Scripting and Programming for Modeling, Simulation, and Control

Copyright © Bruce Char
2008-10
# Contents

1 Introduction -- Technical computing at the turn of the century .............................................................. xi

1.1 Chapter synopsis ................................................................................................................................... 1

1.2 What is technical computing? What kinds of technical computing are there? ..................................... 1

1.3 What are the advantages of doing technical calculation with a computer? ......................................... 1

1.4 The spectrum of devices for technical computing .............................................................................. 2

1.4.1 High performance computers, also known as "supercomputers" .................................................. 3

1.4.2 Hand held or mobile devices ........................................................................................................ 3

1.4.3 Dedicated controllers ................................................................................................................ 4

1.5 Maple, a system for technical computing ............................................................................................ 5

1.6 What about Systems X, Y, Z, ...? ....................................................................................................... 5

1.7 Why pick Maple as the first system? .................................................................................................... 6

1.8 Using more than one system ............................................................................................................. 6

2 Getting started with Maple's Document Mode: doing technical work with a clickable interface .......... 9

2.1 Chapter synopsis ................................................................................................................................ 9

2.2 Starting up Maple, getting a fresh start ............................................................................................. 11

2.3 Evaluating an expression involving exact arithmetic .......................................................................... 14

2.3.1 Grade school arithmetic ........................................................................................................... 11

2.3.2 Making typographical mistakes .............................................................................................. 14

2.3.3 Correcting typographical mistakes ........................................................................................ 16

2.3.4 Exponentiation (powers). Numbers with lots of digits ................................................................ 17

2.3.5 Detecting and fixing vocabulary and "logic" mistakes ............................................................... 18

2.4 Saving and retrieving your work ....................................................................................................... 19

2.5 Retrieving backups ............................................................................................................................ 20

2.6 Algebra, plotting and mouse-clickable operations ............................................................................ 21

2.6.1 Algebraic expressions and equations. Solving equations. Working with pieces of expressions .... 21

2.6.2 For those with previous experience on other systems: some things are different, for a reason .... 21

2.6.3 Plotting and approximate numerical solutions ....................................................................... 22

2.6.4 The Expressions Palette and the Common Symbols Palette: entering Trigs, logs, roots, exponentials 24

2.6.5 Approximate numerical (calculator - type) arithmetic in Maple ............................................ 26

2.6.6 Evaluation, and selection of pieces .......................................................................................... 27

2.7 A quick-reference summary to this chapter ..................................................................................... 27

3 Chapter 3 Technical word processing ................................................................................................. 33

3.1 Chapter Overview ............................................................................................................................. 33

3.2 Maple as a word processor .............................................................................................................. 33

3.3 Shortcuts to entering math symbols ............................................................................................... 36

3.4 Other word processor features ...................................................................................................... 36

3.5 Troubleshooting word processing .................................................................................................. 37

3.6 Summary of Chapter 3 material ...................................................................................................... 37

4 Chapter 4 Assignment ........................................................................................................................ 39

4.1 Chapter Overview ............................................................................................................................ 39

4.2 Assignment: remembering results for future use ............................................................................ 39

4.3 Assignment keyboard shortcut := .................................................................................................. 40

4.4 How to think about assignment: a mental model ............................................................................ 41

4.5 The state of the Maple session and the look of the worksheet ....................................................... 43

4.6 restart causes all assignments to be forgotten ............................................................................. 44

4.7 Evaluation and assignment ............................................................................................................ 44

4.8 Troubleshooting assignments .......................................................................................................... 46

4.9 Summary of Chapter 4 material ...................................................................................................... 48

5 Chapter 5 Building scripts ................................................................................................................ 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.6 Chapter summary for Chapter 15</td>
<td>238</td>
</tr>
<tr>
<td>16 Thinking about iteration</td>
<td>241</td>
</tr>
<tr>
<td>16.1 Chapter Overview</td>
<td>241</td>
</tr>
<tr>
<td>16.2 Constructing iterations and the roles of variables</td>
<td>241</td>
</tr>
<tr>
<td>16.3 Reminder: you can skip writing some looping code through use of built-in operations</td>
<td>244</td>
</tr>
<tr>
<td>16.4 map and the construction of new structures</td>
<td>245</td>
</tr>
<tr>
<td>16.5 The advantages and limitations of shortcuts</td>
<td>247</td>
</tr>
<tr>
<td>16.6 Summary of chapter</td>
<td>248</td>
</tr>
<tr>
<td>Index</td>
<td>251</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An early computer</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A high-end calculator in 2009</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Maple started up with new document in Windows XP</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Maple document with first entry area</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Maple input using 2d math</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Maple input with labeled result</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Arithmetic Operations in Maple</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Examples of Maple error messages</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Create Document Block to force a Math input area wherever the cursor is placed</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of a vocabulary mistake</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>Maple save menu operation</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>Maple save dialog box</td>
<td>20</td>
</tr>
<tr>
<td>2.11</td>
<td>The Maple state display</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>Example of Plotting</td>
<td>23</td>
</tr>
<tr>
<td>2.13</td>
<td>Plot created by right-click -&gt; Plot -&gt; 2DPlot</td>
<td>23</td>
</tr>
<tr>
<td>2.14</td>
<td>User-configured plot using PlotBuilder instead of 2DPlot</td>
<td>24</td>
</tr>
<tr>
<td>2.15</td>
<td>The Expression palette</td>
<td>24</td>
</tr>
<tr>
<td>2.16</td>
<td>Examples of palette-driven computation</td>
<td>26</td>
</tr>
<tr>
<td>2.17</td>
<td>Examples of computing with approximate solving</td>
<td>26</td>
</tr>
<tr>
<td>2.18</td>
<td>Evaluate at a point</td>
<td>27</td>
</tr>
<tr>
<td>2.19</td>
<td>Operations on equations, multi-part expressions</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Maple in math entry mode</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Document after control-T (or Insert-&gt;Text)</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Document with a mixture of text and math</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>control= puts the results of a calculation in the midst of text</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Keyboard shortcuts in math mode through the escape key</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Assignment</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Assignment</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Mental model of assignment</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>Example: The state of a Maple Session</td>
<td>43</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation of expressions involving assigned variables</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation of expressions involving assigned variables</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Reassignment undoes the effect of previous assignment</td>
<td>47</td>
</tr>
<tr>
<td>4.8</td>
<td>Maple won't let you use some names it is already using</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic data structures in Maple and operations to extract parts of them</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Solving simultaneous equations</td>
<td>52</td>
</tr>
<tr>
<td>6.1</td>
<td>Plotting of multiple expressions</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>Plotting points with lists of numbers</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Plot options and labels</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Plot command help</td>
<td>74</td>
</tr>
<tr>
<td>6.5</td>
<td>Examples of plot</td>
<td>75</td>
</tr>
<tr>
<td>6.6</td>
<td>Copying the example</td>
<td>76</td>
</tr>
<tr>
<td>6.7</td>
<td>Modifying the example</td>
<td>77</td>
</tr>
<tr>
<td>6.8</td>
<td>Plot option help page</td>
<td>78</td>
</tr>
<tr>
<td>6.9</td>
<td>Operations on lists</td>
<td>78</td>
</tr>
<tr>
<td>6.10</td>
<td>Solving systems of equations with solve</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>Function results in Maple</td>
<td>89</td>
</tr>
<tr>
<td>7.2</td>
<td>Command Completion</td>
<td>92</td>
</tr>
<tr>
<td>7.3</td>
<td>A function definition in a math textbook</td>
<td>94</td>
</tr>
<tr>
<td>7.4</td>
<td>User-defined functions through the Expression Palette</td>
<td>95</td>
</tr>
<tr>
<td>7.5</td>
<td>Comparing evaluation using eval and functional notation</td>
<td>96</td>
</tr>
</tbody>
</table>
Dedication

To our students, who learn how to work with the new and different.

To my family, whose support is unwavering.

Acknowledgements

Jeremy Johnson started the Computation Lab course. He chose the initial direction and approach of the course. Frederick W. Chapman worked with us in 2006-2008 in the development of the course.

We are grateful to our colleagues David Augenblick, Jeremy Johnson, and Brian Dolhansky for their help in the development of this text. David was a careful proofreader and useful sounding board. Jeremy's shrewd counsel and feedback often led to improvements. Brian's expert and speedy work in tagging and repairing the manuscript kept us on schedule. We also appreciate the assistance of Maplesoft for technical advice and support, especially that of Christina Spirou.
1 Introduction -- Technical computing at the turn of the century

1.1 Chapter synopsis

1. We explain what technical computing is about, why it's done with a machine and what kinds of devices are used to do it.
2. The advantages doing calculation with a computer are surveyed.
3. Maple, a system for technical computing is introduced and contrasted with some of the other choices available today.

1.2 What is technical computing? What kinds of technical computing are there?

Well, obviously, it's "computing done for technical work" -- primarily science and engineering, but any other field where mathematical and scientific reasoning is used, which nowadays can include other fields such as finance and business, medicine, or digital media. The success of such reasoning in these fields is well-known. Students of those fields need to become experienced and develop proficiency in how it's done to be able to obtain success in the same way.

You are probably familiar with numerical computing since it's done even without computers, with paper and pencil or with calculators in high school or even earlier. You probably have also seen graphing calculators do simple kinds of graphical computing, also sometimes known as mathematical visualization. In addition to this, computers can and are used to do symbolic computing -- computing where the answer is a formula rather than a number, as well as deductive or logical computing -- using logic or deduction to find answers.

In this course, we will concentrate on numerical, symbolic and graphical computing, although we will see some instances of deductive computing as well.

1.3 What are the advantages of doing technical calculation with a computer?

One advantage is greater quantity of calculation -- computers can do calculations billions of times faster than humans and thousands or millions of times faster than calculator-driven computation. Using them makes some things feasible that are not possible any other way:

As you've seen with your high school physics, chemistry, and math courses, scientists and engineers often work with mathematical models -- systems of equations, symbols, and mathematical relations that try to describe key aspects of a situation. Calculation with models helps answer questions about the situation being modeled: How much fuel will be used in this operating scenario? How long will it take to heat up the furnace to 265 degrees? Computer simulation loads a model with some initial conditions, and then through calculation produces predictive results. Simulation can be used to make forecasts such as: how fast will the vehicle be moving after 5 seconds? How far will the pollutants move underground after being buried five years? Computer simulations can sometimes generate predictions even when standard techniques of "mathematical solution" are not adequate to find an answer.

Computers make it possible to use models that require much more extensive calculations to reach conclusions. The payoff may be better predictions. Extensive calculation also makes it possible to generate and present more information. Computer visualization goes beyond that to highly detailed pictures or animations of a situation described through a mathematical model. This can lead to better insight and understanding.

Another advantage is that it is easier to make a lasting written record of the work. One of the big differences between professional technical work and homework is that the modeling and calculation has lasting value: it matters to more than one or two people, and it is being used in an on-going project. In a professional situation, a worker may need to go back and review the work a year later, long after the details are faded from easy recall. Other people may want to reuse the programming and will need to be given an explanation about how to use it. Since they have long-term value, it is beneficial to put them into a form suitable for easy future reference and reuse. Thus the work typically includes both programming and documentary explanation.
1.4 The spectrum of devices for technical computing

Back in the days of the original electronic computers developed during and immediately after World War II, the only kind of device you could use for technical computing (which was the only computing that could be justified in those days) filled up a whole room and needed an extensive staff to support.

Table 1.1: An early computer

ENIAC, one of the first electronic computers, being programmed by Herman Goldstine at the University of Pennsylvania circa 1946. (US Army photo) Originally intended to do artillery calculations, it was used for the design and development of the first thermonuclear weapons.

See http://www.seas.upenn.edu/~museum/.

Today's users have a choice of a wide variety of devices:

**Personal computers**

Typically a computer for individual use can be expected to have the following features:

1. Processing capability of a billion or more arithmetic or memory operations per second. Significant amounts of calculation are used in doing the graphics involved in processing digital media or supporting the artificial worlds presented in computer games. This same power can be used to do the simulation and exploration in many common kinds of scientific and engineering situations. This course is about using that calculation power.

2. Standard selection and pointing devices, keyboard and mice. Some computers support "tablet" operation which allow use of a pen writing on the display. The display area and input devices make it easy to enter information via text or by pointing/selection. While the World Wide Web has made much processing possible with just "pointing and clicking", the complexity and mathematical sophistication of the models used in current technical computing outstrip the capabilities of current "point and click" technology. Some (computer programming) language-based interaction is necessary to handle things deftly. This requires a more knowledgeable user, but many people find that the extra power is worth the learning effort.

3. A screen capable of displaying information equivalent to one or more 8 1/2 x 11 inch pieces of paper. This helps support the development of documentation, or of more complex visualizations.

4. Local storage capable of storing a significant fraction of the text of the books in the Library of Congress. Of course, it's easy for a personal media collection to fill up this kind of storage, but it should not be forgotten that the billions of characters that this storage represents can store years or decades of efforts of computer programmers.

5. Connection to the internet, which makes it easier to communicate and share work with others, and to download new programming. From the viewpoint of technical computing, that it's easy to import and run substantial amounts of programming.
High performance computers, also known as "supercomputers"

While hand held devices and personal computers are used widely by the general public for non-technical purposes, there are a class of computers that are used primarily for mathematically-based computing. Typically instead of billions of numerical operations per second ("gigaflops" — billions of floating point operations), they employ multiple processors in parallel to do trillions of operations ("teraflops") or quadrillions ("petaflops"). The hardware, personnel and energy costs of such computers are significant, so typically they are available through centers used through the Internet from many locations. The kinds of technical problems such devices are used for would include: simulation of complicated physical situations, such as for climate prediction, astrophysics, or engineering design (aircraft, groundwater, large building, automobile).

Using such large devices incur significant expense, from hardware, software and support staff costs, as well as electrical power consumed. For example, the U.S. Department of Energy has an IBM computer at its Los Alamos National Laboratory in New Mexico, USA capable of 1 quadrillion ($10^{15}$) numerical operations per second. It consumes 2345.50 kilowatts when it is running. (reference: http://blog.enterprisitplanet.com/green/blog/2008/06/green-petaflop-ibms-roadrunner-wins-supercomputer-top-spot.html) The National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory in Tennessee, USA which has several large supercomputers, has a 2008 annual budget between $80 million and $100 million. (reference: http://news.cnet.com/8301-13772_3-9985500-52.html). A typical personal computer might be capable of approximately 4 billion numerical operations per second, so the Blue Gene computer at Los Alamos provides roughly $25 \times 10^6 = 250,000$ times more computing power.

Typically the programming is developed on personal computers, then moved to the larger devices. Usually the expensive supercomputer time is spent primarily on numerical computation, rather than on providing a nice-to-use interface for users. However, the results of a supercomputer may be shipped over the Internet to a personal computer so that a scientist or engineer may mull over the results in a more contemplative way without incurring additional supercomputer costs.

Multiple computers may be linked together over the Internet to get all the pieces of a particular elaborate computation done. This may have the supercomputers performing the massive numerical computations, while other smaller or personal computers linked in may be displaying the results to a distributed team of investigators who are collectively digesting results and steering the ongoing work.

Hand held or mobile devices

Calculators are useful for casual computation, where one wants to figure out the solution to a small problem once. It's easy to punch in a few numbers and operations and to read the answer on the display. As inexpensive and small mobile devices, these are typically more limited in:

1. memory (limits to computation size and to built-in features)
2. energy consumption (slower processor speed)
3. form factor (keyboard and display too small for more than casual technical use)

Typically calculators are not networked, making it harder to share results to transfer them elsewhere to continue the work.
The TI-Nspire with CAS is a recent generation calculator from Texas Instruments. It can do numerical and symbolic calculations, as well as graphing. According to ticalc.org (http://www.ticalc.org/basics/calculators/ti-nspire-cas.html) it has 16Mb memory, 20Mb storage and has a 150MHz processor. This makes it have about 100 times less memory, 8000 times less storage, and is about twenty times slower than a typical laptop with a dual core 1.5GHz processor, 2Gb memory, and 160Gb disk. Its screen is 240 x 320 pixels, giving it about twenty times less display area than a typical laptop. This is one of the first generation of calculators with limited wireless networking available as an add-on.


Smartphones, personal digital assistants (PDAs), media players have in theory the same processing capabilities as calculators although most of them are not used extensively for technical computation. One advantage of these kinds of devices is that are typically networked so that it's possible to get a more powerful computer somewhere else in the Internet "cloud" of computational resources to do some of the work.

In the future there may be more convergence of the capabilities of all "mobile small form factor" devices. However the small form factor and the limits to energy consumption will probably continue to constrain the capabilities of such devices compared to larger ones.

Dedicated controllers

Even some kinds of toasters have microprocessors in them nowadays. "Smart homes" may network many appliances and home features such as HVAC and lighting control. While these devices are merely configured rather than programmed by end users, students of technical computing should not forget that this is another place where programming is necessary. The economic reasons for switching to computer control of devices appear to be the greater flexibility and variety of control that can be developed at modest cost through programming for many kinds of devices. The kind of programming done for device control often has a mathematical basis. Although the processors in dedicated devices are typically a few orders of magnitude slower than those of personal computers or even smartphones, the programming languages used for them are often the same or similar to those used in personal computing.
1.5 Maple, a system for technical computing

In this course, you will learn how to do various kinds of technical tasks using Maple. The first version of Maple was developed in the early 1980s at the University of Waterloo in Ontario, Canada, but has since undergone many refinements and extension. (In Fall 2010, we will be using Maple 14.) Maple's original emphasis was on algebraic computing, so it was used by those who needed formulas as results for their work. In the early '90s it was used extensively in calculus classes as a way of supporting exploratory experimentation and more extensive experience with "applied" problems. More recently it has expanded its domain into the symbolic, numeric, and graphical calculations done more generally in scientific and engineering modeling and simulation.

Today, Maple supports numerical computing and graphical visualization about as well as it does symbolic computing. It has its own programming language but also has ways of doing calculations through the graphical user interface (GUI) that is a kind of augmented "point and click". The default way of interacting with Maple also allows you to mix documentation and computing instructions and results. Thus it's particularly easy in Maple to produce documents that produce a well-documented solution -- a description of the problem, the explanation for how to solve it, and the computations that produce the details of the answer and evidence to justify its correctness.

1.6 What about Systems X, Y, Z, ...?

There are a variety of systems and programming languages used for technical computing nowadays. In a large technical establishment such as a university or research lab, one might find, in addition to Maple the use of systems such as: Python, Matlab, C, Mathematica, Java, Octave, Macsyma, Sage, Axiom, or Fortran.

The vast array of tasks computers can be used for, the multiplicity of philosophies of system design, and the rapid and unrelenting nature of progress in computing means that there is no "best system" for any situation. Each system has its technical strengths. There may also be historical, cultural, or economic reasons why certain systems are used in certain places which co-exist with the technical justification. As with all popular and demanding activities, there are many products in use.

One important ramification of this is that students in technical fields must expect to become familiar and proficient in more than one programming system. Studying more than one brings greater knowledge of what's possible, and the advantages in judgment that comes from knowledge of diverse ways of achieving a goal. It also allows one to function effectively as a team member in larger cross-organizational efforts and to better to cope with the diversity in computing cultures that's out there.

1.7 Why pick Maple as the first system?

1. It is an interactive system, facilitating quick exploration of new ideas. Compared to languages such as C++ or Java, one can immediately start up an interactive system and calculate results through "point and click" and a little typing. It's also possible to enter computational scripts, which are sequences of steps that are less elaborate to set up than programs. The scripts can be easily re-run on variants of the original situation just by changing a line or two in the script. This allows convenient "what-if" exploration, where a number of different scenarios are explored through computation. The worksheet interface to Maple allows sophisticated mathematical typography and graphics as well as mouse- and palette-driven input.

2. It can handle calculations with formulas. Rather than figuring out the formula yourself, you can even get Maple to calculate the formula for you. While many mathematical calculations produce numerical results, formulas are needed to specify how those results are produced. Designers or developers often need to find and create the formulas rather than to just copy them from a book. A system where formulas are easy to represent and easy to create supports this kind of work. A system that allows you to represent and calculate both numbers and formulas makes it easier to do and document technical calculation than a system that just works with numbers.

3. It supports a variety of data structures that support technical computation: formulas, equations, functions, sets, lists, tables, vectors, and matrices. Rather than having calculations done with a number of digits that's fixed forever, it is easy to change the precision if more (or less) precision is needed. Having higher-level interaction with the computer is usually more productive of human time. Having all of these entities as "first class entities" in the system requires less mental effort by the user, since they...
do not have to translate what they are thinking about (e.g. a table of formulas) into the terms that the computer is using. Rather, the computer language accommodates the human style of thought.

4. **It supports documentation as well as calculation.** From the instructor's point of view, it's easier to create documents that explain Maple because we can use Maple both to handle the calculation and the presentation of results. For students, having a file with both the directions that cause the computation and the results makes it easier to present the work cleanly for grading, and for future reference. For professionals doing technical work, having an integrated environment where text, programming and results can be combined together can be a convenience.

5. **It has a "conventional programming language".** An objective of this course is to make you become familiar enough with some of the standard elements of programming (e.g. assignments, conditionals, loops, procedure definitions) so that you can use them creatively to handle certain common situations that can't be handled through point-and-click or built-in operations. These elements will be found in highly similar form in the languages used by many other systems used for technical work.

6. **The mathematics of modeling and simulation is an explicit feature of the language.** While its programming language is conventional, Maple's language has an important added plus -- you can work over expressions, equations, functions, and other mathematical objects just as conveniently as if they were numbers or characters. Conventional languages (e.g. Java) can represent the arithmetic operations -- multiplication, addition, subtraction, and division -- but after that there are special tricks and conversions that you must perform to bridge the gap between what is written in the program, and the mathematical ideas that you are trying to use in the model. It's more straightforward in Maple to represent the mathematical model and the computation based on it. This ease of expression and comprehension by programmers has a hard-headed dollars and sense payoff. Less programmer time can be spent developing a computation. There is also an efficiency advantage to explicit representation of formulae: it is possible to use systems such as Maple to automatically improve the quality and efficiency of simulation calculations. This again leads to lower costs of doing the work.

We think these things provide a software engineering advantage that will lead most technical computation systems to eventually have such functionality built-in into them.

### 1.8 Using more than one system

Any user of computers who expects to use them professionally for design and investigation must expect to eventually learn multiple systems. Using computer applications for work is like using tools in a workshop-- you would not expect to use one tool to do all tasks, even if the tool, like Maple, has "Swiss Army Knife" capabilities. Features that make it easier to do certain kinds of things may slow down doing other things. Even when systems overlap in features, they typically have differing philosophies and different technical strengths, which means that certain kinds of work may be significantly easier in one system than another. For example, developing something in Mathematica or Maple may be fine and quick for a personal computer, but making the same programming work on a supercomputer may take a lot of effort in a different language. Yet a work environment with multiple languages need not be overwhelmingly complex. Most systems with major development effort behind them (such as Maple and those mentioned in the "section above) have many similarities.

What makes things work out is this: at the introductory level, the difference between casual computing and professional technical computing is the style of working (higher emphasis on documentation, justification and ease of reuse), and the use of language-based commands/programming needed to do the more sophisticated operations in technical work. "Crossing over" to the professional mode of operation means getting over the hurdles of learning the new style of work, and learning how to interact with computers in a typical computer language. Once this hurdle is passed, it should take only incremental effort to acquire expertise in the second, third, or n-th technical system. Having formal instruction on the first system should provide an explicit introduction into the concepts and the work processes to facilitate this.

Most systems realize that they cannot be the sole provider of technical computing services. If Institution A uses Mathematica and Institution B uses Python for their work, then if they expect to use each other's efforts, there has to be a way of interconnecting programs written in one system with that of another. Thus most systems have interconnections. For example, the Matlab Symbolic Toolkit allows Matlab users to call Maple to do formula manipulation. Similarly, Maple users can link to Matlab and run a Matlab program they've gotten from a colleague as part of a problem-solving process written in Maple.
Knowledge of basic programming and the concepts of *software development* make it possible to switch between systems with only a modest amount of additional effort. Software interconnection allows one to reuse programming done in another system without having to translate into another language. Symbolic computation systems like Maple also have the additional bonus of being able to translate some of their programming between languages. There is a "convert-to-Matlab" feature for example, or a "convert-to-C" feature for computations involving just numbers or text.
1 Introduction -- Technical computing at the turn of the century
2 Getting started with Maple's Document Mode: doing technical work with a clickable interface

2.1 Chapter synopsis

1. How to start up Maple and perform simple calculations, algebra, and plots within it.
2. Detecting and fixing typographical mistakes.
3. Introducing mistakes caused by vocabulary misunderstandings and the use of incorrect logic in giving directions.

2. How to save Maple work so that you can refer to it or resume working on it later.
3. How to recover a Maple worksheet if it or your computer crashes.

2.2 Starting up Maple, getting a fresh start

Start up the Maple application (this varies on the type of computer system you have, typically it involves clicking or double-clicking on the Maple 13 icon, but if you can't figure it out yourself ask for a demo for someone who knows). Once the Maple application window appears, a new "document" will appear in the main working area of the Maple application. A flashing cursor will appear with the outline of a small rectangle with dashed lines. The entry mode will read "Math" and "2D Math". You can make the "quick help" black box disappear by clicking on the "close box" circle-X in the upper right hand corner.

Table 2.1: Maple started up with new document in Windows XP

![Image of Maple started up with new document in Windows XP](image.png)
After you close the quick help box, you will see the Maple cursor in a small rectangle with a dashed line outline.

**Table 2.2: Maple document with first entry area**

At this point, what you type will appear in the small rectangle and be regarded as *a mathematical expression*. In the next section, we describe what to type in order to get something useful to happen.
2.3 Evaluating an expression involving exact arithmetic

Grade school arithmetic

In the math area, type $2 + 3$. "$2 + 3"$ is regarded as a mathematical expression by Maple when you type it into the dashed rectangle. As you are typing, the input indicators should say that you are entering Math mode, in "2D Math" input, using "Times New Roman" font:

Table 2.3: Maple input using 2d math

![Image of Maple input using 2D Math]

This expression should show up in the work area. When you hit the enter key, then Maple will evaluate the expression. After the expression is evaluate, you should see the result displayed below the input, as in the figure below:

Table 2.4: Maple input with labeled result

![Image of Maple input with labeled result]

Maple has automatically calculated the answer and given it a label (1). After Maple calculates the answer, the cursor should appear below the result in another dashed-line rectangle. This indicates that Maple is ready to do another calculation.

Maple supports all the basic arithmetic operations in a fashion similar to many other programming languages. One thing that takes some getting used to is that * (asterisk) is used to input multiplication rather than "x" or a centered dot. Another thing that takes some getting used to is that what you type is formatted to look like math notation. Thus if you type a /, Maple understands that you are talking about division and immediately starts formatting your input as if it were a fraction. If you type an asterisk, Maple formats that into a dot (⋅). There is also formatting that occurs with caret (^) since that is the way you enter an exponent in Maple.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Character to type/ character name</th>
<th>Notes</th>
<th>Example</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+ (&quot;plus&quot;)</td>
<td></td>
<td>2 + 2</td>
<td>4</td>
</tr>
<tr>
<td>multiplication</td>
<td>* (&quot;asterisk&quot;)</td>
<td>Typing an asterisk makes a center dot (·) appear in the displayed expression.</td>
<td>2·3</td>
<td>6</td>
</tr>
<tr>
<td>division</td>
<td>/ (&quot;slash&quot;)</td>
<td>Typing a slash draws a baseline and then positions the cursor in the denominator. Subsequent typing appears in the denominator. To get out of the denominator and return to normal typing, use the right-arrow key (→). Multiple divisions are by default conducted left-to-right.</td>
<td>( \frac{2}{6} ) ( \frac{1}{3} )</td>
<td>( \frac{2}{6} ) ( \frac{5}{3} )</td>
</tr>
<tr>
<td>subtraction</td>
<td>- (&quot;dash&quot; or &quot;hyphen&quot;, typically on the same keyboard key as the underscore. Dash and underscore do not mean the same thing.)</td>
<td>Multiple subtractions are conducted leftmost first.</td>
<td>3 − 5</td>
<td>−2</td>
</tr>
<tr>
<td>parentheses</td>
<td>( , ) (&quot;left parenthesis&quot;, &quot;right parenthesis&quot;)</td>
<td>Use parentheses to change the order of calculation. They are also good for removing any guesswork by the reader as the order of operations.</td>
<td>((2 + 3) \cdot 5)</td>
<td>( \frac{25}{29} ) ( \frac{15}{29} )</td>
</tr>
<tr>
<td>negation</td>
<td>- (&quot;dash&quot; or &quot;hyphen&quot;, typically on the same keyboard key as the underscore). This is the same symbol as used for subtraction</td>
<td>Put a dash in front of a number or parenthesized expression to negate it.</td>
<td>−(3·5 − 2)</td>
<td>−13</td>
</tr>
<tr>
<td>power</td>
<td>^ (&quot;caret&quot;, typically on the same keyboard key as the number 6)</td>
<td>Typing a caret moves the cursor to the exponent position. Subsequent typing appears in the exponent. To get out of the exponent and return to the baseline, use the right-arrow key (→).</td>
<td>( 2^3 ) ( 2^3 - 5 )</td>
<td>( \frac{1}{4} )</td>
</tr>
</tbody>
</table>
is the product of all the integers between 1 and n. It is useful in computations that compute the number of possible ways that something could happen. For example, the number of possible orderings of a deck of playing cards is 52!

\[ 4! = 24 \]
\[ (3!)! = 720 \]
\[ 52! = 806,581,751,709,438,785,716,606,368,5 \]
\[ 640,376,697,528,950,544,088,327,7 \]
\[ 8,240,000,000,000,000 \]

Let's try out some of these operations. We can enter a sum of two fractions by using "+" and "/". If we type "2/3→+5/6 enter", this is what we see:

\[ \frac{2}{3} + \frac{5}{6} = \frac{3}{2} \]
We can mix operations. Try to enter and calculate the following:

\[
\frac{1 + \frac{2}{3+4} + 5\cdot6 + 7}{8}
\]

\[
\frac{67}{14}
\]

In order to get that last denominator, we had to select the expression we had entered for the numerator with the mouse, so that the entire contents of the entry rectangle were blue. Then we typed a slash and the denominator appeared beneath it all.

An alternative to using the mouse to enter expression (1.2.1.18) would be to use parentheses. If we type "(1+2/3+4+5*6+7)/8 enter" we will see this:

\[
\frac{\left(1 + \frac{2}{3+4} + 5\cdot6 + 7\right)}{8}
\]

\[
\frac{67}{14}
\]

This allows you to enter complicated expressions without having to use the mouse. Of course, the mouse is still necessary if you want to go back and edit.

We observe in passing that a distinctive feature of Maple is that Maple does exact arithmetic with integers and fractions. It keeps fractions as the ratio of two integers. It will, however, automatically simplify such ratios to lowest terms:

\[
\frac{2}{3} \cdot \frac{6}{7} - \frac{18}{7}
\]

\[
-2
\]

**Making typographical mistakes**

Making mistakes is a normal part of using any tool, be it a computer or otherwise. You'll probably make as many mistakes learning how to use Maple as you would make when learning a new sport, a musical instrument, or when learning how to write a good essay. Some of them will be obvious as soon as you make them, others will be subtle or harder to figure out how to fix.

When you make some kinds of mistakes, the computer may give you an *error message*. For example, if you make a typo and Maple doesn't recognize what you enter as being a valid command, it will complain. Here are some typical error messages. At this stage, some of the messages will make sense and you can react appropriately. Others will use vocabulary that is unfamiliar. For those, the best thing to do is to have a clear idea of what you want to enter (by closely imitating examples that are known to work) and checking carefully that what you have typed was accurate.

**Table 2.6: Examples of Maple error messages**

<table>
<thead>
<tr>
<th>Error, invalid sum/difference</th>
</tr>
</thead>
</table>
We intended to enter "2 + 4" but forgot to type the "4" before we hit enter (return). The appropriate thing to do here is to correct the expression and hit enter again.

\[
2 + 4
\]

\[6\] (2.23)

\[+ 2 4\]

Error, missing operation

\[+ 2.4\]

This time we mistakenly enter the expression with the symbols in the wrong order. Maple complains that it expects some operation to be entered between the two numbers but there isn't one.

\[. + 4\]

Error, invalid matrix/vector product

\[+ 4\]

We intended to enter "2+4" but typed a period instead of a 2 by mistake. Even though we are trying to do the same thing, the error message is different because a different symbol (the period) in this context suggests to Maple that we are trying to do linear algebra. The appropriate thing to do here is to correct the expression and hit enter again.

\[2 + 4\]

\[6\] (2.26)

\[\frac{2}{0}\]

Error, numeric exception: division by zero

If we ask Maple to do an impossible operation, it sometimes gives an error (depending on the operation). The appropriate question to ask yourself here is "what should I be dividing by instead of zero?".

\[\frac{3}{5 + 3\cdot (}\]

Error, unable to match delimiters

\[\frac{3}{5 + 3.}\]

We started a sub-expression with a parentheses but forgot to finish it. In Maple, a delimiter refers to a parenthesis -- ( or ) -- a bracket [ or ], or a brace { or }. Delimiters are symbols that mark the beginning and end of an expression. In many instances they are necessary to unambiguously indicate meaning. For example \(5 \cdot (3 + 5)\) evaluates to 40, where as the expression without parentheses \(5 \cdot 3 + 5\) means 20 because multiplications are always done before additions unless the parentheses indicate otherwise.

\[
\left(3 + \left(5 + \frac{3}{7}\cdot5\right)\right)\cdot2
\]

Error, unable to match delimiters

\[
\left[3 + \left(5 + \frac{3}{7}\cdot5\right)\right]\cdot2
\]

This is another instance of the same mistake. We wanted to enter \(\left(3 + \left(5 + \frac{3}{7}\cdot5\right)\right)\cdot2\) but misplaced several parentheses.
Error, invalid sequence

We intended to enter "1+3" but typed the extra comma in by mistake. We get an error message that talks about sequences, a concept in Maple that we haven't discussed yet (that's coming in the next few chapters). Maple thinks that the sequence is "invalid" because it usually expects commas to be between items, such as "1,2,x".

Maple's language has many other elements in it that we haven't gotten to discuss yet. If you use any of them by mistake then you will see messages with vocabulary we haven't discussed yet.

It's a fairly typical experience for new users to see some messages that you won't be able to gather much intelligence from other than the fact that you made a mistake that you should fix. With knowledge-rich systems such as Maple, you may blunder into sections of the system that you haven't learned yet. The best strategy is to back out of the situation by editing the expression so that it is exactly like something that is known to work.

Error, invalid fraction

This one is fairly obvious. In order to fix it though, we need to know what denominator we intended to enter.

