Chapter 13 More on building and applying scripts with \textit{while} repetition

Section 13.1 Anatomy of a \textit{while} repetition; roles of variables

In Chapter 11, we described the syntactic requirements of repetition through \textit{while}, and showed a few examples of "repetition in action". Examples in sections 11.2.1 and 11.2.3 concerned adding together a number of terms. Each term was expected to be smaller in magnitude than the previous one. The repetition concerned computing the next term and adding it onto a variable $s$ that was used to compute the sum.

While programming languages allow a lot of freedom in what you compute in a \textit{while} repetition, there are standard patterns typically at work. While the situation is not as simple as "memorizing five patterns that you just pick from", being aware of the patterns can often provide useful guidance when you are designing your scripts.

Repetitions work because there is a way to do the task at hand (computing and adding together many terms) one step at a time. Each step uses information available from the variables of the script. Rather than fixing the value of a variable through one assignment and then just using that value for the rest of the script, \textit{while} computations typically change the value of some variables with each repetition.

\textit{while} repetitions usually have these common features:

Before the repetition starts, there are assignments that initialize the values of many variables that appear within the instructions that are "looped".

The variables that occur within the repetition can have the following kinds of roles:

1. \textit{variables that are fixed values}. They are assigned during the initialization but don't change afterwards. \textit{val} is an example of this in Example 11.2.1.1.
2. \textit{variables that are steppers}. For example, with the summations being computed in Examples 11.2.1 and 11.2.3, the variable $i$ is going 1,2,3,4,... or 1,3,5,7,9,... Other possible sequences would be ones that start from zero: 0,1,2,3,4, or 0,2,4,6,8. These variables are set to a value during the initialization, and then updated with an increment (e.g. adding $i := i+1$; or $i := i+2$) with each repetition.
3. \textit{variables that gather the final result}. In the examples in sections 11.2.1 and 11.2.3, the variable $s$ played this role: \textit{term} was computed and added onto $s$. Note that $s$ is initialized to 0 so that final value of $s$ after the repetitions are over will be the sum of the things that were added onto $s$ in the repetition. In the example in section 11.2.2, the variable \textit{frame} plays this role. \textit{frame} is initialized to be an empty table. Each repetition adds another element to the table.

Each repetition of the \textit{while} leaves the accumulation variable one step closer to its final objective (e.g. adding together all the terms of a sum).
4. \textit{Variables that hold the most recent value} going through a succession. Usually (but not always) the most recent value is computed from previous values. For example, \textit{term} in Example 11.2.1.1 is the most recent term of the sum that the script is computing. $m$ in
Example 12.2.1.2 is the value of the most recent move in the simulation/animation.

5. Variables that hold the previous value. Sometimes in order to compute the next value, it is necessary to keep not only the present value, but the one before that. Since a variable can hold only one value at a time, an extra variable needs to be used to play that role. The examples given so far do not have any variables with this role, but we will soon see situations where this is needed.

**Section 13.2 Finding the most recent value from the previously computed information**

For simple tasks such as summing or updating the position of a moving particle, most of the script is fairly easy to write once decisions are made about what the computation is going to be like. For fixed value variables, all you need to do is decide on what value you need for them to be, and then to assign them in the initialization. For steppers, you need to determine the initial value of the stepper and the formula that will change the stepper into the next value. Typically, this is an instruction such as $i := i+1$; or $i := i+2$; or $i:= 2*i$; Gathering variables usually also have a straightforward format; the difficulty may lie more in figuring out the formula you're going to use to calculate the next gathered value.

