19 Calculations that result in true or false

19.1 Chapter Overview

true and false are the two Boolean (sometimes known as "truth") values. Variables can be assigned Boolean values just as they can be assigned numeric values, formulae with symbols in them, equations, ranges, lists, etc. Boolean expressions are any expression which can take on a true or false values. Boolean expressions are commonly found in if or while statement as the conditions controlling execution of the statement. Functions can be defined in Maple that return a Boolean value as their result. Such functions are known as Boolean functions. Boolean functions are often useful in larger programming projects to give a easy way to invoke code where complicated decision-making occurs.

19.2 Boolean values and Boolean expressions

In contrast to numeric values (integer, fractions, floating point), or algebraic ( $x^2 + 1$ ), we have the truth values, true and false. These are sometimes referred to as Boolean values, in recognition of the pioneering work into mathematical logic by the British mathematician George Boole (1815-1864) (ref: "BOOLE, GEORGE." Encyclopedia of Computer Science. Hoboken: Wiley, 2003. Credo Reference. 23 Oct. 2006. Web. 11 May 2010. <http://www.credoreference.com.ez-proxy2.library.drexel.edu/encycics/boole_george>.) Like other types of values, they can be assigned as the value of a Maple variable through :=.

In chapter 12 (Brian, provide hyperlink to textual entry of relational operators table on p. 167), we introduced the relational operators $<$, $>$, $<=$, etc. that could be used to create an equality that could be used as the condition in an if statement or while loop. Such conditions are a type of Boolean expression -- an expression whose value is true or false. One can create more complicated Boolean expressions by the use of the Boolean operators and, or, and not. Using such operators, one can combine Boolean expressions created using inequalities.

Boolean expressions and assignment of Boolean values to variables are often used in while statements, to calculate the next value of the controlling condition, in the midst of the loop. They can also be used to make the controlling logic of if statements more easily readable.

Operations that combine Boolean expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$a$ and $b$ will be true if both $a$ and $b$ are true conditions.</td>
</tr>
<tr>
<td>or</td>
<td>$a$ or $b$ will be true if either $a$ or $b$ (or both) are true conditions.</td>
</tr>
<tr>
<td>not</td>
<td>not(a ) will be true if the condition $a$ is false, and vice versa.</td>
</tr>
</tbody>
</table>

A simple example using Boolean values and Boolean expressions

<table>
<thead>
<tr>
<th>Boolean Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$isRaining := true$</td>
<td>true</td>
</tr>
<tr>
<td>$isPouring := false$</td>
<td>false</td>
</tr>
<tr>
<td>if $isRaining$ and $isPouring$ then $oldManIsSnoring := true$; else $oldManIsSnoring := false$; end if.</td>
<td>We can imagine writing a script doing some data analysis where at some point we determine that it's raining. A little later we decide that the data also say that it isn't pouring.</td>
</tr>
<tr>
<td>This is a way to write code to set up a variable whose Boolean value is determined by the English nursery rhyme (<a href="http://en.wikipedia.org/wiki/Its%27s_Raining__Its%27s_Pouring">http://en.wikipedia.org/wiki/Its%27s_Raining__Its%27s_Pouring</a>). While the rhyme doesn't say what happens if it isn't raining and pouring,</td>
<td></td>
</tr>
</tbody>
</table>

299
false

we decide that we will assign the variable to have the opposite
true value of what it has when it is raining and pouring.

The condition isRaining and isPouring is false because isRaining
and isPouring are not both true. Thus, the variable oldManIsSnor-
ing is assigned false.
\[false\]

oldManIsSnoring
\[false\]

We confirm that this is what the if statement did.