Correcting typographical mistakes

The standard procedure for fixing a mistake is as you would in a word processor: edit the mistaken input and re-execute the computation. Here are ways of doing this:

1. Using the mouse, position the cursor where the mistake is. Then use the backspace key to erase the characters you want to get rid of. Type in more characters to replace it.
2. Use the left arrow key (←) to back up. Typing after backing up then inserts the new typing at the point where the cursor was positioned.
3. Use the mouse or other "pointing device" of your computer to select a section of what you typed. New typing then replaces the selection of what you typed.
4. Use the mouse to select a region, then "cut", which you can do through the Maple menu Edit -> Cut. Of course most people use the keyboard shortcut for cutting, which for Windows or Linux is control-X while on the Mac it's command-X.
5. Copying and pasting (control/command-C and control/command-V) also works in Maple.
You may find that sometimes you attempt to create a 2D Math input area but Maple does not compute a result for the input after you hit the enter key. To create a "clickable math" input area if this happens, place the cursor where you want the input area to be, and use the Format->Create Document Block Menu item of the Maple window:

Table 2.7: Create Document Block to force a Math input area wherever the cursor is placed

Exponentiation (powers). Numbers with lots of digits

Use a caret ^ to specify an exponent (a "power"). In math mode, Maple will position the cursor so that the next things you enter will become the exponent. As with fractions, you can get out of the exponent by using the right arrow key → when you want to go back to non-exponent numbers. Try entering these expressions

\[ 2^3 \]

\[ 2^{1000} - 2 \]

\[ \frac{53575430359313366047421252453000090528070240585276680372187519418517552556246806}{1246599189407847929063797336458776573412593572642846157021799228878734928740} \]

\[ 1967283887412115492710537302531185570938977091076523237491790970633699383779 \]

\[ 582771973038531457285598238843271083830214915826312193418602834034687 \]
We note that Maple does integer and fraction operations exactly. It will not introduce any rounding error into a computation as a calculator would when the answer requires more than ten decimal digits to write down.

There are limits to the number of digits Maple will use for integers or fractions, but they have to do more with ultimate limits of the computer hardware and memory rather than a "pre-ordained" decision about how many digits might be useful to keep. If you type `kernelopts(maxdigits)` into a Math input area on the worksheet, Maple will print out a number which is the maximum number of digits it can handle in any integer or fraction. On the author's computer,\

\[ kernelopts(maxdigits) = 268435448 \]

Note that this is not the value of the maximum number, but how many decimal digits the largest number can have.

For example, Maple can compute the result of\

\[ \frac{1}{52!} + \frac{2^{100}}{3^{27}} \] exactly (try it!).

Exact computation is useful not only for doing algebra but also for things such as computing probabilities through counting, or in deriving mathematical formulas for use in simulation or prediction programs.

**Detecting and fixing vocabulary and "logic" mistakes**

There will be other kinds of mistakes where there is no error message, but the response is not what you want. Sometimes this happens because you say something that while grammatical, means something entirely different from what you meant. This could be as simple as entering "2-3" where you meant to type "2+3". Sometimes it is less obvious, such as mistyping the 23-rd digit of a 55 digit number or the 12th term of a long sum. Or worse yet, it could be because you are using the incorrect vocabulary so what you think you are saying does not have that meaning to the computer.

**Table 2.8: Example of a vocabulary mistake**

<table>
<thead>
<tr>
<th>2x^3 + 5</th>
<th>2x^3 + 5</th>
<th>(2.32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2·3 + 5</td>
<td>11</td>
<td>(2.33)</td>
</tr>
</tbody>
</table>

Suppose we were under the (mistaken) impression could use "x" in Maple to stand for multiplication. We might use the above as a way to calculate "two times three, plus five". However, while there is no error message, what is calculated is not the number we were expecting. The first step towards proceeding to fix this is to realize that it's *not* what we want. To Maple, this is a legitimate calculation -- you want to create a formula that's "two times the variable x3, plus five".

Knowing that the proper way to enter multiplication is through a palette, or symbol "*" (asterisk) as explained in

Finally, there are mistakes made because *you ask Maple to do the wrong calculation*. Even though Maple does what you want, it turns out that what you wanted was wrong!

For example, you may read a word problem and decide to solve the equation \(3·x + 2 = 6\), whose solution is \(x=4/3\). But when you plug the solution into the circumstances of the problem, you find that it doesn't solve the word problem. The problem may be because you used faulty logic to decide that \(3·x + 2 = 6\) was the equation, but it was actually \(2·x + 4 = 6\). This is known as an "error in logic" or just a "logic error".

Sometimes it's obvious that you made a mistake because the answer is obviously wrong. For example, seeing "-1" when you are expecting the answer to "2+3" is obvious because you know already that you shouldn't get an answer which is a negative number. Sometimes it isn't so obvious, so *you need a way to check the correctness of the result*.
2.4 Saving and retrieving your work

You can save your work in a Maple worksheet file through the File -> Save (keyboard shortcut: control/command-S) menu item of the Maple application. A dialog box appears allowing you to type in the name of the worksheet (we've typed in "test" into the box in the .)

The file is saved with a ".mw" suffix (e.g. test1.mw).

To retrieve the file in a subsequent Maple session, start up the Maple application as before and then use the File->Open menu item to initiate the dialog that retrieves the file.

Table 2.9: Maple save menu operation
2.5 Retrieving backups

The "state of Maple" display appears on the bottom left hand side of the Maple window. Usually this display says "Ready", which means that Maple is awaiting your next command. Sometimes it reads "Evaluating..." which indicates that Maple is actively computing an answer. Sometimes it says "Autosaving worksheet", which means that it is saving a copy of the present state of your open worksheets into temporary storage on your computer. The amount of time Maple spends autosaving becomes noticeable in longer Maple sessions when the worksheet contains a lot of results.

Should your computer suddenly lose power or should Maple crash, you can retrieve the last autosaved worksheet by selecting the File -> Recent Documents -> Restore Backup menu item. This will fill your Maple with copies of all the autosaved worksheets. You may then delete them or save them to permanent file space as you wish.
2.6 Algebra, plotting and mouse-clickable operations

Algebraic expressions and equations. Solving equations. Working with pieces of expressions.

We don't have to limit ourselves to just numerical calculation in Maple (even if the ability to use as many digits as we wish and exact fractions allows us to do arithmetic more like the math books do). Another distinctive feature of Maple is that you can do algebra by entering expressions with symbols -- the \( x, y, z, i, \) and \( n \) that we see in algebra books. Maple will automatically collect terms and do some simplifications for us automatically

\[
x^2 + 2 \cdot x + 5 + 3 \cdot x
\]

\[
x^2 + 5x + 5 \tag{2.34}
\]

We can even enter equations:

\[
\frac{3}{5} \cdot x + 1 = 4 - x
\]

\[
\frac{3}{5} x + 1 = 4 - x \tag{2.35}
\]

\[
3\cdot x + 1 + 4\cdot x = a \cdot x + b
\]

\[
7x + 1 = ax + b \tag{2.36}
\]

Note that while Maple automatically collected the \( x \) terms on the left hand side of the equation, it does not try to do the more interventionist operation of moving all the \( x \) terms to the same side of the equation.

Now, enter the following expression, but rather than hitting the enter key after you've entered it, do a right-click. On the Mac, instead of right-clicking hold the control key down then click on the mouse button (this is referred to as "control-click"). A menu of algebraic operations will pop up. Select Factor and see how Maple can factor the polynomial:

\[
x^2 + 5x - 50 \quad \text{factor} \rightarrow (x + 10)(x - 5)
\]

Note that this line does not have a (XX) label for it.

To further demonstrate the right-click (control-click) operations available, enter the following equation. Right click on the expression and a pop-up menu should appear that includes a "solve" item. Select the solve and a submenu will appear where you can specify that you want to solve for \( x \).

\[
\frac{3}{5} \cdot x + 1 = 4 - x \quad \text{solve for } x \rightarrow \left[ x = \frac{15}{8} \right]
\]

For those with previous experience on other systems: some things are different, for a reason

(This section is for people who already know a programming language and are noticing that Maple does some things differently. Other readers can skip this section.)

Most mainstream programming languages work in a similar way. That is why learning Maple has value beyond just being able to use the Maple system -- once you learn Maple, learning Matlab or Java or C is a matter of incremental adjustment. Noticing the differences is educational in that it makes you become aware of the arbitrariness of some of features of languages. If you have used another programming language such as Java or Visual Basic (VB), you will notice that some things do not work the same in Maple as they do there. Some of this can be explained if you know something about the history of programming languages. Some of it is explained by being aware that the designers of Maple had different goals than those who invented Java or VB.)
One thing that you have undoubtedly noted is that symbols do not have to have a value associated with them. For example, in Java if you said

\[ k=5; \]

Then if you were to create another expression in Java such as `System.out.println(k^2 + k + k + 3);` then "5" would be used as the value of \( k \) in the expression and you would end up printing 38. In Maple, you do not have to associate \( k \) with a numerical value before you use \( k \) in an algebraic expression. If there is no prior association, Maple just treats the expression as a formula with symbols in it. It may do some algebraic simplification on what you entered, but it does not need to get a number as a result. Since there was no prior assignment for the expressions in section 2.5.1, the calculation done with them just keeps the formula.

Another thing that is different is that in Maple "=" is used for equations, not assignment. The operator in Maple corresponding to "=" in Java or VB is ":=" (a colon immediately followed by an equals, with no spaces in between). In Maple, if we wanted to associate "5" with the symbol \( k \), then we would do:

\[ k := 5 \]

\[ k^2 + k + 3 + k \]

(2.37)

(2.38)

People who know more than one programming language have a better understanding of which features are change a lot between languages, (such as whether = or := is used for assignment), and which ones are fairly uniform (+ being used for addition, or the use of parentheses in functions and expressions).

Maple does not use "=" for assignment because, being a mathematically oriented language that can handle algebra, it wants to make the entry of equations a natural thing. Its use of ":=" for assignment is a feature borrowed from the Algol/Algol 68/Pascal family of programming languages, which picked this operator to make it clear that the assignment operation is different from algebraic equality.

Is ":=" better than "="? That's a kind of question that is about as hard to answer as trying to decide whether "uno" is better than "one". If one had a language where you had to do ":====\&\%\^\#\+\+\==\==\==\==\==\==\==\==\==\==" instead of "=" or ":="; you could be critical of the choice because it takes much more effort to enter a 35 character operator than a one or two character one. But the Algol-family choice of "=" has reasonable motivation -- studies of novice programmers have shown that beginners using languages where "=" is the assignment make more mistakes because they confuse its use in mathematics with its use in programming. Novices have been observed to write things like "5=k" which does not work as an assignment, even though mathematically the equations "k=5" and "5=k" mean the same thing.

Just as with architectural design of a building, each feature of a programming language is typically carefully considered. Many features are borrowed or copied from predecessor languages, where they have already been subjected to the test of many people using the feature. If you are not familiar with Algol family languages, you can see examples of them at various educational web sites, such as http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/average.html and http://portal.acm.org/citation.cfm?id=154766.155365.

### Plotting and approximate numerical solutions

The right-clickable interface can also activate a plot of a formula you may have entered. Enter this formula, then right-click and select **Plot** \( \rightarrow \) **2d plot**. The automatic defaults for plotting this produce this result.
Table 2.12: Example of Plotting

\[ x^2 - 10 \cdot x + 4 \]

User has clicked on the plot and positioned the cursor at the coordinate (-4.12, 61.60). The cursor was not captured by the screenshot although it is visible under ordinary use.

Table 2.13: Plot created by right-click -> Plot -> 2DPlot

The 2DPlot operation makes pre-set decisions about the plot, such as the range of \( x \) (-10 to 10), the color of the line, axes labelling, etc. Users can inject their own preferences about these things about this by selecting right-click -> Plot -> PlotBuilder and filling in the dialog box with their choices.
Table 2.14: User-configured plot using PlotBuilder instead of 2DPlot

\[ x^2 - 10 \cdot x + 4 \rightarrow \]

The Expressions Palette and the Common Symbols Palette: entering Trigs, logs, roots, exponentials

It's possible to get the common functions of high school algebra, pre-calculus, and calculus by using the Expression palette of the Maple Window:

Table 2.15: The Expression palette
For example, to enter the square root of 36, click on the palette entry for $\sqrt{a}$. That expression will appear in the document, with the "a" selected. If you then type 36 on the keyboard, that number will replace the selected text. If you then hit the enter key, Maple will evaluate the expression and produce the exact result "6".

$$x + y + \frac{1}{2} + \frac{1}{4} + \sqrt{36}$$

$$x + y + \frac{27}{4}$$

(2.39)

You can use the palette multiple times, to create more complicated expressions. Just continue to the use mouse or the arrow keys to move around in the expression. Selecting and typing, backspacing or deleting are all ways of replacing or correcting pieces of the expression.

The Common symbols palette, two panels below the Expression palette, can be used to enter $\pi$ and $e$, the base of the natural logarithm system.

The palette does not have the inverse trig functions, so you have to enter them through typing. Their names are: arcsin, arccos, arctan, etc.
Approximate numerical (calculator - type) arithmetic in Maple

If you enter expressions with integers, exact fractions, and symbols such as \( \pi \) and \( e \), then Maple will perform exact calculations rather than give approximate answers as a conventional calculator would. You can get approximations by selecting the "numerically solve" instead of the "solve" option from the right-click pop-up menu.

### Table 2.16: Examples of palette-driven computation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sin\left(\frac{\pi}{4}\right) + \cos\left(\frac{\pi}{4}\right) )</td>
<td>( \sqrt{2} ) (2.40)</td>
</tr>
<tr>
<td>( \frac{\sqrt{1024} + \ln\left(\frac{2}{3}\right)}{13} \cdot \pi )</td>
<td>( \frac{98}{39} \pi ) (2.41)</td>
</tr>
<tr>
<td>( \arcsin\left(\sin\left(\frac{1}{4}\cdot \pi\right)\right) )</td>
<td>( \frac{1}{4} \pi ) (2.42)</td>
</tr>
</tbody>
</table>

### Table 2.17: Examples of computing with approximate solving

<table>
<thead>
<tr>
<th>Expression</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^2 - 10\cdot x + 4 = x^2 - 10\cdot x + 4 ) ( \text{solve} )</td>
<td>( { x = 5 + \sqrt{21} }, { x = 5 - \sqrt{21} } ) Exact solution of an equation using the &quot;solve&quot; feature of the pop-up menu.</td>
</tr>
<tr>
<td>( x^2 - 10\cdot x + 4 ) ( \text{solve} ) 0.4174243050, 9.582575695</td>
<td>Using the &quot;numerically solve&quot; feature of the pop-up menu</td>
</tr>
</tbody>
</table>

If you have an exact expression, you can ask Maple to approximate it to 5, 10, 20, or more digits. In this mode, Maple can be used as a super-accurate calculator.

### Examples of numerical computation

1. Enter fraction, select approximate->20 from right-click pop-up menu.

\[
\frac{47}{52} + \frac{4}{3} \quad \text{at 20 digits} \quad 2.2371794871794871795
\]

2. Enter exact expression, select approximate->5 from right click pop-up menu

\[
\sin\left(\frac{\pi}{10}\right) \quad \text{at 5 digits} \quad 0.30902
\]

3. Enter equation. Then solve->solve, then select Element->1, then right hand side, then approximate->10
Evaluation, and selection of pieces.

Sometimes you wish to evaluate an expression for a particular value of a variable. There is a right-click operation that does this.

Table 2.18: Evaluate at a point

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 2 \alpha x = 0$</td>
<td>evaluate at point</td>
<td>$\frac{1}{4} - \alpha = 0$</td>
</tr>
<tr>
<td>$x^2 - 2 \alpha x = 0$</td>
<td>evaluate at point</td>
<td>$x^2 - 6y^2 x = 0$</td>
</tr>
<tr>
<td>$3 y + 5$</td>
<td>evaluate at point</td>
<td>$14$</td>
</tr>
</tbody>
</table>

This operation will give a pop-up menu that will allow us to choose values for all the variables. In the first example, we picked $1/2$ for a value of $x$. Note that the pop-up menu will show what you typed rather than displaying 2D math. In the second example, we specified "$3y^2x$" as the value for $\alpha$. In the third example, we picked 3 as the value for $y$.

Using the right-click menu, it's possible to select or extract a portion of an expression for further work.

Table 2.19: Operations on equations, multi-part expressions

<table>
<thead>
<tr>
<th>Right hand side, left hand side</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \frac{\sin(\alpha)}{r^2 - 1}$</td>
<td>$x$</td>
<td>One of the options in the right-click menu is &quot;right hand side&quot;. It only works for equations.</td>
</tr>
</tbody>
</table>

Operations on multi-part expressions

<table>
<thead>
<tr>
<th>Select entry</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 4x = 4$</td>
<td>solve for $x$</td>
<td>$\left[ x = 2 + 2\sqrt{2}, x = 2 - 2\sqrt{2} \right]$</td>
</tr>
<tr>
<td>$\left[ x = 2 + 2\sqrt{2}, x = 2 - 2\sqrt{2} \right]$</td>
<td>select entry 1</td>
<td>$x = 2 + 2\sqrt{2}$ at 5 digits $x = 4.8284$</td>
</tr>
</tbody>
</table>

Solving this quadratic equation reveals that there are two solutions. Right-clicking on the solution and then selecting entry 1 gives the first solution, enclosed in brackets [ ]. Right-clicking on that and again selecting entry 1 gives the first solution, without the brackets. Right-clicking on that and selecting approximation gives a calculator approximation to the root.

2.7 A quick-reference summary to this chapter

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Use $+,*,-,/,^$ for arithmetic. Hitting the Enter key produces a labelled result.</td>
<td>2D Math input mode displays the textbook-like version of what you input. Maple's simplification automatically combined fractions and places things in lowest terms.</td>
</tr>
<tr>
<td>$2 + \frac{3^2}{4} - \frac{1}{6}$</td>
<td>$\frac{49}{12}$</td>
<td>(2.43)</td>
</tr>
<tr>
<td>$5!$</td>
<td>Use $!$ for factorial</td>
<td>Do you know what $5!!$ (double factorial) means?</td>
</tr>
</tbody>
</table>

Making mistakes
\[
2 + \left( \frac{3}{5} \right)
\]

Error, unable to match delimiters

Error message mistakes (from typos or mistakes in intensions)

The easiest ones to detect. You have to figure out what you are doing wrong, though. The error message may not always be helpful in advising you on this, although it often is.

A farmer plants a fence post every foot, for 1250 feet. At that point, he switches to planting the fence posts every 4.7 feet for another 940 feet. How many fence posts does he need in all?

\[
\frac{1250}{1} + \frac{940}{4.7}
\]

\[
1450.000000
\]

Maple did do the arithmetic in the above calculation correctly. The problem is that it's the wrong calculation. Do you see how to get the right answer?

"Logic errors"

You are asking Maple to compute something that it understands, so it gives you an answer. However, this answer doesn't really solve your problem. You need to find a more appropriate computation, which you can only do by thinking about whether you are asking the computer to do something different from what is needed. Often you can find these kinds of mistakes by looking at simpler versions of the problem where the answer can be figured out with paper and pencil. Then you can "scale up" the answer to handle the actual problem you have.

The correct answer is 1251 + 201 1452 fence posts. The computer did what it was asked to do -- the problem was that it was asked to do the wrong thing.

### Editing (fixing mistakes)

- backspace, delete erase starting from current cursor selection
- Arrow keys → ← move cursor within current selection
- Select with mouse/type replaces selected text
- Cut, copy and paste of a selection works as it does with a text processor

### File saves, opens

Save files with File -> Save or File -> Save As. Open a saved file with File -> Open. Other File operations are similar to that of standard word processors.

### Functions and math symbols

\[
\sqrt[3]{\csc \left( \frac{\pi}{2} \right)} + e^{(1 + e)^{1/3}}
\]

Insert math into an expression by using the Expression Palette. You can enter \(\pi\) using the Common Symbols Palette. \(e\) (the natural logarithm base) can also be entered this way. Note: typing e from the keyboard does not enter this symbol.

Chapter 2 demonstrated the following functions and symbols:

| Square roots, \(n\)-th roots | natural logarithms |
| Base 10 logarithms |
| Trig functions: sin, cos, tan, sec, csc |
| summation |

| arcsin, arccos, arctan |

If you are entering a function by the keyboard rather than the palette, you must enclose the function's argument in parentheses.

\[
\ln (e^2 \cdot \sqrt{e}) \quad \text{simplify symbolic} \quad \frac{5}{2}
\]

### Algebra

\[
x^2 - 2x - 15 = 0 \quad \text{left hand side}
\]

\[
x^2 - 2x - 15 \quad \text{factor} \quad (x + 3)(x - 5)
\]

Right-click (control-click on Mac) on an entered expression to get the pop-up menu.

Chapter 2 demonstrated examples of the following operations:
### Plotting

#### Plots→2d plot

The expression must be something that involves a single variable and will result in a number when a value is used for that variable. Thus you can't do a 2d plot of \( x^2 - a \) because you wouldn't get a number if you picked a value just for \( x \) (or just for \( a \)).

Maple uses defaults for the plot range, and the plot color.

Trying to plot an equation produces an implicit plot (see next appendix).

#### Plots→plot builder → 2d plot

A dialog box appears that allows you the select from many more options, such as plot color, the line style, vertical and horizontal ranges, captions, etc.

#### Limited precision (decimal point) numbers

Exact numbers in Maple have no decimal points. Use of limited precision numbers in algebra (e.g. factoring, differentiation, solving) may not produce good results.
Symbolic constants such as $\pi$ and $e$ entered from the Common Symbols Palette are also exact. Numbers with decimal points in Maple cause arithmetic calculations to be done approximately.

Use them in Maple only when an approximate result is desired. Numbers like .25 or .6015 are limited precision. If you want exact algebra done, use 1/4 or $6015/10000$, etc.

In very large calculations, limited precision calculations may be noticeably faster than those with exact arithmetic. Most of the time there isn't an appreciable difference.

### Evaluate at a point

<table>
<thead>
<tr>
<th>Example</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 2 \cdot a \cdot x = 0$</td>
<td>evaluate at point $\frac{1}{4} - a = 0$</td>
</tr>
<tr>
<td>$x^2 - 2 \cdot a \cdot x = 0$</td>
<td>evaluate at point</td>
</tr>
<tr>
<td>$x^2 - 6y^2 \cdot x = 0$</td>
<td>evaluate at point $14$</td>
</tr>
<tr>
<td>$3 \cdot y + 5$</td>
<td>evaluate at point</td>
</tr>
</tbody>
</table>

This operation will give a pop-up menu that will allow us to choose values for all the variables. In the first example, we we picked $1/2$ for a value of $x$. Note that the pop-up menu will show what you typed rather than displaying 2D math. In the second example, we specified "$3*y^2$" as the value for $a$. In the third example, we picked 3 as the value for $y$.

### Operations on equations

**right hand side, left hand side**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| $x = \frac{\sin(a)}{t^2 - 1}$ | left hand side, right hand side $\frac{\sin(a)}{t^2 - 1}$

One of the options in the right-click menu is "right hand side". It only works for equations.

### Operations on multi-part expressions

**select entry**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| $x^2 - 4 \cdot x = 4$ | solve for $x$

Solving this quadratic equation reveals that there are two solutions. Right clicking on these selections and then select entry -> 1 produces the first solution. We can then approximate it by using the

### Commentary

Enter an expression in a document, then right-click (control-click on Mac) followed by:

<table>
<thead>
<tr>
<th>Operations on symbolic expressions</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve-&gt;solve</td>
<td>$x^2 - 1$</td>
<td>$(x = 1), (x = -1)$</td>
</tr>
<tr>
<td>solve-&gt;solve for a variable</td>
<td>$x^2 - 2 \cdot a \cdot x = 0$</td>
<td>$[[x = 0], [x = 2 \cdot a]]$</td>
</tr>
<tr>
<td>solve-&gt;numerically solve</td>
<td>$x = \cos(x)$</td>
<td>0.7390851332</td>
</tr>
</tbody>
</table>

The thing to try when there is a numerical answer but the exact solution is too complicated to understand or Maple can't find an exact solution.

**Factoring**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2 - 1$</td>
</tr>
<tr>
<td>$\cos(x)^2 - \sin(x)^2$</td>
</tr>
</tbody>
</table>

Factoring can simplify an expression sometimes. Factoring doesn't know the trig simplification rules, though.
The expression must be something that involves a single variable and will result in a number when a value is used for that variable. Thus you can't do a 2d plot of \( x^2 - a \) because you wouldn't get a number if you picked a value just for \( x \) (or just for \( a \)).

Maple uses defaults for the plot range, and the plot color.

A dialog box appears that allows you the select from many more options, such as plot color, the line style, vertical and horizontal ranges, captions, etc.

### Operations on equations

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \frac{\sin(a)}{p^2 - 1} )</td>
</tr>
<tr>
<td>( x = \frac{\sin(a)}{p^2 - 1} )</td>
</tr>
</tbody>
</table>
This moves the entire side of an equation to the other side.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Uses approximation methods to turn constants into an approximate (decimal point) number with 5 (10, 20, 50) digits.</th>
</tr>
</thead>
<tbody>
<tr>
<td>move to right, move to left</td>
<td>$x^2 + x + 1 = a \rightarrow 0 = a - x^2 - x - 1$</td>
<td></td>
</tr>
<tr>
<td>Operations on constant expressions</td>
<td>$\tan\left(\frac{\pi}{10}\right) \sqrt{\frac{1}{10}} \rightarrow 0.34157868529293212152$</td>
<td></td>
</tr>
<tr>
<td>approximate-&gt;5 (or 10, 20, 50)</td>
<td>$x = \ln(5000!) \rightarrow x = 37591.143508876766569$</td>
<td></td>
</tr>
</tbody>
</table>
3 Chapter 3 Technical word processing

3.1 Chapter Overview

We learn how to use Maple as a word processor. This allows us to "write up" reports, combining technical writing with math formulae, calculated results, pictures, tables, etc. Many of the features are highly similar to Microsoft Word or similar WYSIWYG (what you see is what you get) word processors. The strength of Maple's word processing is that it makes it easy to enter technical formulae, and that the word processing and calculation can be done in the same document.

3.2 Maple as a word processor

Maple documents allow a mixture of text and mathematics. By default, Maple expects that when you position the cursor by clicking somewhere in the document, you will be entering math and be wanting it do to a calculation. The document is in what is called math entry mode.

Table 3.1: Maple in math entry mode

You can tell whether the document is in math entry mode because the Math button on the Maple toolbar will be gray, and the "C" menu item says 2D Math..

The other mode of operation for Maple documents is text mode. When in text mode, Maple has the behavior of a word processor. It just shows what you typed. Hitting enter while you are in text mode just causes text entry to move to the next line. It does not cause any calculation to be done with what you typed.
You can switch to entering text in the following way:

1. Position the cursor at the spot where you want to enter text.

2. Click on the **Text** button on the Maple toolbar. This places the Maple document in *text entry mode*. Alternatively you can switch to Text mode by typing control-T (on Macintosh, command-T) or by using the Maple menu bar Insert->Text. You can tell when you've switched to text entry mode because the Text button will be gray, and the "C" menu item says **Text**.

3. With the keyboard, enter your verbiage. When you are in text mode, you will also see that the menu bar will be enabled for boldface, italics, underlining, left-/center-/right- justified text, colored text, and colored backgrounds, and bulleted/numbered text.

4. To switch back to math, click on the **Math** button on the Maple tool bar. Alternatively you can type control-R (on Macintosh, command-R use the Maple menu Insert->2-D Math.

**Table 3.2: Document after control-T (or Insert->Text)**

![Text button on Maple toolbar](image)

A Maple worksheet in text mode in OS X. Although it is hard to see, the cursor is positioned at top left of screen.

You can do mathematical word processing without any computation by switching between text and math modes, using the Palettes to help you enter the math. As long as you don't hit the *return (enter)* key, the math will not cause any calculation.
Table 3.3: Document with a mixture of text and math

Richard saw in his physics textbook, *Stephen Hawking for Dummies*, a description of Newton's law of gravitation:

\[ F = \frac{G \cdot m_1 \cdot m_2}{R^2} \]

where it was expected that \( m_1, m_2, \) and \( R \geq 0 \). Although he didn't consider himself a strong physics student, he was glad that hadn't dumbed down the material so much that it lost all the mathematics.

It is possible to mix text and the results of calculations in a paragraph. Typing control-=` (command-=`) when the cursor is in a math expression will cause Maple to print an "=`" and then the result of evaluating the expression on the same line. This is an alternative to hitting the *enter* key and allows those kinds of calculations to be mixed with text.

Table 3.4: control-=` puts the results of a calculation in the midst of text

With a loan of $250,000 and an interest for 3.5% annual interest, the amount of interest after one year would be $250000 \cdot 1.035 = 2,58750000 \times 10^5$ dollars.
3.3 Shortcuts to entering math symbols

Using the Palettes, we can enter a wide variety of mathematics -- expressions, math symbols, Greek letters (using the Greek Palette), arrows, etc. There are additional Palettes not shown by default, which you can get by View → Palettes → Show All Palettes. However, you can enter many symbols in math mode from the keyboard through "shortcuts". Most of the shortcuts consists of typing the textual name of the symbol or some abbreviation of it, and then hitting the escape key -- the key labelled Esc on many keyboards.

For example, to enter the symbol $\infty$ while in math mode, you can type infin and then hit the escape key. A pop-up menu of choices will appear to allow you to complete entry of the symbol. With practice, this can be a faster way of entering "infinity" than using the Palettes.

Table 3.5: Keyboard shortcuts in math mode through the escape key

<table>
<thead>
<tr>
<th>In math mode, we type infin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>After hitting the escape key, a menu of completions appears.</td>
</tr>
<tr>
<td>We pick the first alternative (either by hitting the return key or by operating the mouse to select the first option) and what we typed is replaced by the selection.</td>
</tr>
</tbody>
</table>

Greek symbols can be entered by typing the romanized name of the letter, followed by escape. For example, in math mode, typing omega followed by escape produces $\omega$. Typing Omega followed by escape produces $\Omega$ (the upper case version of the Greek letter).

A shortcut to entering the symbolic constant $e$ (the base of the natural logarithm) is to type e, then hit the escape key, then return.

You can see a summary to all of the conveniences Maple offers through Help → Quick Reference.

3.4 Other word processor features

Inspection of the worksheet toolbar reveals many more word processing features: line justification, bold face and italics, numbered items, colored letters or backgrounds, font sizes, and font types. The Insert operation on the Maple toolbar allows creation of Tables and Images (graphics files). Rudimentary drawings can be inserted through Insert → Canvas. We encourage you to explore and make use of the features on your own.
3.5 Troubleshooting word processing

A phenomenon that you may encounter is *not being able to switch back to math mode from text mode*, even after performing the operation that should do so (clicking on the Text button of the document toolbar, typing control-T, performing Insert → 2D Math, etc. This may be due to the worksheet losing track of where you are in the document. A "sure-fire" cure for switching modes is to position the cursor at the point where you want to enter math, then do Format → Create Document Block. A dashed box will appear at the location of the cursor, indicating that it is again in math mode.

Tools→Spellcheck (alternatively, the F7 key) will run a spelling check on the non-math part of your document.

### 3.6 Summary of Chapter 3 material

<table>
<thead>
<tr>
<th>Name</th>
<th>Menu operation</th>
<th>Key short cut</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important word processing operations in a Maple worksheet</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch entry to 2D Math mode</td>
<td>Insert→2D Math</td>
<td>control-R (command-R on Mac)</td>
</tr>
<tr>
<td></td>
<td>Click on &quot;Math&quot; oval in menu bar just below names of worksheets.</td>
<td></td>
</tr>
<tr>
<td>Switch entry to Text mode</td>
<td>Insert→Text</td>
<td>control-T (command-T)</td>
</tr>
<tr>
<td></td>
<td>Click on &quot;Text&quot; oval in menu bar just below names of worksheets.</td>
<td></td>
</tr>
<tr>
<td>Use a keyboard shortcut in Math mode</td>
<td></td>
<td>Type the shortcut, then hit the escape key. For example, typing omega and then escape will turn the text into $\omega$. Typing e and then escape will allow you to turn the text into the symbolic constant e without needing the Expression Palette.</td>
</tr>
</tbody>
</table>
Chapter 4 Assignment

1. Chapter Overview

We learn how to label results with symbolic names through the operation of "assign to a name", sometimes called assignment. This allows us to reuse the results in subsequent steps of a multi-step calculation without retyping it.

The keyboard operation := provides a keyboard shortcut for assignment. := will be used heavily in later work in programming as we shift from mouse/menu operation to textual specification of calculations.

2. Assignment: remembering results for future use

We can compute a result and label it with name. This action is called assignment. We can do this with the clickable menu by the action right-click (control-click on Macintosh) → assign to a name. A pop-up menu will appear asking us to fill in the name that we want to use.

Once we have assigned a result to a name, we can use the name, and Maple will use the assigned value.

A name can be any sequence of upper- or lower-case letters, digits and the underscore character _. It must start with a letter. Maple distinguishes between upper and lower case letters, so result and Result are considered different names.

In programming, the term variable is used interchangeably with name. Both refer to an identifier which the action of assignment associates with a computed result. Computer books often talk about "assigning the result to a variable" which means the same thing as "assigning the result to a name".

Table 4.1: Assignment

<table>
<thead>
<tr>
<th>Expression</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sin \left( \frac{\pi}{4} \right) + 1 )</td>
<td>( \frac{1}{2} \sqrt{2} + 1 )</td>
<td>assign to a name trigResult (4.1)</td>
</tr>
<tr>
<td>( (\text{trigResult} + 1) \cdot (\text{trigResult} - 1) )</td>
<td>( \frac{1}{2} \left( \frac{1}{2} \sqrt{2} + 2 \right) \sqrt{2} )</td>
<td>at 5 digits 1.9142 (4.4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>After computing a result, we assign it to the name trigResult by clicking on the result, selecting the menu item assign to a name and then typing in the name. (4.2)</td>
</tr>
</tbody>
</table>

We can then use the name instead of repeatedly entering or copying expressions.
4.3 Assignment keyboard shortcut :=

The Maple operator := (colon immediately followed by an equals) also performs assignment.

The general form of the assignment operation when using the keyboard is

\[ name := expression \]

Assigns the name \( name \) the value \( expression \)

<table>
<thead>
<tr>
<th>Form</th>
<th>Examples</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( name := expression whose value will be assigned )</td>
<td>( x := 5 )</td>
<td>Assigns the name ( x ) the value 5.</td>
</tr>
<tr>
<td>( poly := z + \frac{5}{2} + \sin \left( \frac{\pi}{3} \right) )</td>
<td>Assigns the name ( poly ) the value consisting of ( z ), ( \frac{5}{2} ), ( \sin \left( \frac{\pi}{3} \right) ) and whatever the assigned value of ( z ) is. If no value has been assigned the name ( z ), then the result is the algebraic formula: ( z + \frac{5}{2} + \sin \left( \frac{\pi}{3} \right) ).</td>
<td></td>
</tr>
</tbody>
</table>

When you enter an expression in Math mode (even if it's just a name, without any arithmetic), you ask Maple to evaluate what you have entered. Maple also (before it does anything else) figures out the values assigned to names that appear in the expression. Then it does arithmetic, function calculation, and any other operations you've described in the expression.

If you use a name/variable in an expression, and it has no assigned value, then Maple uses the rule that the value of an name with no assigned value is just the name itself.

Table 4.2: Assignment

<table>
<thead>
<tr>
<th>Examples of assignment with :=</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p := x^2 + x + ax + 5 )</td>
<td>We assign the name ( p ) the value of the expression. Note that since ( x ) and ( a ) have not been assigned values, the results of evaluation just leaves them as symbols.</td>
</tr>
<tr>
<td>( x := 3 )</td>
<td>Here we assign the name ( x ) the value 3.</td>
</tr>
<tr>
<td>( p + 1 )</td>
<td>If we enter an expression containing ( p ), its value is plugged in for the calculation of the result.</td>
</tr>
<tr>
<td>( p )</td>
<td>If we now do a calculation with ( p ), the value of ( x ) is used since ( p )'s value mentions ( x ). There may be a chain of assignments that Maple must look at to evaluate an expression.</td>
</tr>
<tr>
<td>( p )</td>
<td>We can solve the result 1.3.4 for ( a ) by right clicking that expression.</td>
</tr>
<tr>
<td>( a = \frac{-17}{3} )</td>
<td>We change the value of ( x ) by assigning it a different value.</td>
</tr>
<tr>
<td>( x := 4 )</td>
<td>When we do another calculation with ( p ), the most recent assigned value of ( x ) is used.</td>
</tr>
<tr>
<td>( p )</td>
<td></td>
</tr>
<tr>
<td>( 25 + 4a )</td>
<td></td>
</tr>
</tbody>
</table>
Examples of assignment with :=

<table>
<thead>
<tr>
<th><img src="image1.png" alt="Picture" /></th>
<th><img src="image2.png" alt="Picture" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>solve</td>
<td>(4.12)</td>
</tr>
<tr>
<td>[ a = - \frac{25}{4} ]</td>
<td>Commentary</td>
</tr>
</tbody>
</table>
| y := x | This a way of assigning 4 -- the current value of \( x \) to the name \( y \).

