

# Introduction to Differential Algebraic Equations

Differential algebraic equations (DAEs) occur in many applications. For example, when solving a PDE with the method of lines or solving a differential equation with an algebraic constraint (pendulum). DAEs have certain difficulties that do not occur with ODEs.

Both a system of ODEs and DAEs can be written in the form:

$$F(t, x(t), x'(t)) = 0$$

where  $F$  and  $X$  are vector valued. When working with ODEs, the system can be written in normal form, where the highest-order derivative is solved for:

$$x'(t) = f(t, x(t))$$

With a system of DAEs, rewriting the system in normal form is not directly possible. Therefore, different numerical techniques need to be applied.

Generally, a system of DAEs can be converted to a system of ODEs by differentiating it with respect to the independent variable. This is a process called index reduction.

---

## Index of a DAE

For a general DAE system, the index of the system refers to the minimum number of differentiations that would be required to solve for all  $X'(t)$  uniquely in terms of  $X(t)$  and  $t$ .

Consider the following DAE. You can recognize this as a DAE since no derivatives of  $y(t)$  are in the equation:

$$\begin{aligned}x''(t) + y(t) &= \cos(t) \\ x(t) &= \cos(t)\end{aligned}$$

In order to get a differential equation for  $y(t)$ , the first equation must be differentiated once. This leads to the new system:

$$\begin{aligned}x'''(t) + y'(t) &= -\sin(t) \\ x(t) &= \cos(t)\end{aligned}$$

The differentiated system now contains  $x'''$ , which must be accounted for. To get an equation for  $x'''$ , the second equation must now be differentiated three times. This leads to:

$$\begin{aligned}x''''(t) + y'(t) &= -\sin(t) \\ x'''(t) &= \sin(t)\end{aligned}$$

Since part of the system had to be differentiated three times, the index of the DAE is 3.

The final differentiated system is said to be an index-0 system. The resulting system of ODEs obtained from index reduction is:

$$x'''(t) + y'(t) = -\sin(t)$$

$$y'(t) = -2 \sin(t)$$

It is typical to reduce the system to either index 0 or index 1, since the underlying integration routines can handle them efficiently.

## Viewing the Index-Reduced System

**NDSolve`ProcessEquations** can be used to view the index-reduced system. You can read about functions like **NDSolve`ProcessEquations** in the **Components and Data Structures** tutorial.

For example, with the preceding system:

```
In[ ]:= eqns = {x'''[t] + y'[t] == Cos[t], x[t] == Cos[t]};
```

```
In[ ]:= state = First@
```

```
  NDSolve`ProcessEquations[eqns, {x, y}, t, Method -> {"IndexReduction" -> Automatic}];
  residual = state["NumericalFunction"] ["FunctionExpression"];
  MatrixForm[residual[[2]]]
```

Out[ ]//MatrixForm=

$$\begin{pmatrix} dd2x + \text{Cos}[t] \\ dd1x + \text{Sin}[t] \\ x - \text{Cos}[t] \\ dd2x + y - \text{Cos}[t] \end{pmatrix}$$

The mapping between variables like  $x'' \rightarrow dd2x$  can also be found:

```
In[ ]:= Flatten[MapThread[Thread[Rule[##]] &, {state["Variables"], state["WorkingVariables"]}]]
```

```
Out[ ]:= {t -> t, x -> x, x' -> dd1x, x'' -> dd2x, y -> y,
  x' -> x$6161, x'' -> dd1x$6161, x^(3) -> dd2x$6161, y' -> y$6161}
```

The derivatives are represented by a new set of variables that are treated as unique algebraic quantities. The reason for doing this is beyond the scope of this course, but more information can be found in the section on the “dummy derivative” method in the **DAE tutorial**.

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

## Index Reduction Methods

The DAE solver methods built into **NDSolve** work with index-1 and index-0 systems. Thus, for higher-index systems, index reduction is necessary.