Discovering that your computation needs a variable with a "most recent value" role and then figuring out how to compute the most recent value is typically the most difficult part of designing a while loop. It takes some amount of experience and insight to determine that the way to compute a term in Example 11.2.1.1 is to do $\text{term} := (-1)^{(i-1)/2}*(\text{evalpt^i})/i!$;

Often times it is easier to find out how to compute the most recent value by looking at how you would modify the previous value. For example, we could initialize term as in Example 11.2.1.1 as before, but recognize that the difference between $x^1$ and $-x^3$ would be to multiply term by $-x^2$ and then divide by $(2*3)$. Thus, if we initialize term := $x/1$; then next assignment we would want to do would be $\text{term} := -\text{term}*x^2/(2*3);$. If we continue this to the next step we begin to see how we can use the value of $i$ to compute the denominator. The next term is $\frac{x^5}{1\cdot2\cdot3\cdot4\cdot5}$. If we want to get that value from the previous value of term, we multiply by $-\frac{x^2}{4\cdot5}$. But since $i$ is 5 at that point, we would do $\text{term} := -\text{term}*x^2/(i-1)*i);$. By inspecting the updates needed for a few terms, we have discovered directions for the update that work in the same way for each "next value" computation. This leads us to the alternative way of computing the sum of 11.2.1.1, displayed below. The entire script is the same as before, except for the line that computes the most recent value of term from the previous value.

The most recent value is usually an incremental modification of the previous values of script variables. For example, term in Example 11.2.1.1. has its most recent value computed by
combining together the previous value, along with the newest value of move doesn't depend in a deterministic way of the sequence counter frameNumber; but each repetition both increments the frameNumber and gives a fresh value for m. xmfp and term have these roles in Example 12.2.1.3.

<table>
<thead>
<tr>
<th>Example 11.2.1.1</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The initial instructions in the sequence set up the variables i, the value of x that we want to use (.3), the value of the first term of the sum, the initial value of the variable that will contain the sum, and the value of the small number. The instructions to be repeated are indented and are found within the while...do... end do. The sequence of four instructions is repeated until the magnitude of term is found to be less than the small number. This check occurs just before the instructions are done for the first time, and then each time after the four instructions are performed. At the very end we print out a message that compares the value of the sum, with Maple's notion of what sin(.3) is.</td>
</tr>
</tbody>
</table>

Value of i
Value of evalpt
Value of term (first time)
Side effect of print
# Example 11.2.1

# initialize variables
i := 1;
evalpt := .3;
term := (evalpt^i)/i!;
print("term is", term);
s := term; # sum so far
tol := 10e-7;

# add on successive terms
while abs(term) >= tol do
    i := i+2;
    term := -term* (evalpt^2)/ ((i-1)*i);
    print("term is", term);
    s := s+term;
end do;

print("sum is: ", s, " compared to:", sin(evalpt));

1
0.3
0.3
"term is", 0.3
0.3

Value of s (first time)
Value tol, the small number
Next value of i (first time repetition)
Next value of term
Side effect of print statement within while.
Learning how to design the actions of a variable with a "most recent value" role is not something that arrives wholly formed in a sudden flash. One studies examples, and tries to find some that seem similar to what your problem needs. Even if an example provides the template for how to do the problem, determining the particulars for the update will require more effort. You should work out by hand what the first few computations would be, and then discover how to make a general formula that would work in the same way for all the steps as far as you have worked out by hand. Studying examples can give you ideas, but it would be overly optimistic to believe that you can take the exact same formula from an example and have it solve your (different) problem.

### Section 13.3 Stepper-controlled loops with for

A very common pattern in scripts with repetitions is to have the repetition counted by and controlled by the value of a variable playing a stepper role. For example, we can print out the trig identities corresponding to \( \sin(i \cdot x) \) for \( i = 1, 2, 3, 4, 5 \) by:
#Example 13.3.1
#Stepper controlled loop, i ranging from 1 to 5

i := 0:
while (i<5) do
    i := i+1;
    printf("sin(%d*x) = %a\n", i, expand(sin(i*x)));
end do:

sin(1*x) = sin(x)
sin(2*x) = 2*sin(x)*cos(x)
sin(3*x) = 4*sin(x)*cos(x)^2-sin(x)
sin(4*x) = 8*sin(x)*cos(x)^3-4*sin(x)*cos(x)
sin(5*x) = 16*sin(x)*cos(x)^4-12*sin(x)*cos(x)^2+sin(x)

Note that the stepper variable $i$ is initialized to zero, but takes on the values 1 through 5 when the repetition is active. Having to remember to initialize the stepper to 0 instead of 1, and remembering that the continuation condition is $i<5$ rather than $i<4$ or $i<=5$, can be tricky to get right.

