



WOLFRAM LANGUAGE & SYSTEM  
Documentation Center



TUTORIAL

...

# Numerical Solution of Boundary Value Problems (BVP)

- ✓ "Shooting" Method
- ✓ Boundary Value Problems with Parameters
- ✓ "Chasing" Method



## "Shooting" Method

The shooting method works by considering the boundary conditions as a multivariate function of initial conditions at some point, reducing the boundary value problem to finding the initial conditions that give a root. The advantage of the shooting method is that it takes advantage of the speed and adaptivity of methods for initial value problems. The disadvantage of the method is that it is not as robust as finite difference or collocation methods: some initial value problems with growing modes are inherently unstable even though the BVP itself may be quite well posed and stable.

Consider the BVP system

$$X'(t) = F(t, X(t)); G(X(t_1), X(t_2), \dots, X(t_n)) = 0, t_1 < t_2 < \dots < t_n.$$

The shooting method looks for initial conditions  $X(t_0) = c$  so that  $G = 0$ . Since you are varying the initial conditions, it makes sense to think of  $X = X_c$  as a function of them, so shooting can be thought of as finding  $c$  such that

$$X_c'(t) = F(t, X_c(t)); X_c(t_0) = c \\ G(X_c(t_1), X_c(t_2), \dots, X_c(t_n)) = 0.$$

After setting up the function for  $G$ , the problem is effectively passed to FindRoot to find the initial conditions  $c$  giving the root. The default method is to use Newton's method, which involves computing the Jacobian. While the Jacobian can be computed using finite differences, the sensitivity of solutions of an initial value problem (IVP) to its initial conditions may be too much to get reasonably accurate derivative values, so it is advantageous to compute the Jacobian as a solution to ODEs.

## Linearization and Newton's Method

Linear problems can be described by

$$X_c'(t) = J(t) X_c(t) + F_0(t); X_c(t_0) = c \\ G(X_c(t_1), X_c(t_2), \dots, X_c(t_n)) = B_0 + B_1 X_c(t_1) + B_2 X_c(t_2) + \dots + B_n X_c(t_n),$$

where  $J(t)$  is a matrix and  $F_0(t)$  is a vector both possibly depending on  $t$ ,  $B_0$  is a constant vector, and  $B_1, B_2, \dots, B_n$  are constant matrices.

Let  $Y = \frac{\partial X_c(t)}{\partial c}$ . Then, differentiating both the IVP and boundary conditions with respect to  $c$  gives



Top

$$Y'(t) = J(t) Y(t); Y(t_0) = I$$

$$\frac{\partial G}{\partial c} = B_1 Y(t_1) + B_2 Y(t_2) + \dots + B_n Y(t_n) = 0.$$

Since  $G$  is linear, when thought of as a function of  $c$ , you have  $G(c) = G(c_0) + \left(\frac{\partial G}{\partial c}\right)(c - c_0)$ , so the value of  $c$  for which  $G(c) = 0$  satisfies

$$c = c_0 + \left(\frac{\partial G}{\partial c}\right)^{-1} G(c_0)$$

for any particular initial condition  $c_0$ .

For nonlinear problems, let  $J(t)$  be the Jacobian for the nonlinear ODE system, and let  $B_i$  be the Jacobian of the  $i^{\text{th}}$  boundary condition. Then computation of  $\frac{\partial G}{\partial c}$  for the linearized system gives the Jacobian for the nonlinear system for a particular initial condition, leading to a Newton iteration,

$$c_{n+1} = c_n + \left(\frac{\partial G}{\partial c}(c_n)\right)^{-1} G(c_n).$$

## "StartingInitialConditions"

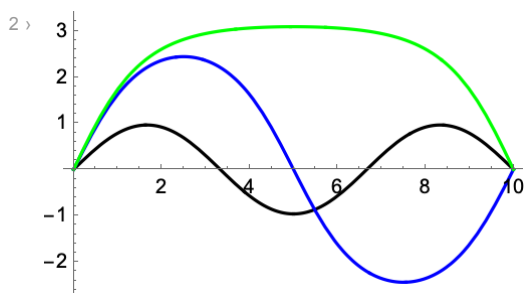
For boundary value problems, there is no guarantee of uniqueness as there is in the initial value problem case. "Shooting" will find only one solution. Just as you can affect the particular solution [FindRoot](#) gets for a system of nonlinear algebraic equations by changing the starting values, you can change the solution that "Shooting" finds by giving different initial conditions to start the iterations from.