An example using boolean values and expressions

\[\text{Velocities} := [95, 47, 86, 35, 16, 98, 87, 87, 90]\]
\[\text{Colors} := ["red", "green", "blue", "black", "red", "green", "white", "red"]\]

\[\text{Licenses} := ["PA", "ON", "PA", "NY", "HI", "BC", "NJ", "IL", "WY"]\]

\[\text{Provinces} := ["PEI", "NB", "QU", "NFL", "ON", "MAN", "SA", "AB", "YU", "BC"]\]

\[\text{member}("ON", \text{Provinces})\]
\[true\]

\[\text{member}("MA", \text{Provinces})\]
\[false\]

\[\text{count} := 0;\]
\[0\]

\[\text{for } i \text{ from 1 to nps(Velocities) do}\]
\[\text{condition1} := \text{Velocities}[i] > 65 \text{ and not(Colors[i]} = "red") ;\]
\[\text{condition2} := \text{Velocities}[i] < 50 \text{ and member(Licenses[i], Provinces)};\]
\[\text{if condition1 or condition2 then count} := \text{count} + 1;\]
\[\text{end if}\]
\[\text{end do}\]

\[\text{printf}("\text{Number of licenses meeting criterion: } %d\text{m}, \text{count});\]
19.3 Boolean functions

We have seen how to design a function that returns a numerical result, and a plot structure result. Occasionally there is need to design a function that returns a Boolean result (either true or false). Such functions are used in larger programming projects where the decision-making is sufficiently complicated that readability may be enhanced by breaking off the code that does the decision-making and giving it a name. This allows it to be conveniently accessed and referred to, as well as reused if it is needed in several places inside the project.

The member function is an example of a built-in function that returns a Boolean result.

While it is not a requirement, Boolean functions are often given a name that starts with the prefix "is", e.g. isPrime, isOdd, isDivisibleBy, isTouching, etc. This naming convention makes it easier to tell at a glance that the function is going to return a Boolean result.

The built-in function is takes a relationship such as an inequality and decides whether it is true or false. This can be helpful in the construction of functions that want to return a Boolean result without having an if statement around.

Example of a user-defined Boolean function

<table>
<thead>
<tr>
<th>Example of a user-defined Boolean function</th>
</tr>
</thead>
</table>
| ```
isWithin := proc(a,b, x0,y0, distance)
   is(evalf(sqrt((a-x0)^2 + (b-y0)^2)<distance));
end proc;

