

In and Out

Edited by Paul Abbott

In and Out offers readers an opportunity to ask questions of the experts. *The Mathematica Journal* encourages readers to submit problems in care of the editor. Answers posted to the *Mathematica* news group, comp.soft-sys.math.mathematica, that appear here are edited for clarity and length.

■ StepMonitor

Q: I am running a complex **FindMinimum** calculation that takes a long time to run. If I abort the computation, is there a way to output the last fully solved iteration that **FindMinimum** has found?

A: Robert Knapp (rknapp@wolfram.com) answers: In Version 5, you can use the **StepMonitor** option to keep track of the iterations.

Here is a simple function that uses **Pause** to simulate running a complex computation.

```
In[1]:= f[x_?NumberQ, y_] := (Pause[1]; (x - 1)^2 + (x^2 - y)^2)
```

StylePrint is used to provide intermediate results as **Output** style cells.

```
In[2]:= FindMinimum[f[x, y], {{x, -2}, {y, 1}},
  MaxIterations -> 3, StepMonitor -> StylePrint[{x, y}, "Output"]]
{-0.903677, 1.21926}
{-0.808228, 1.18443}
{0.475604, 0.501006}
```

```
FindMinimum::cvmit :
Failed to converge to the requested accuracy or precision within 3 iterations. More...
```

```
Out[2]= {0.35051, {x -> 0.475604, y -> 0.501006}}
```

To get a result when you abort the computation, you need to use **CheckAbort**.

```
In[3]:= CheckAbort[FindMinimum[f[x, y],
      {{x, -2}, {y, 1}}, StepMonitor -> (laststep = {x, y}), laststep]
```

```
Out[3]= {2.54791 × 10-15, {x → 1., y → 1.}}
```

If the problem runs to completion, you will get the result from **FindMinimum**. If you abort the computation, you will obtain the last step taken in the iteration.

When **FindMinimum** gets close to a solution, differences between iterations do not show up in **Output** style cells. One can output results with more digits by changing the **PrintPrecision** display option as follows:

1. Open the Option Inspector by clicking Format ▷ Option Inspector. Set the scope to notebook (or global).
2. Locate **PrintPrecision** by typing “printp” in the Lookup text field.
3. Set **PrintPrecision** to the number of digits you want displayed.

■ Beeping Progress

Q: How can I play a sound to alert me regarding my program’s progress during a long looping procedure?

A: You can use the built-in **Message** mechanism. You can specify the required (default) behavior of **Message** by setting **WarningAction** to include **Beep** as follows:

1. Open the Option Inspector by clicking Format ▷ Option Inspector. Make sure the scope is global.
2. Locate **WarningAction** by typing in “warning” in the Lookup text field.
3. Modify **WarningAction** to include **Beep**.

Add a message displaying the iteration number (see Section 2.9.21 of *The Mathematica Book*).

```
In[1]:= Progress::update = "Iteration `1`.";
```

Redundant or repetitive messages are, by default, suppressed in long calculations; therefore, you should turn off **General::stop** (see **General::stop** in the Reference Guide).

```
In[2]:= Off[General::stop]
```

From your loop, call this message.

```
In[3]:= Table[If[Mod[i, 200] == 0, Message[Progress::update, i]]; {i, ζ(i)}, {i, 1000}];
```

```
Progress::update : Iteration 200.
```

```
Progress::update : Iteration 400.
```

```
Progress::update : Iteration 600.
```

```
Progress::update : Iteration 800.
```

```
Progress::update : Iteration 1000.
```

Alternatively, if you have suitable text-to-speech system software installed on your system, you can ask the front end to “speak” the given text, which should be a text string.

```
In[4]:= Table[If[Mod[i, 100] == 0,
  FrontEndExecute[SpeakTextPacket["Iteration " <> ToString[i]]];
  {i, ζ(i)}, {i, 1100, 1500}];
```

To set the speaking voice, use `FrontEndExecute[SetSpeechParameters`
Packet[i]]` where i is a number between 1 and the number of voices you have installed. (For further information, see the following *The Mathematica Journal* articles: “Tricks of the Trade” (SetNotebookStatusLine), 7 (3), 1999 p. 263, and “In and Out” (Why the Beep?), 8 (3), 2002 p. 400.)