<table>
<thead>
<tr>
<th><img src="image3.png" alt="Picture" /></th>
<th><img src="image4.png" alt="Picture" /></th>
</tr>
</thead>
</table>
| unassign('x') | We can undo the connection between \( x \) and any value by unassigning \( x \). This operation produces no output, so no label. We can barely tell that it has happened. The quote marks surrounding the \( x \) -- 'x' are mandatory, otherwise \( x \) would be replaced by its value and Maple would try to unassign the corresponding value rather than \( x \) itself.

\( p \)

\[ x^2 + x + ax + 5 \] | (4.14) |

\( p \) still has a value, but since \( x \) no longer has a value, we are back to the original result.

<table>
<thead>
<tr>
<th><img src="image5.png" alt="Picture" /></th>
<th><img src="image6.png" alt="Picture" /></th>
</tr>
</thead>
</table>
| y | We may have unassigned \( x \), but \( y \)'s unassigned value -- 4 -- is unaffected. The assignment just connects the name and the value determined when the assignment was performed (back at \( 4.13 \)). The information about which variables or expressions were used to figure out what the value was is not retained.

<table>
<thead>
<tr>
<th><img src="image7.png" alt="Picture" /></th>
<th><img src="image8.png" alt="Picture" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>(4.15)</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

4.4 How to think about assignment: a mental model

The operation of assignment uses part of the computer's memory to remember the association of the name with the result. A useful mental model of assignment is to think of the computer creating a memory slot containing the result, labeled with the name being assigned to.

Table 4.3: Mental model of assignment

<table>
<thead>
<tr>
<th><img src="image9.png" alt="Picture" /></th>
<th><img src="image10.png" alt="Picture" /></th>
</tr>
</thead>
</table>
| > \( p := x^2 + x + ax + 5 \) | The mental model has a slot labeled \( p \) with the value \( x^2 + x + \alpha x + 5 \):

\[ x^2 + x + ax + 5 \] | (4.16) |

\( p \)

\[ x^2 + x + \alpha x + 5 \]

<table>
<thead>
<tr>
<th><img src="image11.png" alt="Picture" /></th>
<th><img src="image12.png" alt="Picture" /></th>
</tr>
</thead>
</table>
| x := 3 | If we then assign \( x \) a value, then there are two slots created. The second assignment only works on the slot associated with \( x \).

\[ 3 \] | (4.17) |
If we assign $p$ another value, then the memory slot associated with $p$ is cleared out and replaced with the new value. Note that the act of computing the new value for $p$ causes Maple to use any assigned values for $x$ that currently exist. This is different than with calculation (1.2.5). At that time, $x$ had no assigned value.

\[ p := y^2 + x + a \cdot x + 5 \]

If we unassign a variable, then we can think of the slot as being deleted from the computer's memory. Unassigning $x$ does not unassign $p$ or change $p$'s assigned value.

\[ \text{unassign}(x) \]

\[ p \\
\begin{aligned} y^2 + 8 + 3a 
\end{aligned} \]
4.5 The state of the Maple session and the look of the worksheet

When you first start up Maple with a blank worksheet, you haven't done any assignments. Thus it isn't surprising that a blank worksheet has no assigned variables. Using the mental model of the previous section, Maple has not allocated any memory to remember things -- no slots, no associations between results and names.

The state of a Maple session consists of all the variables that are currently assigned, and what their values are. The mental model of assignments is exactly the state of the session. The state changes every time we do another assignment or unassignment.

Only an assignment operation can change the state of the session. We know that we can jump back and execute a line in the worksheet a second time, just by positioning the cursor there and hitting enter (return). This raises the possibility that the way the worksheet looks is not an accurate reflection of the state of the session -- variables may have different values than what you'd think from reading the worksheet from top to bottom.

If you are not sure what the current value of a variable is, you can find out what it is by entering the name of the variable and hitting enter.

A saved worksheet does not save the state of the session (the variable assignments). Opening a saved worksheet file does not cause it to automatically execute the operations in the worksheet. This gives you a chance to edit the worksheet and possibly change the calculations specified, before carrying out the instructions.

Table 4.4: Example: The state of a Maple Session

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p := x^2 + x + ax + 5 )</td>
<td>When we first load a worksheet, the state of the session is blank. For instance, we may see an equation like this in our worksheet, where it appears that the variable ( p ) stores the function ( x^2 + x + ax + 5 ). However, the variable ( p ) does not store anything yet!</td>
</tr>
<tr>
<td>( x^2 + x + ax + 5 )</td>
<td>(4.20)</td>
</tr>
<tr>
<td>( y := 2:p )</td>
<td>This is evident if we immediately try to use the function ( p ) in another equation. Although we might expect ( y ) to equal ( 2 \cdot x^2 + 2 \cdot x + 2 \cdot ax + 10 ), it actually equals ( 2 \cdot p ). This is because we have just loaded the worksheet, and ( p ) is currently unassigned.</td>
</tr>
<tr>
<td>( 2p )</td>
<td>(4.21)</td>
</tr>
<tr>
<td>( p := x^2 + x + ax + 5 )</td>
<td>To make ( y ) equal to ( 2 \cdot x^2 + 2 \cdot x + 2 \cdot ax + 10 ), we need to go back to the line where we assigned ( p ) and hit enter. Then we can go back to where we assigned ( y ) and hit enter to get the expected result.</td>
</tr>
<tr>
<td>( x^2 + x + ax + 5 )</td>
<td>(4.22)</td>
</tr>
<tr>
<td>( y := 2:p )</td>
<td></td>
</tr>
<tr>
<td>( 2x^2 + 2x + 2ax + 10 )</td>
<td>(4.23)</td>
</tr>
</tbody>
</table>
4.6 restart causes all assignments to be forgotten

We've seen that it's possible to erase a particular assignment using unassign. If we want to forget all assignments we've made so far, then we can use restart. This can be useful in situations where you've done some work and made some assignments, but now want to switch to working on a different problem and would like Maple to forget about the assignments you made before. It is generally a good idea to restart at the beginning of unrelated sections just in case variables were previously assigned values that might not be related to their use in the new section.

Table 4.5: Evaluation of expressions involving assigned variables

<table>
<thead>
<tr>
<th>Examples of assignment with :=</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p := x^2 + x + a\cdot x + 5 )</td>
<td>( 35 + 5a ) ( \text{(4.24)} )</td>
</tr>
<tr>
<td>restart</td>
<td>This has the effect of wiping out all assignments from the state of Maple session. The mental model has no assignments in it at this point.</td>
</tr>
<tr>
<td>( p := x^2 + x + a\cdot x + 5 )</td>
<td>( x^2 + x + ax + 5 ) ( \text{(4.25)} )</td>
</tr>
</tbody>
</table>

4.7 Evaluation and assignment

Assignment really requires two steps. The first is figuring out the result. The second is assigning the result to the name. The "figuring out the result" step is called evaluation.

Evaluation in Maple proceeds in two phases. The first is to determine if any of the symbols in the expression being evaluated have an assigned value. If so, those values are used. If those values involve other symbols, those are in turn checked for values, etc.

Symbols without an assigned value have their own names as their value. This allows you to enter an expression such as \( x^2 + 2\cdot x + 5 \) in \( x \) and use the \( x \)'s as symbols in the normal mathematical style as long as you don't assign \( x \) a value.

Table 4.6: Evaluation of expressions involving assigned variables

<table>
<thead>
<tr>
<th>Examples of assignment with :=</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart</td>
<td>We start fresh through restart.</td>
</tr>
<tr>
<td>( p := x^2 + x + a\cdot x + 5 )</td>
<td>( x^2 + x + ax + 5 ) ( \text{(4.26)} )</td>
</tr>
<tr>
<td>( x := 3 )</td>
<td>( 3 ) ( \text{(4.27)} )</td>
</tr>
<tr>
<td>( p )</td>
<td>( 17 + 3a ) ( \text{(4.28)} )</td>
</tr>
</tbody>
</table>
If we change the assigned value of $x$ and evaluate $p$ again, this time 4 is used everywhere in the expression $x^2 + x + a \cdot x + 5$. At this point, the mental model of the state of the Maple session is

\[ p \quad x^2 + x + a \cdot x + 5 \]

Assigning a a value also changes the result of evaluating $p$. The assigned value of $p$ hasn't changed, but the result of evaluating $p$ takes into account that $a$ now has a value. After \( 1.7.12 \) the mental model of the state of the Maple session is:

\[ p \quad x^2 + x + a \cdot x + 5 \quad a \quad y \]

Changing the value of $a$ causes a different result when evaluating $p$ again, but doesn't change the result of evaluating result2. This is because result2 does not change with the assignment to a done in \( 4.34 \). Its value is still $25 + 4 \cdot y$. The mental model of the state of the session after the operations \( 4.26 \) through \( 4.36 \) are done is

\[ p \quad x^2 + x + a \cdot x + 5 \quad a \quad z+1 \]

\[ x \quad 4 \quad result2 \quad 25 + 4 \cdot y \]
4.8 Troubleshooting assignments

Equations are not the same as assignment

Assignment is an operation that many programming languages have. In some languages (e.g. Maple, Pascal, Eiffel) := is used for the assignment operation. In others (C, Java, Matlab) = is used as the symbol for assignment. Maple uses := because it uses = for equations. It would be confusing to computers and to human readers to use the same symbol for two common but different operations in a single language.

<table>
<thead>
<tr>
<th>= and := mean different things in Maple</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>$a := 3$</td>
<td>We assign $a$ the value 3.</td>
</tr>
<tr>
<td></td>
<td>(4.37)</td>
</tr>
<tr>
<td>$x = 4$</td>
<td>This is an equation. It doesn't assign $x$ any value.</td>
</tr>
<tr>
<td></td>
<td>(4.38)</td>
</tr>
<tr>
<td>$p := a + x$</td>
<td>We assign $p$ the value of the expression $a + x$. $a$ stands for the value 3 at this point since we did an assignment to it. $x$ is just a symbol that has no assigned value.</td>
</tr>
<tr>
<td></td>
<td>(4.39)</td>
</tr>
<tr>
<td>$x := 47$</td>
<td>We can do an assignment to $x$.</td>
</tr>
<tr>
<td></td>
<td>(4.40)</td>
</tr>
<tr>
<td>$p := a + x$</td>
<td>This time $p$'s value is $3 + 47 = 50$.</td>
</tr>
<tr>
<td></td>
<td>(4.41)</td>
</tr>
</tbody>
</table>

The name to be assigned always goes on the left hand side of the :=

Since $5 = x$ and $x = 5$ mean the same thing as mathematical equations, some people think that this should mean that $x := 5$ and $5 := x$ should both assign the value 5 to $x$. However, only $x := 5$ does the assignment.

Assignment := is not symmetric. It matters which side the name is on

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5 := x$</td>
<td>This doesn't mean anything to Maple. The name is supposed to be on the left hand side.</td>
</tr>
<tr>
<td>Error, illegal use of an object as a name</td>
<td></td>
</tr>
<tr>
<td>$x := 5$</td>
<td>This assigns $x$ the value 5.</td>
</tr>
<tr>
<td></td>
<td>(4.43)</td>
</tr>
<tr>
<td>$z := y$</td>
<td>This assigns $z$ the (symbolic) value $y$. It doesn't assign $y$ any value.</td>
</tr>
<tr>
<td></td>
<td>(4.44)</td>
</tr>
<tr>
<td>$z + (z + 1)^2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4.45)</td>
</tr>
</tbody>
</table>
Sometimes you want Maple to forget all the assignments you have made in a session. You can get this to happen either by using \texttt{unassign} on each assigned name, or by entering \texttt{restart} in Math mode and then hitting enter. This will unassign everything, undoing all the assignments.

\texttt{restart} does not erase the worksheet, however. The worksheet still looks the same, including the written record of the assignments you had previously done. What the \texttt{restart} does is to delete all the slots you have set up in your mental model.

\textbf{A subsequent assignment to the same name/variable undoes the previous assignment}

Every name/variable can be assigned at most one value at a time. It is permissible to assign a name several times during a sequence of operations, but each assignment replaces the previous association. While the document will record each assignment, only the most recently performed assignment will be in effect if you use a name after it is assigned.

\begin{table}
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Example} & \textbf{Commentary} \\
\hline
\texttt{a := 3} & This example perform a series of assignments. Its purpose is to demonstrate the effect of reassignment and unassignment. First, we assign a the value 3. \\
& \texttt{(4.47)} \\
& 3 & \texttt{(4.47)} \\
\hline
\texttt{x := 4 + a} & We assign x the value that's 4 plus the value of a. \\
& \texttt{(4.48)} \\
& 7 & \texttt{(4.48)} \\
\hline
\texttt{a := 5} & We assign p the value of the expression \texttt{a + x}. a stands for the value 3 at this point since we did an assignment to it. x is just a symbol that has no assigned value. \\
& \texttt{(4.49)} \\
& 5 & \texttt{(4.49)} \\
\hline
\texttt{x} & If we ask for the value of x we still get 7, which is what it was set to after the last assignment of x at \texttt{(1.8.7)}. Maple does not go back and come up with a new value of x just because a has been assigned subsequent to \texttt{(1.8.7)}. \\
& \texttt{(4.50)} \\
& 7 & \texttt{(4.50)} \\
\hline
\texttt{y := x + a + 1 + z} & We can do an assignment to y. The expression \texttt{x + a + 1} evaluates to the currently assigned value of x which is 7, plus the most recently assigned value of a which is 5, plus 1, plus the most recently assigned value of z. Since z has no assigned value, it is treated as an algebra symbol and left as the symbol z. \\
& \texttt{(4.51)} \\
& 13 + z & \texttt{(4.51)} \\
\hline
\texttt{unassign}(\texttt{a}) & If we unassign a, the previously assigned value is forgotten. But that does not cause the previously assigned values of y and x to be forgotten. \\
\hline
\end{tabular}
\caption{Reassignment undoes the effect of previous assignment}
\end{table}
Some names are already used by Maple. You will get an error message if you try to assign to them yourself.

When you first start up Maple, the names that you would ordinarily think of using to assign to are not assigned. However a few are, such as the symbolic constants \( \pi \) and \( i \). Maple will tell you that such names are reserved for system use. You need to pick another name.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi := 47 )</td>
<td>( \pi ) is a symbolic constant.</td>
</tr>
<tr>
<td>Error, attempting to assign to ( \pi ) which is protected</td>
<td></td>
</tr>
<tr>
<td>( \text{for} := 3.1 )</td>
<td>( \text{for} ) isn't a symbolic constant but it is used in Maple's programming language. So we can't use it as a variable. We can tell that something funny is going on because the ( \text{for} ) is automatically turned into a bold ( \text{for} ), and a red box appears around the initial part of what we typed.</td>
</tr>
<tr>
<td>Error, controlling variable of for loop must be a name</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.8: Maple won't let you use some names it is already using</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>( \text{for} := 3.1 )</td>
</tr>
<tr>
<td>Error, controlling variable of for loop must be a name</td>
</tr>
<tr>
<td>( x := x + 1 )</td>
</tr>
<tr>
<td>Error, attempting to assign to ( x ) which is protected</td>
</tr>
</tbody>
</table>

### 4.9 Summary of Chapter 4 material

#### Assignment

<table>
<thead>
<tr>
<th>General form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment is performed by using the <strong>assign to a name</strong> operation of the clickable menu.</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>( 47 ) (4.56)</td>
</tr>
<tr>
<td></td>
<td>( \text{assign to name} )</td>
</tr>
<tr>
<td></td>
<td>( x ) (4.57)</td>
</tr>
<tr>
<td>Assignment also be performed by typing in the name, followed by ( := ), followed by the expression whose value will be the result to be assigned.</td>
<td>( x := 5 )</td>
</tr>
<tr>
<td></td>
<td>5 (4.58)</td>
</tr>
<tr>
<td>( y := z + \frac{x^2}{2} )</td>
<td>( z + \frac{25}{2} ) (4.59)</td>
</tr>
</tbody>
</table>

#### Unassignment

<table>
<thead>
<tr>
<th>General form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>unassign('symbol name')</td>
<td>( x + 1 )</td>
</tr>
<tr>
<td></td>
<td>6 (4.60)</td>
</tr>
<tr>
<td></td>
<td>( \text{unassign}(x') )</td>
</tr>
<tr>
<td></td>
<td>( x + 1 ) (4.61)</td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
</tbody>
</table>

Undoes all assignments made by the user in the session so far.
5 Chapter 5 Building scripts

5.1 Chapter Overview

We briefly discuss a few extra concepts useful with \texttt{solve}: how to use a combination of relations rather than just a single equation, and how to take apart or combine by the various forms of \texttt{solve}.

We then explore the concept of script: a sequence of operations useful for solving a problem. We find that it's often the case that the need for a computation is driven by its reuse -- doing the same thing but with slight alterations each time. A frequently recurring scenario is a parameterized computation: 1) use variables to assign values to the parameters and 2) have subsequent steps of the computation refer to the parametric variables. Maple is well-equipped for reuse of parameterized scripts, since it has an operation \texttt{Edit \rightarrow Execute \rightarrow Selection} or \texttt{Worksheet}. This makes it easy to solve different versions of a problem by editing the parameter values and re-executing the script.

5.2 The structure of information in Maple: getting information from solve

The result of the \texttt{solve} operation can have multiple parts if there are multiple solutions to the equation. In this case, the result of solve is a sequence, list, or set of solutions, and we can select each part by giving an index (either 1 or 2).

\[ eq1 := 3 \cdot x = x^2 - 28 \rightarrow \texttt{solve} \rightarrow \{ x = -4, \{ x = 7 \} \} \rightarrow \texttt{select entry 1} \rightarrow \{ x = -4 \} \]

\[ eq1 := 3 \cdot x = x^2 - 28 \rightarrow \texttt{solve} \rightarrow \{ x = -4, \{ x = 7 \} \} \rightarrow \texttt{select entry 2} \rightarrow \{ x = 7 \} \]

If we give \texttt{solve} a linear equation, it has only one solution. We can still select the first entry.

\[ eq2 := 3 \cdot x = 28 \rightarrow \texttt{solve} \rightarrow \left\{ x = \frac{28}{3} \right\} \rightarrow \texttt{select entry 1} \rightarrow \frac{28}{3} \]

If we do "solve for x" for the same equation, we see that the answer comes back in a slightly different form. But it still has parts.

\[ eq1L = 3 \cdot x = x^2 - 28 \rightarrow \texttt{solve for x} \rightarrow \{ [x = -4], [x = 7] \} \rightarrow \texttt{select entry 2} \rightarrow [x = 7] \]

Maple (as well as many other programming languages) can compute with objects that have \textit{structure}. Here are four different kinds of structures that Maple can handle:

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Type of structure} & \textbf{What they look like} & \textbf{Examples} \\
\hline
Equations, inequalities & What math books do for equations and inequalities. The clickable menu allows you to get at the \textbf{left hand side} and \textbf{right hand side} of an equation or inequality. & \( x + y = 35 \) \\
\hline
& & \( x + y = 35 \) \hspace{1cm} (5.3) \\
& \texttt{left hand side} & \( x + y \) \hspace{1cm} (5.4) \\
& \texttt{x} \geq \sqrt{57} & \sqrt{57} \leq x \hspace{1cm} (5.5) \\
& \texttt{right hand side} & x \hspace{1cm} (5.6) \\
\hline
\end{tabular}
\end{table}
<table>
<thead>
<tr>
<th>Type of structure</th>
<th>What they look like</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Sequences         | Values or expressions separated by a comma | $x := 19, 47, 92$
|                   |                    | $19, 47, 92$ \hspace{1cm} (5.7) |
|                   |                    | $relations := x^2 - 2 = 0, 0 < x$
|                   |                    | $x^2 - 2 = 0, 0 < x$ \hspace{1cm} (5.8) |
|                   |                    | select entry 1
|                   |                    | $x^2 - 2 = 0$ \hspace{1cm} (5.9) |
| Sets              | A sequence surrounded by curly braces \{ \} | $Scores := \{3, 7, 3, 10\}$
|                   |                    | $(3, 7, 10)$ \hspace{1cm} (5.10) |
|                   |                    | select entry 3
|                   |                    | 10 \hspace{1cm} (5.11) |
| Lists             | A sequence surrounded by square brackets \[ \] | $MyEquations := \{x + y = 3, y - 2 \cdot x = 37\}$
|                   |                    | $[x + y = 3, y - 2x = 37]$ \hspace{1cm} (5.12) |
|                   |                    | solve
|                   |                    | $\{x = \frac{34}{3}, y = \frac{43}{3}\}$ \hspace{1cm} (5.13) |

Note that the result of this solve is a set of equations.

For the time being, we just want you to recognize the different kinds of structures that are output by solve and other functions and be able to select parts from them. Later on we will get a lot of work done by performing more sophisticated operations with them.

### 5.3 Finding simultaneous solutions, constraining solutions.

Suppose we want to solve the system of equations $x + y = 5$ and $-3 \cdot y + 7 = x$. This means finding values of $x$ and $y$ that simultaneous satisfy both equations. We can do this in Maple by typing in the first equation and then the second, separated by a comma. This is called entering a sequence of equations. Right-clicking (control-click on Macintosh) on the sequence will allow you to solve the system.

In Lab 1, you discovered that solve could also handle inequalities as well as equalities. You can enter a sequence of equations and inequalities to solve. This can be used to limit solutions to a particular range of values.

#### Table 5.2: Solving simultaneous equations

<table>
<thead>
<tr>
<th>Equation</th>
<th>The result of this solve is a set of solutions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y = 5, -3 \cdot y + 7 = x$</td>
<td>\hspace{1cm} (5.14)</td>
</tr>
<tr>
<td>solve \hspace{1cm}</td>
<td>\hspace{1cm} (5.15)</td>
</tr>
<tr>
<td>$(x = 4, y = 1)$</td>
<td>\hspace{1cm} (5.15)</td>
</tr>
</tbody>
</table>
Solving this equation produces 4 roots. Two of them are complex numbers (since they have $i$ in them) the others are real.

\[
p := x^4 + 3x^2 - 57.5 = 0
\]
\[
solve \quad x^4 + 3x^2 - 57.5 = 0
\]
\[
(x = 3.0380606341), \ (x = -3.0380606341), \ (x = 2.495959218), \ (x = -2.495959218)
\]

This gets the solver to list only the positive real solutions. We might be interested in only non-negative real solutions, if, for example, we were using the equation to model a physical situation and $x$ was measuring weight. In this case the other values of $x$ wouldn't be relevant to our situation.

\[
p, x \geq 0
\]
\[
solve \quad x^4 + 3x^2 - 57.5 = 0, \ 0 \leq x
\]
\[
(x = 2.495959218)
\]

5.4 Scripting: creating computational work in reusable form

Consider the problem you did in Lab 1, along with a solution:

**Version 1 and solution**


A breeding group of 20 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, $N$, after $t$ years will be given by the formula:

\[
N = \frac{220}{1 + 10 \cdot (0.83)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 80.

(a) Graph $N$ versus $t$.
(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of $t$.)

**Solution to (a)**

\[
N = \frac{220}{1 + 10 \cdot (0.83)^t} \quad \text{right hand side} \quad 220 = \frac{220}{1 + 10 \cdot 0.83^t} \quad \text{assign to a name} \quad \text{sheepPopExpr}
\]
Once we have the sheep population, we need to play with the plotting ranges to see when the leveling off occurs. We'd have to think about it and experiment a bit -- but the computer makes the replotting easy to do once we make our decisions about what to try.

\[
\text{sheepPopExpr}
\]

\[
\frac{220}{1 + 10^{0.83t}}
\] (5.22)

Solution to (b)

\[80 = \text{sheepPopExpr}\]

\[
80 = \frac{220}{1 + 10^{0.83t}}
\] (5.23)

solve

\[\{t = 9.354227718\}\] (5.24)

We can read the leveling off point from the plot, assuming that we have figured out the appropriate time range in (b). Alternatively, we can do a little calculus and take the limit of the expression as \(t\) goes to infinity.

\[
\lim_{t \to \infty} \text{sheepPopExpr}
\]

\[220.\] (5.25)
We can imagine ourselves working as an environmental engineer for the National Forest Service and being very pleased with ourselves for solving the problem with Maple. But now we are handed two more wildlife management problems to do, from other regions in our territory:

**Version 2**

A breeding group of 33 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{330}{1 + 10 \cdot (0.79)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 85.

(a) Graph \( N \) versus \( t \).
(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \)).

**Version 3**

A breeding group of 45 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, \( N \), after \( t \) years will be given by the formula:

\[
N = \frac{450}{1 + 10 \cdot (0.63)^t}
\]

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 90.

(a) Graph \( N \) versus \( t \).
(b) How many years must the state of Montana maintain a program to care for the sheep? Show your work.
(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \)).

We have the feeling that we will shortly be handed problems for a number of other locations as well. How can we reuse our original work with minimal effort?

If we had the first solution, we could produce the second solution through *copy-paste-edit-re-execute*:

**Executing a clone of a script through copy-paste**

1. Copy and paste the original solution into a new document, or to a spot later in the same document. You do this as with a word processor -- select a region of the worksheet with the mouse, type control-C (command-C on a Mac). Then position the document cursor in the destination, and type control-V (command-V on a Mac). A duplicate of the solution should appear there.
2. Using the mouse, edit the formulae as needed. You only need to edit the black (input) formulae, not the blue ones.
3. Position the mouse at the first computation and hit enter. Continue to work your way through the sequence of the commands.
4. Alternatively, select the entire region containing the edited version of the solution and hit Edit->Execute->Selection.
5. If the region to be executed is the entire worksheet, then rather than selecting anything you can do Edit -> Execute -> Worksheet.

The results of executing the edited script are not totally correct. We will have to change a few things by hand: the clickable operation "evaluate at a point" in step (b) will use \( N=80 \) (which is what the copy says) instead of \( N=85 \), so we will have to redo it. Also, the last plot will not show an appropriate vertical range unless we do that. We'd like to do a little less of this hand-tuning in re-execution. Also, we'd like to make it easier to remember what to change between multiple versions.

*Abreeding group (page 58).*
5.5 Rewriting the script using assignment

While copying and editing is probably a little faster than typing in the whole script again, we can reduce the amount of hunting around for changes by writing a script finding the parameters of the problem, and writing the script so that it assigns values to the parameters at the beginning of the problem.

You may be doing assignments at several points in your calculation. Only the ones that would need to be changed between different versions of the problem define problem parameters. You may find the other ones very useful, but they don't have parameter status.

Finding and naming parameters

First, solve at least one version of the problem. Then, imagine what would need to be changed if you were trying to solve alternative versions of the problem. You can find parameters if you have several versions of a problem by looking at what changes in the worksheet from version to version.

For example, in the sheep problem, we note the following things changing in different versions of the problem. We pick names for these.

1. the numerator of the "sheep equation" \( P \)
2. the coefficient in the denominator of the equation \( c \)
3. the value of the stable population \( s \)

The other number in the script is the original population of the sheep. We realize that it can be derived by evaluating the "sheep equation" at \( t=0 \). Thus this value does not need to be a parameter as we can derive it from the other information. It might be a good check though.

We then write the script to assign values to the parameters at the start of the script, and then write the other operations and expressions in terms of the symbols.

This allows us to redo the script just by changing the values at the beginning of the script. We use the word processing features to add extra directions to make the script easier to use, basically saying "here, change these things", and "this is the end of where you should stop copying".

The result of executing the script is Version 2, with use of parameters (page 59).

Having created this script, we can handle the third version of the problem by editing the values of the parameters and executing the script again by selecting the entire script with the mouse, and then doing Edit->Execute->Selection. We get this result:

5.6 Summary of script writing

Script writing is appropriate when you expect to handle several different versions of the same problem. In professional work, this is often the case -- if it's worth doing at all, it's probably going to occur more than once.

Figure out how to solve the problem first. Then write the script. There's really not much point in writing the script if you don't have some idea of the sequence of operations in it.

Once you have a worksheet of instructions for solving one version of the problem, look at it and the other versions and find the parameters. Set up a new worksheet where the first thing you do is to assign values to the parameters. Then work through the rest of the instructions and rewrite them to use the parameters instead of the fixed values from one version of the problem.
5.7 Troubleshooting scripts

Programming books have the tendency to show things that work. They say less about what to do about the things that don't work. However, it is usually the case that computer users often spend more time "getting things to work" than "working". Learning how to get out of jams is at least as valuable as knowing how to enter operations and what they mean. We offer this advice, which is "commonsense about work" as applied to interactive software development:

1. Solve one version of the problem before you try to start scripting. You can use Maple to experiment -- enter and edit snippets of operations that try out the solution technique for part of the problem. Eventually edit them together so that they solve the whole problem. If you have only fuzzy notions about the math or the operations you want to proceed, your computer work will just amplify that. Having a worksheet that solves one version of the problem can remove a lot of the fuzziness.

Where does the inspiration for solving the problem come from? If you are lucky, the solution may be told to you. Or you may find a description of a similar problem as a starter. But the big bucks, as they say, go to those who can devise the solution plan themselves.

2. Limit each step so that it is a small step. If you get into trouble, you will be able to nip it at the bud. This becomes a more prominent tactic in later work when we are tempted to construct long-winded one-line expressions that do everything at once.

3. Test pieces individually, then put them together. For example, if you don't really understand how to make a dotted line plot in teal with a title, you should try to do that with a simple plot (such as $x^2$) rather than an expression whose shape you aren't that familiar with it. Then take what works and substitute the real expression you want to plot in a copy of the $x^2$ plot operation.

If you think about it, this is similar to what happened in Fall 2010 ENGR 101 Lab 2, where they first had you learn how to trigger an oscilloscope with the output of the function generator, rather than the output of the camera/flash sensor. The complexity of troubleshooting is reduced if you halve as many unknowns to worry about. This can be called divide and conquer troubleshooting.

4. If what you have doesn't work, find something similar which does work, and then incrementally edit it. For example, if you can't get $a := x^2 + 3 \cdot x + 1$ in, then first see whether you can get $a := 1$ to work. Once you succeed with that, edit the expression to $a := 3 \cdot x + 1$ and so forth.

5. If strange things continue to happen despite your best efforts to troubleshoot, it may be that previous settings in the Maple session are interfering with your current work. Recall that some people in the ENGR 101 lab couldn't get their oscilloscopes to work because of settings changed in the oscilloscopes by groups earlier in the day. This can be particularly true if you are developing scripts and are assigning parameter values, then switch to development of another script in the same document. The values you assigned will not magically unassign themselves when you start working on something new in the same worksheet.

The remedy for this is to put a restart in as the first operation in your script, then re-execute the worksheet.
5.8 Attachments

Attachment: Version 2 of sheep script without parameters

Version 2 of sheep problem, with edited script

A breeding group of 30 bighorn sheep is released in a protected area in Colorado. It is expected that with careful management the number of sheep, $N$, after $t$ years will be given by the formula:

$$N = \frac{330}{1 + 10 \cdot (0.79)^t}$$

and that the sheep population will be able to maintain itself without further supervision once the population reaches a size of 85.

(a) Graph $N$ versus $t$.

(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.

(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of $t$.)

Solution to (a)

$$N = \frac{330}{1 + 10 \cdot (0.79)^t} \quad \text{right hand side} \quad \frac{330}{1 + 10 \cdot 0.79^t} \quad \text{assign to a name} \quad \text{sheepPopExpr}$$

$$\text{sheepPopExpr}$$

$$\frac{330}{1 + 10 \cdot 0.79^t}$$

Solution to (b)
We can read the leveling off point from the plot, assuming that we have figured out the appropriate time range in (b). Alternatively, we can do a little calculus and take the limit of the expression as \( t \) goes to infinity.

\[
\lim_{t \to \infty} \text{sheepPopExpr} = 330.
\]

**Attachment: Version 2 of Sheep Script, with parameters**

**Version 2, with use of parameters**

**Start of parameters -- change these for each version of the problem**

\[
P := 330
\]

\[
330
\]

\[
c := 0.79
\]

\[
0.79
\]

We call the size of the stable population \( s \).

\[
s := 85
\]

\[
85
\]

**End of parameters**

(a) Graph \( N \) versus \( t \).

(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.

(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

(a)

\[
\text{sheepPopExpr} := \frac{P}{1 + 10 \cdot (c)^t}
\]

\[
\frac{450}{1 + 10 \cdot 0.83^t}
\]
Note that this sheepPopExpr is not a parameter since assignment is always the same for all versions of the problem.

To make the graphing work all the time, we set the vertical axis to "P+30" rather than a fixed value. This will set the vertical axis so that it will be 30 more than the value of P which from the first problem we have realized is the top of the graph.

\[ \text{sheepPopExpr} \]

\[
\frac{330}{1 + 10^{0.79t}} \quad (5.35)
\]

This is the time in years that model predicts it will take for the sheep population to reach self-sustaining status, allowing the wildlife managers to move onto another job.

(c) A little thinking reveals that the leveling off value (as indicated from the graph), is the numerator of \text{sheepPopExpr}, since the denominator goes to 1 as \( t \) goes to infinity.

\[
\lim_{t \to \infty} \text{sheepPopExpr} = 330. \quad (5.37)
\]
Attachment: Version 3 of Sheep Script, with parameters

Version 3 with edited parameters and re-execution

Start of parameters -- change these for each version of the problem

\[ P := 450 \]

\[ c := 0.83 \]

\[ \text{sheepEquation} := N = \frac{P}{1 + 10 \cdot (c)^t} \]

\[ N = \frac{450}{1 + 10 \cdot 0.83^t} \]

We call the size of the stable population \( s \).

\[ s := 100 \]

End of parameters

(a) Graph \( N \) versus \( t \).

(b) How many years must the state of Colorado maintain a program to care for the sheep? Show your work.

(c) How many bighorn sheep can the environment in the protected area support? (hint: examine the graph for large values of \( t \).)

(a)

\[ \text{sheepPopExpr} := \frac{P}{1 + 10 \cdot (c)^t} \]

Note that this sheepPopExpr is not a parameter since assignment is always the same for all versions of the problem.

To make the graphing work all the time, we set the vertical axis to "P+30" rather than a fixed value. This will set the vertical axis so that it will be 30 more than the value of P which from the first problem we have realized is the top of the graph.

\[ \text{sheepPopExpr} \]

\[ \frac{450}{1 + 10 \cdot 0.83^t} \]
This is the time in years that model predicts it will take for the sheep population to reach self-sustaining status, allowing the wildlife managers to move onto another job.

