Suggested Paradigms for Beginners to Mathematica

Many beginners feel that the learning curve for MATHEMATICA® is very steep; for many this steep curve becomes a permanent barrier. This is unfortunate—but perhaps inevitable given the depth and complexity of the language.

My purpose is to provide a few working examples that illustrate good practice for beginners[1].

These suggestions reflect my own experience of using MATHEMATICA® as a tool for my work. Thus, the suggested paradigms and styles reflect my personal tastes. I believe many advanced users would agree with many of my suggestions, but few would agree with all. Of course, there will be exceptions to these suggestions: depending on the type of application or size of a project, I break the general suggestions that are presented below.

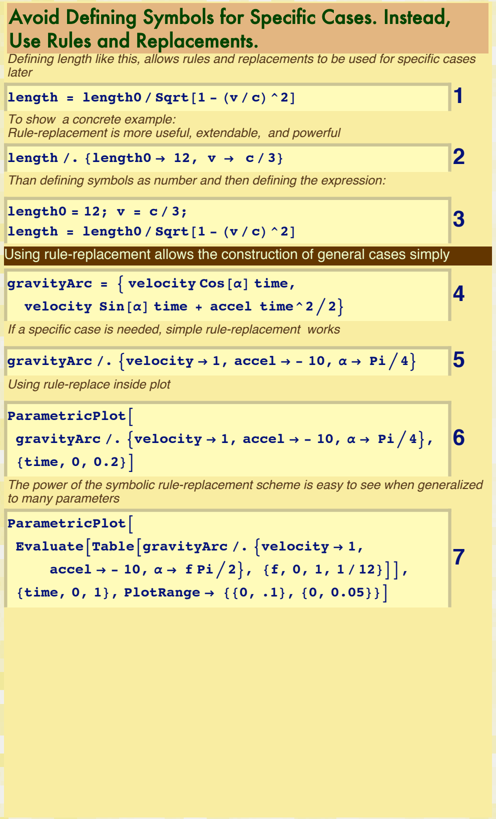
Beginners is probably well-served by starting with a small set of well-worn examples and then extending these to their own purposes. Many of the suggestions won't make sense to first-time explorers—my advice to read the examples and get a general sense of the landscape. Look at them again later when you have developed a working context.

[1] Many thanks to members of the mathgroup mailing list for making suggestions and general discussion. I hope to compile a list of those who have been particularly helpful in future versions of this guide.

Suggested MATHEMATICA® Paradigm 1

Avoid Assignments for Specific Cases and Avoid Assignment to Numbers

Much of the power in MATHEMATICA® comes from the manipulation of symbols. It deals with numbers and numerical calculations just fine, but it is usually best to hold of looking at cases that evaluate numerically until the very end. Think of MATHEMATICA® as having two parts: the first part derives results like you might find in a text book; the second part allows you to compute and visualize the results as you might do in a exercise in a beginning textbook.



Avoid Defining Symbols for Specific Cases. Instead, Use Rules and Replacements.

Defining length like this, allows rules and replacements to be used for specific cases later

```
length = length0 / Sqrt[1 - (v / c) ^2]                                   1
```

To show a concrete example:
Rule-replacement is more useful, extendable, and powerful

```
length /. {length0 → 12, v → c / 3}                                       2
```

Than defining symbols as number and then defining the expression:

```
length0 = 12; v = c / 3;
length = length0 / Sqrt[1 - (v / c) ^2]                                   3
```

Using rule-replacement allows the construction of general cases simply

```
gravityArc = {velocity Cos[α] time,
  velocity Sin[α] time + accel time^2 / 2}                                4
```

If a specific case is needed, simple rule-replacement works

```
gravityArc /. {velocity → 1, accel → - 10, α → Pi / 4}                    5
```

Using rule-replace inside plot

```
ParametricPlot[
 gravityArc /. {velocity → 1, accel → - 10, α → Pi / 4},                  6
 {time, 0, 0.2}]
```

The power of the symbolic rule-replacement scheme is easy to see when generalized to many parameters

```
ParametricPlot[
 Evaluate[Table[gravityArc /. {velocity → 1,
     accel → - 10, α → f Pi / 2}, {f, 0, 1, 1 / 12}]],                    7
 {time, 0, 1}, PlotRange → {{0, .1}, {0, 0.05}}]
```

1: This is a nice physical equation for length. It is not as interesting at a particular values (e.g., zero-velocity length is one meter, relative velocity is 10^8m/s) as is *the behavior of the any length at any velocity*. Here the equation is more descriptive than the numbers one could calculate for a limited set of cases. Setting known values (e.g., $c = 3 \times 10^8$m/s instead of some other set of units) limits the types of applications of the equations.
Leaving the equation in a symbolic form, it will become easy to ask questions such as "At what relative velocity will a length appear to be half of its zero-velocity value?"

2: If a specific case is interesting, then don't change the assignment, but just use a rule-replacement (/. -¿) for the specific case.

3: *As an example of bad practice*, values for an initial length and velocity are assigned. Susequently, the equation is used to make an assigment to length. Here, the *metaphor* for length is lost to a specific instance of length.

4: Here is another example of good practice. A vector (in MATHEMATICA® , a `List`) is defined to give the $x-$ and $y-$components for an idealized falling object. The undefined symbols (*velocity*, *time*, α (take-off angle), and *accel*) are sufficiently numerous that many different cases are immediately derivative. Moreover, the names of the symbols that define the parameters make the result more meaningful and easy to interpret.

5: A specific case can be created by applying one rule-replace to the entire vector.

6: The behavior of a specific case can be visualized by allowing the remaining free parameter (after the rule-replace) to define an $x-$axis in a plot.