When a system is found to have an index greater than 1, **NDSolve** generates a message:

```
In[ ]:= eqns = 
$$\begin{pmatrix} x1'[t] == \cos[t] - x2[t] + x3[t] \\ x2'[t] == \cos[t] - x3[t] + x4[t] \\ x3'[t] == \cos[t] - x4[t] + x5[t] \\ x4'[t] == \cos[t] - x5[t] + x6[t] \\ x5'[t] == \cos[t] + \sin[t] - x6[t] \\ x1[t] + x2[t] + x3[t] + x4[t] + x5[t] - 5 \sin[t] == 0 \end{pmatrix};$$

```

```
In[ ]:= ics = {x1[0] == x2[0] == x3[0] == x4[0] == x5[0] == x6[0] == 0};
```

```
In[ ]:= vars = {x1[t], x2[t], x3[t], x4[t], x5[t], x6[t]};
```

```
In[ ]:= NDSolve[{eqns, ics}, vars, {t, 0, 4 Pi}];
```

**NDSolve:** The DAE solver failed at  $t = 0.$ . The solver is intended for index 1 DAE systems and structural analysis indicates that the DAE index is 2. The option `Method->{"IndexReduction"->Automatic}` may be used to reduce the index of the system.

This is an index-4 system, which has the exact solution  $\sin(t)$  for all variables:

```
In[ ]:= DSolve[{eqns, ics}, vars, {t, 0, 4 Pi}][[1]] // Column
```

```
Out[ ]:= 
$$\begin{matrix} x1[t] \rightarrow \sin[t] \\ x2[t] \rightarrow \sin[t] \\ x3[t] \rightarrow \sin[t] \\ x4[t] \rightarrow \sin[t] \\ x5[t] \rightarrow \sin[t] \\ x6[t] \rightarrow \sin[t] \end{matrix}$$

```

NDSolve uses two methods for index reduction: **"Pantelides"** and **"StructuralMatrix"**.

## Pantelides

The Pantelides algorithm is efficient. However, it can sometimes underestimate the index of the system, which prevents the problem from being solved. In this case, it estimates this index-4 system to be index 2:

```
In[ ]:= NDSolve[{eqns, ics}, vars, {t, 0, 4 Pi}, Method -> {"IndexReduction" -> "Pantelides"}];
```

**NDSolve:** Repeated convergence test failure at  $t == 0.$ ; unable to continue.

## Structural Matrix

The structural matrix method is more general, and may be able to resolve systems where the Pantelides algorithm fails. However, this comes at the cost of higher computational complexity:

```
In[ ]:= NDSolve[{eqns, ics}, vars, {t, 0, 4 Pi}, Method -> {"IndexReduction" -> "StructuralMatrix"}]
```

```
Out[ ]:= { {x1[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t],  
x2[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t],  
x3[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t],  
x4[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t],  
x5[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t],  
x6[t] -> InterpolatingFunction[ Domain: {{0., 12.6}} Output: scalar] [t] } }
```

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

## Equation Simplification

In order to solve a DAE after it has been index reduced, NDSolve converts the system into one of three forms. This step is quite important, since different integration methods are chosen depending on how the system is constructed.

To demonstrate, consider the system:

$$x'(t) + y'(t) + z'(t) = \sin(t)$$

$$y'(t) + z'(t) = \sin(t) + x(t)$$

$$x'(t) + z'(t) = \cos(t) + y(t)$$

NDSolve automatically tries to put the system into normal form by first attempting to solve for the highest-order derivate explicitly with **LinearSolve**. If that fails, then the system is solved symbolically using **Solve**:

```
In[ ]:= eqn = {x'[t] + y'[t] == Sin[t] - z'[t], y'[t] + z'[t] == Sin[t] + x[t],  
x'[t] + z'[t] == Cos[t] + y[t]};  
cond = {x[0] == 1, y[0] == 1, z[0] == 1};
```

To see what NDSolve is doing behind the scenes, NDSolve`ProcessEquations is helpful. You can read about functions like NDSolve`ProcessEquations in the **Components and Data Structures** tutorial:

```
In[ ]:= state = First@NDSolve`ProcessEquations[{eqn, cond}, {x, y, z}, t];
Partition[state["Properties"], UpTo[4]] // TableForm
```

```
Out[ ]//TableForm=
```