(c) A little thinking reveals that the leveling off value (as indicated from the graph), is the numerator of \( \text{sheepPopExpr} \), since the denominator goes to 1 as \( t \) goes to infinity.

\[
\lim_{t \to \infty} \text{sheepPopExpr} = 450.
\]
## 5.9 Summary of Chapter 5 material

<table>
<thead>
<tr>
<th>Type of structure</th>
<th>What they look like</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic data structures in Maple</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equations and inequalities</td>
<td>Expression related by $=$, $&gt;$, $&lt;$, $\geq$, $\leq$, or $\neq$.</td>
<td>( x + y = 0 ) ( \quad ) ( (5.46) ) ( n^2 - 3n &gt; 4 ) ( \quad ) ( (5.47) )</td>
</tr>
<tr>
<td>Sequences</td>
<td>Values separated by a comma</td>
<td>19, 47, 92 ( \quad ) ( (5.48) )</td>
</tr>
<tr>
<td>Lists</td>
<td>A sequence surrounded by square brackets [ ]</td>
<td>( MyData := \quad ) ( \left[ 1.0, x, \frac{3}{4}, a \right] \quad ) ( (5.49) )</td>
</tr>
<tr>
<td>Sets</td>
<td>A sequence surrounded by curly braces { }</td>
<td>( Scores := \quad ) ( { 3, 7, 3, 10 } \quad ) ( (5.50) )</td>
</tr>
<tr>
<td><strong>Solving simultaneous equations</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x + y = 5, -3y + 7 = x )</td>
<td>The result of this <code>solve</code> is a set of solutions.</td>
<td>( x + y = 5, -3y + 7 = x ) ( \quad ) ( (5.51) ) ( { x = 4, y = 1 } ) ( \quad ) ( (5.52) )</td>
</tr>
<tr>
<td>( p := x^4 + 3x^2 - 57.5 = 0 )</td>
<td>Solving this equation produces 4 roots. Two of them are complex numbers (since they have ( i ) in them) the others are real.</td>
<td>( x^4 + 3x^2 - 57.5 = 0 ) ( \quad ) ( (5.53) ) ( { x = 3.0380606341 }, { x = -3.0380606341 }, { x = 2.495959218 }, { x = -2.495959218 } ) ( \quad ) ( (5.54) )</td>
</tr>
<tr>
<td>( p, x \geq 0 )</td>
<td>This gets the solver to list only the positive real solutions. We might be interested in only non-negative real solutions, if, for example, we were using the equation to model a physical situation and ( x ) was measuring weight. In this case the other values of ( x ) wouldn't be relevant to our situation.</td>
<td>( x^4 + 3x^2 - 57.5 = 0, 0 \leq x ) ( \quad ) ( (5.55) ) ( { x = 2.495959218 } ) ( \quad ) ( (5.56) )</td>
</tr>
<tr>
<td>Scripts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating a script for a problem</td>
<td>In a Maple worksheet, take a version of a problem and create a sequence of operations in the worksheet that solve it. Note similarities and differences between different versions of the problem. Envision what you'd have to change in the worksheet in order to solve a different version of the problem, and what would stay the same. You may have to rewrite some of the expressions to refer to the parameter rather than the value. Assign the parameters at the beginning of the script. Rework the rest of the script so that the formulas refer to the parameters by name, rather than the values used in the original version of the problem. For example, if the value 42 appears in several places in your script, define a parameter ( p := 42 ) at the start of the script and edit the other occurrences of 42 to be ( p ) instead. When you have a different version of the problem, you can edit just the single line ( p := 42 ) into say ( p := 47 ) and won't need to edit any other lines of the script.</td>
<td></td>
</tr>
<tr>
<td>Using a script</td>
<td>Copy and paste the script to a new location Edit the assignments to reflect the new version of the problem. Edit-&gt;Execute-&gt;Selection, or just hit enter (return on Macintosh) multiple times to perform the operations in the new version of the script.</td>
<td></td>
</tr>
<tr>
<td>Rationale for using scripts</td>
<td>More work to do than clickable interface the first time. Saves time if you expect to want to reuse the operations on multiple versions of the same problem. Also it is less error prone.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6 More sophisticated scripting

6.1 Chapter Overview

We introduce textual entry of solve and plot operations. This is where the operation is specified by keyboard entry alone, without the use of the mouse or Palettes. Textual entry is often preferred by programmers because it is easier to edit scripts written through textual means. We retain the clickable interface for doing quick one-time calculations, or for developing ideas the first time before we start script-writing.

We being to introduce additional concepts in Maple, to enhance what we can solve and plot:

1. In Maple, a character string is a collection of characters delimited by "s: "This is a string." We see how strings are used in the textual entry of labels and colors in plots.

2. Lists e.g. [1,2, x, 3.5] provide a way of organizing multiple results in a single "data container", making it to operate on the whole collection of results while retaining the ability to getting at individual results from within the collection. Lists are used in both solve and plot.

3. solve uses two other types of data containers: sequences and sets. We describe how to recognize them. and how to extract information from them.

Programmers rely on the on-line documentation to manage the complexity of remembering the details of a knowledge-intensive system such as Maple. They learn/remember how to use a feature by looking up the description, finding an example close to what is desired, and then actively experiment with the example in a fresh worksheet. Reading without experimentation is usually not very productive.

In a previous chapter, we explained assignment and how Maple uses assigned values whenever it sees a name in an expression. We introduce the eval operation, which allows assignments to be done temporarily and immediately forgotten. This can be an attractive alternative if you are concerned about situations where you'd be making many assignments and then undoing them through unassign. eval allows you to evaluate an expression for a particular value of a variable in one line, rather than having to type in the assignment and unassignment as well.

6.2 Textual entry of operations

The textual form of an operation in Maple has the general form:

operationName ( sequence of values )

The operationName can be something like solve or plot. By sequence of values, we mean one or more items, each item separated from the next by a comma. "Sequence" here is the same kind of sequence that was first seen in the previous section on solve (page 52).

Maple will evaluate what you enter in the same way that was described for mathematical expressions possible (page 35). If there are assigned variables mentioned in the sequence, then their values will be used. If Maple knows the operationName (e.g. sin, solve, plot), then it will perform the calculation specified by the built-in programming. Otherwise, the result will be more or less what you typed in.

The technical term for the "values" in this situation is actual parameter or argument. Note that the parentheses around the sequence of values are mandatory -- you will either get an error or a result that's far from what you want if you omit the parentheses.

This style of writing things is sometimes called functional notation. In mathematics examples of functional notation are f(x) or g(3,5). In these examples, the name of the operation is the function name f or g, while the actual parameters or arguments would be x or the sequence 3,5.

Example of textual form of equation solver solve.
The form of the answer returned by the textual version is slightly different from invoking solve through the clickable interface. The former is typically a form that is easier to work with in scripts.

In this, the *operation Name* (or function name) is `solve`. There are two arguments. The first argument is the equation \( x = 3 \cdot x^2 - 2 \). The second argument is the symbol \( x \).

The form of the answer returned by the textual version is slightly different from invoking solve through the clickable interface. The former is typically a form that is easier to work with in scripts.

What you get if you leave out the parentheses -- not much! Because you left out the parentheses, Maple does not think you are asking for any function to be evaluated, which is how the work gets done.

You get another one of those "unable to match delimiters" messages if you forget one of the parentheses.

This is what happens if we forget to use the right arrow key to descend from the exponent of \( x^2 \). Maple thinks that we are talking about \( x \) to a power called \( 2-2,x \), which understandably it doesn't make sense to it as a power.

This is a textual form of plot. The first argument is an expression, the second argument is an equation naming a variable and a range of a plot. Note that if we wanted to change the range from \(-3..3\) to \(-5..2\) then we would just edit that line of the worksheet and hit *enter* again. If we wanted to redo the plot in the clickable interface, we would have to right-click and enter all the information all over again.
If the second argument is just the variable, then `plot` uses default values for the range.

No plotting happens if we forget the mandatory parentheses.

Suppose we entered this instead of `plot(x - 3 \cdot x^2 - 2, x = -3.3)`. There is no error message, but the picture is not at all the same. This example illustrates the fact that just because there is no error message, it does not mean that the computer will produce what you thought you were asking for. The only way that we would discover the mistake is if we already had an idea of what the graph should look like, and noticed that the result differed significantly from what we expected. If you haven't formed a basis for expectations, you won't discover the problem.

Once you realize that the picture must be incorrect, you would be spurred to search for the cause. Since there are no error messages and a plot (albeit a weird one) was produced, the most likely cause is that the arguments to the plot function are wrong. The obvious place to look for correct examples is the on-line documentation. If you look at the on-line documentation for help, you will see that giving `plot` three arguments means something different -- the third argument can be taken as the value of the vertical range of a plot. Evidently what is happening is that you are seeing only the tiniest top slice of the plot produced because of the inadvertent specification of the vertical range.

An attachment at the end of the chapter shows the textual form of common functions, subscripts. These textual forms can be entered from the keyboard wherever the palette entry would work.
6.3 Why are there two different styles for entering operations?

The clickable interface is a good way to get a calculation done quickly, but the actions specified in this way are hard to edit when building scripts. Maple, like most languages, has a textual version of all operations it performs. The editing involved in scripting development can often be easier to do on the textual version. In other words, the clickable interface is good for a one-time calculation, but not so good for the editing and re-execution involved in script reuse.

Another advantage of the textual mode of operation is that the number and variety of operations available in textual form is far greater than what's available in the clickable interface. Maple has several thousand operations. Building a clickable interface to all of them would result in tedious navigation through menus that would either be huge or involve many sublevels.

The downside of using the textual entry is that the developer must spell the text correctly, with the right number and placement of parentheses. Experienced users find that the textual interface is faster to deal with for scripting, while the clickable interface is faster for short, more casual use. Fortunately, in either case one can edit failed attempts and retry, so perfect entry is not necessary to be productive.

Becoming proficient with textual entry of operations is part of the transition technical users make in going from just reuse of other's work to routinely creating their own programming. Without such proficiency, it is hard to realize the full power of the computer in modeling and simulation situations.

6.4 Plotting a list of expressions (multi-plots), plotting lists of numbers (point plots)

Recall that lists in Maple are a way of collecting expressions together into a single object, as discussed in the previous table describing lists and other data containers (page 52). You specify a list by listing the items in the list, enclosed in square brackets [ ].

If the first argument to plot is a list of expressions, then plot will on a single graph display the plots of all the expressions in the collection. Each one will be displayed in a different color.
Table 6.1: Plotting of multiple expressions

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot([x^2, sin(x^2)], x = 0..4)</code></td>
<td>Plot two expressions on a range where both have comparable-sized results. Here we use the list <code>[x^2, sin(x^2)]</code> to indicate the two expressions that should be plotted. Problem: approximately where is the expression <code>-3*x^2 + x - 2</code> equal to 5? While we could use solve to tell us exactly, it's often worthwhile to draw a picture and process the situation visually. If we give plot the two expressions <code>-5</code> and <code>-3*x^2 + x - 2</code> then plot will plot not only the parabola (the second expression, it will also plot the expression that is always -5 for any value of x. This corresponds to the horizontal line drawn on the plot. Visually we can see that the parabola is -5 at roughly -0.8 and 1.2. We can even get a little more precise by ???.</td>
</tr>
</tbody>
</table>

Plotting multiple expressions simultaneously can be useful when you want to compare them. Assuming that the scales are comparable, one can get a sense of similarity or dissimilarity "at a glance".
We can plot data points rather than smooth curves, if we give the textual form of plot separate lists of \( x \) and \( y \) coordinates. can be used with the textual version of \textit{plot}. If we give the textual version of the \textit{plot} operation two lists of numbers that have the same length, then \textit{plot} will regard the first list as a list of \textit{x-coordinates}, and the second list as corresponding \textit{y-coordinates}. If you provide \textit{plot} with the third argument \textit{style=point}, then it will produce a point plot. Otherwise, it will draw lines connecting each point.

**Table 6.2: Plotting points with lists of numbers**

<table>
<thead>
<tr>
<th>( xList ) := [ 1, 2, 3, 4]</th>
<th>We use the textual form of \textit{plot} to plot the points (1,5), (2,6), (3,7) and (4,-1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3, 4]]</td>
<td>(6.6)</td>
</tr>
<tr>
<td>( yList ) := [ 5, 6, 7, -1]</td>
<td>(6.7)</td>
</tr>
<tr>
<td>[5, 6, 7, -1]]</td>
<td></td>
</tr>
<tr>
<td>\textit{plot}(xList, yList, style = point)</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing points (1,5), (2,6), (3,7) and (4,-1) connected by lines or points depending on style.](image-url)
Without the third argument, plot will try to connect the points with a curve.

Maple cares about whether things are capitalized or not. Style is not the same as style.

6.5 Strings -- a way to specify titles and labels in plots

A string in Maple is something enclosed in double-quotes: "red", "this is a string?", "Blink++++++182+++++", are all strings. The double-quote symbol is mandatory for a string. Single-quotes (also known as apostrophes), backquotes (also known as acute accent marks) are not substitutes for double-quotes in writing Maple strings. Characters enclosed by apostrophes or backquotes mean something different to Maple. Use of the wrong punctuation marks will lead to undesired results or error messages.

In addition to being used to input data points in plot, lists also can be used in specifying colors and axis labels.
Table 6.3: Plot options and labels

If one of the arguments to the \textit{plot} operation is of the form \texttt{color = list of color names} then those colors will be used. Most reasonable names will work, but the full list can be seen in on line help (search for \texttt{colornames}). Note that we are using the textual version of the symbolic constant $\pi$. If you can remember how to spell it, it can be easier than selecting it from the Common Symbols Palette.

Note that because the expressions being plotted are given in a set, the color assignment is not in the same order that they were typed in. If we wanted the same order, we should give the plot expressions in a list.

Forgetting brackets for the list of colors.

One of the proficiency issues with textual input is that you have to remember all the ( ) parentheses and [ ] brackets. Can you find the the missing delimiter(s)?

If you leave enough delimiters out, you get error messages that don't complain about missing delimiters. You have to figure out what the problem is, which might involve a missing parentheses even if the message doesn't say so.

The "Error, (in sin)" is a cue that you should look at the places where you included \texttt{sin} in your text and inspect it for problems. It doesn't take too much effort for you to notice that there's no finishing parentheses in the first \texttt{sin(x)}.

One reason why there was no error message about delimiters is that
there are multiple missing parentheses. Because there are equal numbers of missing left and right parentheses, there was no alarm for missing delimiters.

The problem with this plot is that $\pi$ doesn't mean the same thing to Maple as $Pi$. Maple is case-sensitive. Only $Pi$ means the symbolic math constant having to do with the circumference of a circle.

Error, (in plot) expecting a real constant as range endpoint but received -4*Pi

If one of the arguments to the `plot` operation is of the form `labels = list of axes titles` then those will be used. Each title needs to be a string.

In subsequent work, we will see strings used in other situations within Maple other than for plot titles.

### 6.6 Troubleshooting with strings

The most common mistakes with strings is to leave out the delimiting "s, or to use the wrong kind of delimiters. While the similar-looking keyboard characters ' (single quote or apostrophe), and ` (acute accent or backquote) look like would be equivalent, they are use for other purposes in Maple.

<table>
<thead>
<tr>
<th>What happens when you forget to use the &quot; delimiter in strong, or use the wrong character for the delimiter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot(3.5-x^2-2, x = 1..5, labels = [temperature (in degrees C), &quot;pressure in kilopascals&quot;] )</td>
</tr>
</tbody>
</table>
| Forgetting to include "s around one of the titles gives a cryptic error message about an "invalid in". If you were an experienced Maple user, you'd know that `in` is part of the Maple programming language, and should never be flagged in a string. This would be a clue that there's something wrong around where you entered the first label.

The message is not very helpful about telling you how to fix the mistake, though. This unfortunately is typical in most computer programming languages, despite several decades' effort in building systems software to help people program.
Putting the wrong kind of quote -- ` instead of " didn't make a string.
We got the same indication of a problem as before even though the problem is "wrong kind of quote" rather than "no quote".

6.7 Learning through on-line documentation and experimentation

All the options available in the Plot Builder available through the right-click (control-click) interface are also available in the textual version of plot. In fact, there are many additional options and varieties of plotting available. The way to find out what the features are and how to invoke them is to consult the on-line documentation.

We can find out more about the textual forms of plotting by invoking Help -> Maple Help and typing plot into the search field. When we do so, we see the information in the figure below:

Table 6.4: Plot command help
We scroll to the bottom of the page and find an example of this. We are looking for a version of plots where v1 and v2 are lists. We don't see something exactly like that but we do see something with Vectors which are similar. Since the document says this should work for lists or vectors, we take the example and see if we can modify it for our own purposes:

Table 6.5: Examples of plot

![Plot example diagram]

Evidently, the the first list is the values of the $x$ (horizontal) coordinates, and the second list the values of the $y$ (vertical) coordinate. We copy and paste the example into a Maple worksheet and then see if we can get it to work.
According to the documentation, we should be able to get this to work if the first two arguments are lists or vectors. So we edit the example to do lists instead and re-execute the line to see if it works in the same way.
To learn about plot options such as colors and labels, we click on the plot, options item under the search results for plot (see green oval in the figure). Clicking on that item produces this information. We see information about color (with another link to see colors), along with possibilities, for labels, symbols, styles, etc. Again, the way to learn the options is through copying and pasting the examples into a fresh worksheet, getting them to work, and then modifying them to suit your own purposes.
Table 6.8: Plot option help page

Note the "colour". Like Blackberries which are designed just down the road from the Maple company, this is a Canadian product. You will see things like this as well as other indications that it's not an all-American world out there. For example, you can convert pints into Imperial Gallons through Tools -> Assistants -> Units Calculator.

6.8 More operations on lists

So far we have talked only about creating lists, and assigning lists as the value of a variable. You can also generate a sublist of a list, find a particular item in a list by its position index, count the number of items in the list, and convert a list into other types of data.

Table 6.9: Operations on lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list</td>
<td>$s1 := { a, b, c, a }$</td>
<td>Lists can contain symbols, numbers, expressions -- anything, even other lists.</td>
</tr>
<tr>
<td></td>
<td>$s2 := { 1, 3.47, 97, -5.9, 2.1 }$</td>
<td></td>
</tr>
<tr>
<td>Specify a sublist</td>
<td>$ts1 := s1[1..3]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ts1 := { a, b, c }$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s3 := s1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s3 := s2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ts2 := s3[1,3]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ts2 := { a, c }$</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Example</td>
<td>Commentary</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>Specify one item from the list</td>
<td>$sl[2]$</td>
<td>$b$</td>
</tr>
<tr>
<td>Specify the last item in the list</td>
<td>$sl[-1]$</td>
<td>$a$</td>
</tr>
<tr>
<td>Specify a sublist with one item</td>
<td>$s2[3..3]$</td>
<td>$[97]$</td>
</tr>
<tr>
<td>Specify a sublist from the 3rd from the end to the end</td>
<td>$sl[-3..-1]$</td>
<td>$[b, c, a]$</td>
</tr>
<tr>
<td>Count the number of items in the list</td>
<td>$n := \text{nops}(s2)$</td>
<td>$\sum_{i=1}^{n} s2[i]$</td>
</tr>
<tr>
<td>Add together all the items in the list</td>
<td>$\sum_{i=1}^{n} s2[i]$</td>
<td>$\frac{\sum_{i=1}^{n} s2[i]}{n}$</td>
</tr>
<tr>
<td>Compute the average of all the numbers in the list.</td>
<td>$\sum_{i=1}^{n} s2[i]$</td>
<td>$\frac{\sum_{i=1}^{n} s2[i]}{n}$</td>
</tr>
<tr>
<td>Convert a list into a sequence</td>
<td>$\text{op}(ts1)$</td>
<td>$a, b, c$</td>
</tr>
<tr>
<td>Convert a list into a string.</td>
<td>$\text{convert}(s2, \text{string})$</td>
<td>$[1, 3.47, 97, -5.9, 2.1]$</td>
</tr>
</tbody>
</table>

### 6.9 solve, lists and sequences

To solve a system of equations, use a list of expressions or equations for the first argument to `solve`. Use a list of variables as the second argument.

`solve` will return a sequence of lists as the result.

When `solve` finds two solutions for an equation (such as if the equation is quadratic), it will return a sequence of solutions. You can recognize a sequence and distinguish it from a list because, the sequence is missing the enclosing brackets `[ ]` that a list has.

Part-selection operations work in sequences in a similar fashion as they were described here (page 79).

In `solve`, lists and sequences look as if they are almost interchangable. Later on we will see situations where lists and sequences must be handled differently.
Table 6.10: Solving systems of equations with solve

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$eqn := 3x = x^2 - 28$</td>
<td>We get a sequence of solutions since there is a double root.</td>
</tr>
<tr>
<td>$solns := solve(eqn, x)$</td>
<td></td>
</tr>
<tr>
<td>$-4, 7$</td>
<td>(6.25)</td>
</tr>
<tr>
<td>$eval(eqn, x = solns[1])$</td>
<td>We evaluate the equation at the first root and see that it does satisfy the equation.</td>
</tr>
<tr>
<td>$-12 = -12$</td>
<td>(6.26)</td>
</tr>
<tr>
<td>$eval(eqn, x = solns[2])$</td>
<td>Same for the second.</td>
</tr>
<tr>
<td>$21 = 21$</td>
<td>(6.27)</td>
</tr>
<tr>
<td>$system := [3 \cdot x + 5 \cdot y = 6, 2 \cdot x - 5 = y]$</td>
<td>We want to assign a set as the value of the variable system. Maple tells us that the name is already in use as a built-in function, so it won't let us do that.</td>
</tr>
<tr>
<td>$sys := [3 \cdot x + 5 \cdot y = 6, 2 \cdot x - 5 = y]$</td>
<td>We choose a different variable to assign the set to.</td>
</tr>
<tr>
<td>$[3x + 5y = 6, 2x - 5 = y]$</td>
<td>(6.28)</td>
</tr>
<tr>
<td>$vars := [x, y]$</td>
<td>We specify the set of variables we want to solve for, then call $solve$. We get a list with one element (which itself is a list) as a solution.</td>
</tr>
<tr>
<td>$[x, y]$</td>
<td>(6.29)</td>
</tr>
<tr>
<td>$solns := solve(sys, vars)$</td>
<td></td>
</tr>
<tr>
<td>$[[x = \frac{31}{13}, y = \frac{3}{13}]]$</td>
<td>(6.30)</td>
</tr>
<tr>
<td>$soln1 := solns[1]$</td>
<td>We extract the first element of the list. Notice that that are fewer [ ]s.</td>
</tr>
<tr>
<td>$[x = \frac{31}{13}, y = \frac{3}{13}]$</td>
<td>(6.31)</td>
</tr>
<tr>
<td>$eval(sys, soln1)$</td>
<td>We evaluate the system at the solution that $solve$ has found and verify that this really does satisfy the system of equations.</td>
</tr>
<tr>
<td>$[6 = 6, -\frac{3}{13} = -\frac{3}{13}]$</td>
<td>(6.32)</td>
</tr>
<tr>
<td>$sys2 := [x^2 + y^2 = 25, x + y = 5]$</td>
<td>This system of equations has two distinct solutions, so we get a list with two elements in it. Each element is a distinct solution.</td>
</tr>
<tr>
<td>$[x^2 + y^2 = 25, x + y = 5]$</td>
<td>(6.33)</td>
</tr>
<tr>
<td>solve(sys2, [x, y])</td>
<td></td>
</tr>
<tr>
<td>$[[x = 5, y = 0], [x = 0, y = 5]]$</td>
<td>(6.34)</td>
</tr>
<tr>
<td>$eqn2 := \sqrt{x} - 3 \cdot x = 5$</td>
<td>We can find the non-negative roots of an equation by including the appropriate inequality in the list of relations given to $solve$. The result by default is a set. however, if the second argument to $solve$ is a list of variables, then the result will come back as a list. You can select items from a set using the same notation as with lists.</td>
</tr>
<tr>
<td>$solve(eqn2)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(6.35)</td>
</tr>
</tbody>
</table>
6.10 Evaluation, eval, and assignment

Suppose that we had an expression relating time \( t \) to voltage registered by a capacitor as it is being charged by a battery. In a mathematics or electrical engineering textbook, we might see this written as

\[ V(t) = 35 + (65 - 35) \cdot \left(1 - e^{-\frac{t}{3}}\right) \].

We are interested in taking this expression for voltage and doing several calculations with it -- plotting it for a range of \( t \), finding values of \( t \) that correspond to a specified voltage (e.g. "find the time \( t \) when the voltage reached 55 volts"), or finding a voltage corresponding to a specified time (e.g. "find the voltage at \( t=2.5 \) minutes after the start").

If we set up an assignment in Maple, then we could calculate the voltage at \( t=2.5 \) minutes by assigning \( t \) the value 2.5 and then evaluating \( V \). The second evaluation will cause the current value of \( t \) to be used. However, if we wanted to plot the expression \( V \) after that, then we'd have problems because whenever we would type \( t \), Maple would use the value of \( t \) rather than the symbol \( t \).

### Evaluating an expression using a particular value of one of the variables in the expression, and then plotting

\[
V := 35 + (65 - 35) \cdot \left(1 - e^{-\frac{t}{3}}\right)
\]

\[
V := 65 - 30 e^{-\frac{t}{3}}
\]  

\( t := 2.5 \)

\[
t := 2.5
\]

\[
V
\]

\[
51.96205374
\]
Rather than \( t=0..10 \) Maple is seeing \( 2.5=0..10 \) because it is using the value of \( t \) when evaluating what we typed.

We can clear the path for plotting by unassigning \( t \) first. Note that if we did a `restart` instead of an unassign we would lose the assignment to \( V \). Restarting at this point is a bad idea, since it forces us to redefine \( V \) as well as \( t \).

The same problem would happen if we tried to solve an equation involving \( V \) if we had already assigned \( t \) a value.

### Evaluating an expression using a particular value of one of the variables in the expression, and then solving

```maple
restart
V := 35 + (65 - 35) \cdot \left(1 - \frac{t}{3}\right)
t := 4.7
V
```

We set up an expression and then evaluate it at \( t=4.7 \) seconds

```maple
58.73780530
```

Because Maple is evaluating the names \( V \) and \( t \) in the `solve` operation, it is seeing `solve \( \left(35 + (65 - 35) \cdot \left(1 - \frac{4.7}{3}\right), 4.7\right) \)` which it cannot solve because there are no variables in the equation to solve for.

```maple
solve(V = 55, t)
```

Warning, solving for expressions other than names or functions is not recommended.
Error, (in solve) a constant is invalid as a variable, 4.7

```maple
unassign('t')
solve(V = 55, t)
```

We can clear the path for solving by unassigning \( t \) first. Doing a `restart` would not work, because that would also unassign everything, including \( V \). We would lose the expression we want to solve for.

```maple
3 \ln(3)
```

(6.47)
It can be tedious to have to remember to unassign variables if we want to go back to using them as symbols in the expression. We recommend using the `eval` operation (also available in the clickable menu as \( f(x) \bigg|_{x=a} \)) instead of assignment, if you are switching back and forth between using values for a variable and using it as a symbol. `eval` returns the same result as if you had done the evaluation, but the evaluation is automatically undone after the calculation is performed.

You can evaluate using values for several variables by giving a list of equations instead of a single equation as the second argument to `eval`.

---

**Evaluating an expression using a particular value of one of the variables in the expression using eval, and then solving**

<table>
<thead>
<tr>
<th>restart</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V := 35 + (65 - 35) \cdot \left(1 - e^{-\frac{t}{3}}\right) )</td>
</tr>
<tr>
<td>( V := 65 - 30 \cdot e^{-\frac{t}{3}} ) (6.48)</td>
</tr>
<tr>
<td><code>eval(V, t=2.5)</code></td>
</tr>
<tr>
<td>51.96205374 (6.49)</td>
</tr>
<tr>
<td><code>solve(V=55, t)</code></td>
</tr>
<tr>
<td>3 ( \ln(3) ) (6.50)</td>
</tr>
</tbody>
</table>

Evaluating \( V \) produces the same result as before (an expression with the symbol \( t \) in it), so we can immediately set up an equation and solve it for \( t \).

<table>
<thead>
<tr>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t</code></td>
</tr>
<tr>
<td>(6.51)</td>
</tr>
</tbody>
</table>

Note that the `eval` did not assign a value to \( t \). It's still just a symbol at this point.

| restart |

**Begin parameters**

<table>
<thead>
<tr>
<th>( V(t) := 35 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V(t) := 35 ) (6.52)</td>
</tr>
<tr>
<td>( V_{max} := 65 )</td>
</tr>
<tr>
<td>( V_{max} := 65 ) (6.53)</td>
</tr>
<tr>
<td>( vt := 55 )</td>
</tr>
<tr>
<td>( vt := 55 ) (6.54)</td>
</tr>
<tr>
<td>( t0 := 4.7 )</td>
</tr>
<tr>
<td>( t0 := 4.7 ) (6.55)</td>
</tr>
</tbody>
</table>

**End parameters**

\( V_{prime} := V(t) + (V_{max} - V(t)) \cdot \left(1 - e^{-\frac{t}{3}}\right) \)
An example using assignment and eval

This is an example of how to set up a script using parameters while taking advantage of `eval`. Several symbols in the expression for voltage are set up and assigned as parameters. But we use `eval` to maintain $\tau$ and $t$ as symbols in the expression $V'$. 

### Parameters

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_i$</td>
<td>35</td>
</tr>
<tr>
<td>$V_{\text{max}}$</td>
<td>65</td>
</tr>
<tr>
<td>$v_t$</td>
<td>55</td>
</tr>
<tr>
<td>$t_0$</td>
<td>4.7</td>
</tr>
</tbody>
</table>

### Expression

- $V' := V_i + (V_{\text{max}} - V_i) \cdot \left(1 - e^{-\frac{t}{\tau}}\right)$

We use the information that the capacitor is observed to at $v_t$ volts at time $t=t_0$ to find the value of $\tau$ that is consistent with this.
\[ \text{tauValue} := \text{solve(tauExpr = v_t, tau)} \]
\[ \text{tauValue} := 4.278124365 \] (6.67)

\[ tExpr := \text{eval(Vprime, tau = \text{tauValue})} \]
\[ tExpr := 65 - 30 e^{-0.2337472955 t} \] (6.68)

\[ \text{solve(tExpr = v_t, t)} \]
\[ 4.699999999 \] (6.69)

6.11 Summary of Chapter 6 material

<table>
<thead>
<tr>
<th>Troubleshooting textual input in Maple</th>
<th>Examples with error(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remember to...</td>
<td></td>
</tr>
<tr>
<td>Supply a function name and arguments (parameters)</td>
<td>solve ( x^3 - 3x^2 - 2 ) should be ( \text{solve} \left( x^3 - 3x^2 - 2, x \right) )</td>
</tr>
<tr>
<td>Match delimiters (parenthesis and brackets)</td>
<td>solve ( x^3 - 3x^2 - 2, x ) should be ( \text{solve} \left( x^3 - 3x^2 - 2, x \right) )</td>
</tr>
<tr>
<td>Press the right arrow key to exit from variable exponents</td>
<td>solve ( x^3 - 3x^2 - 2, x ) should be ( \text{solve} \left( x^3 - 2, x \right) )</td>
</tr>
<tr>
<td>Set ranges correctly when plotting</td>
<td>plot ( x^3 - 3x^2 - 2, x = 3..3 ) should be ( \text{plot} \left( x^3 - 3x^2 - 2, x = 3..3 \right) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plotting</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plotting multiple expressions</td>
<td>( \text{plot} \left( \left{ x^2, \sin(x^2) \right}, x = 0..4 \right) )</td>
</tr>
</tbody>
</table>
| Plotting with lists | \( xList \) := \{ 1, 2, 3, 4 \}
| | \( xList \) := \{ 1, 2, 3, 4 \} (6.70)
| | \( yList \) := \{ 5, 6, 7, -1 \}
| | \( yList \) := \{ 5, 6, 7, -1 \} (6.71)
| | \( \text{plot} \left( xList, yList, \text{style = point} \right) \) |
| Using multiple colors in a multi-plot | \( \text{plot} \left( \left\{ \sin(x), \sin \left( \frac{x}{2} \right), \sin(2-x) \right\}, x = -\pi..\pi \right) \)
| | \( \text{color} = \left\{ \text{"red"}, \text{"green"}, \text{"blue"} \right\} \) |
| Set the titles of the axes | \( \text{plot} \left( 3.5x^2 - 2, x = 1..5, \text{labels} \right) \)
| | \( \text{"temperature in degrees C"}, \)
| | \( \text{"pressure in kilopascals"} \) |

<table>
<thead>
<tr>
<th>Using Maple's built-in help</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Help&gt;Maple Help or press Ctrl-F1 (Command-F1 on a Mac)</td>
<td>Remember that you can click on related topics when viewing the help for a particular command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on lists</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Create a list | \( sl := [a, b, c, a] \)
| | \( [a, b, c, a] \) (6.72) |
Specify a sublist of values
\[ tsl := s1[1..3] \]
\[ [a, b, c] \]  \hspace{1cm} (6.73)

Specify one item from the list
\[ s1[1] \]
\[ a \]  \hspace{1cm} (6.74)

Specify a sublist with one item
\[ s2[3..3] \]
\[ [97] \]  \hspace{1cm} (6.75)

Count the number of items in the list
\[ n := \text{nops}(s2) \]
\[ 5 \]  \hspace{1cm} (6.76)

Add together all the items in the list
\[ \sum_{i=1}^{n} s2[i] \]
\[ 97.67 \]  \hspace{1cm} (6.77)

Compute the average of all the numbers in the list.
\[ \frac{\sum_{i=1}^{n} s2[i]}{n} \]
\[ 19.5340000 \]  \hspace{1cm} (6.78)

Convert a list into a sequence
\[ \text{op}(ts1) \]
\[ a, b, c \]  \hspace{1cm} (6.79)

Convert a list into a string
\[ \text{convert}(s2, \text{string}) \]
\[ "[1, 3.47, 97, -5.9, 2.1]" \]  \hspace{1cm} (6.80)

### Solving a system of equations using lists

**Create the system of equations**
\[ \text{sys} := \begin{cases} \ 3 \cdot x + 5 \cdot y = 6, \\ 2 \cdot x - 5 = y \end{cases} \]
\[ [3x + 5y = 6, 2x - 5 = y] \]  \hspace{1cm} (6.81)

**Set the variables of the system**
\[ \text{vars} := [x, y] \]
\[ [x, y] \]  \hspace{1cm} (6.82)

**Solve the system and extract the first solution of possibly many solutions**
\[ \text{solns} := \text{solve}(\text{sys}, \text{vars})[1] \]
\[ \begin{cases} x = \frac{31}{13}, y = -\frac{3}{13} \end{cases} \]  \hspace{1cm} (6.83)

### Evaluating an expression with eval instead of assignment

**eval** allows you to substitute a value for a variable within an expression, without assigning that value to the variable.

**Examples**
\[ V := 35 + (65 - 35) \cdot \left(1 - e^{-\frac{t}{3}}\right) \]
\[ V := 65 - 30 e^{-\frac{t}{3}} \]  \hspace{1cm} (6.84)
<table>
<thead>
<tr>
<th>eval($V, t = 2.5$)</th>
<th>51.96205374</th>
<th>(6.85)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>$t$</td>
<td>(6.86)</td>
</tr>
</tbody>
</table>
Chapter 7 Using and Defining Functions

7.1 Chapter overview

Functions occur so much in mathematics that it's natural that Maple knows a lot about them and how to compute with them.

You can also define your own functions.

There are a few pitfalls in the use of functions in Maple:

a) The names of common mathematical functions used in Maple may differ from what you are used to.

b) Some functions in Maple do non-mathematical things, such as solve, and plot. Others take novel arguments -- lists, equations, and ranges, rather than numbers.

c) The way functions are defined uses := and -> rather than the use of = as found in math textbooks. This is because the "context-free" language processing of Maple thinks an equation is being defined whenever it sees an equal sign. It would be difficult for standard computer language-processing technology to use context to determine that an "equals" means "function definition".

7.2 Functions in computer languages: a way of producing an output from inputs

Everyone is introduced to the idea of a function in secondary school mathematics: Calculators can compute many of the common functions found in high school algebra and pre-calculus: sin, cos, ln, √ , etc.