"StartingInitialConditions" is an option of the "Shooting" method that allows you to specify the values and position of the initial conditions to start the shooting process from.

The shooting method by default starts with zero initial conditions so that if there is a zero solution, it will be returned.

This computes a very simple solution to the boundary value problem  $x'' + \sin(x) = 0$  with  $x(0) = x(10) = 0$ :

```
1 » sols = Map[First[NDSolve[{x''[t] + Sin[x[t]] == 0, x[0] == x[10] == 0}, x, t,
    Method -> {"Shooting", "StartingInitialConditions" -> {x[0] == 0, x'[0] == #}}] &,
    {1.5, 1.75, 2}];
Plot[Evaluate[x[t] /. sols], {t, 0, 10}, PlotStyle -> {Black, Blue, Green}]
```



By default, "Shooting" starts from the left side of the interval and shoots forward in time. There are cases where it is advantageous to go backward, or even from a point somewhere in the middle of the interval.

Consider the linear boundary value problem

$$x'''(t) - 2\lambda x''(t) - \lambda^2 x'(t) + 2\lambda^3 x(t) = (\lambda^2 + \pi^2)(2\lambda \cos(\pi t) + \pi \sin(\pi t))$$

$$x(0) = 1 + \frac{1 + e^{-2\lambda} + e^{-\lambda}}{2 + e^{-\lambda}}, x(1) = 0, x'(1) = \frac{3\lambda - e^{-\lambda}\lambda}{2 + e^{-\lambda}}$$



Top

that has a solution

$$x(t) = \frac{e^{\lambda(t-1)} + e^{2\lambda(t-1)} + e^{-\lambda t}}{2 + e^{-\lambda}} + \cos(\pi t).$$

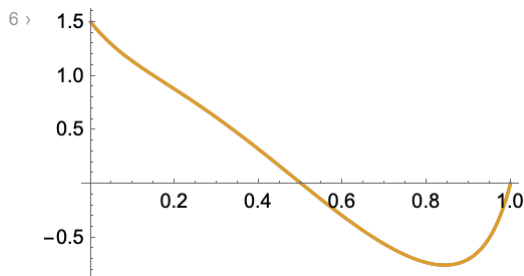
For moderate values of  $\lambda$ , the initial value problem starting at  $t = 0$  becomes unstable because of the growing  $e^{\lambda(t-1)}$  and  $e^{2\lambda(t-1)}$  terms. Similarly, starting at  $t = 1$ , instability arises from the  $e^{-\lambda t}$  term, though this is not as large as the term in the forward direction. Beyond some value of  $\lambda$ , shooting will not be able to get a good solution because the sensitivity in either direction will be too great. However, up to that point, choosing a point in the interval that balances the growth in the two directions will give the best solution.

This gives the equation, boundary conditions and exact solution as Wolfram Language input:

```
3 » eqn = x'''[t] - 2 λ x''[t] - λ² x'[t] + 2 λ³ x[t] == (λ² + π²) (2 λ Cos[π t] + π Sin[π t]);
bcs = {x[0] == 1 +  $\frac{1 + e^{-2\lambda} + e^{-\lambda}}{2 + e^{-\lambda}}$ , x[1] == 0, x'[1] ==  $\frac{3\lambda - e^{-\lambda}\lambda}{2 + e^{-\lambda}}$ };
xsol[t_] =  $\frac{e^{\lambda(t-1)} + e^{2\lambda(t-1)} + e^{-\lambda t}}{2 + e^{-\lambda}} + \text{Cos}[\pi t]$ ;
```