proc(a, b, x0, y0, distance)
   is(evalf(sqrt((a-x0)^2 + (b-y0)^2) < distance))
end proc
```
| We have taken a observation of a moving object in a two-dimensional coordinate system. We want to find those that are within 5 meters of the (3,5). We write a boolean function that does something more general -- it returns true when the point (a,b) is within distance of the point (x0,y0). |
| Obs := [ [1,1,2.1], [1,3,2.4], [1,5,2.7], [1,9,2.9], [2,1,3.0], [2,7,2.5], [2,8,2.4] ]
| [ [1,1,2.1], [1,3,2.4], [1,5,2.7], [1,9,2.9], [2,1,3.0], [2,7,2.5], [2,8,2.4] ] 19.12 |
| for i from 1 to nops(Obs) do
  a := Obs[i][1];
  b := Obs[i][2];
  if isWithin(a, b, 3, 5, 2.5) then print(Obs[i], "is within", 2.5, "meters of", 3, 5);
  end if;
end do. |
| 1.1 |
| 2.1 |
| 1.3 |
| 2.4 |
| 1.5 |
2.7
1.9
2.9
[1.9, 2.9], "is within", 2.5, "meters of", 3, 5
2.1
3.0
[2.1, 3.0], "is within", 2.5, "meters of", 3, 5
2.0
2.7
[2.7, 2.5], "is within", 2.5, "meters of", 3, 5
2.8
2.4
(19.13)

isWithin2 := proc(LO, x0, y0, distance)
    is(evalf(sqrt((LO[1]-x0)^2 + (LO[2]-y0)^2))<distance))
end proc;

proc(LO, x0, y0, distance)
    is(evalf(sqrt((LO[1]-x0)^2 + (LO[2]-y0)^2))<distance))
end proc

for i from 1 to nops(Obs) do
    if isWithin2(Obs[i], 2.7, 4.3, 2.6)
        then print(Obs[i], "is within", 2.6, "meters of", 2.7, 4.3);
    end if;
end do;

[1.3, 2.4], "is within", 2.6, "meters of", 2.7, 4.3
[1.5, 2.7], "is within", 2.6, "meters of", 2.7, 4.3
[1.9, 2.9], "is within", 2.6, "meters of", 2.7, 4.3
[2.1, 3.0], "is within", 2.6, "meters of", 2.7, 4.3
[2.8, 2.4], "is within", 2.6, "meters of", 2.7, 4.3
(19.14)

A procedure that uses a Boolean function

<table>
<thead>
<tr>
<th>A procedure that uses a Boolean function</th>
</tr>
</thead>
<tbody>
<tr>
<td>findWithin := proc(L, pt, distance)</td>
</tr>
<tr>
<td>local ptTab, i;</td>
</tr>
<tr>
<td>end proc</td>
</tr>
<tr>
<td>We define a function procedure that returns a list of all the points in L that are within distance of the specified point pt. This allows us to</td>
</tr>
</tbody>
</table>
for i from 1 to nops(L) do
    if isWithin2(L[i], p[1], p[2], distance) then ptTab[i] := L[i];
end if;
end do;
return convert(ptTab, list);
end proc;

proc(L, pt, distance)
    local ptTab, i;
    for i to nops(L) do
        if isWithin2(L[i], pt[1], pt[2], distance) then ptTab[i] := L[i] end if
    end do;
    return convert(ptTab, list)
end proc

distance1 := 2.5:
pt1 := [3, 5]:
printf("%a are within %.2f meters of %a
", findWithin(Obs, pt1, distance1), distance1, pt1);
distance2 := 2.6:
pt2 := [2.7, 4.3]:
printf("%a are within %.2f meters of %a
", findWithin(Obs, pt2, distance2), distance2, pt2);
distance3 := 2.1:
pt3 := [2.7, 4.0]:
printf("%a are within %.2f meters of %a
", findWithin(Obs, pt3, distance3), distance3, pt3);

[[1.9, 2.9], [2.1, 3.0], [2, 7, 2.5]] are within 2.50 meters of [3, 5]
[[1.3, 2.4], [1.5, 2.7], [1.9, 2.9], [2.1, 3.0], [2.8, 2.4]] are within 2.60 meters of [2.7, 4.3]
[[1.5, 2.7], [1.9, 2.9], [2.1, 3.0], [2.8, 2.4]] are within 2.10 meters of [2.7, 4.0]

19.4 Troubleshooting Boolean calculations

The most common mistake in doing a Boolean calculation is to use a result which Maple cannot decide is true or false. This is typically due to forgetting to assign a variable an expected numerical value, and then using a Boolean expression involving an inequality with the variable.