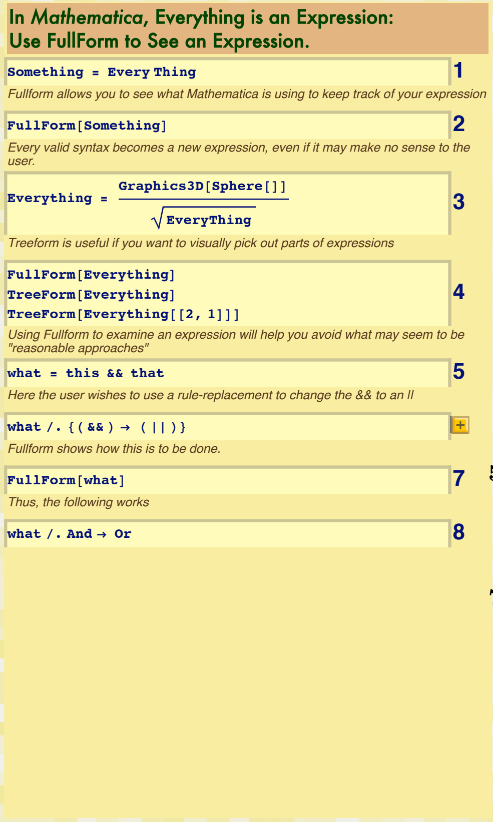
7: More parameters can be easily expored by allowing a second parameter to vary and be visualized. Families of arcs are thus identified; the physical principles become apparent though examination of a how continuum of parameters affect the results.

notebooks, pdf(color), pdf(bw), html

Everything is an Expression; Everything is an Expression.

Everything is an expression. "... I have said it thrice: What I tell you three times is true."

Everything you do or exists in MATHEMATICA® is encoded as an expression. When things start going wrong and misbehaving, this is a good place to start debugging. All the tools in MATHEMATICA® (yes, these are expressions too), are designed to work on expressions; so when using tool, one must understand the form of a expression. What you see on the screen for an expression is *most often* <u>*not*</u> how MATHEMATICA® is keeping track of that expression.



In *Mathematica*, Everything is an Expression:
Use FullForm to See an Expression.

```
Something = Every Thing                                        1
```
Fullform allows you to see what Mathematica is using to keep track of your expression

```
FullForm[Something]                                            2
```
Every valid syntax becomes a new expression, even if it may make no sense to the user.

```
              Graphics3D[Sphere[]]
Everything =  ────────────────────                             3
                 √EveryThing
```
Treeform is useful if you want to visually pick out parts of expressions

```
FullForm[Everything]
TreeForm[Everything]                                           4
TreeForm[Everything[[2, 1]]]
```
Using Fullform to examine an expression will help you avoid what may seem to be "reasonable approaches"

```
what = this && that                                            5
```
Here the user wishes to use a rule-replacement to change the && to an ||

```
what /. {(&&) → (||)}
```
Fullform shows how this is to be done.

```
FullForm[what]                                                 7
```
Thus, the following works

```
what /. And → Or                                               8
```

2: `FullForm` gives you the representation of how MATHEMATICA® is storing an expression. The expression for `Something` has three parts: 1) `Head[Something]` or `Something[[0]]` will return `Times`. 2) `Something[[1]]` is the first argument of an arbitrary number of arguments for `Times`; here `Something[[1]]` is `Every`. 3) `Something[[2]]` (which here is the same as `Something[[-1]]` or `Last[Something]`) is the last argument of `Times`, `Thing`.

3: Any valid syntax produces an expression—even if the intent doesn't make much sense as in this example.

4: Using `FullForm` and `TreeForm` help navigate through MATHEMATICA® 's internal representation.

5–6: Here is an artificial example where `FullForm` comes to the rescue. The user constructs a logical expression, but wants to understand how it behaves when the logical operators are changed. In an attempt to see the effect of changing an `&&` into an `||`, the user tries a rule-replace that results in a syntax error.

7–8: `FullForm` will show that the internal representation of `what` is `And[this,that]`. Thus, using a rule-replace for the `And` works.

3.016 Home
Full Screen
Close
Quit

Non-Dimensionalize Physical Problems: Avoid Units if Possible

This is more of a paradigm about physical sciences than about MATHEMATICA® . However, it is important to see how to use MATHEMATICA® to non-dimensionalize.o For a physical example, consider the simple model for the Bohr one-electron atom:

$$\frac{m_e v^2}{r} = \frac{e^2}{4\pi\epsilon_o r^2} \qquad \text{and} \qquad m_e v r = n\hbar$$

Then, if a characteristic energy $E_{char} = m_e c^2$ (i.e., anything with units of energy that doesn't involve any model parameters), $R_{char} = e^2/(4\pi\epsilon_o m c^2)$ (the classical radius of an electron), and a characteristic velocity c, dimensionless results for the n^{th} Bohr radius, electron velocity, and energy with respect to a completely ionized system are:

$$\frac{r_n}{R_{char}} = \frac{n^2}{\alpha^2} \quad \text{and} \quad \frac{v_n}{c} = \frac{\alpha}{n} \quad \text{and} \quad \frac{E_n}{E_{char}} = \frac{-\alpha}{n^2} \quad \text{where} \quad \alpha \equiv \frac{e^2}{4\pi\epsilon_o \hbar c}$$

α is known as the fine structure constant and is *dimensionless*. Its value is about 1/137 *no matter what units* and is the same on any planet no matter what the size of the inhabitants' feet, their number of fingers, or their arbitrary choice of a system of units. Such dimensionless constants appear and simplify physical results; they make the results easier to compare to physical objects.

It is recommended that non-dimensionalizing is done at the first step. It reduces the number of parameters that appear in equations and reduces the embarassing accident that units don't match.