This solves the system with  $\lambda = 10$  shooting from the default  $t = 0$ :

```
6 » Block[{λ = 10},
  sol = First[NDSolve[{eqn, bcs}, x, t]];
  Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```



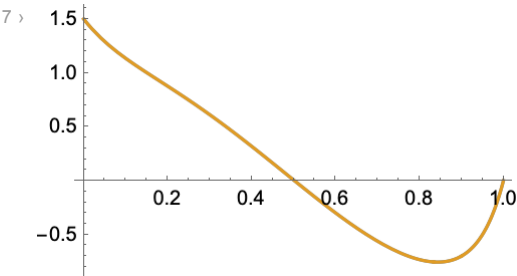
Shooting from  $t = 0$ , the "Shooting" method gives warning messages about an ill-conditioned matrix and that the boundary conditions are not satisfied as well as they should be. This is because a small error at  $t = 0$  is amplified by  $e^{20} \simeq 4 \times 10^8$ . Since the reciprocal of this is of the same order of magnitude as the local truncation error, visible errors such as those seen in the plot are not surprising. In the reverse direction, the magnification will be much less:  $e^{10} \simeq 2 \times 10^4$ , so the solution should be much better.

This computes the solution using shooting from  $t = 1$ :

```
7 » Block[{λ = 10},
  sol = First[NDSolve[{eqn, bcs}, x, t,
    Method -> {"Shooting",
      "StartingInitialConditions" -> {x[1] == 0, x'[1] ==  $\frac{3\lambda - e^{-\lambda}\lambda}{2 + e^{-\lambda}}$ , x''[1] == 0}}]];
  Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```



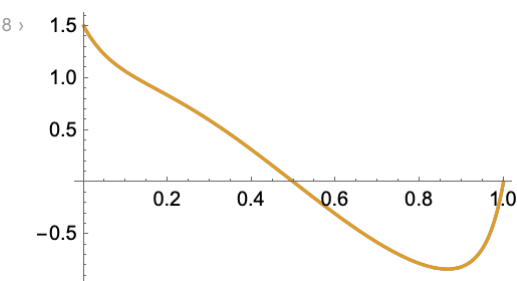
Top



A good point to choose is one that will balance the sensitivity in each direction, which is about at  $t = 2/3$ . With this, the error with  $\lambda = 15$  will still be under reasonable control.

This computes the solution for  $\lambda = 15$  shooting from  $t = 2/3$ .

```
8 » Block[{λ = 15},
  sol = First[NDSolve[{eqn, bcs}, x, t,
    Method → "Shooting",
    "StartingInitialConditions" → {x[2/3] == 0, x'[2/3] == 0, x''[2/3] == 0}]];
  Plot[{xsol[t], x[t] /. sol}, {t, 0, 1}]]
```



Option Summary

option name	default value
"StartingInitialConditions"	Automatic
the initial conditions to initiate the shooting method from	
"ImplicitSolver"	Automatic
the method to use for solving the implicit equation defined by the boundary conditions; this should be an acceptable value for the Method option of FindRoot	
"MaxIterations"	Automatic
how many iterations to use for the implicit solver method	
"Method"	Automatic
the method to use for integrating the system of ODEs	

"Shooting" method options.

"Chasing" Method

The method of chasing came from a manuscript of Gelfand and Lokutsiyevskii first published in English in [BZ65] and further described in [Na79]. The idea is to establish a set of auxiliary problems that can be solved to find initial conditions at one of the boundaries. Once the initial conditions are determined, the usual methods for solving initial value problems can be

applied. The chasing method is, in effect, a shooting method that uses the linearity of the problem to good advantage. As such, the chasing method can be used for linear differential equations.