■ Coplanarity

Q: Given a set of n -dimensional vectors, is there a simple way to check if they are coplanar in \mathbb{R}^{n-1} ?

A: Eckhard Hennig (aidev@kaninkolo.de) answers: To establish coplanarity in \mathbb{R}^{n-1} of a set of n -dimensional vectors, \mathcal{V} , you need to check whether the dimension of the null space (or *nullity*) of the linear mapping defined by \mathcal{V} has dimension ≥ 1 . Here is an example.

Define three linearly independent vectors in \mathbb{R}^3 .

```
In[1]:= a1 = {2, 2, 3}; a2 = {1, -1, 2}; a3 = {1, 0, -1};
```

Generate a set of 10 coplanar vectors in \mathbb{R}^3 by forming random linear combinations of a_1 and a_2 . Use `SeedRandom` to obtain a repeatable sequence of pseudo-random numbers.

```
In[2]:= SeedRandom[1234];
```

```
In[3]:= v2 = Table[Table[Random[Real, {-10, 10}], {2}].{a1, a2}, {10}]
```

```
Out[3]= 
$$\begin{pmatrix} 14.3308 & 0.162139 & 25.0384 \\ 5.2323 & -10.6653 & 11.8228 \\ 15.3128 & 15.8908 & 22.8247 \\ -6.12104 & 10.531 & -13.3446 \\ -23.3773 & -15.7515 & -36.9723 \\ -0.314199 & 10.456 & -3.16384 \\ 2.51425 & -8.06091 & 6.41517 \\ 11.8982 & 12.8355 & 17.6129 \\ 6.58317 & 4.87541 & 10.3017 \\ -3.20984 & 8.46853 & -7.73436 \end{pmatrix}$$

```

The null space of the mapping defined by v_2 has dimension 1; hence, the set is coplanar.

```
In[4]:= Length[NullSpace[v2]]
```

```
Out[4]= 1
```

In Version 5, you can use **MatrixRank** instead to calculate the dimension of the subspace spanned by v_2 . The vectors are coplanar if the rank of v_2 is 2 (or $n - 1$ in the general case).

```
In[5]:= MatrixRank[v2]
```

```
Out[5]= 2
```

NullSpace may be unreliable for large and ill-conditioned systems. A more reliable, but less efficient, alternative is to determine the number of singular values of the mapping.

```
In[6]:= Length[SingularValueList[v2]]
```

```
Out[6]= 2
```

Since there are two singular values, the set of vectors spans a two-dimensional space, that is, the vectors are coplanar in \mathbb{R}^3 .

Now crosscheck for the noncoplanar case: generate 10 vectors in \mathbb{R}^3 by forming random linear combinations of the three independent basis vectors a_1 , a_2 , and a_3 .

```
In[7]:= v3 = Table[Table[Random[Real, {-10, 10}], {3}].{a1, a2, a3}, {10}]
```

```
Out[7]= 
$$\begin{pmatrix} 11.7238 & 10.4453 & 15.8459 \\ 13.4381 & 26.7939 & 8.07186 \\ -30.5407 & -7.09987 & -34.9813 \\ -2.41632 & -14.0657 & -27.2224 \\ 3.17777 & -19.8561 & -12.7652 \\ 7.14493 & 4.95061 & 33.2804 \\ -27.6994 & -10.9223 & -24.3901 \\ -5.41409 & 1.31064 & 14.0318 \\ -25.1416 & 2.88407 & -17.8299 \\ -23.0158 & -11.0079 & -31.1886 \end{pmatrix}$$

```

The null space is empty, which implies that a_1 , a_2 , and a_3 span \mathbb{R}^3 .

```
In[8]:= NullSpace[v3]
```

```
Out[8]= {}
```