3.016

3.016 Home

Full Screen

Close

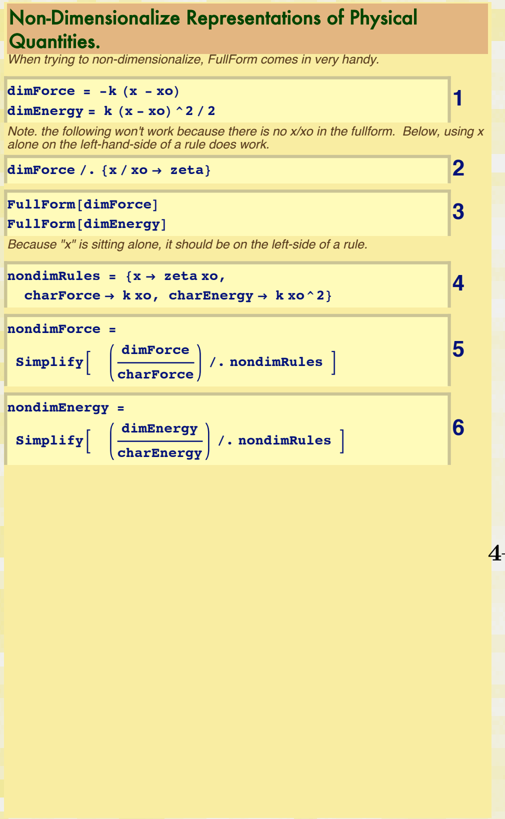
Quit

Suggested Mathematica® Paradigm 3

notebooks, pdf(color), pdf(bw), html

Non-Dimensionaliz Physical Problems

A few examples (and mistakes) are illustrated for the important process of eliminating units.



Non-Dimensionalize Representations of Physical Quantities.

When trying to non-dimensionalize, FullForm comes in very handy.

```
dimForce = -k (x - xo)
dimEnergy = k (x - xo) ^2 / 2
```
1

Note. the following won't work because there is no x/xo in the fullform. Below, using x alone on the left-hand-side of a rule does work.

```
dimForce /. {x / xo → zeta}
```
2

```
FullForm[dimForce]
FullForm[dimEnergy]
```
3

Because "x" is sitting alone, it should be on the left-side of a rule.

```
nondimRules = {x → zeta xo,
  charForce → k xo, charEnergy → k xo^2}
```
4

```
nondimForce =
 Simplify[ (dimForce/charForce) /. nondimRules ]
```
5

```
nondimEnergy =
 Simplify[ (dimEnergy/charEnergy) /. nondimRules ]
```
6

1: The traditional form of Hooke's law and potential energy in a spring are assigned to symbols. There are two parameters in these laws: the spring constant k and the 'force-free length' x_o. The goal is to non-dimensionalize these equations.

2: *This will not work.* An attempt to introduce a non-dimensional length $\zeta = x/x_o$ will fail here because x/x_o does not appear in the expression. (Note `TextForm[x/xo]` is `Times[x, Power[xo, -1]]`)

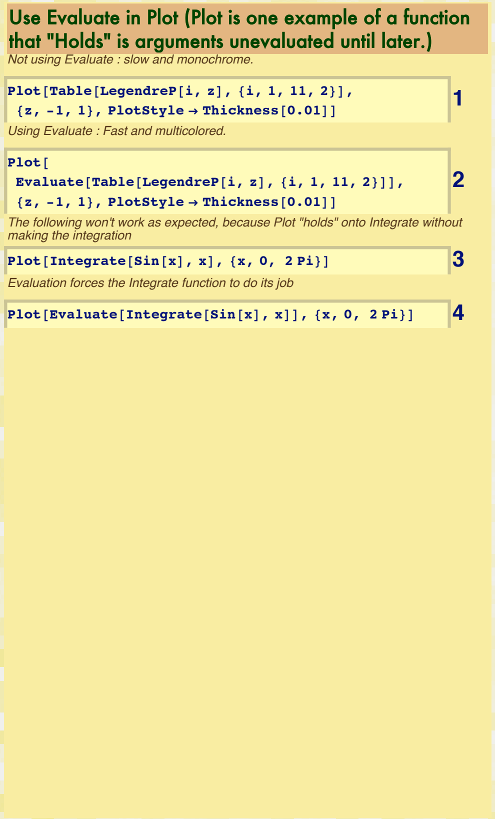
4: A set of rules like this will work in a subsequent replacement: Any x will have to be replaced, and any place that `charForce` or `charEnergy` are used to non-dimensionalize something with force or energy units, the k and $_{x}o$ will cancel.

4–5: Here is an example of using the rules and replacement to find non-dimensional quantities.

notebooks, pdf(color), pdf(bw), html

Using Evaluate in Plot and related functions

The 'normal' behavior of a MATHEMATICA® function is to evaluate its arguments before they are passed onto the function. For example, `Det[amat.bmat]` will first compute the matrix product and pass the result to `Det` the determinant function.
However, there a some functions that 'hold' their arguments unevaluated. This 'hold' behavior is most often encountered in `Plot`.
Without going into the reasons why `Plot` may wish to hold its arguments, this behavior can lead to unexpected results. Here are some examples where the arguments are forced with `Evaluate`.



Use Evaluate in Plot (Plot is one example of a function that "Holds" is arguments unevaluated until later.)

Not using Evaluate : slow and monochrome.

```
Plot[Table[LegendreP[i, z], {i, 1, 11, 2}],
 {z, -1, 1}, PlotStyle → Thickness[0.01]]
```
1

Using Evaluate : Fast and multicolored.

```
Plot[
 Evaluate[Table[LegendreP[i, z], {i, 1, 11, 2}]],
 {z, -1, 1}, PlotStyle → Thickness[0.01]]
```
2

The following won't work as expected, because Plot "holds" onto Integrate without making the integration

```
Plot[Integrate[Sin[x], x], {x, 0, 2 Pi}]
```
3

Evaluation forces the Integrate function to do its job

```
Plot[Evaluate[Integrate[Sin[x], x]], {x, 0, 2 Pi}]
```
4

1: In this case, the `Table` is unevaluated and passed to `Plot`. Here the result is produced more slowly; the lines are not colored the way that they would be if the argument was a list.

2: Forcing the argument of `Plot` to be a list produces a faster and polychromatic result.