Consider the linear ODE

$$X'(t) = A(t) X(t) + A_0(t)$$

where  $X(t) = (x_1(t), x_2(t), \dots, x_n(t))$ ,  $A(t)$  is the coefficient matrix, and  $A_0(t)$  is the inhomogeneous coefficient vector, with  $n$  linear boundary conditions

$$B_i X(t_i) = b_{i0}, i = 1, 2, \dots, n$$

where  $B_i = (b_{i1}, b_{i2}, \dots, b_{in})$  is a coefficient vector. From this, construct the augmented homogeneous system

$$X'(t) = \bar{A}(t) X(t), \quad \bar{B}_i X(t_i) = 0$$

where

$$X(t) = \begin{pmatrix} 1 \\ x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{pmatrix}, \quad \bar{A}(t) = \begin{pmatrix} a_{01}(t) & a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t) \\ a_{02}(t) & a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{0n}(t) & a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}, \text{ and } \bar{B}_i = \begin{pmatrix} b_{i0} \\ b_{i1} \\ b_{i2} \\ \vdots \\ b_{in} \end{pmatrix}.$$

The chasing method amounts to finding a vector function  $\Phi_i(t)$  such that  $\Phi_i(t_i) = \bar{B}_i$  and  $\Phi_i(t) X(t) = 0$ . Once the function  $\Phi_i(t)$  is known, if there is a full set of boundary conditions, solving

$$\begin{pmatrix} \Phi_1(t_0) \\ \Phi_2(t_0) \\ \vdots \\ \Phi_n(t_0) \end{pmatrix} X(t_0) = 0$$

can be used to determine initial conditions  $(x_1(t_0), x_2(t_0), \dots, x_n(t_0))$  that can be used with the usual initial value problem solvers. Note that the solution to system (3) is nontrivial because the first component of  $X$  is always 1.

Thus, solving the boundary value problem is reduced to solving the auxiliary problems for the  $\Phi_i(t)$ . Differentiating the equation for  $\Phi_i(t)$  gives

$$\Phi_i(t) X'(t) + X(t) \Phi_i'(t) = 0,$$

substituting the differential equation,

$$\bar{A}(t) X(t) \Phi_i(t) + X(t) \Phi_i'(t) = 0$$

and transposing

$$X(t) (\Phi_i'(t) + \bar{A}^T(t) \Phi_i(t)) = 0.$$

Since this should hold for all solutions  $X$ , you have the initial value problem for  $\Phi_i$ ,

$$\Phi_i'(t) + \bar{A}^T(t) \Phi_i(t) = 0 \text{ with initial condition } \Phi_i(t_i) = B_i$$

Given  $t_0$  where you want to have solutions to all of the boundary value problems, the Wolfram Language just uses [NDSolve](#) to solve the auxiliary problems for  $\Phi_1, \Phi_2, \dots, \Phi_m$  by integrating them to  $t_0$ . The results are then combined into the matrix of (3) that is solved for  $X(t_0)$  to obtain the initial value problem that [NDSolve](#) integrates to give the returned solution.

This variant of the method is further described in and used by the *MathSource* package [\[R98\]](#), which also allows you to solve linear eigenvalue problems.

There is an alternative, nonlinear way to set up the auxiliary problems that is closer to the original method proposed by Gelfand and Lokutsiyevskii. Assume that the boundary conditions are linearly independent (if not, then the problem is



Top

insufficiently specified). Then in each  $B_i$ , there is at least one nonzero component. Without loss of generality, assume that  $b_{ij} \neq 0$ . Now solve for  $\Phi_{ij}$  in terms of the other components of  $\Phi_i$ ,  $\Phi_{ij} = \tilde{B}_i \cdot \tilde{\Phi}_i$ , where  $\tilde{\Phi}_i = (1, \Phi_{i1}, \dots, \Phi_{ij-1}, \dots, \Phi_{ij+1}, \dots, \Phi_{in})$  and  $\tilde{B}_i = (b_{i0}, b_{i1}, \dots, b_{ij-1}, \dots, b_{ij+1}, \dots, b_{in}) / -b_{ij}$ . Using (5) and replacing  $\Phi_{ij}$ , and thinking of  $x_n(t)$  in terms of the other components of  $x(t)$ , you get the nonlinear equation

$$\tilde{\Phi}_i'(t) = -\tilde{A}^T[t] \tilde{\Phi}_i(t) + (A_j \cdot \tilde{\Phi}_i(t)) \tilde{\Phi}_i(t)$$

where  $\tilde{A}$  is  $A$  with the  $j^{\text{th}}$  column removed, and  $A_j$  is the  $j^{\text{th}}$  column of  $A$ . The nonlinear method can be more numerically stable than the linear method, but it has the disadvantage that integration along the real line may lead to singularities. This problem can be eliminated by integrating on a contour in the complex plane. However, the integration in the complex plane typically has more numerical errors than a simple integration along the real line, so in practice, the nonlinear method does not typically give results better than the linear method. For this reason, and because it is also generally faster, the default for the Wolfram Language is to use the linear method.