Alternatively, the number of singular values equals the dimension of the space.

```
In[9]:= Length[SingularValueList[v3]] == 3
```

```
Out[9]= True
```

■ Autocompilation

Q: I would like to understand the timing difference between the following two code samples. Slow version:

```
In[1]:= n = m = 100; First@Timing[Do[Table[1, {i, 1, n}, {j, 1, m}], {100}];]
```

```
Out[1]= 2.41 Second
```

Fast version:

```
In[2]:= First@Timing[Do[Table[1, {i, 1, 100}, {j, 1, 100}], {100}];]
```

```
Out[2]= 0.46 Second
```

Is this due to autocompilation?

A: Robert Knapp (rknapp@wolfram.com) answers: The issue has to do with autocompilation, but is quite subtle. `Table[1, {i, 1, n}, {j, 1, m}]` is basically equivalent to `Table[Table[1, {j, 1, m}], {i, 1, n}]`. With the second form, it is a bit easier to see why it cannot be autocompiled. Each time the inner `Table` is computed, `m` needs to be evaluated to determine what the limit should be. Therefore the issue boils down to why it needs to be evaluated *each* time through

the outer loop. Before evaluation, m is just a symbol, so it could have any evaluation attached to it, for example, $m := 10 i$. In this example, the result of evaluating with only one i would obviously be wrong. You might think it would be possible to evaluate with i unset, but there are cases where that would be incorrect, for example, $m := \text{If}[\text{NumberQ}[i], 10 i, 0]$. This is, of course, pathological, but it is not too difficult to construct real cases that give incorrect or misleading values. It is very important that *Mathematica* be consistent, so we cannot do this. Thus, since m cannot be known, except for specific values of i , there is no way to autocompile the loop as a whole in a really efficient way. On the other hand, when you evaluate or put in literal numbers, everything is known, so compilation analysis can be done quite completely.

■ Sparse Matrices

Q: One can use **LinearSolve** to solve a system of linear equations. However, I work with very large matrices having only five nonzero diagonals. Are there efficient methods for dealing with such matrices?

A: Daniel Lichtblau (danl@wolfram.com) answers: If you require only machine arithmetic, this can be done effectively using the built-in sparse linear solver in Version 5. Here is an example.

Create a diagonally-dominant random sparse $n \times n$ d -diagonal matrix using **SparseArray**.

```
In[1]:= SeedRandom[1111];
In[2]:= RandomMatrix[n_, d_?OddQ] :=
  SparseArray[{{i_, j_} /; |i - j| ≤ (d - 1) / 2 ⇒ δi,j + Random[]}, {n, n}]
```

With $n = 4$ and $d = 3$, we obtain a sparse array with 10 nonzero elements.

```
In[3]:= RandomMatrix[4, 3]
Out[3]= SparseArray[<10>, {4, 4}]
```

Use **Normal** to obtain the ordinary array corresponding to a sparse array object.

```
In[4]:= % // Normal
Out[4]= 
$$\begin{pmatrix} 1.93339 & 0.814818 & 0 & 0 \\ 0.688359 & 1.1433 & 0.833 & 0 \\ 0 & 0.317132 & 1.91055 & 0.346584 \\ 0 & 0 & 0.0287865 & 1.64717 \end{pmatrix}$$

```

Produce a pentadiagonal sparse matrix of dimension 2000.

```
In[5]:= m = 2000; mat = RandomMatrix[m, 5]
```

```
Out[5]= SparseArray[<9994>, {2000, 2000}]
```

Solve for a random right-hand side.

```
In[6]:= rhs = Table[Random[], {m}];
```

```
In[7]:= Timing[sol1 = LinearSolve[SparseArray[mat], rhs];] // First
```

```
Out[7]= 0.09 Second
```

Compare with the time required for an ordinary (nonsparse) matrix.

```
In[8]:= Timing[sol2 = LinearSolve[Normal[SparseArray[mat]], rhs];] // First
```

```
Out[8]= 18.95 Second
```

Check that the solutions agree using **Norm** (for matrices, **Norm[m]** gives the maximum singular value of m).

```
In[9]:= Norm[sol2 - sol1]
```

```
Out[9]= 4.29374 × 10-11
```

■ ϵ - δ Proofs

Q: How can I do ϵ - δ proofs for limits? For example, prove that the limit of $\frac{x^2-4}{x-2}$, as x approaches 2, is 4.