3: *This will not work as expected.* The *expression* `Integrate`... is passed to `Plot` and plot doesn't know what to do with the expression.

4: Forcing evaluation produces the expected result.

3.016 Home

Full Screen

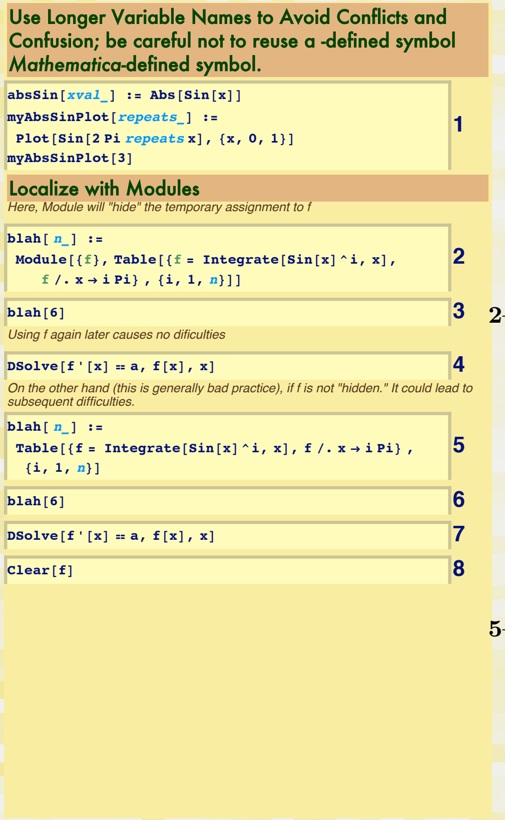
Close

Quit

Longer and Descriptive Variable Names can be Self-Documenting

In books and on scratch paper, it might be wasteful and tedious to use longer notation; this is probably why $f(x)$ dominates discussion even though the result of the function, (e.g., profit) and the argument of the function (e.g., cost) are the underlying ideas.

On a screen, constraints of space and bad handwriting are much less. Why not use variables names that are meaningful and help document what you are thinking about?



Use Longer Variable Names to Avoid Conflicts and Confusion; be careful not to reuse a -defined symbol *Mathematica*-defined symbol.

```
absSin[xval_] := Abs[Sin[x]]
myAbsSinPlot[repeats_] :=
 Plot[Sin[2 Pi repeats x], {x, 0, 1}]
myAbsSinPlot[3]
```
1

Localize with Modules

Here, Module will "hide" the temporary assignment to f

```
blah[ n_] :=
 Module[{f}, Table[{f = Integrate[Sin[x]^i, x],
    f /. x → i Pi} , {i, 1, n}]]
```
2

```
blah[6]
```
3

Using f again later causes no dificulties

```
DSolve[f'[x] == a, f[x], x]
```
4

On the other hand (this is generally bad practice), if f is not "hidden." It could lead to subsequent difficulties.

```
blah[ n_] :=
 Table[{f = Integrate[Sin[x]^i, x], f /. x → i Pi} ,
  {i, 1, n}]
```
5

```
blah[6]
```
6

```
DSolve[f'[x] == a, f[x], x]
```
7

```
Clear[f]
```
8

1: These are examples of naming a function and its argument in a contextual way. MATHEMATICA® 's internal functions use capital letters in standard way—all the first letters are capitalized. If new symbols have a lower-case first letter, then which functions are the user's and which are intrinsic to MATHEMATICA® are unlikely to be confused.

2–3: Sometimes a variable is defined within a function that is not needed once the function is finished. For example, consider `Integrate[f[x],x,0,1]` and `Integrate[f[y],y,0,1]`. These should produce the same result: x and y are used by the function internally and then abandoned. The temporary values of x or y should have no influence *outside* of the function.

Especially with larger complicated functions, creation and debugging are simplified by defining internal variables.

`Module` (as well as `Block` and `With`) provides a way to hide internal variables from the rest of a program.

Using `Module` when an internal variable is used no where else, is good practice.

4: Because f was placed in a module, its previous definition does not interfere.

5–8: *This is an example of bad practice and will lead to an error.* Because `Module` is not used, the appearance of f in **7** will inherent its definition for its last use in **6**.

3.016 Home

Full Screen

Close

Quit

Suggested MATHEMATICA® Paradigm 6

notebooks, pdf(color), pdf(bw), html

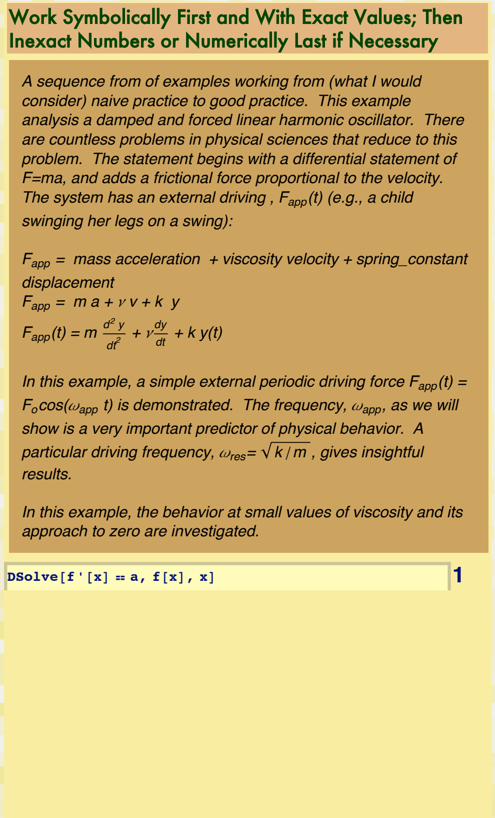
Work Symbolically and Delay the Introduction of Numbers: Introduction

Examples with increasingly better style are presented in the next five versions of exploring the solution to a damped-forced harmonic oscillator.

The goal is to examine the behavior near resonance and in the limit of zero viscosity.