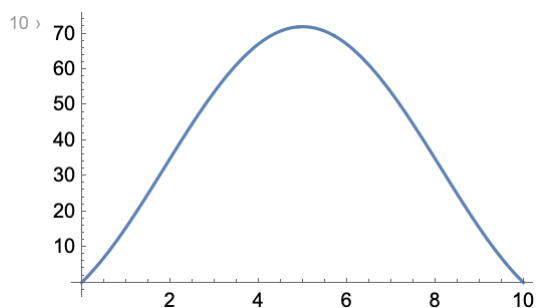
This solves a two-point boundary value problem for a second-order equation:

```
9 » nsol1 = NDSolve[{y''[t] + y[t]/4 == 8, y[0] == 0, y[10] == 0}, y, {t, 0, 10}]
```

```
9 » {{y → InterpolatingFunction[ Domain: {{0., 10.}} Output: scalar ]}}
```



This shows a plot of the solution:

```
10 » Plot[First[y[t] /. nsol1], {t, 0, 10}]
```



The solver can solve multipoint boundary value problems of linear systems of equations. (Note that each boundary equation must be at one specific value of  $t$ .)

```
11 » bconds = {
    x[0] + x'[0] + y[0] + y'[0] == 1,
    x[1] + 2 x'[1] + 3 y[1] + 4 y'[1] == 5,
    y[2] + 2 y'[2] == 4,
    x[3] - x'[3] == 7};
nsol2 = NDSolve[{
    x''[t] + x[t] + y[t] == t, y''[t] + y[t] == Cos[t],
    bconds},
    {x, y},
    {t, 0, 4}]
```

```
12 » {{x → InterpolatingFunction[ Domain: {{0., 4.}} Output: scalar ],
    y → InterpolatingFunction[ Domain: {{0., 4.}} Output: scalar ]}}
```



Top

In general, you cannot expect the boundary value equations to be satisfied to the close tolerance of [Equal](#).

This checks to see if the boundary conditions are "satisfied":

```
13 » bconds /. First[nsol2]
```

```
13 » {True, False, False, False}
```

They are typically only satisfied at most tolerances that come from the [AccuracyGoal](#) and [PrecisionGoal](#) options of [NDSolve](#). Usually, the actual accuracy and precision is less because these are used for local, not global, error control.

This checks the residual error at each of the boundary conditions:

```
14 » Apply[Subtract, bconds, 1] /. First[nsol2]
```

```
14 » {-2.22045 × 10-16, -5.35758 × 10-8, -2.95611 × 10-7, -2.86678 × 10-7}
```

When you give [NDSolve](#) a problem that has no solution, numerical error may make it appear to be a solvable problem. Typically, [NDSolve](#) will issue a warning message.

This is a boundary value problem that has no solution:

```
15 » NDSolve[{x''[t] + x[t] == 0, x[0] == 1, x[Pi] == 0}, x, {t, 0, Pi}, Method → "Chasing"]
```

**NDSolve:** The equations derived from the boundary conditions are numerically ill-conditioned. The boundary conditions may not be sufficient to uniquely define a solution. If a solution is computed, it may match the boundary conditions poorly.

```
15 » {x → InterpolatingFunction[ Domain: {{0., 3.14}} Output: scalar ]]}
```

In this case, it is not able to integrate over the entire interval because of nonexistence.

Another situation in which the equations can be ill-conditioned is when the boundary conditions do not give a unique solution.