Because this "step from 1 to $n$" pattern occurs often, Maple has special language for it:

for $var$ from $init$ to $final$ do .... end do

provides a repetition where the stepper variable $var$ takes on the values $init$, $init +1$, $init +2$, .etc. all the way and including $final$. Example 13.3.2 show this alternative way of setting up the repetition.
# Example 13.3.2
# Stepper controlled loop, i ranging from 1 to 5
for i from 1 to 5 do
  printf("sin(%dx) = %f\n",i,evalf(sin(i*x)));
end do:

\[
\begin{align*}
\sin(1\times x) & = \sin(x) \\
\sin(2\times x) & = 2\times \sin(x)\times \cos(x) \\
\sin(3\times x) & = 4\times \sin(x)\times \cos(x)^2-\sin(x) \\
\sin(4\times x) & = 8\times \sin(x)\times \cos(x)^3-4\times \sin(x)\times \cos(x) \\
\sin(5\times x) & = 16\times \sin(x)\times \cos(x)^4-12\times \sin(x)\times \cos(x)^2+\sin(x)
\end{align*}
\]

The advantage of using `for` instead of `while` in this situation are that it takes less typing and less careful thinking since the initialization and the continuation condition are automatically generated.

There are variants of `for` that are also useful.

<table>
<thead>
<tr>
<th>Table 13.3.1 Varieties of iteration of for</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(a)</strong></td>
</tr>
<tr>
<td><strong>(a)</strong></td>
</tr>
</tbody>
</table>

# Table 13.3.1 (a)
# Stepper controlled loop, i ranging
# from 1 to 5
for i from 1 to 5 do
  printf("sin (%d) = %f\n",i,evalf(sin (i)));
end do:

\[
\begin{align*}
\sin(1) & = 0.841471 \\
\sin(2) & = 0.909297
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>(b)</th>
<th>for var from init to final by stepval do repeated statements end do</th>
<th>Initializes var to <em>init</em>. Repeats with stepper values of <em>init</em>, <em>init</em>+stepval, <em>init</em>+2*stepval, etc. up to and including <em>final</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#Table 13.3.1 (b) #Stepper controlled loop, <em>i</em> ranging # from 1 to 2 in steps of .2 for <em>i</em> from 1 to 2 by .2 do printf(&quot;sin (%f) = %f\n&quot;, <em>i</em>, evalf(sin(<em>i</em>))); end do:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sin(1.000000) = 0.841471 sin(1.200000) = 0.932039 sin(1.400000) = 0.985450 sin(1.600000) = 0.999574 sin(1.800000) = 0.973848 sin(2.000000) = 0.909297</td>
</tr>
<tr>
<td></td>
<td>(c)</td>
<td>for var from init while condition do repeated statements end do</td>
<td>Initializes var to <em>init</em>. If the <em>condition</em> is true then the statements are performed and the value of <em>var</em> is incremented by one as with the original version of for. If the <em>condition</em> is still true, another repetition and another step is taken. This is done over and over until the <em>condition</em> is false.</td>
</tr>
</tbody>
</table>
Table 13.3.1 (c)
Stepper controlled loop, i ranging from 1 to 2 in steps of .2

\[ s := 1; \]
\[ \text{tol} := 10e-10; \]
\[ x := .05; \]
\[ \text{term} := 1; \]
\[ \text{for } i \text{ from 1 while abs } (\text{term}) \geq \text{tol} \text{ do} \]
\[ \text{term} := \text{term} \times x/i; \]
\[ s := s + \text{term}; \]
\[ \text{end do;} \]
\[ \text{printf}("s is:%10.8f, exp (\%f) is:%10.8f", s, x, \exp (x)); \]