Physically, the problem is similar to pushing a pendulum in-time with the pendulum's period. The viscosity correlates the wind or friction forces that would make an observed non-forced pendulum slowly come to rest.

A mathematical development is included in the example.



Work Symbolically First and With Exact Values; Then Inexact Numbers or Numerically Last if Necessary

A sequence from of examples working from (what I would consider) naive practice to good practice. This example analysis a damped and forced linear harmonic oscillator. There are countless problems in physical sciences that reduce to this problem. The statement begins with a differential statement of F=ma, and adds a frictional force proportional to the velocity. The system has an external driving , $F_{app}(t)$ (e.g., a child swinging her legs on a swing):

F_{app} = mass acceleration + viscosity velocity + spring_constant displacement

$F_{app} = m\, a + \nu\, v + k\; y$

$F_{app}(t) = m \frac{d^2 y}{dt^2} + \nu \frac{dy}{dt} + k\, y(t)$

In this example, a simple external periodic driving force $F_{app}(t) = F_o \cos(\omega_{app}\ t)$ is demonstrated. The frequency, ω_{app}, as we will show is a very important predictor of physical behavior. A particular driving frequency, $\omega_{res} = \sqrt{k/m}$, gives insightful results.

In this example, the behavior at small values of viscosity and its approach to zero are investigated.

```
DSolve[f'[x] == a, f[x], x]
```
1

1: The set of examples all use the built-in MATHEMATICA® function `DSolve`. `DSolve` takes one or more equations that involve a function and its derivatives, the function that is being solved for, and the variable on which the function depends.

The result is returned in the form of a rule that can be used in subsequent rule-replaces.

3.016 Home

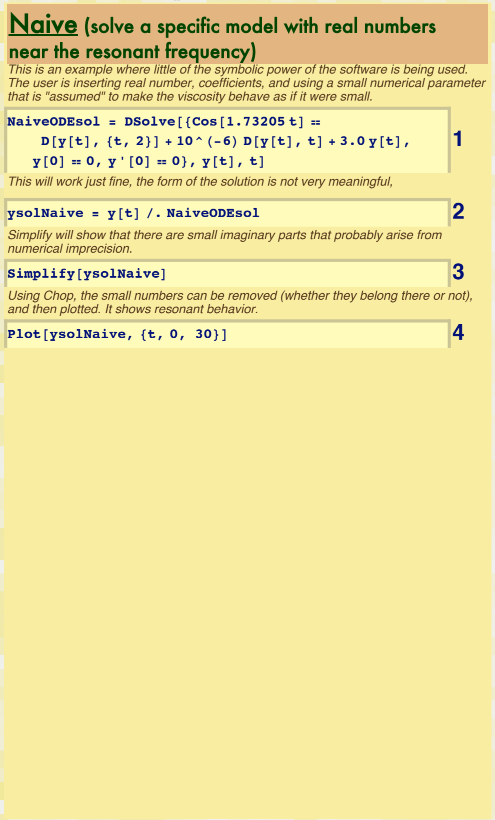
Full Screen

Close

Quit

Work Symbolically and Delay the Introduction of Numbers: Naive Approach

Here, the user puts in numbers for the resonant frequency and makes a guess at what would represent a small parameter.



Naive (solve a specific model with real numbers near the resonant frequency)

This is an example where little of the symbolic power of the software is being used. The user is inserting real number, coefficients, and using a small numerical parameter that is "assumed" to make the viscosity behave as if it were small.

```
NaiveODEsol = DSolve[{Cos[1.73205 t] ==
     D[y[t], {t, 2}] + 10^(-6) D[y[t], t] + 3.0 y[t],
    y[0] == 0, y'[0] == 0}, y[t], t]
```
1

This will work just fine, the form of the solution is not very meaningful,

```
ysolNaive = y[t] /. NaiveODEsol
```
2

Simplify will show that there are small imaginary parts that probably arise from numerical imprecision.

```
Simplify[ysolNaive]
```
3

Using Chop, the small numbers can be removed (whether they belong there or not), and then plotted. It shows resonant behavior.

```
Plot[ysolNaive, {t, 0, 30}]
```
4

1: This will work fine, it just won't be very informative about the general behavior of the system.

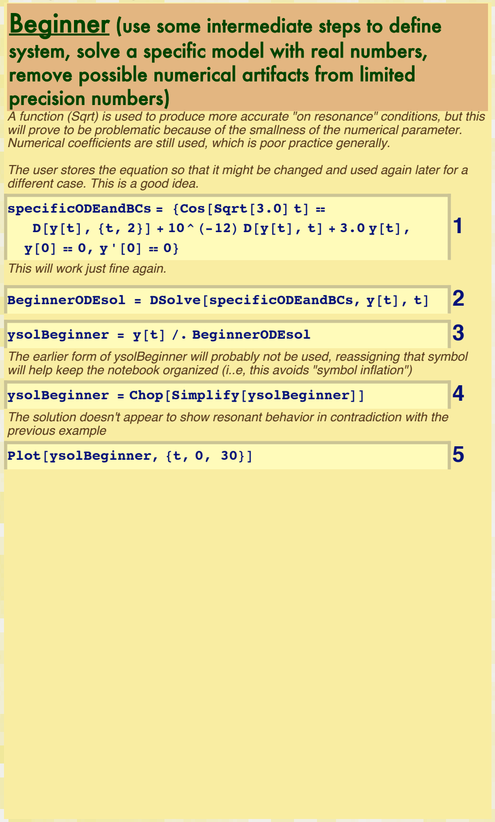
2: The solution can be extracted from the rules produced by `DSolve`.

3: The resulting solution can be simplified and show that numerical imprecision is probably creeping into the results.

4: Nevertheless, the solution be plotted However, `Chop` may be needed to remove small imaginary parts of the solution.

Work Symbolically and Delay the Introduction of Numbers: Beginner Approach

Here the user defines an intermediate expression so that the `DSolve` step is a bit easier to read. However, the user is still using numbers and guessing at a small parameter.