Maple can evaluate these functions. The common ones are found in the Expression palette but there are hundreds more.

When entering a functional expression, the syntax used is:

function name ( sequence of arguments )

The parentheses are mandatory in Maple. Unexpected results, possibly including an error message may result if you forget them.

Table 7.1: Function results in Maple

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(.35), cos\left(\frac{3.14159265}{2}\right), ln(3.72), \sqrt{2.0}</td>
<td>All the common functions know how to give limited-precision results given limited precision inputs</td>
</tr>
<tr>
<td>cos\left(\frac{\pi}{2}\right), ln(1), \sqrt{2} , sin\left(\frac{35}{100}\right), \left(\frac{5}{2}\right), log_{10}(1001), log_{10}(1001.0)</td>
<td>Maple's programming will return an exact number if the function has that kind of result for the given input. However, if a simple exact result can't be found, Maple will return what you typed in. Sometimes there is a little simplification that goes on so what comes out is not literally what comes in, although it usually obvious that it is mathematically equivalent. For example, Maple will always simplify fractions by eliminating the greatest common divisor from the numerator and denominator.</td>
</tr>
<tr>
<td>cos([a, b, c])</td>
<td>Error, invalid input: cos expects its 1st argument, x, to be of type algebraic, but received [a, b, c]</td>
</tr>
<tr>
<td>cos(2, \pi)</td>
<td>Error, (in cos) expecting 1 argument, got 2</td>
</tr>
</tbody>
</table>

89
### Example | Commentary
---|---
\( \cos 1 \ 2 \) | What was this person thinking? Whatever it was, Maple doesn't know what to do with it.  
\( \cos \left( \frac{\pi}{4} \right) \) | Another delimiter message. Look for extra or missing parentheses.  
\( \cos \left( \frac{\pi}{4} \right) \) | This is what happens if you forget the mandatory parentheses. There is no error message, but what Maple is giving you is the product of \( \pi \), the symbol "cos", and 1/4.  
\( \cos \left( \frac{\pi}{4} \right) \) | This is the correct way to compute the cosine of \( \frac{\pi}{4} \) radians.  

The attachment at the end of this chapter shows some of the many other functions available in Maple. Some of them work on lists, sets, equations rather than on numbers or expressions. However, the same principle applies: they have a rule for taking the value of their inputs (also known as arguments) and computing a result from them.

### 7.3 The textual names of common functions: doing math calculations using the keyboard.

Some built-in mathematical functions have textual names that are already quite familiar from their use in mathematics textbooks: \( \sin, \cos, \ln \). Some are used so often that the most convenient thing to do is to remember their names. \( \sqrt{\text{rt}}, \text{abs}, \text{min}, \text{max} \) are straightforward -- they are naturally thought of as functions and have names that are abbreviations of the standard nomenclature. Others, such as \( \text{arcsin, } \text{or log10} \) have names that make sense but you'd have to look them up in the Maple on-line documentation to know.

**Examples of textual names for common math functions**

\[
\text{expr} := \text{abs} (\text{arcsin} (\log10(t)))
\]

Maple knows that "log10" is the base 10 logarithm, but automatically converts it into the ratio of the base e logarithm and \( \ln(10) \).

Many users will find it faster to type in the textual version of plot than to use the plot builder. Furthermore, if we want to change the expression...
or the range or labels of the plot, it's easier to edit the text and re-execute the region than it would be with the clickable interface.

\[
\text{plot}(\text{expr}, t=-1..1, \text{title} = "A funky non-linear plot")
\]

A funky non-linear plot

\[
\text{max}(-3,92,43.7,0,\sqrt{16})
\]

\[
92
\]

(7.8)

\[
\text{min}(5,7,29,x,3)
\]

\[
\text{min}(3,x)
\]

(7.9)

\[
\text{max}\left(\text{abs}(x)\cdot\frac{x}{2},5.7,\text{abs}(x)-2,0\right)
\]

\[
\text{max}\left(5.7,\frac{1}{2}x,|x|\right)
\]

(7.10)

\[
\text{max}\text{ can find the maximum when all of its arguments are numbers. max and min can do limited reasoning if some of the arguments are not numeric. For example, in the third example, Maple knows that abs(x)-2 is not going to be the maximum because abs(x) is must be larger.}
\]

\[
7.4 \text{ A function name to commit to memory: exp}
\]

The only one function whose textual usage may take getting used to is the exponential function \(\text{exp}\). Instead of writing \(e^x\), use \(\text{exp}(x)\). The textual doppelganger \(e^x\) does not work as a way of calculating a power of \(e\), the base of the natural logarithm (where "\(e\)" is just the letter typed at the keyboard, not augmented by command completion as described in section XX). The orientation in college-level mathematics to view "a power of \(e\)" as a function is a pervasive change in point of view from what you may have seen in high school. Making a point to use the new notation frequently is the best way to make the switch.

\[
\text{exp is the name of the exponential function}
\]

\[
\text{exp}(x)\cdot\text{exp}(y)
\]

\[
e^x e^y
\]

(7.11)

\[
\text{exp}(x)\text{ is the textual way of writing "the symbolic constant } e \text{ raised to the power } x. \text{ Maple knows how to simplify symbolic expressions with the simplify } \rightarrow \text{ assuming real operations in the clickable menu.}
\]

\[
\text{assuming real}
\]

\[
e^{x+y}
\]

(7.12)
We can calculate powers of $e$ from the keyboard. For comparison we do the same calculation using the Common Symbols and Expressions Palette. While more of a “sure thing”, proficient users would be able to get the keyboard version entered more quickly.

\[
\begin{align*}
\text{exp(.1)} & \quad 1.105170918 \\
\text{e}^1 & \quad 1.105170918
\end{align*}
\]

(7.13) (7.14)

We can enter an expression involving exponentials using only the keyboard. Maple will regard it as meaning the same thing as the expression entered using the combination of the keyboard and the Palette.

\[
\begin{align*}
V_i + (V_{\text{max}} - V_i) \left( 1 - \exp\left( -\frac{I}{\text{tau}} \right) \right) \\
V_i + (V_{\text{max}} - V_i) \left( 1 - e^{-\frac{I}{\text{tau}}} \right)
\end{align*}
\]

(7.15)

### 7.5 How can I remember so many functions?

A well-developed system such as Maple, Matlab, or C# has thousands of built-in functions. It is unreasonable to expect that you can get the full gamut of professional work done knowing only three or four functions. The bad news is that you will have to remember at least the names of a several dozen functions. The good news is that learning about functions is not that taxing -- if you own a scientific calculator you've already dealt with a situation where you can operate a dozen functions.

A reasonable stance to take is to be familiar (i.e. know "by heart") functions and symbols that you often use and to be adept at using documentation to look up the details of the ones that you need only occasionally. In exams and test about computer functions, you may be quizzed on the details of the most common, but things rarely end up in a place where your grade will depend on how well you memorize thick lists of names.

A quick-recall method for access to common functions with textual entry is to type the escape key (Esc) after typing a few characters. A pop-up menu will appear that will list possible ways of completing what you typed. This is called command completion. Recall that you've already used this feature to enter symbolic constants such as $e$, the natural logarithm base. If what you are entering is a Maple operation such as `solve`, command completion will provide a textual template to fill in the rest of the arguments. It will also provide the parentheses required when entering functional notation textually.

<table>
<thead>
<tr>
<th>Table 7.2: Command Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>We type <code>sol</code> and then type the ESC key. A pop up menu shows option, including several forms of <code>solve</code>.</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{sol} & \\
\text{solve(eqn, x)} & \quad \text{Clicking on one of the options provides something to fill out. We are not compelled to have a second variable of $x$, it's just short-hand reminder that the second argument is the variable and the first one is the}
\end{align*}
\]
equation. We can then edit these to be what we need for the purpose at hand.

7.6 Be savvy about using on-line documentation

Experienced users refer to the on-line documentation to help remember details about functions. As has mentioned earlier, this is available through Help -> Maple Help menu feature (key shortcut: press the control key and then the F1 key). If you recall a phrase or a name of a function, you can type it into the search field and the on-line help system will, like Google, produce the pages it has about your text entry. You can then explore further by pressing links. Trying the examples typically given at the end of the description is a good way to get a form that you can use for your own purposes.

Using on-line help

We want to find information about how to use the inverse sine function in Maple. We start up on-line help and type in "inverse trigonometric" into the search field, then hit the "search" button. The page we see does tell us that it's probably called "arcsin" but we'd like to see more. We see a link in the "see also" which we click on.

This uncovers more links. The one that says "invtrig" seems promising so we click on that link.
7.7 Defining your own functions with \( \rightarrow \) (arrow)

Maple allows you to define simple functions with the use of \( \rightarrow \). The general form is

\[
\text{function name} := (\text{sequence of arguments}) \rightarrow \text{expression that describes result}.
\]

These can be entered through the Expression Palette, or textually. The arrow is entered textually by typing a - and then a >, with no spaces separating them. We give an example of function definition in the following example.

Table 7.3: A function definition in a math textbook

Fruit flies are placed in a half-pint milk bottle with a banana (for food) and yeast plants (for food and to provide a stimulus to lay eggs). Suppose that the fruit fly population after \( t \) days is given by

\[
P(t) = \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}}.
\]

(a) What is the population after five days?

(b) How long does it take for the population to reach 180?

Analyzing the text, we see that it defines a function named \( P \). It takes one input (argument), \( t \), and produces as output whatever you get from evaluating the expression \( \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}} \). Assuming that Maple understands the definition of \( P \) like it does a built-in function, then the description of what happens when you evaluate \( P(5) \) would be:

"Substitute 5 for wherever you see \( t \) in the expression \( \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}} \). This gives you

\[
\frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot 5}}
\]

. Perform all the arithmetic and relevant simplifications in this expression and return that as the result of the function."
Even though 5 is an exact number, because there are limited precision numbers in the expression we expect that the result will be a limited-precision number. If the expression had only exact numbers in it, then the calculation would be done exactly.

What we would like to do is to tell Maple about the definition of \( P \) and use it in our work.

We can do this through the clickable interface. We anticipate reuse of this for other days and population levels, and turn it into a parameterized script:

Table 7.4: User-defined functions through the Expression Palette

<table>
<thead>
<tr>
<th>User-defined functions through the Expression Palette</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>Start of parameters</td>
<td></td>
</tr>
<tr>
<td>numDays := 5</td>
<td></td>
</tr>
<tr>
<td>( 5 )</td>
<td>(7.16)</td>
</tr>
<tr>
<td>popLevel := 180</td>
<td></td>
</tr>
<tr>
<td>( 180 )</td>
<td>(7.17)</td>
</tr>
<tr>
<td>End of parameters</td>
<td></td>
</tr>
<tr>
<td>[ P := t \rightarrow \frac{230}{1 + 56.5 \cdot e^{-0.37 \cdot t}} ]</td>
<td>(7.18)</td>
</tr>
<tr>
<td>Compute the number of flies after ( numDays = 5 ) days.</td>
<td>With this definition, we can compute ( P(5) ) using the natural mathematical notation.</td>
</tr>
<tr>
<td>( P(numDays) )</td>
<td></td>
</tr>
<tr>
<td>23.27016688</td>
<td>(7.19)</td>
</tr>
</tbody>
</table>

We plot the function to see how the population grows. This is not needed by the problem but it helps us understand the situation better.

We can plot \( P(t) \) like we would any other expression. One the options to plot (see plot options in on-line help) is the ability to specify the title of the graph by giving title=string as an additional argument.
User-defined functions through the Expression Palette

```plaintext
plot(P(t), t = 0..numDays, title
    = "Fruit flies like a banana", labels = ["t",
    "# of flies"])
```

Fruit flies like a banana

Compute when the population reaches desired level by solving the equation \( P(t) = \text{popLevel} \).

\[ \text{soln := solve}(P(t) = \text{popLevel}, t) \]

14.36533644

(7.20)

In anticipation of using the value in later work, we assign it to the variable \( \text{soln} \). We would then intend to take further steps using \( \text{soln} \).

End of script

We could have defined the function textually just by typing \( P := (t) \rightarrow \ldots \) followed by the rest of the expression instead of using the Expression Palette.

## 7.8 Using functional evaluation instead of using assignment or eval

The problem with assigning variables is that it may cause unwanted side-effects, such as when trying to \( \text{solve} \) or \( \text{plot} \) with expressions involving those variables. We have seen in the section on eval (page 79) that we can use \( \text{eval} \) instead of assignment, which saves us the chore of having to assign and then unassign variables so that they remain as symbols when solving or plotting. We can more succinctly avoid this problem by defining a function based on the expression, and then causing evaluation to occur with standard functional notation.

### Table 7.5: Comparing evaluation using eval and functional notation

<table>
<thead>
<tr>
<th>restart</th>
<th>We set up an expression and then evaluate it at ( t = 4.7 ) seconds.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V := 35 + (65 - 35) \cdot \left( 1 - e^{\frac{t}{3}} \right) )</td>
<td>We set up an expression and then evaluate it at ( t = 4.7 ) seconds.</td>
</tr>
<tr>
<td>( V := 65 - 30 e^{\frac{t}{3}} )</td>
<td>( V := 65 - 30 e^{\frac{t}{3}} )</td>
</tr>
</tbody>
</table>
We create a function of \( t \) using the arrow notation. We can then evaluate the function at any value of \( t \) that we want through the standard notation. This takes a little less typing than using `eval`.

\[
V_f := (t) \rightarrow 35 + (65 - 35) \left(1 - e^{-\frac{t}{3}}\right)
\]

\[
V_f := t \mapsto 65 - 30 e^{-\frac{t}{3}}
\]

We can plot the function \( V_f \) in a natural way. `plot` also has an abbreviated way of plotting functions -- just give the name of the function and the plotting range, and omit the name of the functional argument. You can read more about this in the on-line help for `plot`. 
Troubleshooting function definitions

Function definition is another place where the standard notation in mathematics does not work in Maple. Recall that the technology that is standard in most computer language understanding systems needs to assign a unique meaning to input from the way it looks. Equations already use "=" so if you use \( f(x) = ... \) Maple will understand you to be talking about an equation, not a function definition. Use :=

<table>
<thead>
<tr>
<th>Table 7.6: Troubleshooting function definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F := (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
</tr>
<tr>
<td>( g : = 6.673 \times 10^{-11} )</td>
</tr>
<tr>
<td>( F(1, 2, 10) )</td>
</tr>
<tr>
<td>( F_{bad} := (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} )</td>
</tr>
<tr>
<td>Error, invalid operator parameter name</td>
</tr>
</tbody>
</table>
This illustrates an alternative way of defining a function, although it is not the form prescribed by these notes. Rather than "=" as would appear in a math textbook, the assignment operation ":=" is used instead. This produces the following pop-up:

Clicking "ok" to function definition will create the proper function definition, as the subsequent line of the computation indicates. We get the same result as with F.

### 7.10 Using functions from library packages, with

Although we have seen a number of built-in functions so far in Maple, there are several thousand more. Some of them are defined in your Maple program when it starts up. However, it is not done for most of the built-in functions. There are so many that if that were done for all of them, Maple would take a long time to start up and would require large amounts of memory even before you had done any work in it.

Most built-in functions are organized into collections called *packages*. The general way to access a function belonging to a package is through `package[function]`. The least squares function belongs to a package named CurveFitting, hence its full name is `CurveFitting[LeastSquares]`.

The `with(package name)` operation in Maple will load all the functions in the specified package into Maple. After this operation, functions can be referred to with just their "short name", e.g. `LeastSquares` rather than `CurveFitting[LeastSquares]`. Doing a `with(package)` can save you typing if you expect to use a function, or several functions, from a package several times during a Maple session. Ending the line with a colon (:) will suppress printing of all the functions in the package that usually occurs.

<table>
<thead>
<tr>
<th>with</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td><strong>Commentary</strong></td>
</tr>
<tr>
<td><code>restart</code></td>
<td>The CurveFitting package has a number of functions for data fitting. One is called <code>LeastSquares</code>. However, <code>LeastSquares(...)</code> does nothing. Although there is no error message, this is a mistake -- we didn't load the package in with a &quot;with&quot; so the least squares data fitting function isn't known. We get the same kind of behavior as if we had typed in <code>f(1,2,3)</code> with <code>f</code> undefined -- Maple just spits back what we typed in.</td>
</tr>
<tr>
<td><code>pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(7.35)</td>
</tr>
</tbody>
</table>
Doing a "with" gives the names of all the functions in the package.

Once we do the with, LeastSquares works.

7.11 Attachment: some built-in problem-solving functions

The functions in the Expression palette have the same name and work similarly to those described in math textbooks. The operations discussed in this attachment are also found in math textbooks, but they are usually not given function names. It may seem novel to you that the rules for solving equations, factoring polynomials, or plotting can be collected together and given a function name. Yet this way of writing about such actions allows us to combine mathematics and working on it. Thus solve, plot, factor, etc. are true functions -- they have names, they are invoked with arguments, and return results that can be assigned to a variable.

<table>
<thead>
<tr>
<th>Textual names of common operations in Maple</th>
<th>Function</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>solve an expression or an equation</td>
<td>solve</td>
<td>solve((x^2 - 7\cdot x - 98))</td>
<td>If there is more than one root, returns a sequence of values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14, -7</td>
<td>(7.41)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>solve((x^2 = 7\cdot x + 98))</td>
<td>(7.42)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14, -7</td>
<td></td>
</tr>
<tr>
<td>solve an expression or an equation numerically</td>
<td>fsolve</td>
<td>fsolve((x^2 = \cos(x)))</td>
<td>If there is more than one root, returns a sequence of values</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.8241323123</td>
<td>(7.43)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fsolve((x^2 - 7\cdot x - 97))</td>
<td>(7.44)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-6.952272480, 13.95227248</td>
<td>fsolve won't work if the answer is not a number.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>fsolve(a \cdot x^2 + b \cdot x + c = 0, x)</code></td>
<td>Error, in <code>fsolve</code> {a, b, c} are in the equation, and are not solved for.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>plot</code></td>
<td>Plot takes two inputs. The first is an expression, the second is an equation naming the variable and the horizontal plot range. <em>plot</em> may take more than two arguments, optionally. The rest of the arguments are referred to as <em>plot options</em>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>plot(x^2 - 7 \cdot x - 98, x = -20 .. 20)</code></td>
<td><code>plot</code> may take more than two arguments, optionally. The rest of the arguments are referred to as <em>plot options</em>.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
<tr>
<td><code>plot([t^2 + 7 \cdot t, 4], t = -8 .. 2, color = [&quot;DodgerBlue&quot;, &quot;Purple&quot;], labels = [&quot;time&quot;, &quot;velocity&quot;])</code></td>
<td><code>plot</code> may take more than two arguments, optionally. The rest of the arguments are referred to as <em>plot options</em>.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
<tr>
<td><code>eval(x^2 + y^2, x = 1)</code></td>
<td>Eval takes two arguments (inputs). The first is an expression, the second is an equation or a list of equations indicating what values to use for one or more variables in the expression.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
<tr>
<td><code>eval(x^2 + y^2, [x = 1, y = \pi + 2])</code></td>
<td>Eval takes two arguments (inputs). The first is an expression, the second is an equation or a list of equations indicating what values to use for one or more variables in the expression.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
<tr>
<td><code>rhs(x = 3 \cdot x^2 + 1)</code></td>
<td>The third example shows that Maple thinks that &quot;x=0..10&quot; is an equation even if it isn't one in the standard mathematical sense.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
<tr>
<td><code>lhs(x = 3 \cdot x^2 + 1)</code></td>
<td>The third example shows that Maple thinks that &quot;x=0..10&quot; is an equation even if it isn't one in the standard mathematical sense.</td>
<td><img src="image" alt="Plot" /></td>
<td></td>
</tr>
</tbody>
</table>
7.12 Summary of Chapter 7 material

<table>
<thead>
<tr>
<th>Common mathematical functions (see on-line help for index of functions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin, cos, tan, sec, csc, cot</td>
</tr>
<tr>
<td>exp</td>
</tr>
<tr>
<td>arcsin, arccos, arctan, arcsec, arccsc, arccot</td>
</tr>
<tr>
<td>sinh, cosh, tanh, ....</td>
</tr>
<tr>
<td>ln, log10, log[b]</td>
</tr>
<tr>
<td>min, max</td>
</tr>
<tr>
<td>abs, sqrt</td>
</tr>
<tr>
<td>ceil, floor, trunc, frac</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>When entering math, type the first part of the name of the function and then hit the escape key. A pop-up window will appear with a list of all- $sol$ (then hit the ESC key)</td>
</tr>
</tbody>
</table>
ternative competitions of what you typed. Pick one, and a template will automatically be entered for you. Edit the spots of the template to fit your situation.

This is a way to get the computer to automatically type the right kind of delimiters.

### Defining custom functions with the arrow (\(\rightarrow\)) notation

Create a custom function by naming it and its parameters. A custom function defined using arrow notation is of the form:

\[ \text{FunctionName} := \text{ParameterName} \rightarrow \text{Function} \]

In this case the function name is \(P\) and its sole parameter is \(t\). For multiple parameters, use the form:

\[ \text{FunctionName} := \left( \text{ParameterName1}, \text{ParameterName2}, \ldots \right) \rightarrow \text{Function} \]

To call the function, use the form:

\[ \text{FunctionName}(\text{ParameterValues}) \]

To plot the function, set the range of an independent variable, and use it as a parameter to your custom function.

To solve the function for a particular value, use the solve function.

### Troubleshooting function definitions

**Error**

- Forgetting to use the assignment (\(\Leftarrow\)) operator.
- Using the equality operator (\(=\)) instead of the arrow in function definition.

**Examples**

<table>
<thead>
<tr>
<th>Error</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forgetting to use the assignment ((\Leftarrow)) operator.</td>
<td></td>
</tr>
</tbody>
</table>

\[ F_{\text{bad}} = (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} \]

Error, invalid operator parameter name

\[ F_{\text{bad}} = (m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2} \]

Using the equality operator (\(=\)) instead of the arrow in function definition.

\[ FF := (m_1, m_2, r) = \frac{g \cdot m_1 \cdot m_2}{r^2} \]

\[ (m_1, m_2, r) = \frac{g \cdot m_1 \cdot m_2}{r^2} \]
### Troubleshooting function definitions

Although technically not an error, defining a function without using the arrow notation is not prescribed by these notes.

\[
F_2(m_1, m_2, r) := \frac{g \cdot m_1 \cdot m_2}{r^2}
\]

\[
(m_1, m_2, r) \rightarrow \frac{g \cdot m_1 \cdot m_2}{r^2}
\]

(7.64)
8 Chapter 8 Programming with functions

8.1 Chapter overview

Most computer languages regard functions or procedures as the "lego blocks" for building programs. Not only is it expected that there will be a lot of different kinds of blocks that will be provided, but that you will build things by putting them several of them together. The way functions are used in this way is through daisy chaining -- by making the output of one function the input of another. These chains can then be defined to be functions themselves. By defining a few chains and then using them together, powerful combinations of operations can be custom-built quickly for the user's needs.

Most computer languages extend the concept of functions to go beyond the numbers or formulas that "mathematical functions" provide. In Maple, as in most of languages, a function can return other kinds of results. It is fairly common in Maple and other languages to have functions that produce as output a list, an equation, or a string as a result. As we shall see, we can even return a Maple plot as the result of a function. In a symmetric fashion, it is possible for computer functions to have lists, equations, plots, or strings as inputs.

8.2 Designing functions from context

In doing technical work, we often see functions defined as an equation relating the name of the function, its argument(s), and the function definition.. Those are easy to translate into Maple's notation and use. For example if we see in a mathematics book "define \( f(x) = x^2 + x - v0 \) " then we can just transcribe it into the Maple function notation: \( f := (x) \rightarrow x^2 + x - v0 \).

In word problems, we have to "read between the lines" and design the function. This requires answering the questions:

a) What will the inputs to the function be? Try to give symbolic names for it.

b) What will the output be? Sometimes to realize what the output is, you can create a worksheet with several steps. If there is only one final result, then that should be the output.

c) How do you calculate the output from the inputs? Hopefully, there's a simple formula that describes this.

We illustrate this process with an example:

**Designing a function for pressure/temperature problems**


The Ideal Gas Law, as stated in *Introduction to Engineering* is:

\[
P \cdot V = n \cdot R \cdot T
\]

where

- \( P \) is pressure in Pascals (Pa)
- \( V \) is volume in \( m^3 \).
- \( n \) is the amount of gas in moles (mol),
- \( T \) is the temperature in degrees K,

\( R \) is the gas constant, approximately \( 8.31 \cdot \frac{m^3 \cdot Pa}{K \cdot mol} \).
We want to solve the following problem (actually, various versions of it):

**Problem**

We measure the temperature and pressure of a gas. It has a pressure of 100 \( \text{kPa} \) and a temperature 473 degree Kelvin. Then cool it so it has a pressure of 89.8 \( \text{kPa} \). What is its temperature?

Finding the answer

First, we do the mathematical thinking and informal calculation that allows us to build a function that will solve all problems of this type:

For a fixed cylinder volume, according to the Ideal Gas Law:

\( \frac{P_1}{T_1} = \frac{P_2}{T_2} \), so \( T_2 = \frac{P_2}{P_1} \cdot T_1 \).

\[
T_2 = \frac{89.8}{100} \cdot 473 = 424.7540000
\]

**Function design**

We see that the problem wants us to calculate a temperature \( T_2 \) given the atmospheric pressure \( P_1 \), the internal pressure \( P_2 \), and the first temperature, \( T_1 \).

The output is \( T_2 \), the inputs are \( P_1, P_2, \) and \( T_1 \). We are free to name the function anything we want since the problem statement doesn't name this. We decide to call it something that reminds us of the purpose.

\[
\text{secondTemp} := (P1, P2, T1) \rightarrow \frac{P2}{P1} \cdot T1
\]

\[
(P1, P2, T1) \rightarrow \frac{P2T1}{P1}
\] (8.2)
Testing and troubleshooting the function

We see whether we get the intended result with the numbers we've already worked out. Note that if we hadn't done the analysis, we wouldn't have any way of testing what we designed.

\[ \text{secondTemp}(89.8, 100, 473) \]

\[ 526.7260579 \quad (8.3) \]

Oops, that isn't the same result. What did we do wrong? The formula 1.2.2 seems like the right thing. What else could go wrong? Close inspection indicates that the first argument to \text{internalTemp} is P1, which appears in the denominator of the formula for the output. In (1.2.3), that would put the "89.8" in the denominator, but our example had 89.8 in the numerator. Oops, we gave the values in the wrong order for the function. There's nothing wrong except that we should invoke the function with the information given in the correct order:

\[ \text{secondTemp}(100, 89.8, 473) \]

\[ 424.7540000 \quad (8.4) \]

Using the function

We are given a different version of the problem:

We measure the temperature and pressure of a gas. It has a pressure of 2000 [kPa] and a temperature 473 degree Kelvin. Then cool it so it has a pressure of 53.6 [kPa]. What is its temperature?.

\[ \text{secondTemp}(2000, 56.6, 473) \]

\[ 13.38590000 \quad (8.5) \]

Since the answer is in degrees Kelvin, this is only about 14 degrees above absolute zero. That's pretty cold!

The usefulness of alternative function designs

Suppose we had this new problem:

We measure the temperature and pressure of a gas. It has a pressure of 100 [kPa] and a temperature 473 degree [K]. We then heat it to 512 degrees Kelvin. What is its pressure then?

Another function designed

A little thought produces the calculation:

\[ P_2 = \frac{P_1}{T_1} = \frac{100}{473} = 108.2452431 \]
This leads to the function definition:

\[
\text{secondPressure} := (P_1, T_1, T_2) \rightarrow \frac{P_1}{T_1} \frac{T_2}{T_2}
\]

We test this (remembering what happened before about the order of arguments)

\[
\text{secondPressure}(100, 473, 512) \rightarrow 108.2452431
\] (8.7)

**Conclusion**

To develop functions, it helps to have worked through some typical calculations interactively. Once you have realized which quantities you are starting with and named them, and have developed the formula for the calculation using those names, you can create a function definition. You can use the names given in the problem description, or you can make up names based on their purpose. Unlike mathematics, you are not limited to single letters for names of variables or names of functions. Computer programmers know that longer names are often easier to remember or understand.

### 8.3 Function composition: daisy-chaining functions together

In the scripts we have developed so far, we have developed a result through a sequences of actions. These sequences can often be described through functional composition -- an expression that chains together several actions. Consider the following example:

**Problem**

On November 1, 2007, one Euro was worth 1.002908434 US dollars. We are buying widgets that cost 30 Euros each and importing them into the US. We then put the widgets into packages that cost .075 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?
Finding the solution: step 1, first do the calculation interactively.

Doing this in the style of scripts, we first assign the values to variables, and then do the calculational steps.

\[
\text{convRate} := 1.002908434
\]

\[1.002908434\] (8.8)

\[
\text{costInEuros} := 30
\]

\[30\] (8.9)

\[
\text{pkgCost} := .075
\]

\[0.075\] (8.10)

\[
\text{markupPct} := .10
\]

\[0.10\] (8.11)

\[
\text{totalCost} := \text{pkgCost} + \text{convRate} \cdot \text{costInEuros}
\]

\[30.16225302\] (8.12)

\[
\text{sellingPrice} := (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[33.17847832\] (8.13)

We foresee using this calculation several times as the conversion Rate, the manufacturing cost in Europe, and the packaging cost change. We even see that the markup might change. We can try to boil down these steps into a few functions.

Designing the solution: step 2, design functions to do the calculational steps

\[
\text{totalCostFunc} := \text{convRate} \cdot \text{costInEuros} \cdot \text{pkgCost} \rightarrow (\text{pkgCost} + \text{convRate} \cdot \text{costInEuros})
\]

\[\text{(convRate, costInEuros, pkgCost) \rightarrow pkgCost + convRate \cdot costInEuros}\] (8.14)

\[
\text{priceFunc} := \text{markupPct, totalCost} \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[\text{(markupPct, totalCost) \rightarrow (1 + markupPct) \cdot totalCost}\] (8.15)

\[
\text{sellingPriceFunc} := \text{convRate} \cdot \text{costInEuros} \cdot \text{pkgCost} \cdot \text{markupPct} \rightarrow \text{priceFunc}(\text{markupPct},
\text{totalCostFunc}(\text{convRate, costInEuros, pkgCost}))
\]

\[\text{(convRate, costInEuros, pkgCost, markupPct) \rightarrow priceFunc(markupPct, totalCostFunc(convRate, costInEuros, pkgCost))}\] (8.16)

Note that the way that the third function \text{sellingPriceFunc} is defined, it takes the output of \text{totalCostFunc} and makes it one of the inputs to \text{priceFunc}.\]
Testing the solution: step 3, test the building blocks in the order that they are used

We test the first two functions. After we see that they agree with our preliminary version of the calculations, we test the third function that depends on the correctness of the first two.

\[
\text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost})
\]

\[30.16225302\] \hspace{1cm} (8.17)

\[
\text{priceFunc}(0.10, (1.3, 10))
\]

\[33.17847832\] \hspace{1cm} (8.18)

\[
\text{sellingPriceFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}, \text{markupPct})
\]

\[33.17847832\] \hspace{1cm} (8.19)

We could write a script that just used totalCostFunc and priceFunc, but by designing and using a third function, we reduce the work of handling an instance of the problem to just pasting in the values for the four parameters in one line. This is probably less work than changing four lines of parameters that we have had with our previous approach to scripts.

While function composition is a succinct way of ordering many operations, its advantages are apparent only after the chain is built and tested as working correctly. It maybe easier to develop the chain as a script of assignments and then refactor -- rewrite without changing the meaning -- the script so that it uses user-defined functions to replace some of the chains of assignments.

Using the solution: step 4, present a script that defines the functions, then invoke the "answer function" repeatedly to handle various versions of the problem

A script that uses functional composition (chaining), and its use

Begin function definitions

\[
\text{totalCostFunc} := (\text{convRate}, \text{costInEuros}, \text{pkgCost}) \rightarrow (\text{pkgCost} + \text{convRate} \cdot \text{costInEuros})
\]

\[
\text{(convRate, costInEuros, pkgCost)} \rightarrow \text{pkgCost} + \text{convRate} \cdot \text{costInEuros}
\] \hspace{1cm} (8.20)

\[
\text{priceFunc} := (\text{markupPct}, \text{totalCost}) \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\]

\[
\text{(markupPct, totalCost)} \rightarrow (1 + \text{markupPct}) \cdot \text{totalCost}
\] \hspace{1cm} (8.21)

\[
\text{sellingPriceFunc} := (\text{convRate}, \text{costInEuros}, \text{pkgCost}, \text{markupPct}) \rightarrow \text{priceFunc}(\text{markupPct}, \text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}))
\]

\[
\text{(convRate, costInEuros, pkgCost, markupPct)} \rightarrow \text{priceFunc}(\text{markupPct}, \text{totalCostFunc}(\text{convRate}, \text{costInEuros}, \text{pkgCost}))
\] \hspace{1cm} (8.22)

End function definitions

Problem solving

Version 1

On November 1, 2002, one Euro was worth \(\frac{1}{9971}\) = 1.002908434 US dollars. We are buying widgets that cost 30 Euros each and importing them into the US. We then put the widgets into packages that cost .075 dollars. How much should we sell the product in the US if we want a 10% markup from our costs?
8.4 Expressions with units of measurements: convert

Maple has facilities for converting between various English and metric units. It is useful for doing multi-step calculations because the conversions happen automatically.

In the first way of using convert, one thinks of a value as implicitly expressing a number of units and wants another number expressing those number of units converted to another unit. One uses convert( value, units, fromUnit, toUnit).

Table 8.1: Examples of Unit Conversion

<table>
<thead>
<tr>
<th>Examples of unit conversion</th>
<th>Note that the answer to this was expressed as a floating point number rather than a fraction because the input was floating point (1.0)</th>
</tr>
</thead>
</table>
| How many inches in a meter? How many feet in a kilometer? How many millimeters in a mile? | convert(1, units, inch, meter) \[
\begin{align*}
127 \\
5000
\end{align*}
\] (8.26) |
| convert(1, units, ft, km) | \[
\begin{align*}
381 \\
1250000
\end{align*}
\] (8.27) |
| convert(1.0, units, mile, mm) | \[1.6093440 \times 10^6\] (8.28) |
### Examples of unit conversion

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert(5.4, units, kilowatt, horsepower)</code></td>
<td><code>7.241519284</code> (8.29)</td>
</tr>
<tr>
<td><code>convert(2.0, units, angstroms, micrometers)</code></td>
<td><code>0.0002000000000</code> (8.30)</td>
</tr>
<tr>
<td><code>convert(15.0, units, miles, meters, 'second')</code></td>
<td><code>6.705600000</code> (8.31)</td>
</tr>
<tr>
<td><code>convert(13.3, units, gallons, liters, meter^3)</code></td>
<td><code>65.85005144</code> (8.32)</td>
</tr>
</tbody>
</table>

Maple can convert between most compatible units.

Sometimes units are expressed as ratios of other units. Maple can handle such conversions as well.

In many examples in the Maple documentation, some of the arguments to `convert` are quoted -- surrounded by apostrophes -- to prevent evaluation from using the value of the names of the units. For example, if you have assigned a value to the variable `s`, then you cannot convert to seconds with this name without quotation.

### Troubleshooting unit conversion

#### 1. `seconds := convert(3, units, days, seconds)`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>259200</code></td>
<td>(8.33)</td>
</tr>
</tbody>
</table>

As long as the various names used as arguments to the `convert` function don't have values, things work fine.