1
1.0 10^-9
0.05
1

s is:1.05127110, exp (0.050000) is:1.05127110

| (d) | for var from init to final by stepval while condition do repeated statements end do | The repetition stops when either var reaches final or the while condition becomes false. |
# Table 13.3.1 (d)

Stepper controlled loop, i ranging from 1.0 to 0 in steps of -.1

# Stops when it reaches 0 or when ln(x) becomes less than -1.

```
s := 1;
x := 1.0;
tol := 1.0;
term := ln(x);
for x from 1.0 to 0 by -.01 while abs(term)<tol do
    term := ln(x);
    printf("ln(%4.2f) is: %10.8f\n", x, term);
end do:
```


1
1.0
1.0
0.

ln(1.00) is: 0.00000000
ln(0.99) is: -0.01005034
ln(0.98) is: -0.02020271
ln(0.97) is: -0.03045921
ln(0.96) is: -0.04082199
ln(0.95) is: -0.05129329
ln(0.94) is: -0.06187540
ln(0.93) is: -0.07257069
ln(0.92) is: -0.08338161
ln(0.91) is: -0.09431068
ln(0.90) is: -0.10536052
ln(0.89) is: -0.11653382
ln(0.88) is: -0.12783337
ln(0.87) is: -0.13926207
ln(0.86) is: -0.15082289
ln(0.85) is: -0.16251893
ln(0.84) is: -0.17435339
ln(0.83) is: -0.18632958
\[\ln(0.82) \text{ is: } -0.19845094\]
\[\ln(0.81) \text{ is: } -0.21072103\]
\[\ln(0.80) \text{ is: } -0.22314355\]
\[\ln(0.79) \text{ is: } -0.24846136\]
\[\ln(0.78) \text{ is: } -0.27443685\]
\[\ln(0.77) \text{ is: } -0.26136476\]
\[\ln(0.76) \text{ is: } -0.28768207\]
\[\ln(0.75) \text{ is: } -0.30110509\]
\[\ln(0.74) \text{ is: } -0.31471074\]
\[\ln(0.73) \text{ is: } -0.32850407\]
\[\ln(0.72) \text{ is: } -0.34249031\]
\[\ln(0.71) \text{ is: } -0.35667494\]
\[\ln(0.70) \text{ is: } -0.37106368\]
\[\ln(0.69) \text{ is: } -0.38566248\]
\[\ln(0.68) \text{ is: } -0.40047757\]
\[\ln(0.67) \text{ is: } -0.41551544\]
\[\ln(0.66) \text{ is: } -0.43078292\]
\[\ln(0.65) \text{ is: } -0.44628710\]
\[\ln(0.64) \text{ is: } -0.46203546\]
\[\ln(0.63) \text{ is: } -0.47803580\]
\[\ln(0.62) \text{ is: } -0.49429632\]
\[\ln(0.61) \text{ is: } -0.51082562\]
\[\ln(0.60) \text{ is: } -0.52763274\]
\[\ln(0.59) \text{ is: } -0.54472718\]
\[\ln(0.58) \text{ is: } -0.56211892\]
\[\ln(0.57) \text{ is: } -0.57981850\]
\[\ln(0.56) \text{ is: } -0.59783700\]
\[\ln(0.55) \text{ is: } -0.61618614\]
\[\ln(0.54) \text{ is: } -0.63487827\]
\[\ln(0.53) \text{ is: } -0.65392647\]
\[\ln(0.52) \text{ is: } -0.67334455\]
\[\ln(0.51) \text{ is: } -0.69314718\]
\[\ln(0.50) \text{ is: } -0.71334989\]
\[\ln(0.49) \text{ is: } -0.73396918\]
\[\ln(0.48) \text{ is: } -0.75502258\]
\[\ln(0.47) \text{ is: } -0.77652879\]
\[\ln(0.46) \text{ is: } -0.79850770\]
\[\ln(0.45) \text{ is: } -0.82098055\]
\[\ln(0.44) \text{ is: } -0.84397007\]
\[\ln(0.43) \text{ is: } -0.86750057\]
\[\ln(0.42) \text{ is: } -0.89159812\]
Section 13.4  More advice about developing scripts