Beginner (use some intermediate steps to define system, solve a specific model with real numbers, remove possible numerical artifacts from limited precision numbers)

A function (Sqrt) is used to produce more accurate "on resonance" conditions, but this will prove to be problematic because of the smallness of the numerical parameter. Numerical coefficients are still used, which is poor practice generally.

The user stores the equation so that it might be changed and used again later for a different case. This is a good idea.

```
specificODEandBCs = {Cos[Sqrt[3.0] t] ==
    D[y[t], {t, 2}] + 10^(-12) D[y[t], t] + 3.0 y[t],
   y[0] == 0, y'[0] == 0}
```
1

This will work just fine again.

```
BeginnerODEsol = DSolve[specificODEandBCs, y[t], t]
```
2

```
ysolBeginner = y[t] /. BeginnerODEsol
```
3

The earlier form of ysolBeginner will probably not be used, reassigning that symbol will help keep the notebook organized (i..e, this avoids "symbol inflation")

```
ysolBeginner = Chop[Simplify[ysolBeginner]]
```
4

The solution doesn't appear to show resonant behavior in contradiction with the previous example

```
Plot[ysolBeginner, {t, 0, 30}]
```
5

1: Defining the set of equations this way allows one to make changes to an expression and reuse it in the solution step.

4: Because `ysolBeginner` will probably not be used in its previous form, the expression is reassigned. This reduces "symbol inflation," but could possible introduce errors because the use of `Chop` could (but is unlikely to) remove relevant parts of the solution.

3.016 Home

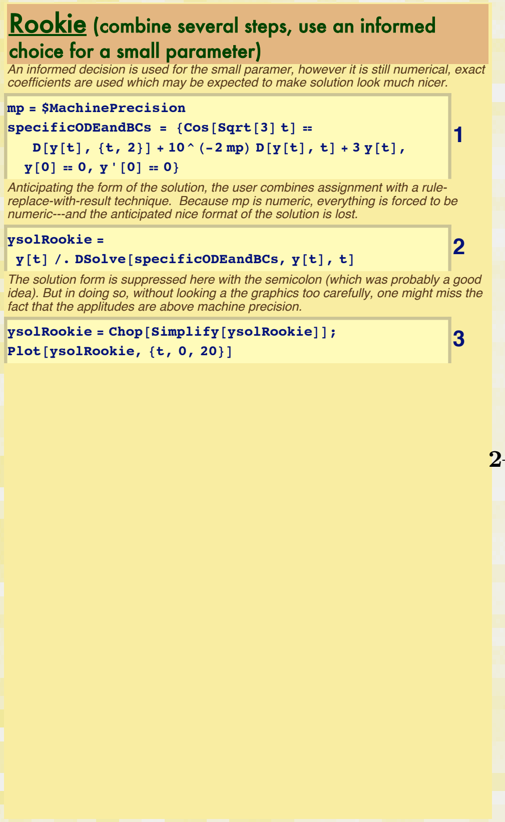
Full Screen

Close

Quit

Work Symbolically and Delay the Introduction of Numbers: Rookie Approach

The user uses a more informed method to introduce a meaningful small parameter and intends to uses exact numbers. Although it is a bit deceptive, the clever introduction of a small parameter inserts numerical values into the equation. Thus, `DSolve` will produce a numerical approximation as well—which was not the intent of the rookie.



Rookie (combine several steps, use an informed choice for a small parameter)

An informed decision is used for the small paramer, however it is still numerical, exact coefficients are used which may be expected to make solution look much nicer.

```
mp = $MachinePrecision
specificODEandBCs = {Cos[Sqrt[3] t] ==
    D[y[t], {t, 2}] + 10^(-2 mp) D[y[t], t] + 3 y[t],
   y[0] == 0, y'[0] == 0}
```
1

Anticipating the form of the solution, the user combines assignment with a rule-replace-with-result technique. Because mp is numeric, everything is forced to be numeric---and the anticipated nice format of the solution is lost.

```
ysolRookie =
  y[t] /. DSolve[specificODEandBCs, y[t], t]
```
2

The solution form is suppressed here with the semicolon (which was probably a good idea). But in doing so, without looking a the graphics too carefully, one might miss the fact that the applitudes are above machine precision.

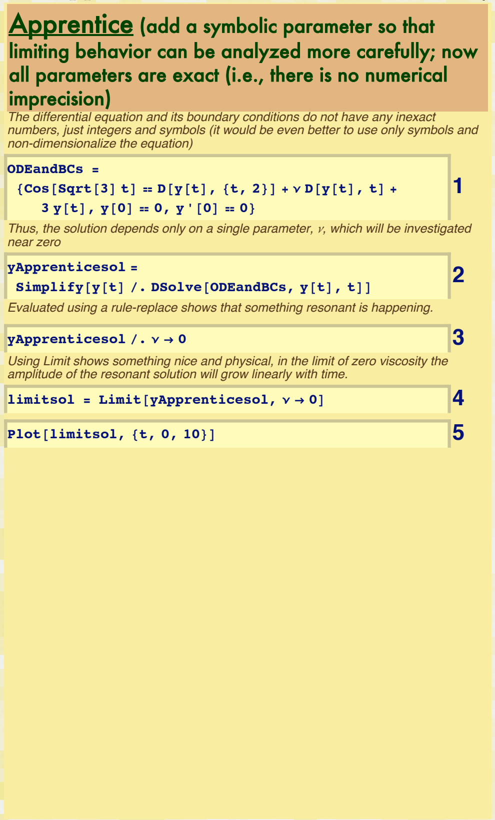
```
ysolRookie = Chop[Simplify[ysolRookie]];
Plot[ysolRookie, {t, 0, 20}]
```
3

1: The user probably looked in the Help Browser for something about "precision" and found that MATHEMATICA® keeps track of what the precision is on the current CPU. These system specific values are stored as dollar-sign values, such as `$MachinePrecision`.