A: Andrzej Kozłowski (akoz@mimuw.edu.pl) answers: If you restrict attention to rational functions, you can use quantifiers and **Resolve** to do this.

```
In[1]:= Resolve[ $\forall \epsilon > 0 \left( \exists \delta, \delta > 0 \left( \forall x, |x-2| < \delta \wedge x \in \mathbb{R} \wedge \lambda \in \mathbb{R} \left| \frac{x^2-4}{x-2} - \lambda \right| < \epsilon \right) \right)$ ]
```

```
Out[1]=  $\lambda == 4$ 
```

■ Integer Solutions

Q: Is there a simple way to compute the solutions to $p + q = n$ for $2 \leq p \leq \frac{n}{2}$, $p \leq q$, $p, q \in \mathbb{P}$ for a specified $n > 4$, $n \in \mathbb{Z}$?

A: Adam Strzebonski (adams@wolfram.com) answers: In Version 5, you can use **Reduce**, though it requires changing the value of a system option. By default,

Reduce writes out integer solutions of $a \leq x \leq b$ explicitly if the number of integers in $[a, b]$ is less than 11.

In[1]:= **Reduce**[$1 \leq x \leq 10, x, \mathbb{Z}$]

Out[1]= $x = 1 \vee x = 2 \vee x = 3 \vee x = 4 \vee x = 5 \vee x = 6 \vee x = 7 \vee x = 8 \vee x = 9 \vee x = 10$

Otherwise, the solutions are represented as $x \in \mathbb{Z} \wedge a \leq x \leq b$.

In[2]:= **Reduce**[$1 \leq x \leq 11, x, \mathbb{Z}$]

Out[2]= $x \in \mathbb{Z} \wedge 1 \leq x \leq 11$

Here is the general *parametric* solution to the given problem with $n = 500$.

In[3]:= **m1**[n_] := **Reduce**[$p + q = n \wedge 2 \leq p \leq \frac{n}{2} \wedge p \leq q, \{p, q\}, \mathbb{P}$]

In[4]:= **m1**[500]

Out[4]= $(p | q) \in \mathbb{P} \wedge c_1 \in \mathbb{Z} \wedge 2 \leq c_1 \leq 250 \wedge p = c_1 \wedge q = 500 - c_1$

Changing the value of a **Reduce** system option yields *explicit* solutions.

In[5]:= **Developer`SetSystemOptions**
 "**ReduceOptions**" \rightarrow "**DiscreteSolutionBound**" $\rightarrow \infty$;

In[6]:= **Reduce**[$1 \leq x \leq 11, x, \mathbb{Z}$]

Out[6]= $x = 1 \vee x = 2 \vee x = 3 \vee x = 4 \vee x = 5 \vee$
 $x = 6 \vee x = 7 \vee x = 8 \vee x = 9 \vee x = 10 \vee x = 11$

Here are all the solutions to the given problem with $n = 500$.

In[7]:= **m1**[500]

Out[7]= $p = 13 \wedge q = 487 \vee p = 37 \wedge q = 463 \vee p = 43 \wedge q = 457 \vee p = 61 \wedge q = 439 \vee$
 $p = 67 \wedge q = 433 \vee p = 79 \wedge q = 421 \vee p = 103 \wedge q = 397 \vee$
 $p = 127 \wedge q = 373 \vee p = 151 \wedge q = 349 \vee p = 163 \wedge q = 337 \vee$
 $p = 193 \wedge q = 307 \vee p = 223 \wedge q = 277 \vee p = 229 \wedge q = 271$

Although this approach is simplest in that it requires just a single command and no manual preprocessing of the input, it is not very efficient. **Reduce** has no special methods for solving equations and inequalities over the primes, other than by generating primes in a given finite interval. If we give the problem to **Reduce** directly, it uses a general method for solving systems of equations and inequalities over the integers, and then removes the numeric solutions that are not prime. Changing the system options causes $2 \leq c_1 \leq 250$ to be represented as 249 explicit values, from which we get 249 integer solutions for p and q , and only then does it remove those that are not prime.