To reiterate the information presented about "debugging" scripts in Chapter 11:

1. Enter the first few lines of the script, probably no more than three or four. Get them to work, then add more. As you get better, you may think you can get away with entering many lines at once. Resist the temptation, as all that will happen will be that you will have more places to look for a problem.
2. If your function involves the definition of several functions, have you tested each function individually before you try to use them together?
3. Identify the portions of the script that are tricky to get right. See if you can develop a simplified version that is easier to get correct. Then through editing, gradually change it into the more complex goal.

Examples of this include:
   - If you are making a movie, can you draw just the first frame (as a non-movie)?
   - If you are using a built-in operation that has many parameters, can you get it to work without including the optional ones? For example, in plotting, an extra parameter might be color=red. Can you get the remainder of the plotting script to work without specifying the color? Having fewer things in the script means fewer things to check out for the "bug".

4. If you are making a while or for, does the first repetition work correctly? You will need to look at the execution trace of the initialization and the first repetition, so put in print or printf instructions that make reading the trace easier. Often times once you get the first two repetitions to work correctly, then things work correctly.
5. Experienced programmers often insert print or printf statements into the script to display the values of variables in an easier to read fashion. It takes more effort to enter the print statement, but the time savings in debugging can be huge.

   After you think you no longer need these diagnostic print statements, you can put a comment symbol # in front of them. This will turn the print instruction into just a comment, so it won't be performed by the computer. If you ever discover that another bug has popped up in the script, you can restore the diagnostic output by removing the #s, rather than having to type in the entire print instruction again.

6. If you are making a while or for, does the repetition stop at the right time?

Here is a problem. Quick, what's the answer?

_A farmer is making a fence and needs to put in a fencepost hole every 5 feet. How many poles does she need to make a fence 50 feet long?_

If you thought the answer was "10", you made an off by one error. It is easy to make "fencepost problems", particularly when setting up a repetition to perform actions a number of times. Like other kinds of human error, you can't will yourself to never make such
Section 13.Z  Chapter summary

Roles for variables in *while* or *for* repetitions

<table>
<thead>
<tr>
<th>Role</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>Typically assigned a value before the repetitions start, and never changed inside the loop. <em>val</em> is an example of this in Example 11.2.1.1.</td>
</tr>
<tr>
<td>Stepper</td>
<td>Initialized to a value, and then changed each time through incrementation. Stepper situations include an integer variable initialized to 0 with 1 added onto it each repetition, or a variable initialized to 5.3 with .2 added each time, or a variable initialized to 10 with 2 subtracted each time. Typically the termination condition of the loop is based on the stepping variable reaching a final value. <em>for</em> loops automatically initialize a stepper variable and typically include the step value and the final value. The variable <em>i</em> in Example 11.2.1.1 is an example of a stepper variable.</td>
</tr>
<tr>
<td>Gatherer</td>
<td>A variable used to gather the final result. For example, if a loop is adding together values into a sum, the variable that contains the sum typically appears in a gatherer role in the <em>while</em> or <em>for</em> loop. The variable <em>s</em> in example 11.2.1.1 is a gatherer. Gatherers can appear in situations other than in summation. For example, <em>frame</em> is the example given given Section 11.2.2 is a table but is used to gather all the frames of a movie being computed one frame at a time in a loop.</td>
</tr>
<tr>
<td>Most recent value</td>
<td>A variable that changes with each repetition. Its value is typically a function or expression of other variables such as a stepper, or can be an expression or function of its previous value. <em>term</em> of Example 12.2.1.1 is an example of a variable with this role. They are usually the most challenging parts of the loop to design correctly.</td>
</tr>
<tr>
<td>Previous value</td>
<td>Some computations need not only the most recent value of a variable, but it's previous value as well, so an extra variable is used to keep that handy as well. We haven't seen any examples of this yet but will soon.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 13.3.1 Varieties of iteration of <em>for</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) <strong>for</strong> var from init to final do. repeated statements end do</td>
</tr>
</tbody>
</table>
including final.