#### 2. `seconds2 := convert(4, units, minutes, seconds)`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>240</code></td>
<td>(8.34)</td>
</tr>
</tbody>
</table>

Maple performs evaluation of names as it figures out what the inputs to `convert` is. Since `seconds` has a value, Maple tries to compute `convert(4, units, minutes, 259200)`. Since the 4th argument to `convert` has to be a name, an error results.

#### 3. `seconds2 := convert(4, units, minutes, 'seconds')`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>240</code></td>
<td>(8.34)</td>
</tr>
</tbody>
</table>

Quoting the 4th argument causes the name `seconds` to be given as the 4th input to `convert`. This works.

#### 4. `seconds3 := convert(5, 'units', 'hours', 'seconds')`

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>18000</code></td>
<td>(8.35)</td>
</tr>
</tbody>
</table>

Quoting all the names as a prophylactic measure is acceptable. You see this in a lot of the Maple on-line documentation.

In *Star Wars Episode IV: A New Hope*, Han Solo says that the Millennium Falcon made the Kessel Run in "less than twelve parsecs". We want to know how many days a parsec is.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert(12.0, units, parsecs, days)</code></td>
<td><code>2.300821388 10^14</code> (8.36)</td>
</tr>
</tbody>
</table>

A parsec is a non-fictional unit of distance, not time, so we can convert 12 parsecs to miles, kilometers, inches... But we can't convert it to days any more than we can convert inches to volts.

#### A problem solved, a script built using function definitions

A car travels 45 miles per hour. How many minutes does it take to travel 900 kilometers?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>We build a sequence of calculations to understand how to solve this problem. This is the informal phase of development, while we are trying to understand what to do. Once we have an idea, we start designing functions and testing them.</code></td>
<td></td>
</tr>
</tbody>
</table>
A problem solved, a script built using function definitions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance := 900.0</td>
<td>900.0 (8.37)</td>
</tr>
<tr>
<td>speed := 45</td>
<td>45 (8.38)</td>
</tr>
<tr>
<td>( d := \text{convert}(\text{distance, units, kilometers, miles}) )</td>
<td>559.2340730 (8.39)</td>
</tr>
<tr>
<td>( t := \frac{d}{\text{speed}} )</td>
<td>12.42742384 (8.40)</td>
</tr>
<tr>
<td>( \text{convert}(t, \text{units,}'hours','minutes') )</td>
<td>745.6454304 (8.41)</td>
</tr>
</tbody>
</table>

It would pretty obvious how to make a script out of this to handle any problem of the form: A car travels \( \text{speed} \) miles per hour. How many minutes does it take to travel \( \text{distance} \) kilometers... With a few user defined functions, we can get the answer with less typing/cutting/pasting:

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d\text{Convert} := (\text{distance}) \rightarrow \text{convert}(\text{distance, units, kilometers, miles}) )</td>
<td>( \text{distance} \rightarrow \text{convert}(\text{distance, units, kilometers, miles}) ) (8.42)</td>
</tr>
<tr>
<td>( d\text{Convert}(900) )</td>
<td>( \frac{781250}{1397} ) (8.43)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t\text{Calc} := (d, \text{speed}) \rightarrow \frac{d}{\text{speed}} )</td>
<td>( (d, \text{speed}) \rightarrow \frac{d}{\text{speed}} ) (8.44)</td>
</tr>
<tr>
<td>( t\text{Calc}(1.4.18, 45) )</td>
<td>( \frac{156250}{12573} ) (8.45)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t\text{Conv} := (t) \rightarrow \text{convert}(t, \text{units,}'hours','minutes') )</td>
<td>( t \rightarrow \text{convert}(t, \text{units,}'hours','minutes') ) (8.46)</td>
</tr>
<tr>
<td>( t\text{Conv}(1.4.15) )</td>
<td>( \frac{3125000}{4191} ) (8.47)</td>
</tr>
</tbody>
</table>

The first step was to convert the distance from kilometers to miles. We create a function that does it.

We test it to the result that we got in the script above and see that it agrees.

The next step was to calculate the time (in hours) from the distance in miles and the speed in mph.

The test shows that \( t\text{Calc} \) seems to be built correctly.

The third step was to convert the time from hours to minutes.

The test of this step agrees with what the script does, too.
A problem solved, a script built using function definitions

\[
\text{solveIt} := (\text{speed}, \text{distance}) \\
\rightarrow \text{tConv}(\text{tCalc}(d\text{Convert}(\text{distance}), \text{speed})) \\
\rightarrow \text{tConv}(\text{tCalc}(d\text{Convert}(\text{distance}), \text{speed}))
\]

\[\text{solveIt}(45, 900.0) = 745.6454304 \quad (8.49)\]

The solution function chains together the three functions we've developed. This concludes the development and testing. We present a script and several solved problems in the figure below.

The above table showed the thinking behind the design and testing of the multi-step calculation. However, in "what we would hand in", we don't show the testing or the initial script, just the definition of the functions, and then the repeated invocation of the "solution function". Define the functions once, then invoke it repeatedly. This eliminates the need for repeated cutting/pasting/selection for execution.

Solving multiple versions of a problem through functions

Begin function definitions

\[
d\text{Convert} := (\text{distance}) \rightarrow \text{convert}(\text{distance}, \text{units}, \text{kilometers, miles})
\]

\[
\text{distance} \rightarrow \text{convert}(\text{distance}, \text{units}, \text{kilometers, miles})
\]  
\[\text{dConvert} := (\text{distance}) \rightarrow \text{convert}(\text{distance}, \text{units}, \text{kilometers, miles}) \quad (8.50)\]

\[
t\text{Calc} := (\text{d, speed}) \rightarrow \frac{\text{d}}{\text{speed}}
\]

\[
(\text{d, speed}) \rightarrow \frac{\text{d}}{\text{speed}}
\]  
\[\text{tCalc} := (\text{d, speed}) \rightarrow \frac{\text{d}}{\text{speed}} \quad (8.51)\]

\[
t\text{Conv} := (\text{t}) \rightarrow \text{convert}(\text{t, units, 'hours', 'minutes'})
\]

\[
\text{t} \rightarrow \text{convert}(\text{t, units, 'hours', 'minutes'})
\]  
\[\text{tConv} := (\text{t}) \rightarrow \text{convert}(\text{t, units, 'hours', 'minutes'}) \quad (8.52)\]

\[
\text{travelSoln} := (\text{speed, distance}) \\
\rightarrow \text{tConv}(\text{tCalc}(d\text{Convert}(\text{distance}), \text{speed}))
\]

\[
(\text{speed, distance}) \\
\rightarrow \text{tConv}(\text{tCalc}(d\text{Convert}(\text{distance}), \text{speed}))
\]  
\[\text{travelSoln} := (\text{speed, distance}) \rightarrow \text{tConv}(\text{tCalc}(d\text{Convert}(\text{distance}), \text{speed})) \quad (8.53)\]

End of function definitions

Problem version A

A car travels at 45 miles per hour. How many minutes does it take to travel 900 kilometers?

\[\text{travelSoln}(45, 900.0) = 745.6454304 \quad (8.54)\]

Problem version B

A car travels at 45 miles per hour. How many minutes does it take to travel 452 kilometers?

\[\text{travelSoln}(45, 452.0) = 374.4797052 \quad (8.55)\]
Solving multiple versions of a problem through functions

<table>
<thead>
<tr>
<th>Problem version C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A car travels at 65 miles per hour. How many minutes does it take to travel 1500 kilometers?</td>
</tr>
</tbody>
</table>

\[
\text{travelSolv}(65, 1500.0) = 860.3601126
\]

For casual unit conversion, it can still be useful to rely upon Maple's encyclopaedic knowledge of how to convert units. You can access this through Tools->Assistants->Unit Calculator

Table 8.2: Unit Converter Assistant

![Image of Unit Converter Assistant]

8.5 Inputs and outputs to user-defined functions don't have to be numbers

Although you don't see much mention of this in mathematics texts, it is fairly common while programming to define and use functions that take inputs and produce outputs that are not numbers. For example, if we have a list \( L \) of numbers, we can create a function that takes a list as input and produces the average of all the numbers as its output.

Table 8.3: A function that takes a list as its input

<table>
<thead>
<tr>
<th>A function that takes a list as its input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
</tr>
</tbody>
</table>
| \[
\text{average} := L \rightarrow \frac{\sum_{i=1}^{\text{nops}(L)} L[i]}{\text{nops}(L)}
\] | We are expecting the input to be a list of numbers. As explained in chapter 4, \( L[i] \) uses indexing to get the \( i \)-th value of the list. (See section 4.2.) \( \text{nops}(L) \) is the number of elements in the list. |
| \[
\text{average}([5, 7, -3, 2, 6])
\] | \[
\frac{17}{5}
\] | When we invoke the function, \( L \) is \([5, 7, -3, 2, 6]\), so \( \text{nops}(L) \) is 5. Since at least one of the elements of the list was a limited precision number (5. has a decimal point), the limited precision arithmetic is performed with it and subsequent steps of the sum. |
| \[
\text{average}([5, 7, -3, 2, 6, 2, 2, 6])
\] | \[
3.375000000
\] | (8.59) |
In analyzing mathematical models as we have been doing, it is also useful to produce abbreviations for common combinations of plot options by creating a function that produces a plot as its result.

**Table 8.4: A function that returns a plot as its output, rather than a number**

<table>
<thead>
<tr>
<th>A function that returns a plot as its output, rather than a number</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td>We are given two lists of data, ( pData ) and ( tData ). Plot ( pData ) as a function of ( tData ), and vice versa.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td>Build a function that becomes an abbreviation for the operations in the plot. Provide a third argument that is the string for the color.</td>
</tr>
<tr>
<td>( PlotIt := (xData, yData, c, L) \rightarrow plot(xData, yData, style = point, color = c, labels = L) )</td>
</tr>
<tr>
<td>( PlotIt := (xData, yData, c, L) \rightarrow plot(xData, yData, style = point, color = c, labels = L) ) (8.60)</td>
</tr>
<tr>
<td>( pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2] );</td>
</tr>
<tr>
<td>( pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2] ) (8.61)</td>
</tr>
<tr>
<td>( tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3] );</td>
</tr>
<tr>
<td>( tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3] ) (8.62)</td>
</tr>
</tbody>
</table>

Plot pressure versus temperature, in red. Note: there seems to be a bug in Maple that suppresses the printing of the horizontal axis label.

\( PlotIt(pData, tData, "red", ["pressure", "temperature"]) \)

Plot temperature versus pressure, in red. Copying and pasting the invocation of the \( PlotIt \) function we defined is easier than changing the insides of the original plot operation.
A function that returns a plot as its output, rather than a number

PlotIn(tData, pData, "blue", ["temperature", "pressure"])

It is possible to return a list or sequence as a result of a function. Such a function can be put in a chain.

Table 8.5: A problem solved with a function that outputs a sequence of two numbers

<table>
<thead>
<tr>
<th>A problem solved with a function that outputs a sequence of two numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem A</strong></td>
</tr>
<tr>
<td>A right triangle has a hypotenuse of length 5 feet. The angle between it and one of its sides is 10 degrees. What is the length of the perimeter?</td>
</tr>
<tr>
<td>We first build a function that computes the two sides of the right triangle and returns the two values as a sequence. We have to convert degrees into radians in order to do this because the Maple trig functions all use radians.</td>
</tr>
</tbody>
</table>
| \[
\text{sideSide} := (\text{hypo, angle}) \rightarrow (\text{hypo} \cdot \sin(\text{convert(angle degrees, radians)}), \text{hypo} \cdot \cos(\text{convert(angle degrees, radians)}))
\] |
| (8.63) |
| Let's test the sideSide function. |
| \[
\text{sideSide}(5, 10.0)
\] |
| \[
5 \sin(0.055555555556 \pi), 5 \cos(0.055555555556 \pi)
\] |
| (8.64) |
| Now, develop a function that takes a sequence of three numbers and adds them together. |
| \[
\text{sumSides} := (a, b, c) \rightarrow a + b + c
\] |
| (8.65) |
| By chaining together the output of sideSide and making it part of the input of sumSides, we can get the whole computation done in one function. |
A problem solved with a function that outputs a sequence of two numbers

\[
\text{perimeter} := (\text{hypo}, \text{angle}) \rightarrow \text{sumSides}((\text{sideSide}(\text{hypo}, \text{angle})), \text{hypo})
\]

(8.66)

\[
\text{perimeter}(5, 10)
\]

\[
5 \sin\left(\frac{1}{18} \pi\right) + 5 \cos\left(\frac{1}{18} \pi\right) + 5
\]

at 5 digits

10.792

(8.67)

Problem B

A right triangle has a hypotenuse of length 10 feet. The angle between it and one of its sides is 42 degrees. What is the length of the perimeter?

\[
\text{evalf}(\text{perimeter}(10, 42))
\]

24.12275432

(8.68)

Once we have done the work to design and test the functions out on a problem, we can present a script that can solve several different versions of the problem:

Solving several versions of a function with function definitions

Begin function definitions

A function that computes the two sides of a right triangle given the angle and the length of the hypotenuse

\[
\text{sideSide} := (\text{hypo}, \text{angle}) \rightarrow (\text{hypo} \cdot \sin(\text{convert}((\text{angle} \cdot \text{degrees}, \text{radians}))), \text{hypo} \cdot \cos(\text{convert}((\text{angle} \cdot \text{degrees}, \text{radians}))))
\]

(8.70)

A function that takes a sequence of three numbers and adds them together.

\[
\text{sumSides} := (a, b, c) \rightarrow a + b + c
\]

(8.71)

Compute the perimeter by summing the three sides.

\[
\text{perimeter} := (\text{hypo}, \text{angle}) \rightarrow \text{sumSides}(\text{sideSide}(\text{hypo}, \text{angle})), \text{hypo})
\]

(8.72)

End function definitions

Problem A Solution

A right triangle has a hypotenuse of length 5 feet. The angle between it and one of its sides is 10 degrees. Approximately, what is the length of the perimeter in feet?

\[
\text{evalf}(\text{perimeter}(5, 10))
\]

10.79227965

(8.73)
Solving several versions of a function with function definitions

Problem B Solution

A right triangle has a hypotenuse of length 10 feet. The angle between it and one of its sides is 42 degrees. Approximately, what is the length of the perimeter in feet?

`evalf(perimeter(10, 42))`

24.12275432  

(8.74)

8.6 Chapter Summary

Function design

Designing functions from context

a) What will the inputs be?

b) What will the output be?

c) How do we calculate the output from the inputs?

Function composition

\[ A := (x, y) \rightarrow \frac{1}{x} + 3x^2 + 3 \cdot y \]

\[ (x, y) \rightarrow \frac{1}{x} + 3x^2 + 3y \]  

(8.75)

\[ B := (x, y) \rightarrow \frac{3}{A(x, y)} \]

\[ (x, y) \rightarrow \frac{3}{A(x, y)} \]  

(8.76)

\[ B(3, 1) \]

\[ \frac{9}{253} \]  

(8.77)

Unit conversion

Units can be converted directly into compatible units.

\[ convert(1, \text{units}, \text{inch}, \text{meter}) \]

\[ \frac{127}{5000} \]  

(8.78)

Compound units expressed in ratio form can also be converted into compatible compound units.

\[ convert(15.0, \text{units}, \text{miles} \text{hour}^{-1} \text{meters} \text{second}) \]

6.705600000  

(8.79)

Converting between incompatible units will generate an error message.

\[ convert(3, \text{units}, \text{days}, \text{miles}) \]

Error, (in convert(units)) unable to convert ‘d’ to ‘mi’

If we create a variable with the same name as a unit, trying to convert using the variable name will throw an error.

\[ \text{seconds} := 5 \]

\[ 5 \]  

(8.80)

\[ convert(3, \text{units}, \text{days}, \text{seconds}) \]

Error, (in convert(units)) unable to convert ‘d’ to ‘s’
### Unit conversion

<table>
<thead>
<tr>
<th>convert(3, units, 'days', 'seconds')</th>
</tr>
</thead>
<tbody>
<tr>
<td>259200</td>
</tr>
</tbody>
</table>

We can use Maple's built-in unit converter to convert units using drop-down menus. This tool is located in Tools>Assistants>Unit Calculator.

### Non-number inputs and outputs of a function

#### Using a list of numbers as an input

Average of a list of numbers:

\[
\text{average} := L \mapsto \frac{\sum_{i=1}^{\text{nops}(L)} L[i]}{\text{nops}(L)}
\]

\[
\text{average}([5, 7, -3, 2, 6])
\]

\[
\frac{17}{5}
\]

#### Returning a plot instead of a number

Plot function:

\[
\text{Plot} := (xData, yData, c, L) \mapsto \text{plot}(xData, yData, \text{style} = \text{point}, \text{color} = c, \text{labels} = L)
\]

\[
\text{Plot}([1, 2, 3], [2, 1, 2, 1], \text{blue}, ['x', 'y'])
\]

![Plot of a function](image)

#### Returning a sequence of numbers instead of a single number

Side of a right-angled triangle:

\[
\text{sideSide} := (\text{hypo}, \text{angle}) \mapsto (\text{hypo} \cdot \sin(\text{convert(\text{angle}, \text{degrees}, \text{radians})}), \text{hypo} \cdot \cos(\text{convert(\text{angle}, \text{degrees}, \text{radians}))))
\]

\[
\text{sideSide}(1, 45)
\]

\[
\frac{1}{2} \sqrt{2}, \frac{1}{2} \sqrt{2}
\]
9 Chapter 9 Visualization, modeling, and simulation

9.1 Chapter Overview

1. *Plot structures* are the result of the *plot* operation. They can be assigned to variables through := just as numbers, formulas, lists, and function definitions, can be.

2. The *display* operation of the *plots* package allows you to combine together plots. Often times a single picture can be more enlightening or easier to understand than looking at multiple pictures separately.

3. *Simulation* is the art of predicting the behavior of system entities as they change over time, through the use of mathematical models. It can be as simple as using functions that, given the time \( t \) as input, calculate the position, size, weight, or other changing properties of a situation. With the appropriate mathematics, personal computer or supercomputer-class calculations can be used to come up with reasonably accurate descriptions of phenomena. Computational simulations have become a mainstay of modern engineering because the "build it and see" methodology often seen in elementary student work is not cost-effective once one moves about beyond simple scenarios that are inexpensive to test.

4. The *animate* operation of the *plots* package is explained. Its use is illustrated with a session of question-answering using a mathematical models of moving bodies. Computer-generated animations are another useful tool besides *solve*, and *plot*.

9.2 plot structures

Like *solve*, Maple plot is a function: it has inputs and produces outputs. What kind of output does the plot function produce? In Maple, the result of *plot* is a special type of result called a *plot structure*. When you evaluate an expression in Maple that invokes the plot function, a plot structure is created. If the plot structure is then assigned to a variable (through :=, for example), then an ellipsis of the plot structure is displayed. If the plot structure is the entire result and there is no assignment, then the plot is displayed.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plot(sin(x), x = 0..10)</code></td>
<td>The result of this operation is a plot structure. Since it is the entire result and is not being assigned to a variable through :=, Maple displays the structure in a pretty way -- by drawing a picture of all the points and labels established by the plot operation and put into the plot structure.</td>
</tr>
<tr>
<td><code>p := plot(cos(x), x = 0..10)</code></td>
<td>Evaluating an expression and then assigning it to the name <code>p</code>. This does not display the plot. We just see <code>PLOT(...)</code> which is a sign that the value of <code>p</code> is a plot structure.</td>
</tr>
</tbody>
</table>
CommentaryExample

Evaluating an expression -- just \( p \) -- causes the result (the plot) to be displayed.

The value of assigning a plot to a variable is explained in the next example, which uses another Maple operation, `display`, to combine plots together.

---

**9.3 plots[display] and combining plots**

The display function from the plots package takes as its first argument a list of plot structures. It will produce a plot structure that combines all the plots together. `display` is the way to get a multi-plot in a script without doing cutting and pasting of plots.

**Table 9.2: Display combines plots**

<table>
<thead>
<tr>
<th>display combines plots</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeData := [4.5, 5.5, 6, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10, 10.5, 11, 11.5, 12, 12.5, 13]</td>
<td>Someone we know has used a variant on LeastSquares Curve Fitting, to derive a formula with that fits the data. We wish to plot both the data points and the formula on a single graph.</td>
</tr>
<tr>
<td>tempData := [58, 57.5, 57, 56, 55.5, 54, 53, 52, 51, 50, 49, 48.5, 48.5, 47.5, 47.5, 46, 45.5, 45]</td>
<td>(9.2)</td>
</tr>
<tr>
<td>[58, 57.5, 57, 56, 55.5, 54, 53, 52, 51, 50, 49, 48.5, 48.5, 47.5, 47.5, 46, 45.5, 45]</td>
<td>(9.3)</td>
</tr>
</tbody>
</table>
This is the point plot we get from this data.

\[
p1 := \text{plot}(\text{timeData}, \text{tempData, style = point, color = "red"})
\]

Let's assume that we have a friend who has done curve fitting to \(\text{timeData}\) and \(\text{tempData}\) and gotten this formula. In ENGR 101 Fall 2010, Matlab was used to do exponential curve fitting like this. We could replication that calculation in Maple, but don't show it here. If you're interested in seeing how to do it, ask your instructor about it.

\[
\text{formula} := 25.0 + 33.61425396 e^{-0.06172329065 t + 0.2777548079}
\]

We load the plots package. Because we ended the line with a colon (:), the list of functions in the package is suppressed. In general, ending a line with a colon suppresses the normal output.
9.4 plottools, lines, and other shapes

The plottools package has a number of functions that are useful for inclusion in visualizations (plots). For example, you can create a line segment of any desired color and line thickness with the `line` function.

One can then use the `display` function to merge together lines, plots, and other shapes.

The on-line documentation on plottools contains links to further description.

Table 9.3: plottools: lines and other shapes: a frivolous drawing

We create a bit of "modern art" by drawing a circle and a point plot.

```maple
with(plottools);
[arc, arrow, circle, cone, cuboid, curve, cutin, cutout, cylinder, disk, dodecahedron, ellipse, ellipticArc,
  hemisphere, hexahedron, homothety, hyperbola, icosahedron, line, octahedron, parallelepiped,
  pieSlice, point, polygon, project, rectangle, reflect, rotate, scale, semitorus, sphere, stellate,
  tetrahedron, torus, transform, translate]
```

As the on-line documentation indicates, the first two arguments to `line` are lists indicating the coordinates of the starting point and ending point of the line segment. There are optional arguments that indicate color and line thickness, etc.

\[
I_1 := \text{line}([[0, 1], [1, 1], \text{color} = \text{"orange"}]);
\]

\[
\text{CURVES}([[0., 1.], [1., 1.], \text{COLOUR}(\text{RGB}, 0.80000000, 0.19607843, 0.19607843))}
\]

\[
I_2 := \text{line}([[1, 1], [1, 2], \text{color} = \text{"blue"}, \text{thickness} = 20])
\]

\[
\text{CURVES}([[1., 1.], [1., 2.], \text{COLOUR}(\text{RGB}, 0., 0., 1.00000000), \text{THICKNESS}(20))}
\]
We suppress printing of the plot structure CURVES(....) for c3 with a colon because it's long and we don't want to see it, we want to see the picture it describes. Note that Maple exposes its Canadian roots by using "colour".

\[ c3 := \text{circle([2, 2.5], 3, color = "Purple", thickness = 5)} : \]

\[ \text{with(plots)} : \]

\[ \text{display(\{11, l2, c3, plot(2 \cdot r^2, t = 0 .. 2, style = point, symbol = circle, symbolsize = 30)\})} \]

One of the options to plot (and display) is to not show the axes. Another option is to indicate that the scaling should be constrained to be equivalent in both horizontal and vertical directions (to make the circle look like a circle). That gives us an unframed work of art!

\[ \text{display(\{11, l2, c3, plot(2 \cdot r^2, t = 0 .. 2, style = point, symbol = circle, symbolsize = 30)\}, axes = none, scaling = constrained)} \]

We can use lines and circles for less frivolous purposes, too.
Table 9.4: Combining a line with other plots using display

We create a plot structure that is a green line segment running between the points (12,25), and (12, 58).

We can create a picture with the data plot and the exponential curve plot of the previous section, plots and the vertical green line. This highlights the value of the curve at t=12 minutes.

9.5 Mathematical models and simulation

Mathematical models try to describe a "real" situation in terms of equations and formulae. The point of modeling is to try, through mathematical or computational means, to determine what will happen without having to run experiments in the "real" situation. As Dr. Jay Brockman of the University of Notre Dame says in his book Introduction to Engineering:

Some engineering students have been fortunate enough to participate in pre-engineering programs such as the first LEGO(TM) League robotics design competition or American Society of Civil Engineering bridge-building contests. In addition to fostering creative problem-solving skills, such projects also introduce students to the important notion that seemingly good ideas don't always work out in practice. Often in such programs, students have ample opportunity to test and modify their designs before they formally evaluate them. If the design doesn't work, then like a sculptor working with clay, the designer adds something here or removes something there until the design is acceptable.

This cut-and-try methodology is also sometimes used in industry, particularly in circumstances where the design is simple, or where, the risk or cost of failure is low. In many situations, however, there is no second chance in the event of failure. For engineering systems such as buildings, bridges, or airplanes -- top name just a few -- failure to meet specifications could mean a loss of life. For others -- such as the integrated circuit chip -- the cost of fabrication is so high that a company may not be able to afford a second chance. In these situations, it's critical for the engineering team to be highly confident that a design will be acceptable before it's built. To do this, engineers use models to predict the behavior of their designs. A model is an approximation to a real system, such that when actions are performed on the model, it will respond in a manner similar to the real system. Models can have many different forms, ranging from physical prototypes such as a crash-test dummy to complex computer simulations.

We can think of a mathematical model as a kind of virtual system... whose input is a set of variables that represent either aspects of the design or aspects of the environment, and whose output is a set of variables that represent the behavior of the system. Inside is a set of mathematical relationships that describe the operation of the system.
The point of expressing a situation mathematically is to use mathematics and computation to better understand the situation. Usually we are given or derive a formulas that allow us to calculate key properties of the system. For models involving only a few variables, this can involve the following kinds of actions:

1. Get a single number, by evaluating a formula or function.
2. Get a single number, by solving an equation.
3. Gain an understanding of a relationship between one or more entities of interest and the "input variables" by producing a formula.
4. Gain a visual understanding of the relationship by plotting a function, or possibly several plots merged together.
5. Gain an understanding of how a a system changes over time. Rather than computing the value of a variable once, we repeatedly compute the value of the variable at several different points at time. This is called computational simulation of the system. We can view plots changing over time by producing an animation.

### Examples of the first four types of computation

#### Third type: producing a formula

In the temperature-pressure data fitting example (page 99) the mathematical model is the formula that expresses the relationship between temperature and pressure. We had only data and no formula to begin with, but we developed the formula using the CurveFitting[LeastSquares] operation. This was a computation of the third type mentioned above.

#### First type: Evaluating a formula or function

Once we had the formula, we used the relationship to calculate pressure at several given temperatures. This was a computation of the first type mentioned above.

#### Second type: solving an equation to get a desired value

We also found a temperature corresponding to a specified pressure by using solve -- a computation of the second kind mentioned above.

#### Fourth type: visualization (plotting)

In the example with an exponential curve fit (page 122), we got a visual impression of how the formula fit the data by combining the point plot of the data and the plot of the formula together -- the fourth kind of computation mentioned above.

We haven't explained how to do the fifth type of computation -- animation -- yet. This will be discussed in upcoming section Animations (movies) using animate (page 129)

### 9.6 Drawing x-y position as a function of time through parameterized plots

The mathematical models often describe the position of a system entity as a function(s) of time. If the entity's position is two dimensional, then we have two functions, often called $x(t)$ and $y(t)$. We can generate a plot of position for various values of $t$ with a special form of plot.

```plaintext
plot( [ x-position expression, y-position expression, var = low..high], plot options)
```

will draw a two dimensional graph connecing the $(x,y)$ points traced out for the values of the expression as the variable $var$ takes on values between $low$ and $high$.

### Table 9.5: Examples of plots where the x and y positions are parameterized by t

<table>
<thead>
<tr>
<th>Examples of plots where the x and y positions are parameterized by t</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example 1</strong></td>
</tr>
</tbody>
</table>

In this example, we describe the $x$ and $y$ positions as periodic functions of time. Every $2\pi$ time units, the positions return back to where they were originally.
Examples of plots where the x and y positions are expressions parameterized by t

\[ \text{xpos} := (t) \rightarrow 3 \cdot \cos(t) \]
\[ t \rightarrow 3 \cos(t) \quad (9.11) \]

\[ \text{ypos} := (t) \rightarrow 2 \cdot \sin(t) \]
\[ t \rightarrow 2 \sin(t) \quad (9.12) \]

We plot the position using parameterized plotting. By default the scaling used for the horizontal and vertical axes are different, but in this case the default makes the graph look misleadingly like a circle when it should look more like an ellipse. To compensate, we use the `scaling = constrained` option to `plot`.

```plaintext
plot([xpos(t), ypos(t), t = 0 .. 2*Pi], scaling = constrained)
```

Example 2

In this example, we have parameterized expressions for an object shot out of a cannon with horizontal velocity 10 feet/second and vertical velocity 10 feet/second minus the acceleration due to gravity.

\[ \text{xpos2} := (t) \rightarrow 10 \cdot t \]
\[ t \rightarrow 10 \cdot t \quad (9.13) \]

\[ \text{ypos2} := (t) \rightarrow 10 \cdot t - \frac{32 \cdot t^2}{2} \]
\[ t \rightarrow 10 \cdot t - 16 \cdot t^2 \quad (9.14) \]
Examples of plots where the $x$ and $y$ positions are expressions parameterized by $t$

```
plot([xpos2(t), ypos2(t), t = 0 .. .5], scaling = constrained, color = "blue", labels = ["x", "y"])
```

9.7 A first animation example using animate, animation controls

Let's look an animation. First we have to create it. We can use the `animate` function of the `plots` package. For the time being, let's not worry the details of why the operation is entered the way it is. Rather we focus on what the animation is trying to do, and how to view it in Maple once it has been created.

**Table 9.6: First animation example**

This produces a movie of a point moving through the points $(0,0), (1, .01), (2, .04)$, etc. up to $(10, 100)$.

`with(plots) :`
If we click on the plot, the Maple tool bar changes and shows us *animation controls*.

**Table 9.7: Animation controls**

<table>
<thead>
<tr>
<th>Text</th>
<th>Math</th>
<th>Drawing</th>
<th>Plot</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="pause.png" alt="Pause" /></td>
<td><img src="play.png" alt="Play" /></td>
<td><img src="current-frame.png" alt="Current Frame" /></td>
<td><img src="frame.png" alt="Frame" /></td>
<td><img src="animation.png" alt="Animation" /></td>
</tr>
</tbody>
</table>

These controls are highly similar to video playback controls found in many applications (e.g. You Tube), so we won't discuss at length here. Note that using them you can:

1. Start playing the animation.
2. Stop the animation.
3. Display only a particular frame, "frozen".
4. Control the number of frames per second it plays.
5. Set it to play once or continually repeat in a loop.

See Graphics->Animation->Animation Toolbar under the Table of Contents of the on-line Maple help.

Right-clicking on the animation will also produce a menu of operations that provide an alternative for controlling the animation.

```maple
animate(plot, [[t], [t^2], style = point, symbolsize = 30, color = "Purple"], t = 0..10)
```
If we play the animation, we will see a point move upwards in a parabolic path:

### Frames 1, 5, 10, 15, 20, and 25 of the animation

- **Frame 1**: $t=0$.
- **Frame 5**: $t=1.6667$.
- **Frame 10**: $t=3.3333$.
- **Frame 15**: $t=5.0000$.
- **Frame 20**: $t=6.6667$.
- **Frame 25**: $t=8.3333$.
Frames 1, 5, 10, 15, 20, and 25 of the animation

9.8 The first animate example, part 2

Now that we've gotten the general idea of what animate's results are like, let's look again at what was entered and look at the details of what was computed. The general form of the operation is:

<table>
<thead>
<tr>
<th>General form of the animate operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>with(plottools):</td>
</tr>
<tr>
<td>animate(plotting function, [ parameters to plotting function], time variable = time range)</td>
</tr>
</tbody>
</table>

What this does is to apply the plotting function to the parameters for various values of the time variable. Each value of the time variable used is a separate frame of the movie.

In the previous section we used:

\[
\text{animate}\left(\text{plot}, [t, t^2], \text{style = point}, \text{symbolsize = 30}, \text{color = "Purple"}, t = 0..10\right)
\]

In the first frame of the movie, it used the value \(t=0\) for the time variable \(t\). Thus it plotted

\[
\text{plot}([0],[0], \text{style = point}, \text{symbolsize = 30}, \text{color = "Purple"});
\]

plots a small circle at the coordinate (0,0).

The next time value used is \(t=0.4667\). The second frame of the movie is the result of

\[
\text{plot}([0.41667],[0.41667^2], \text{style = point}, \text{symbolsize = 30}, \text{color = "Purple"});
\]

which plots a small circle at the coordinate (0.4167, 0.1736138889).
9.9 Designing a function to use in animate

In this section, we explore the design of a function that we use with animate to create an animation.

Problem

Create an animation of a circle whose radius expands from $r=1$ at time $t=0$ to $r=5$ at time $t=10$.

Solution

The actions we create are a short script that defines a function $\text{drawCircle}(t)$. Then it uses it in animate. While we there are other ways to use animate other than this style, we choose to do things this way because the $\text{drawCircle}$ function allows us to test the plotting one frame at a time before we try to create the animation.