ActiveDirection	Caller	FiniteElementData	IndependentVariables
IntegrationDirection	MassMatrix	MaxSteps	Norm
NumericalFunction	OptionValues	OriginalCaller	Parameters
Properties	SystemSize	TemporalVariable	VariableDimensions
VariablePositions	VariableRanges	VariableTransformation	WorkingPrecision

To view the form NDSolve put the DAE into, the "NumericalFunction" option of the state variable should be used with the property "FunctionExpression":

```
In[ ]:= Partition[state["NumericalFunction"] ["Properties"], UpTo[3]] // TableForm
```

```
Out[ ]//TableForm=
```

ArgumentDimensions	ArgumentNames	ArgumentUnits
CompiledFunction	FunctionExpression	InputIndexes
InputTypes	Properties	ResultUnits
SolutionDataComponents	WorkingPrecision	

The three options of "EquationSimplification" put the equation into one of three forms.

## Solve

This is the default method, which attempts to solve for the highest-order derivative explicitly. This transforms the DAE into a system of ODEs. Robust and efficient numerical methods can then be taken advantage of to solve the system of ODEs:

```
In[ ]:= state = First@NDSolve`ProcessEquations[{eqn, cond},
{ x, y, z }, t, Method -> {"EquationSimplification" -> "Solve"}];

In[ ]:= state["NumericalFunction"] ["FunctionExpression"] [t, x, y, z] // Column
```

```
Out[ ]:=
```

$$\begin{aligned} & -x \\ & -y - \cos[t] + \sin[t] \\ & x + y + \cos[t] \end{aligned}$$

Compare with using the Solve function explicitly:

```
In[ ]:= First[Solve[eqn, {x'[t], y'[t], z'[t]}]] // Column
```

```
Out[ ]:=
```

$$\begin{aligned} x'[t] & \rightarrow -x[t] \\ y'[t] & \rightarrow -\cos[t] + \sin[t] - y[t] \\ z'[t] & \rightarrow \cos[t] + x[t] + y[t] \end{aligned}$$


## "TimeConstraint" Option

The "TimeConstraint" option defines the time in seconds to allow Solve to explicitly solve for the highest-order derivatives. By default, "TimeConstraint" is set to 1 second. Increasing the time constraint can sometimes resolve a common error message when solving DAEs:

```
In[ ]:= eqn = {x'[t] + y'[t]^2 + z'[t]^2 == Sin[t], x'[t] * y'[t] + z'[t] == Sin[t] + x[t],
x'[t] + z'[t] == y[t] + Cos[t]};
```

```
In[ ]:= cond = {x[0] == 1, y[0] == -1, z[0] == 1};
```




```
In[ ]:= NDSolve[{eqn, cond}, {x, y, z}, {t, 0, 1}]
```

 **NDSolve**: Cannot solve to find an explicit formula for the derivatives. Consider using the option  
Method->{"EquationSimplification"->"Residual"}.

```
Out[ ]:= NDSolve[{ {x'[t] + y'[t]^2 + z'[t]^2 == Sin[t], x'[t] y'[t] + z'[t] == Sin[t] + x[t],  
x'[t] + z'[t] == Cos[t] + y[t] }, {x[0] == 1, y[0] == -1, z[0] == 1} }, {x, y, z}, {t, 0, 1}]
```

The error message suggests **Method** -> {"EquationSimplification" -> "Residual"}. This method does work, but due to the nature of the method, only one of the multiple solutions is found:

```
In[ ]:= NDSolve[{eqn, cond}, {x, y, z}, {t, 0, 1}, Method -> {"EquationSimplification" -> "Residual"}]
```

```
Out[ ]:= { {x -> InterpolatingFunction[ Domain: {{0, 1}}  
Output: scalar ],  
  
y -> InterpolatingFunction[ Domain: {{0, 1}}  
Output: scalar ],  
  
z -> InterpolatingFunction[ Domain: {{0, 1}}  
Output: scalar ] ] }
```