### Troubleshooting Boolean calculations

| for i from 1 to nops(Velocities) do
| if Velocities[i] < minUS and not(member(Licenses[i], Provinces))
| then minUS := Velocities[i];
| end if;
| end do;
| printf("Minimum US speed observed is: %a\n", minUS);
| Error, cannot determine if this expression is true or false: 95 < minUS |
| Using the velocity/color/license plate information of the previous examples, we attempt to find the slowest speed by a non-Canadian car by writing a loop that updates the variable minUS whenever it finds a smaller value. However, there's an error because we forgot to initialize minUS, so when i=1, the truth or falsity of the expression Velocities[i]<minUS can't be determined. The computer is trying to decide whether 95<minUS is true or false, but since minUS has no assigned value, this can't be determined. |
Minimum US speed observed is: 16

The printf statement also gives an error because it tries to print out the integer value of minUS, which has not assigned any numerical value by the time the printf statement is executed.

We fix this by initialize minUS to a value that we know will be larger than the eventual outcome.

isCanadianMin := proc(velocity, license, minCAN, provList)
    return velocity<minCan and member(license, provList);
end proc;

minCAN := 200;

for i to nops(Velocities) do
    if isCanadianMin(Velocities[i], Licenses[i],
        minCAN, Provinces)
        then minCAN := Velocities[i];
    end if;
end do;
printf("Minimum Canadian speed observed is: %d\n", minCAN);

We create a Boolean function which, when given appropriate parameters, is supposed to decide whether the velocity parameter is less than the current minimum, and if it belongs to a Canadian car, returns true.

We try to use the function in a loop similar to that of the previous example. However, we get an error message.

proc(velocity, license, minCAN, provList)
    return velocity < minCan and member(license, provList)
end proc

200

Error, cannot determine if this expression is true or false: 47 < minCan

Minimum Canadian speed observed is: 200

We note that the cause is the same as the previous example: minCan doesn't have a value so the computer can't decide whether 47<minCan has no value. We fix the function to use the name of the parameter, minCAN, and things now work properly.

isCanadianMin := proc(velocity, license, minCAN, provList)
    return velocity<minCAN and member(license, provList);
end proc;

minCAN := 200;

for i to nops(Velocities) do
    if isCanadianMin(Velocities[i], Li-
censes[i],
    minCAN, Provinces)
    then minCAN := Velocities[i];
end if;
end do;
printf("Minimum Canadian speed observed is:
%d\n", minCAN);

proc(velocity, license, minCAN, provList)
    return velocity < minCAN and member(license,
        provList)
end proc

Minimum Canadian speed observed is: 47

19.5 Chapter summary

<table>
<thead>
<tr>
<th>Boolean term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>There are only two Boolean values in Maple: true and false. Maple variables can be assigned Boolean values.</td>
</tr>
<tr>
<td>expression</td>
<td>An expression whose value is either true or false. Maple variables can be assigned the result of evaluating Boolean expressions.</td>
</tr>
<tr>
<td>function</td>
<td>A function whose return value is either true or false</td>
</tr>
</tbody>
</table>