A more efficient way of solving this particular problem is to make **Reduce** generate only prime values for p , and then select those for which $q = n - p$ is prime.

In[8]:= **m2**[n_] := **Or** @@ ($p = \# \wedge q = n - \# \& / \&$

Select[$p /. \{ \text{ToRules}[\text{Reduce}[2 \leq p \leq \frac{n}{2}, p, \mathbb{P}]] \}, \text{PrimeQ}[n - \#] \& \}$]

Here is a comparison of the two methods.

```
In[9]:= (s1 = m1[105];) // Timing // First
Out[9]= 124.31 Second

In[10]:= (s2 = m2[105];) // Timing // First
Out[10]= 0.35 Second

In[11]:= s1 === s2
Out[11]= True
```

■ Interpolation

Q: Given a set of nonuniformly sampled $\{x, y\}$ data, how can I compute numerical values of the slope dy/dx , second derivative d^2y/dx^2 , and integral $\int y^2 dx$? Also, is it possible to compute y as a function of uniformly incremented arc length?

A: Here is an exact function $y(x) = x \sin(x) - x/2$.

```
In[1]:= y(x_) = x sin(x) - x/2;
```

Sample $y(x)$, taking x from a uniform random distribution over the interval $(a, b) = (1, 12)$, and append the function values at the endpoints.

```
In[2]:= {a, b} = {1, 12};
```

```
In[3]:= f(x_) = {x, y(x)};
```

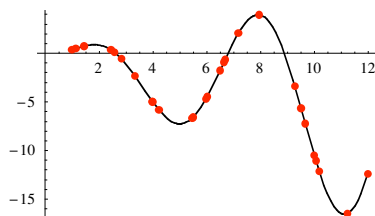
```
In[4]:= SeedRandom[4321];
```

```
In[5]:= data = N[Sort[f /@ Join[{a}, Table[Random[Real, {a, b}], {25}], {b}]]];
```

Superimpose the sampled data over a plot of the function to visualize the $\{x, y\}$ data.

```
In[6]:= Plot[y(x), {x, a, b}, Epilog -> {Hue[1], AbsolutePointSize[3], Point /@ data}]
```

From In[6]:=



Interpolate the data using **Interpolation** of sufficiently high order.

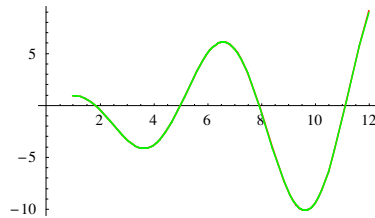
```
In[7]:= y_int = Interpolation[data, InterpolationOrder -> 5]
```

```
Out[7]= InterpolatingFunction[{{1. 12.}}, <>]
```

Compare the first and second derivatives; the red curve denotes the exact value and the green curve denotes the interpolated function in each plot. The agreement is very good.

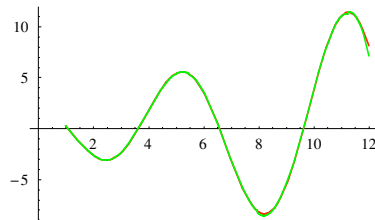
```
In[8]:= Plot[{y'(x), y'_int(x)}, {x, a, b}, PlotStyle -> {Hue[0], Hue[1/3]}]
```

From In[8]:=



```
In[9]:= Plot[{y''(x), y''_int(x)}, {x, a, b}, PlotStyle -> {Hue[0], Hue[1/3]}]
```