2–3: The user plots the results, but without regard to the range over which the interesting behavior appears and misses the point.

Work Symbolically and Delay the Introduction of Numbers: Appretice Approach

The apprentice uses exact numbers and symbols and then analyzes solutions for particular values of the symbols.



Apprentice (add a symbolic parameter so that limiting behavior can be analyzed more carefully; now all parameters are exact (i.e., there is no numerical imprecision)

The differential equation and its boundary conditions do not have any inexact numbers, just integers and symbols (it would be even better to use only symbols and non-dimensionalize the equation)

```
ODEandBCs =
 {Cos[Sqrt[3] t] == D[y[t], {t, 2}] + ν D[y[t], t] +
    3 y[t], y[0] == 0, y'[0] == 0}                      1
```

Thus, the solution depends only on a single parameter, ν, which will be investigated near zero

```
yApprenticesol =
 Simplify[y[t] /. DSolve[ODEandBCs, y[t], t]]           2
```

Evaluated using a rule-replace shows that something resonant is happening.

```
yApprenticesol /. ν → 0                                 3
```

Using Limit shows something nice and physical, in the limit of zero viscosity the amplitude of the resonant solution will grow linearly with time.

```
limitsol = Limit[yApprenticesol, ν → 0]                 4
```

```
Plot[limitsol, {t, 0, 10}]                              5
```

2: The user decides to eliminate a few intermediate steps (hoping that this more concise version will be easy to read at some time in the future). The solution rules appear as an internal expression and the user wraps the result inside a `Simplify`. MATHEMATICA® keeps track of what the precision is on the current CPU. These system specific values are stored as dollar-sign values, such as `$MachinePrecision`.

3: The goal is to analyze the behavior of the resonant solution in the limit of small viscosity; the user looks at a particular case, the solutions value at $\eta = 0$. In this case, the evaluation at zero gives an unphysical result.

4: Allowing for the possibility that the function's limit doesn't equal its value, the function `Limit` is used to check this case.

5: Because the resulting expression has a very simple form, the user can see the relevant bounds for plotting by inspection.

3.016 Home

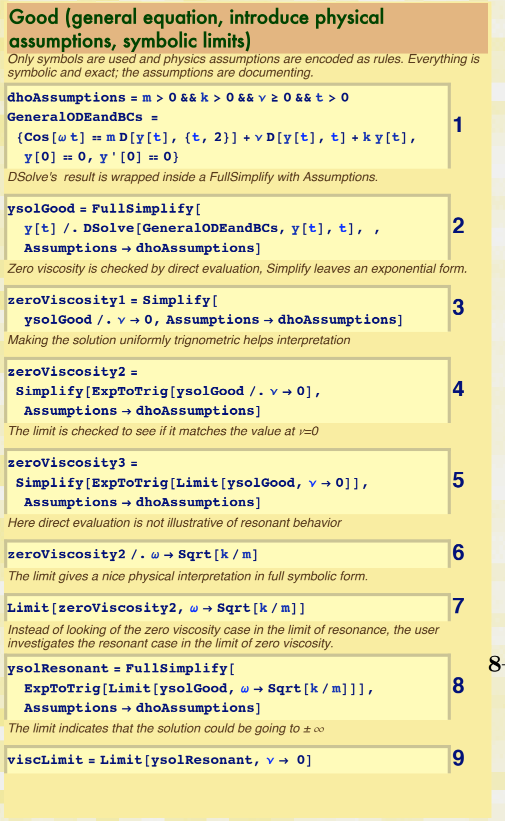
Full Screen

Close

Quit

Work Symbolically and Delay the Introduction of Numbers: Good Approach

Symbols are used instead of numbers. Physical assumptions are encoded so that simplifying procedures will give the nicer results that apply in the more specific physical situation. Some effort is invested into getting the solution into a nice readable form and this produces informative results. The rules for resonance appear and the distinction between two different kinds of physical limits arise naturally.



Good (general equation, introduce physical assumptions, symbolic limits)

Only symbols are used and physics assumptions are encoded as rules. Everything is symbolic and exact; the assumptions are documenting.

```
dhoAssumptions = m > 0 && k > 0 && ν ≥ 0 && t > 0
GeneralODEandBCs =
 {Cos[ω t] == m D[y[t], {t, 2}] + ν D[y[t], t] + k y[t],
  y[0] == 0, y'[0] == 0}                                  1
```

DSolve's result is wrapped inside a FullSimplify with Assumptions.

```
ysolGood = FullSimplify[
  y[t] /. DSolve[GeneralODEandBCs, y[t], t], ,             2
  Assumptions → dhoAssumptions]
```

Zero viscosity is checked by direct evaluation, Simplify leaves an exponential form.

```
zeroViscosity1 = Simplify[                                 3
  ysolGood /. ν → 0, Assumptions → dhoAssumptions]
```

Making the solution uniformly trignometric helps interpretation

```
zeroViscosity2 =
 Simplify[ExpToTrig[ysolGood /. ν → 0],                    4
  Assumptions → dhoAssumptions]
```

The limit is checked to see if it matches the value at ν=0

```
zeroViscosity3 =
 Simplify[ExpToTrig[Limit[ysolGood, ν → 0]],               5
  Assumptions → dhoAssumptions]
```

Here direct evaluation is not illustrative of resonant behavior

```
zeroViscosity2 /. ω → Sqrt[k / m]                          6
```

The limit gives a nice physical interpretation in full symbolic form.

```
Limit[zeroViscosity2, ω → Sqrt[k / m]]                     7
```

Instead of looking of the zero viscosity case in the limit of resonance, the user investigates the resonant case in the limit of zero viscosity.

```
ysolResonant = FullSimplify[
  ExpToTrig[Limit[ysolGood, ω → Sqrt[k / m]]],             8
  Assumptions → dhoAssumptions]
```

The limit indicates that the solution could be going to ± ∞

```
viscLimit = Limit[ysolResonant, ν → 0]                     9
```

1: Only symbols are used for the parameters in the equation to be solved. However, the user knows something about the values for the physical system that is being modeled.