#Table 13.3.1 (a)
#Stepper controlled
loop, i ranging
# from 1 to 5
for i from 1 to 5 do
    printf("sin (%d) = %f\n", i, evalf(sin (i)));
end do:

sin(1) = 0.841471
sin(2) = 0.909297
sin(3) = 0.141120
sin(4) = -0.756802
sin(5) = -0.958924

(b) for var from init to final by stepval do
    repeated statements
end do

Initializes var to init. Repeats with stepper values of init, init+stepval, init+2*stepval, etc. up to and including final.

#Table 13.3.1 (b)
#Stepper controlled
loop, i ranging
# from 1 to 2 in steps of .2
for i from 1 to 2 by .2 do
    printf("sin (%f) = %f\n", i, evalf(sin (i)));
end do:

sin(1.000000) = 0.841471
sin(1.200000) = 0.932039
sin(1.400000) = 0.985450
sin(1.600000) = 0.999574
sin(1.800000) = 0.973848
sin(2.000000) = 0.909297

(c) for var from init while condition do
    repeated statements
end do

Initializes var to init. If the condition is true then the statements
are performed and the value of var is incremented by one as with the original version of for. If the condition is still true, another repetition and another step is taken. This is done over and over until the condition is false.

<table>
<thead>
<tr>
<th>Table 13.3.1 (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stepper controlled loop, i ranging from 1 to 2 in steps of .2</td>
</tr>
<tr>
<td>s := 1;</td>
</tr>
<tr>
<td>tol := 10e-10;</td>
</tr>
<tr>
<td>x := .05;</td>
</tr>
<tr>
<td>term := 1;</td>
</tr>
<tr>
<td>for i from 1 while abs(term)&gt;=tol do</td>
</tr>
<tr>
<td>term := term*x/i;</td>
</tr>
<tr>
<td>s := s+term;</td>
</tr>
<tr>
<td>end do:</td>
</tr>
<tr>
<td>printf(&quot;s is:%10.8f, exp (%f) is:%10.8f&quot;,s,x,exp(x));</td>
</tr>
</tbody>
</table>

| 1 |
| 1.0 10⁻⁹ |
| 0.05 |
| 1 |
| s is:1.05127110, exp (0.050000) is:1.05127110 |

| for var from init to final by stepval while condition do |
| repeated statements |
| end do |

The repetition stops when either var reaches final or the while condition becomes false.
Table 13.3.1 (d)  
Stepper controlled loop, i ranging from 1.0 to 0 in steps of -.1  
# Stops when it reaches 0 or when ln(x) becomes less than -1.  
s := 1;  
x := 1.0;  
tol := 1.0;  
term := ln(x);  
for x from 1.0 to 0 by -.01 while abs(term)<tol do  
    term := ln(x);  
    printf("ln(%.2f) is: %.8f\n",x,term);  
end do:  

1  
1.0  
1.0  
0.  