From In[9]:=



Next, examine $\int y(x)^2 dx$. For comparison purposes, compute the exact value of $\int_a^x y(t)^2 dt$.

```
In[10]:= i_exact(x_) = Integrate[y(t)^2, t]
```

```
Out[10]= x^3/4 + Cos[x] x^2 - 1/4 Sin[2 x] x^2 - 1/4 Cos[2 x] x - 2 Sin[x] x -
2 Cos[x] + 1/8 Sin[2 x] + Sin[2]/8 + 2 Sin[1] + Cos[2]/4 + Cos[1] - 1/4
```

Compute the numerical value of $\int_a^x y(t)^2 dt$ as an **InterpolatingFunction** using **FunctionInterpolation** and **Integrate**.

```
In[11]:= i_int(x_) = Integrate[FunctionInterpolation[y_int(x)^2, {x, a, b}][x], x]
```

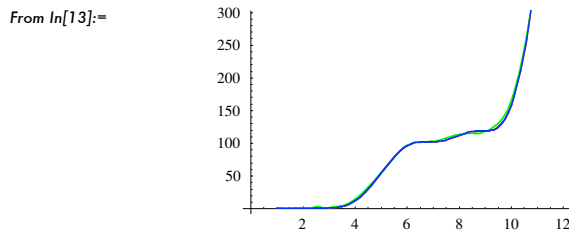
```
Out[11]= InterpolatingFunction[{{1. 12.}}, <>][x]
```

Alternatively, compute the indefinite integral as an **InterpolatingFunction** using **NDSolve** (since $g(x) = \int_a^x y(t)^2 dt$ is equivalent to $g'(x) = y(x)^2$ with $g(a) = 0$).

```
In[12]:= i_ndsol(x_) = g(x) /. First[NDSolve[{y_int(x)^2 == g'(x), g(a) == 0}, g, {x, a, b}]]
Out[12]= InterpolatingFunction[({1. 12.}), <>][x]
```

Compare the exact value i_{exact} (red) to the two numerical approaches, i_{int} (green) and i_{ndsol} (blue). The curves overlay one another quite well.

```
In[13]:= Plot[{i_exact(x), i_int(x), i_ndsol(x)}, {x, a, b},
  PlotStyle -> {Hue[0], Hue[1/3], Hue[2/3]}]
```



The arc length, $r(x)$, of a parametric curve is

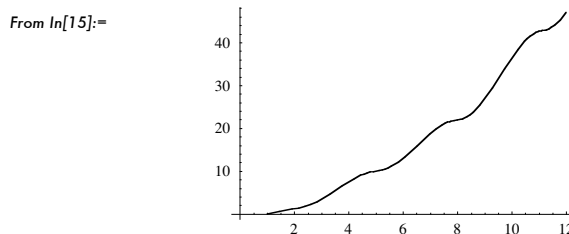
$$r(x) = \int_a^x \sqrt{1 + y'(t)^2} dt,$$

so we have $r'(x) = \sqrt{1 + y'(x)^2}$ with $r(a) = 0$. Here is the arc length, r , as an **InterpolatingFunction**.

```
In[14]:= r = r /. First[NDSolve[{r'(x) == Sqrt[y_int(x)^2 + 1], r(a) == 0}, r, {x, a, b}]];
```

The arc length is, as expected, monotonically increasing.

```
In[15]:= Plot[r(x), {x, a, b}]
```

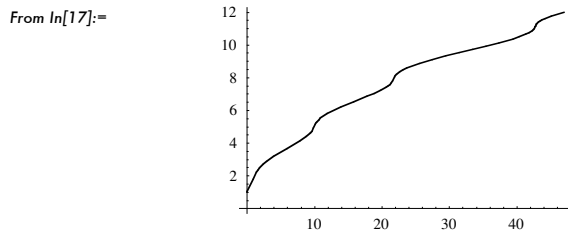


Since $x'(r) = 1/r'(x)$ we can also determine the inverse function $x(r)$.

```
In[16]:= x = x /. First[NDSolve[{1/x'(r) == Sqrt[1 + y_int(x(r))^2], x(0) == a}, x, {r, r(a), r(b)}]];
```

Here is a plot of $x(r)$.