2: The form of the solution is obtained by simplifying with assumptions. However, the resulting form will not be as simple as hoped.

3: Without trying to finesse the solution into a nicer form first, the user evaluates at zero-viscosity. The result will indicate that the conditions for resonance are appearing, but will need more careful inspection—perhaps the numerator and denominator both go to zero.

4: The user enforces that all oscillatory terms appear as trigonometric functions and then looks at the value at zero viscosity. The result is more informative and makes it clearer that the limit needs more careful attention.

5: Here, the user finds that the value of the expression and its limit have the same value.

6: So, the user takes the zero-viscosity case and evaluates its value at resonance. The result will not help understand the physics of the problem—the user suspects that the numerator and denominator both go to zero.

7: The limit is checked and gives the nice physical result that the amplitude grows linearly with time as the system approaches the resonance condition at zero viscosity.

8–9: The other way of looking at the limit—working at resonance and letting viscosity go to zero—yields a different result. The solution will diverge to $\pm\infty$ depending on the sign of $\sin(t)$.

3.016 Home

Full Screen

Close

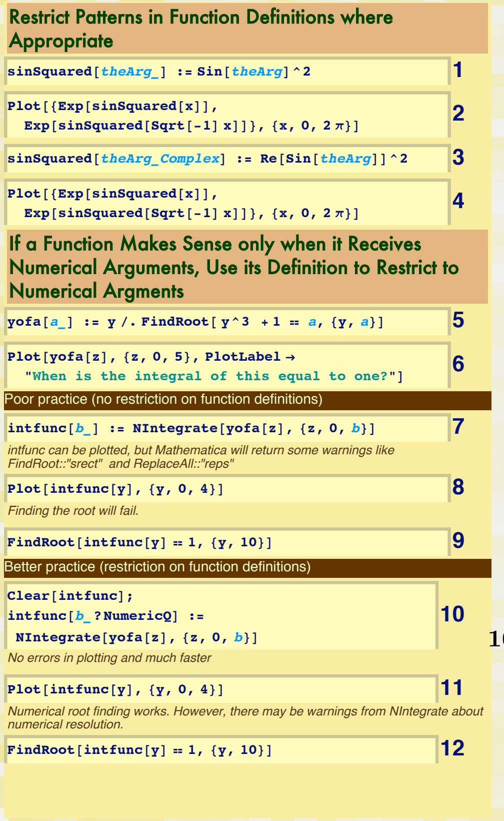
Quit

Suggested MATHEMATICA® Paradigm 12

notebooks, pdf(color), pdf(bw), html

Restrict Patterns where Appropriate, Especially Numerical Patterns for Numerical Functions

When a function is expected to be used for a limited type of arguments, build in the type of argument by restricting a pattern. When a function *only* makes sense for numerical arguments, define the function for only numerical arguments.



Restrict Patterns in Function Definitions where Appropriate

```
sinSquared[theArg_] := Sin[theArg]^2
```
1

```
Plot[{Exp[sinSquared[x]],
  Exp[sinSquared[Sqrt[-1] x]]}, {x, 0, 2 π}]
```
2

```
sinSquared[theArg_Complex] := Re[Sin[theArg]]^2
```
3

```
Plot[{Exp[sinSquared[x]],
  Exp[sinSquared[Sqrt[-1] x]]}, {x, 0, 2 π}]
```
4

If a Function Makes Sense only when it Receives Numerical Arguments, Use its Definition to Restrict to Numerical Argments

```
yofa[a_] := y /. FindRoot[ y^3 +1 == a, {y, a}]
```
5

```
Plot[yofa[z], {z, 0, 5}, PlotLabel →
  "When is the integral of this equal to one?"]
```
6

Poor practice (no restriction on function definitions)

```
intfunc[b_] := NIntegrate[yofa[z], {z, 0, b}]
```
7

intfunc can be plotted, but Mathematica will return some warnings like FindRoot::"srect" and ReplaceAll::"reps"

```
Plot[intfunc[y], {y, 0, 4}]
```
8

Finding the root will fail.

```
FindRoot[intfunc[y] == 1, {y, 10}]
```
9

Better practice (restriction on function definitions)

```
Clear[intfunc];
intfunc[b_?NumericQ] :=
 NIntegrate[yofa[z], {z, 0, b}]
```
10

No errors in plotting and much faster

```
Plot[intfunc[y], {y, 0, 4}]
```
11

Numerical root finding works. However, there may be warnings from NIntegrate about numerical resolution.

```
FindRoot[intfunc[y] == 1, {y, 10}]
```
12

1: Here there is no restriction on the pattern, the function will work the same for any argument.

2: The behavior may not be what was intended for complex arguments.

3: Here, the function is defined to behave differently if the argument is complex (i.e., using the restriction `_Complex`).

4: The plot shows that the real part stays at +1.

5: The **5–12** parts of the code will demonstrate how a numerical process (here root-finding) will fail if functions are *not* defined for numerical arguments. The example is a numerical function that finds a y where $y^3 + 1 - a$ is zero as a function of a.

6: This function will work because `Plot` only supplies numerical arguments; `yofa` would throw an error if it received an undefined symbol; but `Plot` is not supplying any of those. In the following, two numerical functions are called in sequence to find where the integral of these values is 1.

7: *This is poor practice and will lead to an error.* `NIntegrate` will only make sense if the integrand evaluates to a number, but here no restriction to numerical arguments is made.

8: This result can be plotted properly, although `Plot` will issue some warnings.

9: However, a subsequent numerical operation on the function, like `FindRoot`, will fail.

10–12: If the pattern in the function is restricted to numerical arguments, then plotting will be faster and will issue no warnings. Subsequent numerical operations work fine (although in this case, `NIntegrate` will issue some warnings about convergence.

3.016

3.016 Home

Full Screen

Close

Quit