D[u[t, x, y], x, x] + D[u[t, x, y], y, y] + Sin[u[t, x, y]],
  u[t, -L, y] == u[t, L, y],
  u[t, x, -L] == u[t, x, L],
  u[0, x, y] == Exp[-(x^2 + y^2)]
};
```

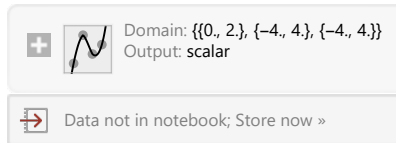
```

In[ ]:= NDSolve[{eqns, (D[u[t, x, y], t] /. t -> 0) == 0}, u, {t, 0, L/2}, {x, -L, L}, {y, -L, L}];
NDSolve[{eqns, Derivative[1, 0, 0][u][0, x, y] == 0},
  u, {t, 0, L/2}, {x, -L, L}, {y, -L, L}];
sol = NDSolve[{eqns, (D[u[t, x, y], t] /. t -> 0) == 0},
  u[t, x, y], {t, 0, L/2}, {x, -L, L}, {y, -L, L}];

In[ ]:= res = u[t, x, y] /. sol[[1]]

```

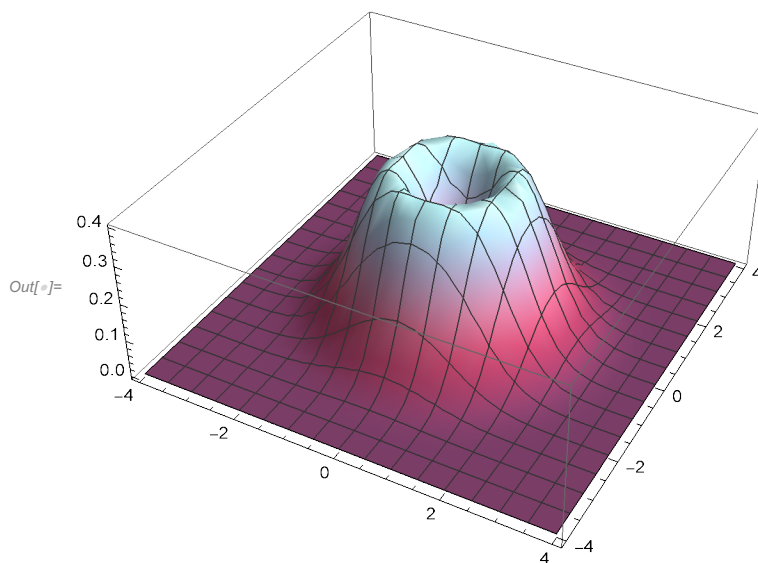
```
Out[ ]:= InterpolatingFunction[
```



```

In[ ]:= Plot3D[Evaluate[res /. t -> 1], {x, -4, 4}, {y, -4, 4},
  ColorFunction -> "CandyColors", PlotPoints -> 25, PlotRange -> All]

```



```
In[ ]:= Clear["Global`*"]
```

Automatic Adaptive Step Size

All numerical methods must discretize the independent variables with a certain step size. A larger step size results in a faster calculation but can decrease the accuracy of the result. Adaptive step-size methods try to balance and readjust these conflicting requirements along the integration path.

Adaptive Step Sizes in NDSolve

NDSolve uses an adaptive procedure to determine the size of the steps in the independent variable. In general, if the solution varies rapidly in a particular region, NDSolve will reduce the step size to be able to better track the solution, then increase the step size when the solution varies more slowly.

Consider the following second-order ODE:

```


In[ ]:= eqn = y''[x] + Sin[(1 + 50 Exp[-10 x^2]) x] == 0;
cond = {y[0] == 1, y'[0] == 1/100};
sol = NDSolveValue[{eqn, cond}, y[x], {x, -1, 1}]

```

```

Out[ ]:= InterpolatingFunction[

```

 Domain: {{-1., 1.}}
Output: scalar

```

] [x]

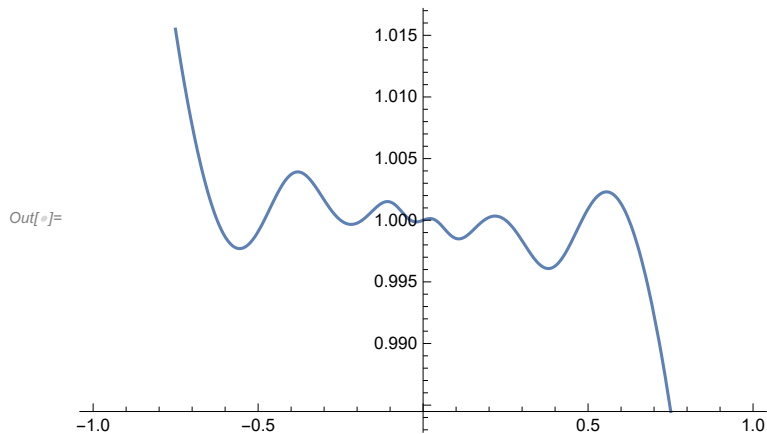
```

Visualizing the solution:

```

In[ ]:= Plot[sol, {x, -1, 1}]

```

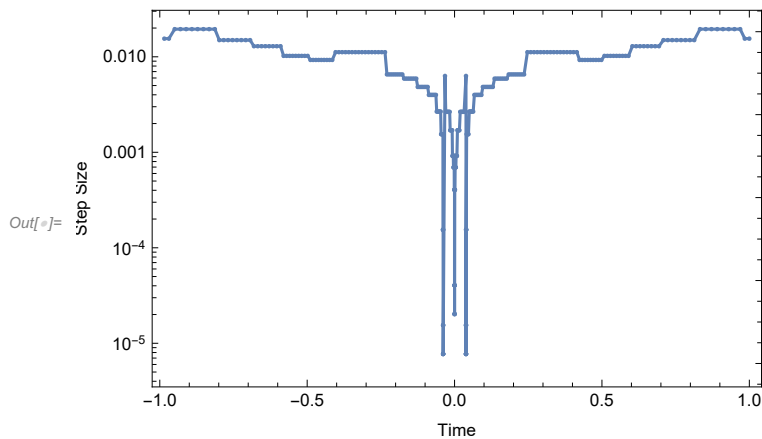


The step sizes NDSolve used while solving the problem can be visualized with the **DifferentialEquations`NDSolveUtilities`** package, which gives access to the **StepDataPlot** function:

```

In[ ]:= Needs["DifferentialEquations`NDSolveUtilities`"]
StepDataPlot[sol, FrameLabel -> {"Time", Rotate["Step Size", Pi/2]}]

```



```

In[ ]:= Names["DifferentialEquations`NDSolveUtilities`*"]
Out[ ]:= {CompareMethods, FinalSolutions, InvariantDimensions,
InvariantErrorFunction, InvariantErrorPlot, InvariantErrorSampleRate,
RungeKuttaLinearStabilityFunction, StepDataCount, StepDataPlot}
In[ ]:= Clear["Global`*"]

```

NDSolve Options

NDSolve has many built-in options that allow the user to control the method that is used to solve a problem, the accuracy of the final result, the precision of the numbers used in the calculation and the maximum step size to use, and to monitor the calculation as it happens.

Options of particular importance are:

```
In[ ]:= CellPrint@ExpressionCell[Grid[{
  {Style["", Bold, 20], SpanFromLeft},
  {}},
  {Style["Option Name", Italic, 16],
   Style["Default Value", Italic, 16], Style["Description", Italic, 16]}},
  {"Method", "Automatic", "Specifies the method to use with NDSolve."},
  {"MaxSteps", "Automatic", "Maximum number of steps to take."},
  {"MaxStepSize", "Automatic", "Maximum size of each step."},
  {"StartingStepSize", "Automatic",
   "Specifies the initial step size to use in trying to generate results."},
  {"AccuracyGoal", "Automatic",
   "Specifies the absolute tolerance to use in solving the nonlinear system."},
  {"PrecisionGoal", "Automatic",
   "Specifies the relative tolerance to use in solving the nonlinear system."},
  {"WorkingPrecision", "MachinePrecision", "Specifies how many digits of
   precision should be maintained in internal computations. "},
  {"InterpolationOrder", "Automatic", "Specifies what order of
   interpolation to use."},
  {"EvaluationMonitor", "None", "Gives an expression to evaluate whenever
   functions derived from the input are evaluated numerically. "},
  {"StepMonitor", "None", "Gives an expression to evaluate whenever
   a step is taken by the numerical method used. "},
  {"", SpanFromLeft}
}],
Background → {None, {None, {LightBlue}, None}},
Dividers →
  {{False}, {False, {2 → {RGBColor[0.85, 0.5, 0.4], Thick}, 4 → GrayLevel[0.6]}}},
Spacings → {3, 1},
ItemSize → {{10, 10, 40}},
Alignment → Left], "Output", "GeneratedCell" → False]
```


<i>Option Name</i>	<i>Default Value</i>	<i>Description</i>
Method	Automatic	Specifies the method to use with NDSolve.
MaxSteps	Automatic	Maximum number of steps to take.
MaxStepSize	Automatic	Maximum size of each step.
StartingStepSize	Automatic	Specifies the initial step size to use in tr
AccuracyGoal	Automatic	Specifies the absolute tolerance to use in solving the nonlinear system.
PrecisionGoal	Automatic	Specifies the relative tolerance to use in solving the nonlinear system.
WorkingPrecision	MachinePrecision	Specifies how many digits of precision should be maintained in internal computati
InterpolationOrder	Automatic	Specifies what order of interpolation to use
EvaluationMonitor	None	Gives an expression to evaluate whenever fun derived from the input are evaluated numer
StepMonitor	None	Gives an expression to evaluate whenever a step is taken by the numerical method us

Example: Controlling Maximum Step Size

The step size can sometimes become quite large for some values of the independent variable. Large step sizes are one possible source of error in a solution.

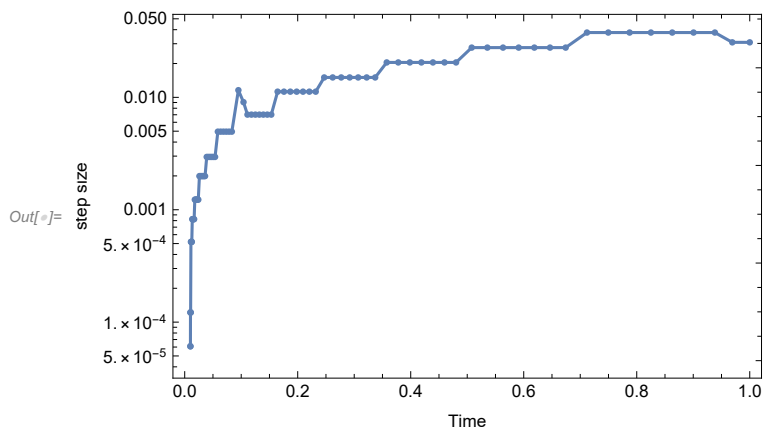
Solve with the default options in NDSolve:

```
In[ ]:= eqn = y''[x] + Sin[Log[x]] == 0;
cond = {y[1/100] == 1, y'[1/100] == -1};
sol = NDSolve[{eqn, cond}, y[x], {x, 1/100, 1}][[1]]
```

```
Out[ ]:= {y[x] → InterpolatingFunction[ Domain: {{0.01, 1.}} Output: scalar][x]}
```

```
In[ ]:= Needs["DifferentialEquations`NDSolveUtilities`"] // Quiet
```

```
In[ ]:= StepDataPlot[sol, FrameLabel -> {"Time", Rotate["step size", Pi/2]}]
```



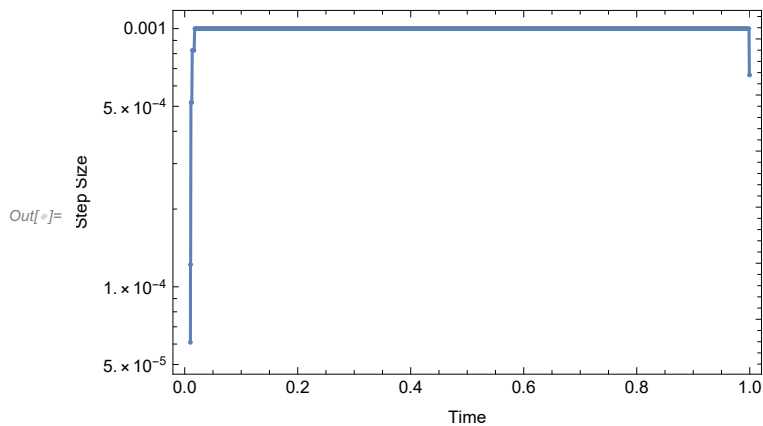
Specifying **MaxStepSize** will prevent the step size from getting too big. This comes with a computation cost, but may resolve some problems with numerical errors.

Solve the equation, but with a smaller maximum step size:

```
In[ ]:= sol = NDSolve[{eqn, cond}, y[x], {x, 1/100, 1}, MaxStepSize -> 0.001];
```

Check the step sizes that were used. Notice the change in scale on the plot:

```
In[ ]:= StepDataPlot[sol, FrameLabel -> {"Time", Rotate["Step Size", Pi/2]}]
```



```
In[ ]:= Clear["Global`*"]
```

Summary

The Wolfram Language is a powerful tool for working with and solving differential equations. This section of the course covered:

- Solving differential equations in the Wolfram Language
- The basics of using NDSolve
- Specifying boundary/initial conditions

- Using the options to control the numerical accuracy of the solution

Further Reading

Tutorials

- **Advanced Numerical Differential Equation Solving in the Wolfram Language**
- **Numerical Solution of Differential Equations**

Guides

- **Differential Equations**
- **Partial Differential Equations**
- **Differential Operators**

Glossary

AsymptoticDSolveValue	D	Derivative	DSolve	DSolveValue
FullSimplify	Grid	Integrate	MaxStepSize	NDSolve
NDSolveValue	Plot	StepDataPlot	TraditionalForm	