Increasing the time constraint gives **Solve** more time to solve for the derivative terms explicitly. As a result, all branches for the solution set are found:

```
In[ ]:= NDSolve[{eqn, cond}, {x, y, z}, {t, 0, 1},
  Method -> {"EquationSimplification" -> {"Solve", "TimeConstraint" -> 10}}]
```


**NDSolve:** Maximum number of 116222 steps reached at the point  $t == 0.00019737558639288739$ .


Out[ ]:= { {x -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,


y -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,


z -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] } ,

{x -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,


y -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,


z -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] } ,


{x -> InterpolatingFunction[ Domain: {{0., 0.000197}} Output: scalar ] ,

 Data not in notebook; Store now »

y -> InterpolatingFunction[ Domain: {{0., 0.000197}} Output: scalar ] ,


 Data not in notebook; Store now »

z -> InterpolatingFunction[ Domain: {{0., 0.000197}} Output: scalar ] } ,

 Data not in notebook; Store now »













{x -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,

y -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] ,

z -> InterpolatingFunction[ Domain: {{0, 1}} Output: scalar ] } }

One of these branches has numerical problems near  $t = 0$  due to a stiff system, which is the cause of the error message. This message can be resolved by using the **"StiffnessSwitching"** method:

```
In[ ]:= NDSolve[{eqn, cond}, {x, y, z}, {t, 0, 1}, Method →  
{"StiffnessSwitching", {"EquationSimplification" → {"Solve", "TimeConstraint" → 10}}}]
```

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

The resulting equations from the **"EquationSimplification"** method are quite complicated, which explains the need for an increased **TimeConstraint**. `NDSolve`ProcessEquations` will explicitly display the equations that are found:



```

In[ ]:= state = First@NDSolve`ProcessEquations[{eqn, cond}, {x, y, z}, {t, 0, 1},
      Method → {"EquationSimplification" → {"Solve", "TimeConstraint" → 10}}];
eqns = state["NumericalFunction"] ["FunctionExpression"]

```

Out[ ]:=

$$\text{Function}[\{t, x, y, z\}, \left\{ \frac{\dots 117 \dots + \frac{1}{4} \sqrt{\dots 7 \dots + \frac{\dots 1 \dots}{\dots 1 \dots} + \frac{(\dots 1 \dots)^{1/3}}{3 \cdot 2 \cdot 1}} \sqrt{\dots 1 \dots} - \frac{1}{8} (\dots 1 \dots)^{3/2}}{-x+y+\cos[t]-\sin[t]}, \right. \\ \left. \frac{1}{2} - \frac{1}{2} \sqrt{\dots 8 \dots + \dots 1 \dots} - \frac{1}{2} \sqrt{\dots 1 \dots}, \frac{x}{4} - \frac{y}{4} + \dots 113 \dots + \frac{1}{4} \sqrt{\dots 1 \dots} \sqrt{\dots 1 \dots} - \frac{1}{8} (\dots 1 \dots)^{3/2} \right\}]$$

large output

show less

show more

show all

set size limit...

## Mass Matrix

Generate a system of the form  $Mx' = F(x, t)$  using the option "MassMatrix". This option is particularly useful for taking advantage of efficient linear algebra functionality. However, difficulties may arise if the mass matrix is singular:

```

In[ ]:= eqn = {x'[t] + y'[t] == Sin[t] - z'[t], y'[t] + z'[t] == Sin[t] + x[t],
      x'[t] + z'[t] == y[t] + Cos[t]};
cond = {x[0] == y[0] == z[0] == 1};

```

```

In[ ]:= state = First@NDSolve`ProcessEquations[{eqn, cond},
      {x, y, z}, t, Method → {"EquationSimplification" → "MassMatrix"}];

```

```

In[ ]:= state["NumericalFunction"] ["FunctionExpression"] [t, x, y, z]