```maple
#Returns a result that is true or false depending on
#whether the point (a,b) is within the specified
distance from the point (x0, y0)
isWithin := proc(a,b, x0, y0, distance)
is(evalf(sqrt((a-x0)^2 + (b-y0)^2)<distance));
end proc;
```

<table>
<thead>
<tr>
<th>Boolean operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>( a ) and ( b ) will be true if both ( a ) and ( b ) are true conditions.</td>
</tr>
<tr>
<td>or</td>
<td>( a ) or ( b ) will be true if either ( a ) or ( b ) (or both) are true conditions.</td>
</tr>
<tr>
<td>not</td>
<td>( \text{not}(a) ) will be true if the condition ( a ) is false, and vice versa.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Built-in Boolean functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>member(y, [x, y, z])</code></td>
<td>The function <code>member</code> tests whether something is an element of a set, and returns true or false accordingly.</td>
</tr>
<tr>
<td><code>is(15 &lt; 10)</code></td>
<td>The function <code>is</code> inspects its argument and decides whether the supplied expression is true or false.</td>
</tr>
</tbody>
</table>
Calculations that result in true or false
20 The end, for now

20.1 Chapter Overview

We summarize the fundamental message of this course: that you can expect certain functionality can be available for your use without having to make heroic efforts to get it. The conceptual foundation has been laid for you to continue growing your proficiency in technical computing through your own practice regime, to handle more sophisticated mathematics and mathematical models, using the syntax of whatever computer language you find the most expedient to use.

20.2 What you have done in the course: technical features

Over 12 labs and 12 homework/quiz sessions we have covered the following technical features:

a) Interactive calculation -- arithmetic, arithmetic with multitudinous math functions (trig, log, exp, roots), arithmetic with varying precision, exact arithmetic.

b) Intermixing of documentation and calculation so that results and their explanations can performed in the same document where they are presented.

c) "Clickable" menus, and palettes to make entry of mathematics and operations require less typing and be more accurate.

d) Higher-level operations: solve, and plot. Eventually we found problems that motivated use of more of the large repertoire technical systems have: least squares and spline data fitting, or minimize and maximize, for example.

e) Scripts -- creating sequences of calculations to tackle longer or multi-part problems, parameterized to facilitate reuse.

f) Structures for holding collections of data: lists, sets, plots, and tables. Part of the work in understanding this is how to initialize the structure, how to insert data into the structure, and how to extract it. Maple has many more data structures -- vectors, matrices, and arrays for example, that more advanced applications motivate the use of.

g) One-line functions using ->.

h) Plotting and animation -- showing results to facilitates quick comprehension. In addition to function plotting, we have plotted data, done parameterized plotting, done multiple multi-color plots, and created animations of mathematically-described phenomena.

i) Programming -- looping and conditional execution. Encapsulating scripts into procedures (functions). Learning how to specify scripts through textual means, which meant getting used to a particular style of notation with balancing parentheses, a particular order of information, exactly correct spelling.

j) An expanded repertoire of mathematical representations. In addition to numbers, we have learned how to compute with formulae, such as calculus operations for differentiation and integration. Piecewise expressions can make it easier to represent information found in "real world" situations such as motivated by physics word problems. Allowing results to contain arbitrary symbols allows some symbols in certain formulae to represent units of measurement such as meters/hour rather than numbers.

k) On-line help for everything.

l) "Widgets" for allowing simple ways of running and rerunning scripts with different values of parameters. The paradigm is to get a "stock widget" and connect it to a computational procedure you have through small (one line) amounts of code. If the procedure produces the right kind of result, it can be displayed in another widget.

We have learned how to do these things using Maple, which supports all of this. But another key effect of our work is the ability to discuss and plan the work by speaking in human terms: "now plot the function" rather than a discussion of what exactly to type. The important idea was the decision to do plotting, and the function to plot. The secondary action was to figure out how to say it in Maple (or the language of whatever system you happen to be using). If you
do not practice regularly, the particular details of how to do function plotting in Maple may fade from your memory. The important thing to remember when you return to the computer in the future after this course is over is that it is reasonable to expect to do these sorts of things without heroic effort, and that there are usually many examples lying around that you can use as a basis for (re)-learning and imitation. The upside of "so simple that even a freshman can do it" is that it, if you've done your job as a student well, it is routine to come back and learn how to do it again if you've forgotten, or have to do it in another language. This is hardly unique to programming, it's true for all the mathematics and science that you learn now as a freshman and will have need to use again in later years.

20.3 What you have done in this course: how to work with computation

Besides our tour of "features you should expect to have, and how to get them", we have toured a variety of small technical problems and the things that can be done to understand them with the help of computation. A key skill in all of this is to be able to discover and state mathematical relationships that help to describe real situations -- the often painfully acquired "skill with word problems". These relationships have been referred to as a mathematical model. Sometimes finding the model is a matter of reading about it in an article created by an expert. The skill then is being able to translate the relationship into the syntax of whatever computer system you are using to do your work with. Sometimes you have to determine it yourself.