Here is a boundary value problem that does not have a unique solution. Its general solution is shown as computed symbolically with [DSolve](#):

```
116 » dsol = First[x /. DSolve[{x''[t] + x[t] == t, x'[0] == 1, x[Pi/2] == Pi/2}, x, t]]
```

**DSolve:** Unable to resolve some of the arbitrary constants in the general solution using the given boundary conditions. It is possible that some of the conditions have been specified at a singular point for the equation.

```
116 » Function[{t}, t + c1 Cos[t]]
```

[NDSolve](#) issues a warning message because the matrix to solve for the initial conditions is singular, but has a solution:

```
117 » onesol = First[x /. NDSolve[
  {x''[t] + x[t] == t, x'[0] == 1, x[Pi/2] == Pi/2}, x, {t, 0, Pi/2}, Method → "Chasing"]]
```

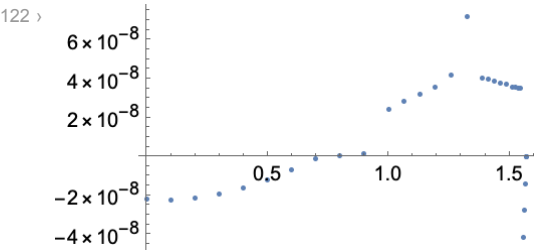
**NDSolve:** The equations derived from the boundary conditions are numerically ill-conditioned. The boundary conditions may not be sufficient to uniquely define a solution. If a solution is computed, it may match the boundary conditions poorly.

```
117 » InterpolatingFunction[ Domain: {{0., 1.57}} Output: scalar ]]
```

You can identify which solution it found by fitting it to the interpolating points. This makes a plot of the error relative to the actual best fit solution:



```
118 » ip = onesol@"Coordinates"[1];
points = Transpose[{ip, onesol[ip]}];
model = dsol[t] /.  $\mathfrak{C}_1 \rightarrow \alpha$ ;
fit = FindFit[points, model,  $\alpha$ , t];
ListPlot[Transpose[{ip, onesol[ip] - ((model /. fit) /. t -> ip)}]]
```



Typically, the default values the Wolfram Language uses work fine, but you can control the chasing method by giving `NDSolve` the option `Method` -> { "Chasing", *chasing options* } . The possible *chasing options* are shown in the following table.

option name	default value
<b>Method</b>	<b>Automatic</b>
the numerical method to use for computing the initial value problems generated by the chasing algorithm	
"ExtraPrecision"	0
number of digits of extra precision to use for solving the auxiliary initial value problems	
"ChasingType"	"LinearChasing"
the type of chasing to use, which can be either "LinearChasing" or "NonlinearChasing"	

Options for the "Chasing" method of `NDSolve` .

The method "ChasingType"->"NonlinearChasing" itself has two options.

option name	default value
"ContourType"	"Ellipse"
the shape of contour to use when integration in the complex plane is required, which can either be "Ellipse" or "Rectangle"	
"ContourRatio"	1/10
the ratio of the width to the length of the contour; typically a smaller number gives more accurate results, but yields more numerical difficulty in solving the equations	

Options for the "NonlinearChasing" option of the "Chasing" method.

These options, especially "ExtraPrecision", can be useful in cases where there is a strong sensitivity to computed initial conditions.

Here is a boundary value problem with a simple solution computed symbolically using `DSolve` :

```
23 » bvp = {x''[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1};
dsol = First[x /. DSolve[bvp, x, t]]

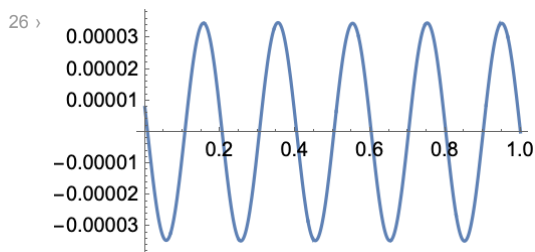
24 › Function[{t}, Csc[10  $\sqrt{10}$ ] Sin[10  $\sqrt{10}$  t]]
```