In[17]:= `Plot[x(r), {r, r(a), r(b)}]`



Here is a table of uniformly incremented arc length values of the interpolated function y_{int} .

In[18]:= `arcdata = Table[{x(t), y_int(x(t))}, Evaluate[{t, r(a), r(b), $\frac{r(b) - r(a)}{20}$ }]]]`

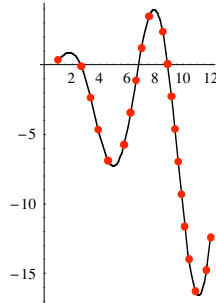
Out[18]=

1.	0.341471
2.67154	-0.125759
3.35219	-2.37684
3.90786	-4.66386
4.61717	-6.9037
5.76061	-5.75552
6.2294	-3.44957
6.61897	-1.12848
7.01705	1.19114
7.56797	3.47666
8.53554	2.37315
8.89554	0.0476488
9.17178	-2.28962
9.41651	-4.63042
9.64925	-6.97245
9.88311	-9.31438
10.1326	-11.6547
10.4256	-13.9899
10.8973	-16.292
11.6967	-14.7726
12.	-12.4389

A **Plot** with **AspectRatio** \rightarrow **Automatic** shows that the sampling does appear to be uniform with respect to arc length.

```
In[19]:= Plot[yint(x), {x, a, b},
  Epilog -> {Hue[1], AbsolutePointSize[3], Point /@ arcdata},
  AspectRatio -> Automatic]
```

From In[19]:=



■ NonlinearRegress

Q: Consider the following initial-value problem.

```
In[1]:= de[a_, b_][t_] = y'(t) == a - b y(t)^3;
```

```
In[2]:= ic = y(0) == 1;
```

The differential equation cannot be solved explicitly so you have to use **NDSolve**. How can I use the output of **NDSolve** as the model function argument to **NonlinearRegress**?

A: Carl Woll (carlw@u.washington.edu) writes: Basically, there are four steps.

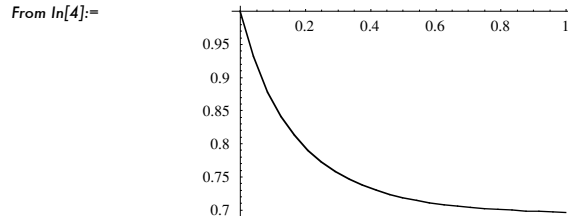
1. Define the model with both the parameters and variables restricted to numeric quantities to prevent **NDSolve** from being given symbolic input.
2. Compute the derivatives of the model with respect to each parameter.
3. Teach *Mathematica* about these derivatives.
4. Use **NonlinearRegress** as usual.

After restricting the input parameters a and b to be numeric quantities, numerical solution is straightforward. The variable t used inside **NDSolve** is in a **Module** to ensure that it does not already have a value. Dynamic programming improves efficiency by computing and saving the solution for fixed a and b .

```
In[3]:= f[a_?NumericQ, b_?NumericQ] :=
      f[a, b] = Module[{t}, y /. First[NDSolve[{de[a, b][t], ic}, y, {t, 0, 1}]]]
```

Plot the solution for $a = 1$ and $b = 2$.

```
In[4]:= Plot[f[1, 3][t], {t, 0, 1}]
```



Next, compute the derivatives of $f[a, b][t]$, with respect to the *parameters* a and b , by differentiating the parameters of the differential equation and the initial condition.

```
In[5]:=  $\frac{db}{da} \wedge = 0; \frac{dt}{da} \wedge = 1; \text{dea}[a_, b_] [t_] = \frac{d \text{de}[a, b][t]}{da} /. y^{(n)}(t_) \rightarrow ya^{(n-1)}(t)$ 
```

```
Out[5]= ya'(t) == 1 - 3 b y(t)^2 ya(t)
```

```
In[6]:=  $\frac{da}{db} \wedge = 0; \frac{dt}{db} \wedge = 1; \text{deb}[a_, b_] [t_] = \frac{d \text{de}[a, b][t]}{db} /. y^{(n)}(t_) \rightarrow yb^{(n-1)}(t)$ 
```

```
Out[6]= yb'(t) == -y(t)^3 - 3 b yb(t) y(t)^2
```