Once you gotten the description of the model into the computer, you have to figure out which operations (e.g. "curve fit", "find right hand side of", "differentiate", "solve", "plot", "minimize", etc.) you want to perform with the mathematical model in order to find out the desired information. Often times the desired information is a prediction, which happens through a simulation. Extensive understanding of a phenomenon that can be modeled mathematically often involves many calculations. Often tables, plots, or animations are produced to present the computed information in a form that more easily explains the results.

Having a good troubleshooting and test regimen is an important part of any effort to develop code that's more than a line or two long. An important thing to remember about your work in the labs is that the development process was punctuated by frequent testing along the way, so that every additional part that was built was confirmed to work before anything else was added on. Testing a long code only after you have built and entered it is a good way to write a program that you will never get to troubleshoot successfully.

20.4 Continuing an education in computing

There are several different additional ideas that a full education in computational science and engineering should include:

1. More sophisticated models of physical situations often requires the use of linear algebra, and differential equations. Any of the technical systems (e.g. Maple, Matlab, Mathematica) can handle these calculations, either symbolically/exactly or with floating point calculation. It only awaits your becoming familiar enough with the mathematical ideas and taking the time to learn how the computational machinery you have already seen -- scripts, procedures, visualization, etc. -- is used with them.

2. More sophisticated calculations often require ideas from "advanced programming": recursion, object-oriented programming, tree and network-graph data structures. The history of computational science and engineering indicates that while for many decades mainstream applications avoided such features because of the additional effort needed by programmers to understand them, eventually the need for handling more sophisticated or complex situations has led to their extensive adaptation.

3. Computational efficiency becomes an issue as you write more code and rely less on built-in functions to handle major portions of your computation. The study of algorithms -- the development of a step-by-step description for how to solve a problem which can be straightforwardly turned into code --gives you access to the accumulated wisdom and cleverness about how to solve problems faster or using less memory. For large problems, the difference between efficient and inefficient methods can be striking -- seconds instead of hours or even years.

4. With calculations involving floating point numbers, numerical accuracy can sometimes be an issue. Relying upon built-in techniques such as fsolve or Optimization/Minimize/ gives you access to the techniques that are usually accurate, but sometimes built-in code is insufficient to solve your problem, and you must write your own. Poor approximation
techniques can mean that your computed results are so far from the exact answer that they are useless for design or prediction purposes. The study of numerical analysis will give you access to what is known about getting good approximations and avoiding bad approximations with floating point numbers.

5. Several different types of computing have arisen that we haven't covered but are of growing value in technical fields: artificial intelligence -- the ability to get programs to tackle problems that most people regard as requiring thinking to solve, and data mining -- extracting conclusions from monumentally large amounts of information.

6. Most languages also support some form of parallel computing -- running parts of a program on several computer processors at once in hopes to getting results faster. For example, Maple can use all the processors of a "multicore" computer such as is typical on many personal computers nowadays through a parallel version of map.

If you want to teach yourself about these things, how do you go about doing it? We have seen that most computer systems have ample amounts of documentation explaining their features. There are many books and web pages devoted to explaining these kinds of ideas. It takes a familiarity with fundamental concepts (which this course has exposed you to), time, and motivation. The latter is particularly important. We recommend having a good excuse for learning the new material -- a new problem that you want to solve, for example. Then it's a matter of reading, experimentation, and talking to knowledgeable and responsive consultants when you get really stuck.

Proficiency in programming, like almost every other skill needs practice and periodic contemplation of your progress. Making it a point to solve a stream of small problems will allow you to learn how to do all the things necessary to handle larger projects. If you wait until some enormous project comes along to revisit programming, then you will find that an enormous amount of time will be needed to get up to speed. Incremental acquisition of knowledge is a way of living, not a course to take and be done with. It's the only way to become proficient with modern technical computing systems, given the large amount of functionality built into them and the large and diverse nature of problems they are used to help solve.

As you get more experienced and ambitious, you should also become familiar with more of the principles of software engineering. As might be expected, once a programming project encounters success, it attracts many users who want to reuse the software. They put pressure on the developers to fix and extend the programming to do more. The techniques used to design and maintain a large system -- say one that has $10^6 - 10^7$ lines of programming, are far different from the informal and intuitive style we have used in this course. A program built for personal use -- taking a few hours to build -- is not at all like a large software systems that takes dozens of programmers a few years to get working. Assuming that the "personal project" approach to software construction works on a large system makes about as much sense as assuming that the experience of building a backyard garden shed will sufficient to let you build a 40 story skyscraper. Without organizational principles, a regimen of programming style, and software tools to do extensive testing and recordkeeping, it becomes infeasible to make even small changes to a large program without great expense. We've shown you a few software engineering ideas in this course -- frequent testing and incremental development, and the use of variable names that are words rather than single mathematical symbols. The field of software engineering can provide you with the best practices and formal analysis taken from the construction of large computer systems over the past half century.