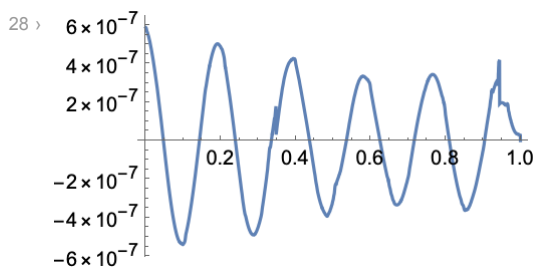
This shows the error in the solution computed using the chasing method in `NDSolve`:

```
25 » sol = First[x /. NDSolve[
      {x'[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1}, x, {t, 0, 1}, Method → "Chasing"];
      Plot[sol[t] - dsol[t], {t, 0, 1}]
```



Using extra precision to solve for the initial conditions reduces the error substantially:

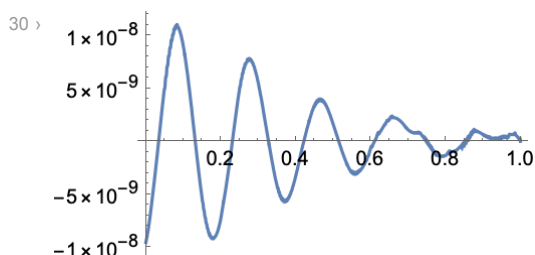
```
27 » sol = First[x /. NDSolve[{x'[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1},
      x, {t, 0, 1}, Method → {"Chasing", "ExtraPrecision" → 10}]];
      Plot[sol[t] - dsol[t], {t, 0, 1}]
```



Increasing the extra precision beyond this really will not help because a significant part of the error results from computing the solution once the initial conditions are found. To reduce this, you need to give more stringent `AccuracyGoal` and `PrecisionGoal` options to `NDSolve`.

This uses extra precision to compute the initial conditions along with more stringent settings for the `AccuracyGoal` and `PrecisionGoal` options:

```
29 » sol = First[x /. NDSolve[{x'[t] + 1000 x[t] == 0, x[0] == 0, x[1] == 1}, x, {t, 0, 1}, Method →
      {"Chasing", "ExtraPrecision" → 10}, AccuracyGoal → 10, PrecisionGoal → 10]];
      Plot[sol[t] - dsol[t], {t, 0, 1}]
```



## Boundary Value Problems with Parameters

In many of the applications where boundary value problems arise, there may be undetermined parameters, such as eigenvalues, in the problem itself that may be a part of the desired solution. By introducing the parameters as dependent variables, the problem can often be written as a boundary value problem in standard form.

For example, the flow in a channel can be modeled by



Top

$$f''' - R(f')^2 - f f'' + R a = 0$$

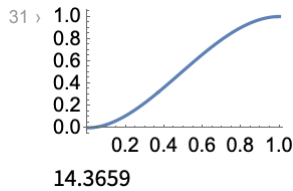
$$f(0) = f'(0) = 0, f(1) = 1, f'(1) = 0,$$

where  $R$  (the Reynolds number) is given, but  $a$  is to be determined.

To find the solution  $f$  and the value of  $a$ , just add the equation  $a' = 0$ .

This solves the flow problem with  $R = 1$  for  $f$  and  $a$ , plots the solution  $f$  and returns the value of  $a$ :

```
31 » Block[{R = 1}, sol = NDSolve[{f'''[t] - R ((f'[t])^2 - f[t] f''[t]) + R a[t] == 0,
    a'[t] == 0, f[0] == f'[0] == f'[1] == 0, f[1] == 1}, {f, a}, t];
    Column[{Plot[f[t] /. First[sol], {t, 0, 1}],
    a[0] /. First[sol]}]]
```



#### Related Tech Notes

- [Advanced Numerical Differential Equation Solving](#)

Feedback Top



Introduction for Programmers



Introductory Book

[Wolfram Function Repository](#) | [Wolfram Data Repository](#) | [Wolfram Data Drop](#) | [Wolfram Language Products](#)

© 2023 Wolfram. All rights reserved.

[Legal & Privacy Policy](#) | [Site Map](#) | [WolframAlpha.com](#) | [WolframCloud.com](#)



Top