ln(1.00) is: 0.00000000  
ln(0.99) is: -0.01005034  
ln(0.98) is: -0.02020271  
ln(0.97) is: -0.03045921  
ln(0.96) is: -0.04082199  
ln(0.95) is: -0.05129329  
ln(0.94) is: -0.06187540  
ln(0.93) is: -0.07257069  
ln(0.92) is: -0.08338161  
ln(0.91) is: -0.09431068  
ln(0.90) is: -0.10536052  
ln(0.89) is: -0.11653382  
ln(0.88) is: -0.12783337  
ln(0.87) is: -0.13926207  
ln(0.86) is: -0.15082289  
ln(0.85) is: -0.16251893  
ln(0.84) is: -0.17435339
\ln(0.83) \text{ is: } -0.18632958
\ln(0.82) \text{ is: } -0.19845094
\ln(0.81) \text{ is: } -0.21072103
\ln(0.80) \text{ is: } -0.22314355
\ln(0.79) \text{ is: } -0.23572233
\ln(0.78) \text{ is: } -0.24846136
\ln(0.77) \text{ is: } -0.26136476
\ln(0.76) \text{ is: } -0.27443685
\ln(0.75) \text{ is: } -0.28768207
\ln(0.74) \text{ is: } -0.30110509
\ln(0.73) \text{ is: } -0.31471074
\ln(0.72) \text{ is: } -0.32850407
\ln(0.71) \text{ is: } -0.34249031
\ln(0.70) \text{ is: } -0.35667494
\ln(0.69) \text{ is: } -0.37106368
\ln(0.68) \text{ is: } -0.38566248
\ln(0.67) \text{ is: } -0.40047757
\ln(0.66) \text{ is: } -0.41551544
\ln(0.65) \text{ is: } -0.43078292
\ln(0.64) \text{ is: } -0.44628710
\ln(0.63) \text{ is: } -0.46203546
\ln(0.62) \text{ is: } -0.47803580
\ln(0.61) \text{ is: } -0.49429632
\ln(0.60) \text{ is: } -0.51082562
\ln(0.59) \text{ is: } -0.52763274
\ln(0.58) \text{ is: } -0.54472718
\ln(0.57) \text{ is: } -0.56211892
\ln(0.56) \text{ is: } -0.57981850
\ln(0.55) \text{ is: } -0.59783700
\ln(0.54) \text{ is: } -0.61618614
\ln(0.53) \text{ is: } -0.63487827
\ln(0.52) \text{ is: } -0.65392647
\ln(0.51) \text{ is: } -0.67334455
\ln(0.50) \text{ is: } -0.69314718
\ln(0.49) \text{ is: } -0.71334989
\ln(0.48) \text{ is: } -0.73396918
\ln(0.47) \text{ is: } -0.75502258
\ln(0.46) \text{ is: } -0.77652879
\ln(0.45) \text{ is: } -0.79850770
\ln(0.44) \text{ is: } -0.82098055
\ln(0.43) \text{ is: } -0.84397007
\ln(0.42) \text{ is: } -0.86750057
| Develop incrementally. | \( \ln(0.41) \) is: -0.89159812  
|                      | \( \ln(0.40) \) is: -0.91629073  
|                      | \( \ln(0.39) \) is: -0.94160854  
|                      | \( \ln(0.38) \) is: -0.96758403  
|                      | \( \ln(0.37) \) is: -0.99425227  
|                      | \( \ln(0.36) \) is: -1.02165125  
|                      |  
| Make it easy to figure out the execution trace. |  
|          | Type in at most one or two lines before you try to get them to work.  
|          | If you develop your script through using several functions, test each one individually before you start invoking chains of them together.  
|          | If you anticipate needing an action that will result in a very long Maple statement, create a simple form of it first, get it to work, and then edit it into progressively longer and more complicated forms until you reach your goal. For example, if you are making a movie, first create just one frame and get that to work before you start entering the additional verbiage needed to make the movie.  
|          | If you are developing a \texttt{while} or \texttt{for} intentionally initialize it and set its stopping condition so that it will run only one or two times. Check that it works correctly. Then change the statement so that it runs as many times as you need to meet your goals.  
| Check for fencepost errors, also known as "off by one errors". | Everyone spend a lot of time figuring out what a program is up to. Make life easy on you. Put in extra \texttt{print} or \texttt{printf} instructions into the script so that when you are debugging you will see what is going on more easily. You can remove them later on by putting a \# symbol in front of them, which will turn them into a program comment.  
| | You can also suppress portions of the execution trace by changing semi-colons into colons. Changing the semi-colon after an \texttt{end do} into a colon will suppress all of the execution trace from the repeated instructions inside.  
| | It's a common mistake to make when setting up \texttt{while} or \texttt{for} loops to have the repetition proceed one too many or one too few times. The best you can do is to make checking for them habitual, just as you check for balanced parentheses or missing commas.  

Table 13.Z.2  More advice about script construction