Solution, part 1

```maple
with(plottools):
with(plots):

display([[circle([0,0], 1, color=\text{"red"})]])
```

While this seems to do the job that we wish for the starting frame, we have to consider how this will vary as $t=0$ to $t=10$. We want the radius of the circle to be 1 when $t=0$ (as in the figure above), and 5 when $t=10$. We can do some line fitting to come up with the formula if we can't remember enough high school algebra to figure it out ourselves:

Figuring out the formula for the radius as a function of time

$$\text{rad}(t) := \text{CurveFitting}[[\text{LeastSquares}][[[0, 10], [1, 5]], t]]$$

$$1 + \frac{2}{5} t$$

(9.15)

```maple
with(plots):

We plot the formula we get from the data fitting with the data points to convince ourselves that the line agrees exactly with the two points. We'd
get a similar result for curve fitting a line any time we fit a line with only two data points.

**Testing the solution function**

We can create a function that creates a circle of the appropriate size given a value for \( t \). 

\[
drawCircle := (t) \rightarrow \text{display} \left( \text{circle} \left( [0, 0], 1 + \frac{2}{5} \cdot t, \text{color} = \text{"red"} \right) \right)
\]

\[
t \rightarrow \text{plots:-display} \left( \text{plottools:-circle} \left( [0, 0], 1 + \frac{2}{5} \cdot t, \text{color} = \text{"red"} \right) \right)
\]

(9.16)
We try this function out at $t=0$.

The function evaluated at $t=5$ looks the same, but we see from the axis labels that it is actually a bigger circle.
Once we are convinced that the function `drawCircle` works for the range of values of $t$ that we need it to, we can use it in animate. The way we invoke `drawCircle` within `animate` is different than when we were using it one frame at a time. The name of the function is kept separate from the parameters, and the range. This delays the creation of any plot structures until `animate` starts computing them.

Complete script solving the "expanding circle animation" problem

```plaintext
with(plots):

with(plottools):

drawCircle := (t) -> display([circle([0, 0], 1 + \frac{2}{5} \cdot t, color = "red")])

t->plots:-display([plottools:-circle([0, 0], 1 + \frac{2}{5} \cdot t, color = "red")])
```

We see the first frame of the animation below. When we click on the animation, the animation toolbar appears and we can then play it. We see the circle expand over time. Unlike the individual plots we created during our testing, all the circles use the same scaling.
Some frames from the expanding circle animation

\[ \text{animate(drawCircle, } [t], t = 0..10) \]
9.10 Designing a more elaborate animation

Problem

A satellite is in a circular orbit around the earth, at a distance of five earth radii. Create an animation of it circling.

Solution — discussion

We will create a function \( \text{drawPlanetAndSat}(t) \) that for any time \( t \) draws both the Earth and the satellite at time \( t \). We browse through the on-line plottools package and discover the function \( \text{disk}(c,r, \text{color}=...) \) that draws a solid disk whose center is at the point \( c \), has radius \( r \), and specified color, in a fashion similar to the \( \text{circle} \) function we used in the Designing a function to use in \( \text{animate} \) (page 133). Browsing through on-line help for the color names known to plot reveals that one color is "DarkKhaki". We decide to draw the Earth as a disk of radius 1 centered at \((0,0)\).

### Drawing a disk

```maple
with(plots):
with(plottools):

display([disk([0, 0], 1, color = "DarkKhaki")])
```

We decide to represent the satellite as a point. Recall that \((\cos(t), \sin(t))\) describes circular motion moving around a circle of radius \( r=1 \). To parameterize orbital motion at radius \( 5 \), we use \((5\cos(t), 5\sin(t))\).

### Drawing a point at \((0,5)\)

```maple
with(plots):
with(plottools):
```
We can combine the two together using `display`.

**Drawing a disk, and a point at (0,5)**

```maple
with(plots) :

with(plottools) :

display([disk([0, 0], 1, color = "DarkKhaki"),
        plot([5*cos(0), 5*sin(0)], style = point,
             symbolsize = 20, color = "red")])
```

We invoke `display` with a list of two plot structures: the disk, and the point plot.

After we draw this we see the two plots together, but it isn't exactly what we want, because the horizontal and vertical scaling is not equal.
We include the plot option `scaling=constrained` as an extra argument to `display`.

We're almost done. As with the animation design of the previous section, we write a function of $t$ that describes the disk and the position of the point at time $t$.

### A function describing a frame to draw at time $t$

```maple
with(plots):

with(plottools):

drawEarthAndSat := (t) -> display([disk([0, 0], 1, color = "DarkKhaki"),
plot([5*cos(t)], [5*sin(t)], style = point, symbolsize = 20, color = "red"),
scaling = constrained])
```

\[(9.18)\]
We test this function out at a few values of $t$. The idea is that the satellite makes one orbit during the period $t=0 \cdot 2 \pi$.
We now can use this function with 

```maple
animate
to draw the orbiting satellite.
```

**Complete script to solve the "orbiting satellite animation problem"**

```maple
with(plots):
with(plots[plottools]):

\[
drawEarthAndSat := (t) \rightarrow display([disk([0, 0], 1, color = "DarkKhaki"), plot([5 \cdot \cos(t)], [5 \cdot \sin(t)], style = point, symbolsize = 20, color = "red"), scaling = constrained)]
\]

\[
t \rightarrow plots\cdot\text{display}([plottools\cdot\text{disk}([0, 0], 1, color = "DarkKhaki"), plot([5 \cos(t)], [5 \sin(t)], style = plottools\cdot\text{point}, symbolsize = 20, color = "red"), scaling = constrained])
\]

(9.19)
Problem

A boy throws a ball straight up in the air with an initial velocity of 15 miles per hour. Once released, the ball’s position is described by the function \( x(t) = v_0 t - \frac{1}{2} g t^2 \), where \( v_0 \) is the initial velocity, and \( g \) is the force of gravity, \( g = -\frac{32 \text{ feet}}{\text{sec}^2} \).

(a) How many seconds does the ball stay in the air?

(b) Generate an animation that shows the ball's motion. \( t \) should be measured in seconds, and position in feet from ground level. Assume that the ball starts at 0 feet altitude even though this would be a bit unrealistic for a boy to do unless he was standing in a pit!

(c) Use the animation to determine roughly when the maximum altitude is, and what that altitude is.
(d) Find how fast the initial velocity should be so that the ball goes up over 30 feet.

**Solution**

First convert 15 miles per hour into a velocity in feet per second using the convert function first described in the section on problem solving functions in Chapter 7 (page 102).

\[
v_0 := \text{convert} \left( 15, \text{units}, \frac{\text{miles}}{\text{hour}}, \frac{\text{feet}}{\text{second}} \right)
\]

\[22\]  
(9.20)

Define the gravitational constant

\[g := -32\]

\[-32\]  
(9.21)

Next, define the function for position.

\[x := (t) \rightarrow v_0 \cdot t + \frac{g \cdot t^2}{2}\]

\[t \rightarrow v_0 \cdot t + \frac{1}{2} \cdot g \cdot t^2\]  
(9.22)

Doing a rough plot of position versus time shows us roughly when the ball will hit the ground. We guess that it might take three seconds.

\[\text{plot}(x(t), t = 0 . . 3)\]

![Graph showing position vs time plot]

Oh, that's too much. The answer seems to be about 1.4 seconds, but we can use solve to come up with an exact value.

\[\text{solve}(0 = x(t), t)\]

\[0, \frac{11}{8}\]  
(9.23)

There are actually two solutions -- the obvious one is when \(t=0\) and the ball hasn't yet been thrown.
To get the larger one, we compose the \( \text{max} \) function with \( \text{solve} \).

\[
\text{flightTime} := \text{max}(\text{solve}(x(t) = 0, t))
\]

\[
\begin{array}{c}
11 \\
8
\end{array}
\]

(9.24)

This answers part (a) of the problem.

We verify our computation by evaluating \( x \) at that time and finding that the position of the ball really is at altitude 0.

\[
x(\text{flightTime})
\]

\[
0
\]

(9.25)

Now we need to make the movie. We need to create a function which for time \( t \), draws a point at the coordinate \((0, x(t))\). To illustrate what we mean, let's compute the position of the ball at \( t=1 \) second.

\[
x(1.0)
\]

\[
6.000000000
\]

(9.26)

We want the ball to be at position \((0,6)\). A plot command that would do this would be:

\[
\text{plot}([0, [6], \text{style = point, color = red, symbol = circle, symbolsize = 30})
\]
Similarly, at \( t=0.5 \), the ball's position would be at \( x(0.5) \):

\[
\text{plot}([0], [x(0.5)], \text{style} = \text{point}, \text{color} = \text{red, symbol} = \text{circle, symbolsize} = 30)
\]

```

Evidently \( x(0.5) \) is 7.

We create a user-defined function that creates a plot structure as a result.

\[
\text{ballFrame} := (t) \rightarrow \text{plot}([0], [x(t)], \text{style} = \text{point}, \text{color} = \text{red, symbol} = \text{circle, symbolsize} = 30)
\]

\[
  t \rightarrow \text{plot}([0], [x(t)], \text{style} = \text{point}, \text{color} = \text{red, symbol} = \text{circle, symbolsize} = 30)
\]  

(9.27)
```
Let's try this out for $t=.5$ and see if we get the same result.

\[
\text{ballFrame}(0.5)
\]

We can now use this function with \textit{animate}. Note that we needed \texttt{flightTime} in order to describe how long the movie runs. If we did more or less than that, then the ball wouldn't have landed, or would be shown as going below ground level.

\[
\text{with(plots):}
\]

\[
\text{animate(ballFrame, } [t], t = 0 .. \texttt{flightTime})
\]

This answers part (b) of the problem.
By playing the movie, we see that the maximum altitude is about 7.5 feet, at time $t=0.687$ seconds. This answers part (c) of the problem.

To answer part (d), we need to do more programming. We first modify the plot so that it draws a line segment at 30 feet as well as plotting the position of the ball. We use the `line` function of the plottools package discussed in an ??? to draw a line segment at (-5,30) to (5,30), and to color it green:

$$\text{height} := 30$$

$$\text{with(plottools)} :$$

$$\text{pLine} := \text{line}([-5, \text{height}], [5, \text{height}, \text{color} = \text{"green"}])$$

$$\text{CURVES}([[\text{-5.}, 30.], [5.\text{,} 30.]], \text{COLOUR} (\text{RGB, 0.}, 1.00000000, 0.))$$

The display function can be used to combine this line with a frame of the movie. Here is an example of this:

$$\text{with(plots)} :$$

$$\text{display}([\text{ballFrame}(0.5), \text{pLine}])$$

The automatic scaling of plot chops off the vertical distance between 6 feet and 0 because there is nothing in this frame that needs that. In the animation, the scale is adjusted so that all frames operate in the same axes.

Now we can create a new user-defined function that plots both the line and the ball.

$$\text{ballWithLine} := (t) \rightarrow \text{display}([\text{ballFrame}(t), \text{pLine}])$$

$$\text{t\rightarrowplots\_display([\text{ballFrame}(t), \text{pLine}])}$$

(9.30)
To look at the behavior of the ball at a particular velocity, we can now execute a two line script, consisting of assigning $v_0$ to the desired initial velocity, and then the operation that draws the movie.

$$v_0 := \text{convert}\left(15, \text{units}, \frac{\text{miles}}{\text{hour}}, \frac{\text{feet}}{\text{second}}\right)$$

(9.31)

$\text{animate}\left(\text{ballWithLine}, [t], t = 0..\text{max}(\text{solve}(x(t) = 0, t))\right)$

As we already have seen, $v_0 = 22$ is not fast enough. We set it higher and recalculate the movie:

$v_0 := 50$

(9.32)

$\text{animate}\left(\text{ballWithLine}, [t], t = 0..\text{max}(\text{solve}(x(t) = 0, t))\right)$
That was too high. Let's try 40.

\[ v_\theta := 40 \]  
\[ 40 \]  
\[ \text{(9.33)} \]

\[ \text{animate}(\text{ballWithLine}, [t], t = 0..\max(\text{solve}(x(t) = 0, t))) \]

Too low. Let's try 45

\[ v_\theta := 45 \]  
\[ 45 \]  
\[ \text{(9.34)} \]

\[ \text{animate}(\text{ballWithLine}, [t], t = 0..\max(\text{solve}(x(t) = 0, t))) \]

So 45 feet per second seems to be about right. We could get a more precise determination through movie-watching, but for high accuracy we should use more mathematics. In a subsequent chapter, we will introduce additional Maple operations that can calculate the velocity exactly (or a close approximation) without the trial-and-error of movie watching. Having the movies did give us a better understanding of the phenomenon.
9.12 Exporting animations and non-animated plots

One operation available in the popup menu is Export. Right-click (or control-click) -> Export -> Graphics Interchange Format will produce an animation file in .gif format. As the animation file is being created, a dialog box will appear asking you to specify the directory where the .gif file should be written. Once created, the file can be included on web pages or other documents.

This feature is also available for ordinary (non-animated) plots. Right-clicking (control-click for Macintosh) will create a file of the plot in .gif, .jpeg, or .ps format. However, .gif file is the only format of the three that is supported by web browsers for animations.

Table 9.9: Exporting animations through the pop-up menu
9.13 Summary of Chapter 9

**Combining plots and shapes using display**

Supplying the `display` function with a list of two or more plots will cause those plots to be plotted on top of one another.

We can save a plot in a variable to display later. Marking a specific value on the plot can be accomplished using the `line` or `circle` function.

```plaintext
plot4 := plot(
  sin(x), x = 0 .. 2*Pi)
```

```plaintext
maxAmp := line([0, 1], [2*Pi, 1], color = blue)
```

Parameterized plots

For plots that may have values corresponding to 2d positions, we use multiple functions to define both $x(t)$ and $y(t)$.

```plaintext
xpos := (t) -> 3*cos(t)
ypos := (t) -> 2*sin(t)
```

(9.35)  
(9.36)  
(9.37)  
(9.38)
### Parameterized plots

We put these functions in a list, along with the range of the independent variable \( t \), and plot.

\[
\text{plot}\left([x(t), y(t), t = 0..\Pi], \text{scaling} = \text{constrained}\right): 
\]

### Animating plots using animate

Creating an animation

\[
x := (t) \rightarrow 50 \cdot \sin\left(\frac{\pi}{4}\right) t:
\]

\[
y := t \rightarrow 50 \cos\left(\frac{\pi}{4}\right) t - 16 t^2:
\]

\[
posPlot := (t) \rightarrow \text{plot}\left([x(t)], [y(t)], \text{style} = \text{point}, \text{symbol} = \text{circle}\right):
\]

\[
\text{animate}(posPlot, [t], t = 0..2):
\]

Controlling the animation using the animation tool bar or the animation popup menu

The animation tool bar is located above the workspace window, while the popup menu can be displayed by right-clicking the animation.

Exporting an animation to a graphics file

Right click animation>Export>Graphics Interchange Format
10 Moving into programming

10.1 Chapter Overview

The interactive document interface that we've worked with so far is good for quick development of calculations involving a few steps. Because feedback occurs after each step entered, this style of working is often the fastest way of getting such short calculations done. We've also seen how the effort to develop a script can be made to yield greater payback by finding and exploiting situations that require re-use, by editing parameters of a script and re-executing it.

In our previous work, we've executed a script by the following process:

a) Setting up the values needed for the parameters assigned at the beginning of the script. This makes it easy to find and change the parameter values when you want to re-use it.

b) Listing the actions to solve the problem after the parameters are listed. The notion is that if all you want is re-use, you don't change anything in this part of the script.

b) Positioning the mouse cursor on the first line of the script and hitting return (or enter) repeatedly, hitting the !!! button on the Maple Toolbar, or selecting Edit->Execute->Selection.

With longer scripts or extensive re-execution, this process becomes tedious even if we are saving a lot of time by having the computer do the work. In this chapter, we introduce a new way of entering and executing scripts: code edit regions. While such regions make it more convenient to work and execute blocks of Maple instructions, the programmer must enter things using the textual version of Maple expressions and operations. Individual lines must be separated by semi-colons (;) so that the Maple language processor can more easily tell where one instruction ends and the next begins.

With code edit regions, the programmer must also work harder to see how the state of the computation is changing during the computation, rather than only looking at the final output. This is because if the final result is wrong, the programmer must find at what point in the script mistakes (called program bugs) occur. Work with code edit regions often requires adding print or printf statements into the script to better see changes in variables and intermediate results.

10.2 More on printing: print, printf and sprintf

print is a function that takes a sequence of values as arguments, and returns NULL (first introduced in ???) as a result. Its use is in its side effect -- it causes the sequence of values to be displayed in the worksheet in two dimensional format. print is used to provide more intelligible displays of information by providing words to go along with quantities or formulas that are computed. The words, of course, are provided by including strings as part of the sequence.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of print</td>
<td></td>
</tr>
<tr>
<td>( L := \left[ \exp\left( -\frac{1}{10.0} \right), \exp\left( -\frac{2}{10.0} \right), \exp\left( -\frac{3}{10.0} \right) \right] )</td>
<td>We set up a list of values. L has all floating-point values because the &quot;10.0&quot; in the denominator causes all computations to be done using limited-precision arithmetic. The display of the value of L is suppressed because we ended the assignment with a colon.</td>
</tr>
<tr>
<td>print(&quot;This list presents some values of the function ( e^x ), ( \exp(x) ); print (&quot;. The last element of the list is:&quot;, L[-1]);&quot;.</td>
<td>print prints out the sequence of values given to it as arguments. You can give any number of items in the sequence. The commas separating the string and the number are from the sequence. Note that as first mentioned ???, the result is not displayed because the line ends with a colon.</td>
</tr>
</tbody>
</table>

"This list presents some values of the function \( e^x \), \( e^{-1} \)\( \). The last element of the list is:", 0.7408182207 (10.1)
printf is a function that returns NULL (first introduced in ???) as its result, and is used primarily for its side effect of causing information to be displayed in the worksheet when the code is executed. The first item in the sequence is a string that contains ordinary words plus special format codes that begin with %. ?? summarizes the commonly used format codes. The on-line documentation for printf describes all format codes comprehensively. Some of them allow for quite intricate effects such as right or left justification, padding with leading or trailing zeroes, etc.

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
printf("There are ", nops(L),
  " elements in this list. The largest value is", max(L))
  "There are ", 3,
  " elements in this list. The largest value is",
  0.9048374180
```
| (10.2) |

Use of printf

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$L := \exp(-.1), \exp(-.2), \exp(-.3), \frac{4}{5}$; </code></td>
<td>We set up a list of values as in the previous example.</td>
</tr>
</tbody>
</table>
| ```
printf("The last element of the list is \%f\,", L[-1])
``` |
| The last element of the list is 0.800000. |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
printf("There are \%d elements in this list. The largest is \%d\,",
  nops(L), max(L))
``` |
| There are 4 elements in this list. The largest is 9.048374e-01. |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
L has \%d elements. The largest is 9.048374e-01.
printf("The largest is \%d\,", max(L));
``` |
| Note that a printf statement does not automatically cause the output on a new line. Thus, several printf statements executed together as these are will have all the output on a single line. |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
printf("There are \%f elements in this list. The largest is \%d\,",
  nops(L), max(L))
``` |
| If you attempt to use \%f format on an integer, or \%d format on a number that is not an integer, you will get incorrect values (with no warning that they are incorrect), or possibly an error message. Let the user beware; the computer isn't going to necessary save you from making a mistake. |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
printf("\%d", L)
``` |
| You can't print a list of floating point numbers using just \%f format. |

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| ```
printf("\%a", L)
``` |
| The \%a format will handle arbitrary numerical or non-numeric values. The value being printed out is a list, which is a "non-numeric value" because it isn't a single number. However, the output uses the textual format rather than the "pretty" format used by printf. |
sprintf is like printf in that its first argument is a string with format codes, and the rest of its arguments are values to be formatted. However, unlike printf it does not print (display) any information. It returns a string which is the formatted information instead. This is useful in situations where functions need formatted information, such as titles or labels of plots.

Table 10.1: Use of sprintf

| L := [exp(-1), exp(-2), exp(-1.5), exp(-2.7)] : |
| message := sprintf("plot of x versus exp(-x), using %d points", nops(L)) |
| "plot of x versus exp(-x), using 4 points" |

(10.4)

We set up a list of data points. Using sprintf, we create a string. The %d format is used to insert the number of points into the message.

We need the string to put a title onto our pointplot. You can read more about the title= argument to pointplot by entering plot,options in Maple's on-line help.

The advantage of using sprintf is in re-use. If we want to plot a different set of points, we can change the line assigning L (and the plot line of x coordinates), but we don't have to alter the sprintf since it will automatically use the right value corresponding to the length of the list.

Table 10.2: Common formatting codes for printf and sprintf

<table>
<thead>
<tr>
<th>Format code</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>printf(&quot;The answer is: %a&quot;, x = x^2 + 1)</td>
<td>Algebraic format. Works for any type of value. The value is printed out in a format suitable for textual input.</td>
</tr>
<tr>
<td>%f</td>
<td>printf(&quot;The answer is : %f&quot;, L[-1])</td>
<td>Fixed format for floating point numbers. w and d are optional. If given, w describes the number of columns for the entire number, and d the number of columns for the number after the decimal point.</td>
</tr>
<tr>
<td>%w.df</td>
<td>printf(&quot;The answer is: %f&quot;, L[-1])</td>
<td>Fixed format for floating point numbers. w and d are optional. If given, w describes the number of columns for the entire number, and d the number of columns for the number after the decimal point.</td>
</tr>
</tbody>
</table>

The answer is: \( x = x^2 + 1 \)
<table>
<thead>
<tr>
<th>Format code</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%10.5f&quot;, L[-1])</code></td>
<td><code>0.00910</code></td>
<td>Values will be converted to floats where possible.</td>
</tr>
<tr>
<td><code>printf(&quot;%10.5f&quot;, 20!)</code></td>
<td><code>2432902008176640000.00000</code></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;, \( \frac{3}{4} \))</code></td>
<td><code>0.750000</code></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;The answer is: %f&quot;, x=x^2 + 1)</code></td>
<td>Error, (in fprintf) number expected for floating point format</td>
<td></td>
</tr>
<tr>
<td><code>%e</code></td>
<td><code>printf(&quot;%e&quot;, L[-1])</code></td>
<td>Scientific format for floating point numbers.</td>
</tr>
<tr>
<td><code>%we</code></td>
<td><code>printf(&quot;%12.5e&quot;, L[-1])</code></td>
<td></td>
</tr>
<tr>
<td><code>%d</code></td>
<td><code>printf(&quot;%d&quot;, 10!)</code></td>
<td></td>
</tr>
<tr>
<td><code>%we</code></td>
<td><code>printf(&quot;%e&quot;, 10!)</code></td>
<td></td>
</tr>
<tr>
<td><code>%w.de</code></td>
<td><code>9.095277e-03</code></td>
<td></td>
</tr>
<tr>
<td><code>%d</code></td>
<td><code>2432902008176640000</code></td>
<td></td>
</tr>
<tr>
<td><code>%we</code></td>
<td><code>2.43290e+18</code></td>
<td></td>
</tr>
<tr>
<td><code>%we</code></td>
<td><code>7.500000e-01</code></td>
<td></td>
</tr>
<tr>
<td><code>%w.de</code></td>
<td><code>7.500000e-01</code></td>
<td></td>
</tr>
<tr>
<td><code>%d</code></td>
<td><code>printf(&quot;%d&quot;, 3.0)</code></td>
<td>Format for integers.</td>
</tr>
<tr>
<td></td>
<td><code>Error, (in fprintf) integer expected for integer format</code></td>
<td></td>
</tr>
<tr>
<td><code>%d</code></td>
<td><code>printf(&quot;%d&quot;, 20!)</code></td>
<td></td>
</tr>
<tr>
<td><code>%w.de</code></td>
<td>Error, (in fprintf) number expected for floating point format</td>
<td></td>
</tr>
</tbody>
</table>

wand d are optional. If given, w describes the number of columns for the entire number, and d the number of columns for the number after the decimal point. If w is wider than the number of digits required, then it is filled out with leading white space.

Values will be converted to floats where possible.
### 10.3 For the curious: why printf is so cryptic

The format codes for `printf` are a "programming language within a programming language" in that they describe in abbreviated form how the computer should output values. The style used for format codes is very much unlike the rest of Maple, in that brevity rather than readability seems to be the criterion used to design this portion of the language. A language for science and engineering popular starting in the middle of the twentieth century was Fortran, which had formatting that was similar in approach but was much different in syntax. Given your knowledge of Maple (and English), the intent of the program is fairly obvious: it will print There were 3 values, and the answer was 4.7.

**A Fortran program that demonstrates formatted printing**

```fortran
n = 3
x = 4.7
print (*,35) n,x
35 format('There were',i2,' values, and the answer was',f5.1,'.')
stop
end
```

The design of `printf` can be explained by the fact that Maple's `printf` is a close imitation of the `printf` found in the programming language C (popular with many programmers at the time Maple was invented in the '80s, and still popular among computer engineers and scientists doing device programming). Matlab, being invented in the same era, also a `sprintf` function of highly similar design. Imitation makes it easier to transfer programming skill from one language to another. Inventors of programming languages will tend to imitate unless they have a goal or insight that justifies a difference. Programming languages designed by skillful designers...
do not introduce arbitrary differences without a good cause. When you notice similarities between languages, take it as a recognition that many people liked that style of doing things, across many different kinds of programmers and kinds of problems. When you notice a difference between languages, you should explore the reasons why the designers made different choices. Usually there is a problem or issue that the designer is trying to handle better by being different from the rest of the pack.

Programming language features features that work across tens of thousands of programmers and programs are usually motivated by significant design rationales. The design of most programming languages is nowadays informed by what worked well or didn't work well in the past.

10.4 Code edit regions: executing a series of actions at once

We have seen that even some of the easiest technical calculations break down into a series of operations, chained together. Realizing that we will typically need the computer to perform a "series of operations" is the motivation for making the transition to programming, where execution of many operations in a block is typical.

The Maple document interface that we have been using so far easily supports calculations where the user prompts to computer to do a series of steps by positioning the cursor at the first operation and then hitting return (or enter) repeatedly. We now introduce a second way to enter instructions and have them executed as a block. This alternative is often easier to work with when you have a few dozen instructions and will want to execute them all in a chain.

One can open a text field in the Maple document where one can enter a series of instructions. To do this, position your cursor where you want the text field to appear in the document, and with the mouse perform Insert->Code Edit Region

| Insert->Code Edit Region menu |
Once the field has been created, you can type the textual version of the Maple instructions you want executed. Each instruction (commonly referred to as a statement) must be separated by either a semi-colon, or a colon. If the statement ends in a semi-colon, then its value will be printed during execution of the region. If the statement ends in a colon, then printing of its value will be suppressed just as it is in operation of documents.

Once all the instructions have been entered, you can run them all in a series by typing control-e (command-e on Macintosh), or by entering right-click->Execute Region (on Macintosh, control-click->Execute Region). Instead of typing control-e, you can as an alternative click on the "!" icon on the Maple toolbar. The results for each statement will appear below the region in blue, except for the statements whose printing is suppressed by a colon.

**Execute Group icon ("!" icon) on Maple Toolbar**
Any portion of a line that begins with a "#" is a *program comment*. The rest of the line is regarded by Maple as something that people will read, not an instruction to be performed by the computer. Typically what appears after the # is commentary written in English (or whatever language is convenient for communication with the intended audience) that helps explain/remind human readers what a segment of code is about. As you read programs with comments, you will see that some programmers are "chattier" than others. This is because they feel that their intended reading audience needs more explanation.

In professionally written code, you will often see comments at the beginning of the region that give an overview of what the code region does, the name of the author(s), and the date of creation/modification. Professional organizations often mandate a particular style and content for such "header" comments, so that it will be easy to find information about any code written by several programmers working on a single project.

### A code edit region with actions entered, separated by semi-colons.

<table>
<thead>
<tr>
<th>Code region</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># Figure 11.1.1</code></td>
<td>Segments of lines that begin with # are regarded as program comments</td>
</tr>
<tr>
<td><code># initialize variables</code></td>
<td>(for the program reader's eyes), not operations for Maple to carry out.</td>
</tr>
<tr>
<td><code>i := 1;</code></td>
<td>The results in blue are displayed after we position the cursor in the</td>
</tr>
<tr>
<td><code>val := .3; # evaluation point</code></td>
<td>code edit region and type control-E (command-E on Macintosh), or enter</td>
</tr>
<tr>
<td><code>term := (val^i)/i!; # a term to compute</code></td>
<td>Execute Code Region via the clickable menu.</td>
</tr>
<tr>
<td><code>print(&quot;term is&quot;, term); # message</code></td>
<td>Result of first assignment (to <code>i</code>)</td>
</tr>
<tr>
<td><code>s := term; # s has a copy of the term</code></td>
<td>Result of assignment to <code>val</code>.</td>
</tr>
<tr>
<td><code>tol := 10e-7; # A small value.</code></td>
<td>Result of assignment to <code>term</code>.</td>
</tr>
<tr>
<td></td>
<td>Result of calling the <code>print</code> function.</td>
</tr>
<tr>
<td><code>1</code></td>
<td>Result of assignment to <code>s</code>.</td>
</tr>
<tr>
<td><code>0.3</code></td>
<td>Result of assignment to <code>tol</code>.</td>
</tr>
<tr>
<td><code>0.3</code></td>
<td></td>
</tr>
<tr>
<td>&quot;term is&quot;, 0.3</td>
<td></td>
</tr>
<tr>
<td><code>0.3</code></td>
<td></td>
</tr>
<tr>
<td><code>0.0000010</code></td>
<td></td>
</tr>
</tbody>
</table>

The code region will develop a scroll bar if the amount of text entered exceeds the size of the window. The size of the field can be adjusted by right click->Component Properties, and then modifying the integers listed for the width and height.

### Enlarging a code Region window by changing component properties via the clickable menu
# Figure 11.1.1
# Initialize variables
i := 1;
val := .3; # Evaluation point
term := (val^i)/i!; # A term to compute
print("term is", term); # Message
s := term; # s has a copy of the term
tol := 10e-7; # A small value.
The "Collapse Code Edit Region" menu item of the clickable menu will reduce the entire window to an icon. If the first line of the region is a program comment, it will be listed to the right of the icon. Clicking on the code icon will execute the code within.

A collapsed code edit region
10.5 Turning intentions into code

More complicated problems or longer computations requires a planning phase before any programming is done. Once the plan has been developed, the programmer then goes about the business of writing the actions described in the plan in the programming language being used. This business of translating intentions into instructions in a language a computer can understand is called coding, or writing program code. In concrete terms for this course, what this means is that the problems we will be working on will need some planning before anything is written in Maple. As with all plans, it is beneficial to be able to see how well the plan is going to work (i.e. to evaluate the plan) and fix/polish it before moving to the coding phase.

The purpose of programming languages is to express a solution to the computer. They are not ideal languages for people to think in to develop the outline of a solution.

10.6 Making code easier to read and understand

As has been mentioned, an important part of training for a programmer is to make code easier to reuse, since reuse allows the cost of writing the program to be amortized over more invocations of it. Part of reuse is allowing others to understand what they should do with the code, or to allow yourself to more quickly recall the details of what you were doing when you return to a program after a few weeks or months away from it. This kind of reuse is helped by making certain stylistic choices in how you write the code. Here are some ideas:

1. Use mnemonic variable names -- names that suggest what the variable's purpose is. This is different from the variable naming practices of the "mathematics culture", which is to use symbols of one letter in length, e.g. \( x, i \).

   a. The reason for the difference in practices is that it is relatively easy to keep track of the purpose of variables if the expression is only one line long, as is typical in elementary mathematics. Even elementary programs however consist of many lines. A first-year student's program might be a dozen or even 100 lines, using ten or twenty variables. Remembering what they all do becomes too hard without reminders.

2. Invent and use user-defined functions to encapsulate common operations. Reusing a name two or more times has advantages beyond not doing so much typing. It allows the reader to see quickly that you are doing the "same thing as before" instead of something subtly or radically different.

3. Mnemonically name functions. If the function has a name that describes what is doing (e.g. "plotCoolingFunction" instead of "f") then it becomes even easier for the reader to understand/recall what the function is intended to do even the first time that they read about it.

4. Blank lines are usually used to indicate the natural conceptual divisions between different groups of instructions. For example, the first few instructions establishing initial values of variables might be separated by a few blank lines from the rest of the instructions which establish the main body of work, which in turn may be separated by the "finishing up" instructions.

5. Indentation is used for indicating nuances of control, such as the extent of code blocks that are repeated. We will discuss indentation more in the next chapter where the kind of code that benefits from it is first discussed.
## 10.7 Troubleshooting common errors in entry

New kinds of errors and warnings can appear in code regions, due primarily to the new requirement that statements be separated by semi-colons and colons. Trying to enter too many lines at once before testing what you have so far can make older errors appear more mysterious.

### Table 10.3: Example Commentary

<table>
<thead>
<tr>
<th>Code regions with entry errors</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Commentary</td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2) ;</code></td>
<td></td>
</tr>
<tr>
<td>Error, <code>;</code> unexpected</td>
<td>The error message indicates that a semi-colon was not expected where Maple scanned it. This is typically semi-colons can appear only at the end of a complete Maple statement. This message typically indicates that something went wrong in what was entered before the semi-colon. Edit the code region to make the statement correct, and re-execute the region.</td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2)</code></td>
<td>The problem here is a missing parentheses ). To fix this, you'd put a closing ) after the &quot;3.5&quot;, and then re-execute the region with a control-e(or command-e).</td>
</tr>
<tr>
<td><code>Warning, inserted missing semicolon at end of statement</code></td>
<td>This is only a warning, and executing this region does perform the computation correctly.</td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2);</code></td>
<td>If we add another line without a semi-colon between, we get just an error. Both lines are messed up because Maple needs an explicit separator between the statements. Just putting the next instruction on another line is not enough.</td>
</tr>
<tr>
<td><code>b := cos(3.5) + sin( Pi/2)</code></td>
<td>If you try to enter several lines at once before entering, you have the additional problem of figuring out which line the mistake occurred on. In this case, you have to decide which of the two semi-colons was unexpected.</td>
</tr>
<tr>
<td><code>Warning, premature end of input, use &lt;Shift&gt; + &lt;Enter&gt; to avoid this message.</code></td>
<td>There's both a missing closing parentheses and a missing semi-colon here. The symptom is that there is a warning message (which isn't necessarily a sign of trouble, but is here), plus the fact that you are expecting a value to be printed from the computation but just see a red &quot;&gt;&quot;, with no numeric output. Fixing both problems and re-executing the region will produce the correct result.</td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2)</code></td>
<td></td>
</tr>
<tr>
<td>Error, <code>;</code> unexpected</td>
<td>Sometimes when you enter a line, you make more than one error. Even though the same mistake has been made as in the previous example,</td>
</tr>
</tbody>
</table>
there is a different error message than before. Executing this region causes the message indicated, plus the cursor will be observed to flash at the start of the second line.

Rather than being in an "I was expecting more" state which produced the first warning message, Maple is in a "I was expecting more, but wait what I've seen isn't what I was expecting". Since there's nothing before the cursor started flashing before any of the second line was parsed, it means that the error must have happened before that on the previous line.

There are still two things to fix. Fixing them will produce two lines of output.

## 10.8 Summary of Chapter 10

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use of print</strong></td>
<td></td>
</tr>
</tbody>
</table>
| \[
L := \left[ \exp\left( -\frac{1}{10.0} \right), \exp\left( -\frac{2}{10.0} \right), \exp\left( -\frac{3}{10.0} \right) \right];
\] | We set up a list of values. L has all floating-point values because the "10.0" in the denominator causes all computations to be done using limited-precision arithmetic. The display of the value of L is suppressed because we ended the assignment with a colon. |
| print("This list presents some values of the function ", \exp(x)); print(". The last element of the list is ", L[-1]); | |
| "This list presents some values of the function \( e^x \). The last element of the list is ", 0.7408182207 | (10.13) |
| \[\text{print("There are ", \text{nops}(L), 
\" elements in this list. The largest value is", \text{max}(L))}
\] | |
| "There are ", 3, 
\" elements in this list. The largest value is", 0.9048374180 | (10.14) |

| **Use of printf** | |
| \[
L := \left[ \exp(-1), \exp(-2), \exp(-3), \frac{4}{5} \right];
\] | We set up a list of values as in the previous example. |
| \[\text{printf("The last element of the list is \%f\
The last element of the list is 0.800000.}
\] | The first argument to printf is a string describing how to format the rest of the values. The string contains format codes describing where to insert the values of the later arguments into the message described in the string. In this example, there is one format code \%f, for printing numbers in conventional decimal point notation. Since printf joins together the message and the value together into a single message without a sequence into a single result, what is printed is without commas. |
| \[\text{printf("There are \%d elements in this list. 
\nThe largest is \%e.", 
\text{nops}(L), \text{max}(L))}
\] | Additional printf codes are \"\%d\" (for integer values), and \"\%e\" (for floating point numbers in scientific notation). The format code \n causes the next output to occur on the next line. |
| There are 4 elements in this list.
The largest is 9.048374e-01. | |
Commentary

Example

```
printf("L has %d elements ", nops(L));
printf("The largest is %e , max(L));
```
L has 4 elements. The largest is 9.048374e-01.

Note that a printf statement does not automatically cause the output on a new line. Thus, several printf statements executed together as these are will have all the output on a single line.

```
printf("There are %f elements in this list. The largest is %d",
        nops(L), max(L))
```
There are 4.000000 elements in this list. The largest is

Error, (in fprintf) integer expected for integer format

If you attempt to use %f format on an integer, or %d format on a number that is not an integer, you will get incorrect values (with no warning that they are incorrect), or possibly an error message. Let the user beware; the computer isn't going to necessary save you from making a mistake.