```

```

Out[ ]:= {Sin[t], x + Sin[t], y + Cos[t]}

```

```

In[ ]:= state["MassMatrix"] // MatrixForm

```

```

Out[ ]:= MatrixForm

```

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

The system  $Mx' = F(x, t)$  that is to be solved is given by:

```

In[ ]:= MatrixForm[Rationalize[state["MassMatrix"].{x'[t], y'[t], z'[t]}] ==
      MatrixForm[{Sin[t], x + Sin[t], y + Cos[t]}]

```

$$\text{Out[ ]:=} \begin{pmatrix} x'[t] + y'[t] + z'[t] \\ y'[t] + z'[t] \\ x'[t] + z'[t] \end{pmatrix} = \begin{pmatrix} \sin[t] \\ x + \sin[t] \\ y + \cos[t] \end{pmatrix}$$

## Residual

The "Residual" method generates a system of the form  $F(x, x', t) = 0$ .

This method is quite general and can be used to solve most DAEs. However, consistent initialization of the DAE can be difficult using this method, as there can be hidden boundary conditions that also need to be satisfied. This is discussed later:

```
In[ ]:= state = First@NDSolve`ProcessEquations[{eqn, cond},
      {x, y, z}, t, Method -> {"EquationSimplification" -> "Residual"}];
state["NumericalFunction"] ["FunctionExpression"]

Out[ ]:= Function[{t, x, y, z, x$25458, y$25458, z$25458}, {x$25458 + y$25458 + z$25458 - Sin[t],
      -x + y$25458 + z$25458 - Sin[t], x$25458 - y + z$25458 - Cos[t]}]
```

When the system is put in a residual form, the derivatives are represented by a new set of variables that are treated as unique algebraic quantities. The reason for doing this is beyond the scope of this course, but more information can be found in the section on the “dummy derivative” method in the DAE tutorial.

```
Clear["Global`*"]
```

---

## DAE Initialization

Finding consistent initial conditions that satisfy all necessary consistency conditions can be one of the most difficult parts of solving a DAE system. Not only must the initial conditions satisfy the DAE itself, but they must also satisfy derivatives of the DAE as well.

### Example

Try to find the unique set of initial conditions for the following DAE:

$$x_2'(t) + x_1(t) = \sin(t)$$

$$x_3'(t) + x_2(t) = \sin(t)$$

$$x_3(t) = \cos(t)$$

Differentiate the third equation once:

$$x_3'(t) = -\sin(t).$$

Substituting into the second equation:

$$x_2(t) = 2 \sin(t).$$

Differentiating  $x_2(t)$  and substituting into the first equation gives:

$$x_1(t) = -2 \cos(t) + \sin(t).$$

Therefore, at  $t = 0$ , the only consistent set of initial conditions is  $x_1(0) = -2$ ,  $x_2(0) = 0$ ,  $x_3(0) = 1$ .

While there are many initial conditions that satisfy the equation  $F(t, x(0), x'(0)) = 0$ , the conditions found in the preceding are the only consistent set of initial conditions.

Since the constraints on the initial conditions are not always obvious, the problem of finding consistent initial conditions can be quite challenging.

---

## DAE Initialization Methods

For high-index systems, NDSolve automatically tries to initialize the system using the collocation algorithm. For larger systems, the QR or BLT initialization algorithm may be more appropriate.

## Collocation

Collocation is the most general method. This method tries to satisfy the residual equation near the point of initialization. This allows the method to be used to initialize high-index systems. The collocation algorithm does not perform index reduction and operates on the original equations.

A detailed discussion of the Collocation method, as well as the QR and BLT methods, is presented in the DAE tutorial.

## QR

For an  $n$ -variable DAE system, the QR algorithm needs to handle at most a matrix of size  $n$ , as opposed to a matrix of size  $n^2$  when the collocation method is used. This makes the method quite efficient in handling very large systems.

## BLT

The BLT (Block Lower Triangular) method examines the structure of the system and splits the system into a number of smaller subsystems that can be solved efficiently. As a result, much smaller matrices are dealt with (compared to the entire system), and thus it is computationally efficient for large systems.