Then use **NDSolve** on the system of ordinary differential equations, consisting of the original and the differentiated equations.

```
In[7]:= fa[a_?NumericQ, b_?NumericQ] :=
      fa[a, b] = Module[{t}, ya /. First[NDSolve[
        {de[a, b][t], dea[a, b][t], ic, ya[0] == 0}, {y, ya}, {t, 0, 1}]]]
```

```
In[8]:= fb[a_?NumericQ, b_?NumericQ] :=
      fb[a, b] = Module[{t}, yb /. First[NDSolve[
        {de[a, b][t], deb[a, b][t], ic, yb[0] == 0}, {y, yb}, {t, 0, 1}]]]
```

After you have computed the derivatives of the model with respect to the parameters, teach *Mathematica* this information.

```
In[9]:= f /:  $\frac{\partial f[a, b][t]}{\partial a} := \text{fa}[a, b][t]$ 
```

```
In[10]:= f /:  $\frac{\partial f[a, b][t]}{\partial b} := \text{fb}[a, b][t]$ 
```

Next, use **NonlinearRegress** in the usual way. Load the package.

```
In[11]:= << Statistics`NonlinearFit`
```

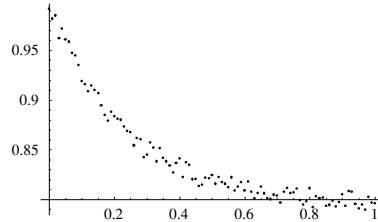
Create some noisy data.

```

In[12]:= SeedRandom[10]
In[13]:= data = Table[{t, f[1, 2][t] + Random[Real, {-0.01, 0.01}]}, {t, 0, 1, 0.01}];
In[14]:= lp = ListPlot[data]

```

From In[14]:=



Then run **NonlinearRegress**.

```

In[15]:= NonlinearRegress[data, f[a, b][t], t,  $\begin{pmatrix} a & 0 \\ b & 1 \end{pmatrix}$ ]

```

Out[15]= {BestFitParameters → {a → 0.940321, b → 1.89889}, ParameterCITable →

	Estimate	Asymptotic SE	CI
<i>a</i>	0.940321	0.0233763	{0.893937, 0.986704},
<i>b</i>	1.89889	0.038682	{1.82213, 1.97564}

EstimatedVariance → 0.0000335898, ANOVATable →

	DF	SumOfSq	MeanSq
Model	2	71.8804	35.9402
Error	99	0.00332539	0.0000335898,
Uncorrected Total	101	71.8837	
Corrected Total	100	0.277771	

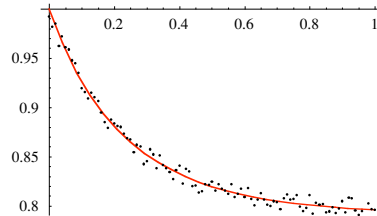
AsymptoticCorrelationMatrix → $\begin{pmatrix} 1. & 0.992543 \\ 0.992543 & 1. \end{pmatrix}$,

	Curvature
FitCurvatureTable → Max Intrinsic	0
Max Parameter-Effects	0
95. % Confidence Region	0.569042

View the best-fit solution.

```
In[16]:= Plot[f[a, b][t] /. (BestFitParameters /. %),  
          {t, 0, 1}, Epilog -> First[lp], PlotStyle -> Hue[1]]
```

From In[16]:=



This method allows you to use the full power of **NonlinearRegress** in a simple and very natural way. There are other approaches, but most of them require that you either use the **FindMinimum** method of **NonlinearRegress**, or that you abandon **NonlinearRegress** (and all of its statistical feedback) and use **FindMinimum** directly.

Paul Abbott

*School of Physics, M013
The University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
tmj@physics.uwa.edu.au
physics.uwa.edu.au/~paul*