## Comparison of Methods

For cases with inconsistent initial conditions, these methods can modify the initial conditions to find a consistent set of initial conditions. The results obtained from the various methods may be different, since multiple consistent sets of initial conditions may be possible.

Consider the pendulum example with inconsistent initial conditions. The initial condition  $x[0] = 2$  violates the constraint  $x^2 + y^2 = 1$ :

```
In[ ]:= PendulumDAE = {
  x''[t] == x[t] * λ[t],
  y''[t] == y[t] * λ[t] - 1,
  x[t]^2 + y[t]^2 == 1
};

In[ ]:= solQR = NDSolveValue[{PendulumDAE, x[0] == 2, y'[0] == 1}, {x, y, λ}, {t, 0, 1},
  Method -> {"IndexReduction" -> True, "DAEInitialization" -> "QR"}];
{Through[solQR[0]], D[Through[solQR[t]], t] /. t -> 0}
```

 **NDSolveValue:** NDSolve has computed initial values that give a zero residual for the differential–algebraic system, but some components are different from those specified. If you need them to be satisfied, giving initial conditions for all dependent variables and their derivatives is recommended.

```
Out[ ]:= {{0.925965, 0.37761, 0.268815}, {-0.124551, 0.30542, 0.}}
```

Solve this system using the BLT algorithm:

```
In[ ]:= solBLT = NDSolveValue[{PendulumDAE, x[0] == 2, y'[0] == 1}, {x, y, λ},
  {t, 0, 1}, Method → {"IndexReduction" → True, "DAEInitialization" → "BLT"}];
{Through[solBLT[0]], D[Through[solBLT[t]], t] /. t → 0}
```

... **NDSolveValue**: NDSolve has computed initial values that give a zero residual for the differential–algebraic system, but some components are different from those specified. If you need them to be satisfied, giving initial conditions for all dependent variables and their derivatives is recommended.

```
Out[ ]:= {{0.000153905, -1., -1.00007}, {0.00856202, 1.31774 × 10-6, 3.98657 × 10-6}}
```

In this particular example, the initial conditions found by the BLT algorithm are the same as those found by the more general Collocation algorithm:

```
In[ ]:= solCollocation = NDSolveValue[{PendulumDAE, x[0] == 2, y'[0] == 1}, {x, y, λ},
  {t, 0, 1}, Method → {"IndexReduction" → True, "DAEInitialization" → "Collocation"}];
{Through[solCollocation[0]], D[Through[solCollocation[t]], t] /. t → 0}
```

... **NDSolveValue**: NDSolve has computed initial values that give a zero residual for the differential–algebraic system, but some components are different from those specified. If you need them to be satisfied, giving initial conditions for all dependent variables and their derivatives is recommended.

```
Out[ ]:= {{0.000153905, -1., -1.00007}, {0.00856202, 1.31774 × 10-6, 3.98657 × 10-6}}
```

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

## IDA Method for Solving DAEs

Once the DAE has been index reduced and properly initialized, it can finally be solved.

NDSolve uses the Implicit Differential-Algebraic (IDA) package, which is part of **SUNDIALS** (SUite of Nonlinear and Differential/ALgebraic Equation Solvers) developed at the Center for Applied Scientific Computing of Lawrence Livermore National Laboratory.

IDA solves DAEs with backward differentiation formula methods of orders 1 through 5, implemented using a variable-step form. You can read about the numerical details of this method and its options in the **IDA tutorial**.

## Summary

Differential algebraic equations (DAEs) occur naturally when solving PDEs or differential equations with algebraic constraints. DAEs have certain computational difficulties not present with ODEs or PDEs. This section covered:

- Properties of DAEs
- Index-reduction methods
- Methods to simplify the form of a DAE
- Finding consistent initial conditions

## 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

"BLT"	"Collocation"	DSolve	"EquationSimplification"	"IndexReduction"
"MassMatrix"	NDSolve	NDSolve`ProcessEquations	"NumericalFunction"	Pantelides
"QR"	"Residual"	Solve	"StiffnessSwitching"	"StructuralMatrix"
"TimeConstraint"				