```
printf("%f", L)
```
Error, (in fprintf) number expected for floating point format

You can't print a list of floating point numbers using just %f format.

```
printf("%a", L)
```
[.9048374180, .8187307531, .7408182207, 4/5]

The %a format will handle arbitrary numerical or non-numeric values. The value being printed out is a list, which is a "non-numeric value" because it isn't a single number. However, the output uses the textual format rather than the "pretty" format used by `print`.  

<table>
<thead>
<tr>
<th>Table 10.4: Use of sprintf</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td><strong>Commentary</strong></td>
</tr>
</tbody>
</table>
| ```
L := [ exp(-1), exp(-2), exp(-1.5), exp(-2.7) ]:
message := sprintf("plot of x versus exp(-x), using %d points", nops(L))
```

"plot of x versus exp(-x), using 4 points"

(10.16) |
| We set up a list of data points. Using sprintf, we create a string. The %d format is used to insert the number of points into the message.
| We set up a list of data points. Using sprintf, we create a string. The %d format is used to insert the number of points into the message. |
We need the string to put a title onto our `plot`. You can read more about the `title` argument to `plot` by entering `plot, options` in Maple's on-line help.
The results in blue are displayed after we position the cursor in the code edit region and type control-E (command-E on Macintosh), or enter Execute Code Region via the clickable menu.

Result of first assignment (to \(i\))

Result of assignment to \(val\).

Result of assignment to \(term\).

Result of calling the \(print\) function.

Result of assignment to \(s\).

Result of assignment to \(tol\).
Creating and developing code

11.1 Chapter Overview

Being able to work conveniently with longer scripts also allows us to solve more complicated problems. But keeping all the ideas in your head for what you want to do on the computer becomes too error-prone as you start dealing with more complicated tasks. The programmer takes on two kinds of work -- the plan for how to solve the problem, and the coding where the plan is communicated to the computer in a programming language. Despite Maple's powerful repertoire of functions (e.g. "solve" or "fit data"), it is often the case that the steps of a plan can require several Maple actions to implement. In those situations it is productive to outline a solution before writing the details of the code. The outline (as well explanations of any tricky details of the code) can be included in the code edit region by including *program comments*.

The ability to include comments in coding regions also can be used by the programmer as they figure out how to create the code. All the details of the plan need not be made in one fell swoop. A programmer can begin by writing a code outline in comment form. Since the Maple language processor does not perform any actions when it executes a comment, the code edit region will both be helpful to the programmer, and also not contain anything that would cause error messages from Maple.

Once the programmer is satisfied that the outline is satisfactory, the finer details can be filled in much as an outline for a report or paper can be used. When the details are fine enough, the programmer can write the short bits of code that correspond to that detail.

As with all multi-part projects, it is best to develop code *incrementally*, adding a line or two of code at a time, running the entire region and checking that it works as far as it goes, and then proceeding onto another few lines. The alternative to incremental development -- typing in the entire solution and then trying to fix its problems -- is much more time consuming because there are too many places to look for problems. Incremental development works faster because there are only slight changes from the previously tested version of the code, hence only a few places that could be causing problems if there are any.

11.2 The "outline approach" for developing a solution in a code region.

Many students learn to write a report through the "outline method". See for example, http://owl.english.purdue.edu/owl/resource/544/02/. The outline method suggests that the way to proceed goes roughly like this:

1. After considering what you would like to say, create the major sections for your content, and put them into an order that makes sense.

2. Refine each section, by adding subsections that group the major points of each section. Put the subsections in an order that will make sense to the reader.

3. Continue in this way to create subsubsections, etc. until the level of detail is small enough that you can see an easy writing task for each segment in your outline.

The final version of the report will keep the content organized in this way. This kind of refinement can makes it easier to keep things well-organized and coherent at each level of detail. It also helps avoid getting lost in the fine details.

Code development using this style has the initial outline written as program comments. Only when the subpoints or subsubpoints can be expressed straightforwardly as something the programmer finds easy to code (or one line of code) does the writing turn from writing comments into writing of code.

This makes commenting something you do even before you are completely certain as to what you will do. It's a lot easier to change comments than it is to redo the details of code that has already been written. Having the plan description included into the code region means that you do not have to jump between pieces of paper and the Maple worksheet to check whether the plan is being followed by the code.

Table 11.1: A problem to solve

| From Sullivan, Pre-calculus, p. 342 |
A fast-food restaurant wants a special container to hold coffee. The restaurant wishes the container to quickly cool the coffee from 200 °F to 130 °F as quickly as possible, and keep the liquid between 110 °F and 130 °F (optimal drinking temperature) as long as possible. The restaurant has three containers to select from.

(a) The CentiKeeper Company has a container that reduces the temperature of a liquid from 200 °F to 100 °F in 30 minutes by maintaining a constant temperature of 70 °F.
(b) The Temp Control Company has a container that reduces the temperature of a liquid from 200 °F to 100 °F in 25 minutes by maintaining a constant temperature of 60 °F.
(c) The Hot’n’Cold Company has a container that reduces the temperature of a liquid from 200 to 120 °F in 20 minutes by maintaining a constant temperature of 65 °F.

How long does it take each container to lower the coffee temperature from 200 °F to 130 °F?
How long will the coffee temperature remain between 110 °F and 130 °F?

Using the method we suggest, we would first create a code region that has only comments, giving the basic outline of what we would do. In this situation, it would also be good to create a preface describing the problem. In Maple, we could just place the description as ordinary text before the code region with the comments.

We think about what we would do and realize that these questions can be answered by doing the "heating and cooling equation" calculations we've seen ???. Such equations typically require finding a "heating constant" used in an exponential formula, and then solving one or more equations for t.

Table 11.2: Initial code outline to solve the problem

<table>
<thead>
<tr>
<th>Initial code outline to solve the problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of problem</td>
</tr>
<tr>
<td>(From Sullivan, Pre-calculus, p. 342)</td>
</tr>
</tbody>
</table>

A fast-food restaurant wants a special container to hold coffee. The restaurant wishes the container to quickly cool the coffee from 200 °F to 130 °F as quickly as possible, and keep the liquid between 110 °F and 130 °F (optimal drinking temperature) as long as possible. The restaurant has three containers to select from.

(a) The CentiKeeper Company has a container that reduces the temperature of a liquid from 200 °F to 100 °F in 30 minutes by maintaining a constant temperature of 70 °F.
(b) The Temp Control Company has a container that reduces the temperature of a liquid from 200 °F to 100 °F in 25 minutes by maintaining a constant temperature of 60 °F.
(c) The Hot’n’Cold Company has a container that reduces the temperature of a liquid from 200 to 120 °F in 20 minutes by maintaining a constant temperature of 65 °F.

How long does it take each container to lower the coffee temperature from 200 °F to 130 °F?
How long will the coffee temperature remain between 110 °F and 130 °F?

#Define shared parameters for problem
Each of these steps needs to be fleshed out a bit more before we can start writing lines of code. We don't have to flesh out things starting from the beginning, we can add more details to the outline in any appealing order. In this case, we talk about how to answer the questions for CentiKeeper, using the solution process discussed in the outline approach as a model for the computation. We still have only comments, not code. We can use two ##s at the start of the line (which, since it begins with a #, is still regarded as a comment) to indicate that the comment is at a lower level than before.

Table 11.3: Next step of refinement for code outline

<table>
<thead>
<tr>
<th>Next step of refinement for code outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Define shared parameters for problem</td>
</tr>
<tr>
<td>#Define parameters for each company</td>
</tr>
<tr>
<td>#Answer the questions for CentiKeeper</td>
</tr>
<tr>
<td>##Find the heating constant k1 for CentiKeeper</td>
</tr>
<tr>
<td>##Substitute the constant into the heating equation.</td>
</tr>
<tr>
<td>##Find the value of t such that the heat equation produces 130 degrees.</td>
</tr>
<tr>
<td>##Find the value of t such that the heat equation produces 110 degrees.</td>
</tr>
<tr>
<td>##Compute the difference, print out the answer.</td>
</tr>
<tr>
<td>#Answer the questions for Temp Control</td>
</tr>
<tr>
<td>#Answer the questions for Hot'n'Cold</td>
</tr>
</tbody>
</table>

The general heating formula we're going to use is \( u(t) = T + (u_0 - T) e^{kt} \). \( u_0 \) is the initial temperature, and \( T \) is "room temperature". \( k \) is the heating constant. We realize that there are other entities too which we give mathematical names to: \( \tau \), a period of time elapsed after the start, and \( N \), the temperature of the liquid in the cup after \( \tau \) minutes has elapsed. We will use different \( u_0 \), \( T \), \( \tau \), \( N \) and \( k \) for each company. We will give different names to the programming variables that we will use for each company, rather than trying to re-use the symbols several times.

We can flesh out the outline further with these decisions.

Table 11.4: Next step of refinement for code outline

<table>
<thead>
<tr>
<th>Next step of refinement for code outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Define shared parameters for problem</td>
</tr>
<tr>
<td>##There are no shared parameters</td>
</tr>
</tbody>
</table>

#Define parameters for each company

#Answer the questions for CentiKeeper

#Answer the questions for Temp Control

#Answer the questions for Hot'n'Cold
# Define parameters for each company
## \( u_0^1, T_1, k_1, \tau_1, N_1 \) for CentiKeeper
## \( u_0^2, T_2, k_2, \tau_2, N_2 \) for Temp Control
## \( u_0^3, T_3, k_3, \tau_3, N_3 \) for Hot'n'Cold

## Answer the questions for CentiKeeper
### Find the heating constant \( k_1 \) for CentiKeeper
### Substitute the constant into the heating equation.
### Find the value of \( t \) such that the heat equation produces 130 degrees.
### Find the value of \( t \) such that the heat equation produces 110 degrees.
### Compute the difference, print out the answer.

## Answer the questions for Temp Control
### Do the same thing as we did for CentiKeeper, only use the variables we have for Temp Control

## Answer the questions for Hot'n'Cold
### Do the same thing as we did for CentiKeeper, only use the variables we have for Hot'n'Cold

We still haven't written any code, but we are about ready to do that. First however, we should learn a few more "tricks" to efficient code development.

### 11.3 A crucial skill in debugging: identify the first line of code where things go wrong

The execution trace of a code region is the output that results from executing each line of a block of code in a region. Full execution traces can be created only after the syntax errors have been removed from the code region so that the region can be executed without any error messages. Removing error messages is only the beginning to the work, however. The code you write may still have mistakes in it -- the instructions you give may make grammatical sense, but be the wrong actions to solve the problem. A mistake in code is often called a **bug**, and removing mistakes from code is called **debugging** it. Programmers spend a lot of time debugging, usually because getting something new to work is difficult. Any ideas, approaches, or strategies for making debugging less difficult are usually welcomed by programmers.

#### Using an execution code trace to find a bug in a short code block

```plaintext
# Define parameters for each company
## u01, T1, k1, tau1, N1 for CentiKeeper

u01 := 200;
T1 = 70;
N1 := 100;
tau1 := 30;
```

\[
\begin{align*}
200 \\
70 \\
100 \\
30
\end{align*}
\]
After creating the code region and entering the lines, we get the execution trace -- the lines in blue -- by clicking on the code edit region and typing either control-e (command-e for Macintosh), or entering right-click->Execute Code. Each line in blue corresponds to result of execution of a line of code.

A major difference between executing a script interactively in a document, and in a code edit region, is that execution trace for interactive execution occurs intermingled with the lines of code, while with code regions the execution trace occurs after the listing of all the lines of the block.

Programmers debugging code inspect execution traces to find mistakes in their programming. They try to find the first line of code where things go wrong. While the cause of that "buggy result" may lie in something that happened even earlier, it is certain that the first program bug did not happen later. Knowing the first buggy line of the execution trace typically reduces the number of lines of code that must be inspected for the mistake, since the error must have occurred in the code at or before the line that generated the "bad output".

In the execution trace above, we see that the first line of output is consistent with the "u01 := 200;" line of code -- the value of the assigned variable is printed out. We can see immediately that the T1=70 line looks different than the others, which were just numbers from the assignments. Looking at the code, we see that there is a difference between the second line of code (excluding comments) and the others -- there is an "=" rather than a ";=". We have used the execution trace to discover a mistake we have made in a line of code -- that line should really be T1 := 70.

11.4 Using print and printf to include more information in the execution trace

Sometimes the execution trace does not provide enough information to discover a bug. For that reason, programmers sometimes rely on placing print or printf statements in code regions just before steps that they want to look at. Usually the statements that do the printing output a message that includes not just values of variables, but what variables they are, and other information that makes it easier to identify which line of code out of many in the code block is doing the printing.

Continuing the example above, we enter more lines of code and begin to execute what we have. We ran into problems where some of the variables got mangled from bugs in previous executions of the code region. So we place a restart at the beginning to make sure that we unassign all variables before we begin execution again, leaving only the assignments made this time around.

We put in a printf statement to make it clear what we're solving for, before we solve it. We do this because we know (from previous efforts working with this kind of problem) that solving this "heat equation" can be tricky.

<table>
<thead>
<tr>
<th>Execution trace augmented by a printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Define parameters for each company</td>
</tr>
<tr>
<td>##u01, T1, k1 , tau1, N1 for CentiKeeper</td>
</tr>
<tr>
<td>restart; #Unassign variables from previous executions</td>
</tr>
<tr>
<td>u01 := 200;</td>
</tr>
<tr>
<td>T1 := 70;</td>
</tr>
<tr>
<td>N1 := 100;</td>
</tr>
<tr>
<td>tau1 := 30;</td>
</tr>
<tr>
<td>#Answer the questions for CentiKeeper</td>
</tr>
<tr>
<td>##Find the heating constant k1 for CentiKeeper</td>
</tr>
<tr>
<td>###Describe the heat equation</td>
</tr>
<tr>
<td>heat1 := T1+(u01-T1)<em>exp(k1</em>t);</td>
</tr>
<tr>
<td>h1 := eval(heat1,t=tau1);</td>
</tr>
<tr>
<td>printf(&quot;Solving for %a&quot;,ht1);</td>
</tr>
<tr>
<td>k1soln := solve(ht1,k1);</td>
</tr>
<tr>
<td>printf(&quot;Solving %a for heat constant: k1 is %f&quot;, ht1, k1soln);</td>
</tr>
</tbody>
</table>
We see a lot of gibberish after awhile, but our attention turns to the "Solving for 70+130*exp(30*k1)" and the next line of the execution trace, which we infer from its placement is the result assigned to k1soln. It appears to be an imaginary number (because it has an "I" in it), which is definitely not the right kind of value for a cooling constant. Looking again at the printf output, we see that what we're solving for is wrong -- according to the math we should be solving the equation $100 = 70 + 130e^{30k1}$. The Error, message is triggered by the result of solve. printf can't handle expressions with I in them which is what the buggy computation is producing. We use these insights (which come only after we understand what we would be doing by hand with this calculation), to rework the code region so that it sets up an equation that uses the "N1" information.

Table 11.6: Execution trace from a repaired code region

```plaintext
#Define parameters for each company
##u01, T1, k1, tau1, N1 for CentiKeeper
restart; #Unassign variables from previous executions
u01 := 200;
T1 := 70;
N1 := 100;
tau1 := 30;
##u02, T2, k2, tau2, N2 for

#Answer the questions for CentiKeeper
##Find the heating constant k1 for CentiKeeper

###Describe the heat equation
heat1 := N1 - T1+(u01-T1)*exp(k1*t);
ht1 := eval(heat1,t=tau1);
printf("Solving for %a.,ht1);
k1soln := solve(ht1,k1);
printf("Solving %a for heat constant: k1 is %f.", ht1, k1soln);
```
Solving for \( 100 = 70 + 130 \exp(30k1) \).

\[
\frac{1}{30} \ln \left( \frac{3}{13} \right)
\]

Solving \( 100 = 70 + 130 \exp(30k1) \) for heat constant: \( k1 \) is \(-0.048878\).

This presents information in the trace that convinces us that the right thing is going on.

### 11.5 Incremental coding and testing

The old adage "don't bite off more than you can chew" also applies to coding. Suppose you enter thirty lines of code and then execute them. If you find a mistake at the end, there are thirty lines that could be the source of the problem. Having to look through thirty of anything (the execution trace, or the code) can be time consuming and complicated to keep track of. You may have noticed that we have developed code in the past few sections only a few lines at a time -- entering them, viewing the execution trace, finding things to fix, and proceeding. It would be nice if we could just get everything written correctly the first time, but it is more realistic to assume that you won't. There will be typos and misconceptions to fix all along the way.

Proceeding incrementally makes it likely that it's always the last lines you entered where the problem is most likely to lie. Combining this with information from `print` statements to augment the execution trace can be a very efficient way of developing code because the execution trace information is presented intelligibly, and its typically only the last few lines that you need to scrutinize carefully.

Finally, you can use the "comment trick" to handle things incrementally even if you've already written more than two lines of code. You can type in all the lines of code, but add a `#` in front of all the lines except the first one. If you then execute the code region, you will see the results from only the first line of code because all the others will be treated as comments and not executed. Once you have that line working well, then you can remove the `#` from the second line, and re-execute the region. Because two lines are now not comments, you will see the execution trace from two lines of code. You can continue "uncommenting" lines in this fashion to achieve incremental development.

The comment trick can be used to on `print` or `printf` statements that you have been using for troubleshooting but are not completely sure that you no longer need. By putting a `#` in front of a `print` statement, you can turn off its effects without erasing it completely. If you decide that you need to see it again, you can remove the `#` and re-activate it. This can save a lot of typing.

Having proceeded to solve one version of the problem partially, let's see if we can create the whole ball of wax for CentiKeeper. We enter one new line at a time, execute the region, and see what needs fixing. After about five minutes, we have all the lines working, and have printed the desired answer at the end. Note that we have used an advanced feature of so that it prints only two digits after the decimal point. We also rely on the fact that even if the value being given to a `%f` format code isn't a floating point number, the result is automatically approximated so that it can appear in floating point format. You can read about these feature in the on-line help for `printf`.

At this point, we'll clone the process for the other two situations through copying, pasting, and editing to use different variable names.
Table 11.7: Execution trace augmented by a printf

<table>
<thead>
<tr>
<th>Execution trace augmented by a printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define parameters for each company</td>
</tr>
<tr>
<td>u₀₁, T₁, k₁ , τ₁, N₁ for CentiKeeper</td>
</tr>
<tr>
<td>restart; #Unassign variables from previous executions</td>
</tr>
<tr>
<td>u₀₁ := 200; T₁ := 70; N₁ := 100; τ₁ := 30;</td>
</tr>
<tr>
<td>#Answer the questions for CentiKeeper</td>
</tr>
<tr>
<td>#Find the heating constant k₁ for CentiKeeper</td>
</tr>
<tr>
<td>###Describe the heat equation</td>
</tr>
<tr>
<td>heat₁ := T₁+(u₀₁-T₁)<em>exp(k₁</em>t); ht₁ := N₁ =eval(heat₁,t=τ₁); printf(&quot;Solving for %a.&quot;,ht₁); k₁soln := solve(ht₁,k₁); printf(&quot;Solving %a for heat constant: k₁ is %f.&quot;, ht₁, k₁soln);</td>
</tr>
<tr>
<td>###Substitute the constant into the heating expression.</td>
</tr>
<tr>
<td>eqn₁ := eval(heat₁, k₁=k₁soln); #Find the value of t such that the heat equation produces 130 degrees.</td>
</tr>
<tr>
<td>CentiKeeper₁₃₀ := solve(eqn₁=130,t);</td>
</tr>
<tr>
<td>#Find the value of t such that the heat equation produces 110 degrees.</td>
</tr>
<tr>
<td>CentiKeeper₁₁₀ := solve(eqn₁=110,t);</td>
</tr>
<tr>
<td>#Compute the difference, print out the answer.</td>
</tr>
<tr>
<td>printf(&quot;The CentiKeeper cup keeps coffee warm for %5.2f minutes.&quot;, CentiKeeper₁₁₀-CentiKeeper₁₃₀);</td>
</tr>
</tbody>
</table>
Solving for \( 100 = 70 + 130 \exp(30k1) \).

\[
\frac{1}{30} \ln \left( \frac{3}{13} \right)
\]

Solving \( 100 = 70 + 130 \exp(30k1) \) for heat constant: \( k1 \) is \(-0.048878\).

\[
70 + 130 e^{\frac{1}{30} \ln \left( \frac{3}{13} \right) t}
\]

\[
\frac{30 \ln \left( \frac{6}{13} \right)}{\ln \left( \frac{3}{13} \right)}
\]

\[
\frac{30 \ln \left( \frac{4}{13} \right)}{\ln \left( \frac{3}{13} \right)}
\]

The CentiKeeper cup keeps coffee warm for 8.30 minutes.

### 11.6 Using interactive development first for tricky code

We've already mentioned an idea to speed up development of code that you're not completely sure about -- copy examples from documentation and get them to work in a worksheet, then modify the working example to do what you really would like to do. When developing a sizable script in a code region, there will still be spots where you won't know what to enter immediately. When you encounter such a situation, open a fresh Maple document (i.e. by doing File->New->Document Mode) and use it as an experimental sandbox working interactively to get those lines of code to work. Then copy the working code into the code region when it is "fully baked" or nearly so. If you have developed the code interactively using 2D math notation, you may need to convert it into ordinary textual format by applying Format->Convert To->1D Math Input.

<table>
<thead>
<tr>
<th>Converting 2D expressions into 1D expressions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x0 = \frac{mv0\left(-1 + e^{\frac{ht}{m}}\right)}{b} )</td>
<td>We enter an expression in the usual interactive two dimensional format. It's probably easier to enter the expression this way and see that it agrees with the mathematical conception.</td>
</tr>
<tr>
<td>( \frac{x0}{m}\frac{v0x}{\left(-1+\exp(-b*1/m)\right)}{b} )</td>
<td>We copy the expression and paste it into a separate place on the worksheet. We then select the expression and do Format-&gt;Convert To-&gt;1D Math Input. The result in red replaces the copy of the expression. We can then copy this into a code region.</td>
</tr>
</tbody>
</table>
11.7 Preventing catastrophic loss of worksheet information

The advice is: Save the current state of your work after making significant changes to the content of your worksheet.

When you start working for longer periods of time with computers, the unexpected may become more likely. Several of these unexpected scenarios could cause your work to be seriously disrupted.

1. There could be a power failure, or a system crash or freeze.

Saving your document before embarking on a test run will at least allow you to recover what you had before the crash. Crashes or freezes are not totally unheard in highly interactive programs with complicated interfaces, such as web browsers, Java-based applets, or systems such as Maple (whose interface is Java based). While you can try to recover from what Maple saved automatically while you were working (see the discussion in ???), saving a file yourself gives you explicit control over what you will recover after the crash.

2. Your most recent changes to your work are a big mistake. Suddenly you wish you could revert back to the way things were an hour ago, or yesterday.

Unless you are working in an environment that automatically preserves successive versions of your work, you will have to keep copies the old fashioned way -- by saving them under different names. One simple way is to append a version number, e.g. MyLab1-1.mw, MyLab1-2.mw, etc. If you aren't up to keeping track of the latest version number yourself, you could also use a time index label, e.g. MyLab1-103109-1014.mw.

3. Your computer could break down, making the files on it unavailable until the computer repaired, or even permanently unreadable should repairs not be possible. Computer theft is another way of losing the files on that computer permanently.

This doesn't happen all that often, but probably happens to everyone who uses a computer for a few years. Hopefully you have saved copies of your valuable files elsewhere. This process is called creating a backup copy of your files, or just backing up. In this era of "cloud computing" and inexpensive external storage (writable DVDs, external flash or hard drives), this is fairly easy to do for those who have the discipline to back up regularly. There are even some commercial programs (e.g. Mozy https://mozy.com/registration, SpiderOak https://spideroak.com/, or Macintosh Time Machine http://www.apple.com/macosx/what-is-macosx/time-machine.html, for example) that create "back up copies" automatically at periodic intervals without any effort on your part other than to make sure that the computer is powered on when the back ups should happen. The main point is to have the discipline to create backups before you end up in a situation where you have to work many weeks to recreate lost work.

Creating and having convenient access to back ups is expected for professionals who use computers. Losing the most recent version of a file may happen to everyone occasionally; losing a valuable file completely is as lame an excuse for a professional to use as "I forgot to do my homework" is for a student barring true calamities such as a natural disaster.
## 11.8 Summary of Chapter 11

<table>
<thead>
<tr>
<th>Code Development Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a code outline as a series of comments in a code region before you start coding. Refine the outline by adding subpoints until you have supplied enough details to write code.</td>
</tr>
<tr>
<td>Add <code>print</code> or <code>printf</code> statements to code in order to make the execution trace more intelligible and to make key results easier to find within it.</td>
</tr>
<tr>
<td>The goal in debugging is typically to inspect the execution trace and find the first place where things go wrong. Use this information to determine the line(s) of code you should look at to find and fix the causes of problems. In order to do this you need to know what is right, and you should create situations where you don't have to look in too many places.</td>
</tr>
<tr>
<td>Develop incrementally -- enter only a few lines of code at a time, and get them to work before adding more. Since comments always work, you can enter entire outlines without having to be particularly incremental.</td>
</tr>
<tr>
<td>Use an interactive worksheet to test out small segments of code before you enter them into a worksheet. This can be an easy way of entering a mathematical expression, for example.</td>
</tr>
<tr>
<td>Save your work to avoid losing it. Save your work using versioned names in order to avoid having new work mess up clean copies of things that used to work. Save your work on other computers to avoid losing it completely should your computer break.</td>
</tr>
</tbody>
</table>
12 Tables and other data structures

12.1 Chapter Overview

This brief chapter introduces Maple tables. Tables are another way in Maple to store multiple values in a single container. Like most other structures in Maple, they can hold any number of items.

12.2 A review of Maple data structures and properties

The value of computers in technical work comes not only from doing long calculations conveniently, but being able to do thousands or millions of shorter computations quickly. In order to harness this power without the programming becoming very tedious, we need to store inputs and outputs in data containers or data structures. We've already seen lists and sets as containers. Maple also has specialized containers used for particular purposes: ranges, equations, and expressions.

Let's review the data containers we've seen and used in Maple so far. Each kind of container has a different syntax to create it, and to access its parts. Variables can be assigned as their value their whole container. Some Maple functions use the whole container, while in other cases you use "parts access" to use particular items within the container. One common aspect of these data containers is that they are immutable: once you've created them, you can't change them, you can only access (find out the value of) their value. In order to "change" a list or expression, for example, what you have to do is to create a new list/expression with the changes in it, and assign it to the same variable as had the old list.

<table>
<thead>
<tr>
<th>Type of container</th>
<th>Examples</th>
<th>How to create it</th>
<th>How to access a part of it</th>
<th>Operations that can be performed with container</th>
<th>Ways of changing data container after it's been created</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math expression</td>
<td>a + b + c</td>
<td>Type it in using an arithmetic operation or mathematical function</td>
<td>( \text{op}(i, \text{expr}) ) gets the ( i )-th piece of the expression</td>
<td>Arithmetic and math functions: (+, -, \times, \div, \sqrt{}, !), etc., ( \text{nops} \text{(expr)} ) tells you the number of terms in the expression</td>
<td>Can't do it. Can only create a new container with edited content.</td>
</tr>
<tr>
<td>Range</td>
<td>1..10 as in ( \text{plot}(x^2, x=1..10) )</td>
<td>Type it in using a &quot;..&quot;</td>
<td>( \text{op}(1, r) ) gets the low end of the range. ( \text{op}(2, r) ) gets the high end</td>
<td>None other than access functions, but many functions use ranges as one of their parameters.</td>
<td>Can't do it.</td>
</tr>
<tr>
<td>Equation</td>
<td>( 2<em>x-3=5 ) as in ( \text{solve}(2</em>x-3=5, x) )</td>
<td>Type it in using a &quot;=&quot;</td>
<td>( \text{rhs(eqn)}, \text{lhs(eqn)} ) gets the right and left hand sides of the equation</td>
<td>( \text{solve}, \text{fsolve}, \text{plot}, \text{etc.} ) although usually other arguments are necessary</td>
<td>Can't do it.</td>
</tr>
<tr>
<td>List</td>
<td>[1.1, 2.3, 5.5]</td>
<td>Type in items separated by commas, enclosed by square brackets.</td>
<td>( L[i] ) gets the ( i )-th item from the list. If ( i ) is negative, counting starts from the end. ( L[i..j] ) gets a list of the ( i )th through ( j )th items.</td>
<td>( \text{nops} ) tells you the number of items in the list</td>
<td>Can't do it.</td>
</tr>
<tr>
<td>Sequences</td>
<td>1.1, 2.3, 5.5</td>
<td>Type in items, separated by commas.</td>
<td>( L[i] ) gets the ( i )-th item from the list. If ( i ) is negative, counting starts from the end. ( L[i..j] ) gets a list of the ( i )th through ( j )th items.</td>
<td>All Maple functions accept sequences, although not all of them work with sequences that are the wrong length or provide the wrong kinds of values.</td>
<td>Can't do it.</td>
</tr>
<tr>
<td>Set</td>
<td>{1.1, 2.3, 5.5}</td>
<td>Type it in: items separated by commas, enclosed by curly brackets.</td>
<td>( S[i] ) gets the ( i )-th item from the set.</td>
<td>( \text{nops} ) tells you the number of items in the list union, intersect, minus, perform set operations</td>
<td>Can't do it.</td>
</tr>
</tbody>
</table>
### 12.3 Tables

In this section, we introduce another kind of data structure, tables. A table is a structure where each element is accessed using a value called a key or index. Individual elements of the table can be assigned, in any order. Individual elements can be accessed using the index notation, e.g., \( t[2] \) to get the element of the table \( t \) whose access key is 2. This is similar to how lists worked, where \( L[2] \) for a list \( L \) was an expression whose value was the 2nd element of the list \( L \).

The key can be any valid Maple value, not necessarily an integer (although they often are). Tables can add elements over time with additional assignments.

Tables are useful in situations where we will be adding items to the data structure frequently, but we don't know in advance how many entries we will ultimately have. They will be more efficient to use than lists or sets in that kind of situation. In future chapters, we will also see ways to exploit the other useful property of tables, where it is more convenient to have non-integer values for access keys.

<table>
<thead>
<tr>
<th>Demonstration of tables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>( T := table() )</td>
<td></td>
</tr>
<tr>
<td>( table([ ]) )</td>
<td></td>
</tr>
<tr>
<td>(12.1)</td>
<td></td>
</tr>
<tr>
<td>( T[1] := 3.5 )</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>(12.2)</td>
<td></td>
</tr>
<tr>
<td>( T[2] := 4.6 )</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td>(12.3)</td>
<td></td>
</tr>
<tr>
<td>17.3</td>
<td></td>
</tr>
<tr>
<td>(12.4)</td>
<td></td>
</tr>
<tr>
<td>( T[2] := 4.7 )</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>(12.5)</td>
<td></td>
</tr>
<tr>
<td>( T[-1] )</td>
<td></td>
</tr>
<tr>
<td>( T_{-1} )</td>
<td></td>
</tr>
<tr>
<td>(12.6)</td>
<td></td>
</tr>
<tr>
<td>( indices(T) )</td>
<td></td>
</tr>
<tr>
<td>([1],[2],[4])</td>
<td></td>
</tr>
<tr>
<td>(12.7)</td>
<td></td>
</tr>
</tbody>
</table>

\( S[i..j] \) gets a set of the \( i^{th} \) through \( j^{th} \) items.
### Example

- $\textbf{entries}(T)$
  
  $[3.5, [4.7], [17.3]]$  

  (12.8)  

  The $\text{entries}$ function will return a sequence of values stored in the table. The sequence is generally listed in the same order that the $\text{indices}$ function lists access keys.

- $\textbf{nops}([\text{indices}(T)])$
  
  4  

  (12.9)  

  This is the way to tell how many items are in a table at any time. Note that $\text{nops}(%)$ does not work because $\text{indices}$ produces a sequence of items, so that $\text{nops}$ would typically be given several arguments instead of the one that it expects.

- $L := \text{convert}(T, \text{list})$
  
  $[3.5, 4.7, 17.3]$  

  (12.10)  

  The $\text{convert}$ command can be used to put all the entries of a table into one list.

- $T[x] := 92 \cdot \exp(-3 \cdot i)$
  
  $92 e^{-3 i}$  

  (12.11)  

  The indices of a table can also be letters. Here we assign a value to the index $T[x]$.

- $\text{indices}(T)$
  
  $[1], [2], [4], [x]$  

  (12.12)  

  - $\text{entries}(T)$
    
    $[3.5, [4.7], [17.3], [92 e^{-3 i}]]$  

    (12.13)

  
  $T'[2]$  

  (12.14)  

  This is the way to remove an entry from the table -- putting quotes (' , not backquotes `) around the item being removed. This is also the way unassignment of values were done for ordinary variables in section 4.2.

- $\text{indices}(T)$

  $[1], [4], [x]$  

  (12.15)

- $\text{Lnew} := \text{convert}(T, \text{list})$
  
  $[3.5, 17.3, 92 e^{-3 i}]$  

  (12.16)

- $S := \text{table}([1 = 2, 2 = 4, 3 = 9])$

  $\text{table}([1 = 2, 2 = 4, 3 = 9])$  

  (12.17)

  If we want to fill up a table by giving a multiple entries at once, we can give the $\text{table}$ function a list of equations indicating indices and the associated value.

- $S[1]$

  $2$  

  (12.18)

- $S[3]$

  $9$  

  (12.19)

There are three major differences between tables in Maple and the data containers we've seen previously: a) there is a separate creation step (creating a table), b) you can change the contents of the table be assigning , c) items within a table are stored under a key or index value. The latter works similar to the indexing used for lists, but the index does not have to be an integer. A table can store any kind of item under any kind of key. For the time being, we will stick with integer keys, but eventually we will take greater advantage of this "key" feature.
12.4 Troubleshooting with tables

1. Table creation $tbl := table( )$;
   should always appear at the beginning of a script, which assigns $tbl$ a fresh, empty table. If you don't do this, you may see items left over in $tbl$ from the last time you executed a script in the same worksheet with $tbl$ in it. Alternatively, you can do a restart; before running a script that uses a table.

2. Use entries and indices to see what's in your table. Check the results to make sure that there are no unexpected indices and that the values are computed correctly.

3. Know when to use square braces and when to use parentheses in working with tables.

<table>
<thead>
<tr>
<th>Troubleshooting with tables: mistaking [ ] and ()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T := table( )$;</td>
<td>Accessing a table value requires the use of square brackets [ ] to enclose the index. Use of parentheses makes Maple think you are talking about functions rather than tables.</td>
</tr>
<tr>
<td>$\text{table([ ])}$</td>
<td>(12.20)</td>
</tr>
<tr>
<td>$T(3) := 4.7$</td>
<td>We place an entry into the table -- the value 4.7 accessed by the key 3.</td>
</tr>
<tr>
<td>Error, invalid left hand side in assignment</td>
<td>(12.21)</td>
</tr>
<tr>
<td>$T[3] := 4.7$</td>
<td>This does not access values in the table. It uses ( )s rather than the necessary [ ]s.</td>
</tr>
<tr>
<td>$4.7$</td>
<td>(12.22)</td>
</tr>
<tr>
<td>$T(3)$</td>
<td></td>
</tr>
<tr>
<td>$\text{table([3 = 4.7])(3)}$</td>
<td></td>
</tr>
<tr>
<td>$T[3]$</td>
<td></td>
</tr>
<tr>
<td>$4.7$</td>
<td>(12.23)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Troubleshooting tables: checking for extra keys</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T := table( )$</td>
<td>This a demonstration of how the fact that &quot;a table key can be any value&quot; can create confusion. We create a table and place three entries into it.</td>
</tr>
<tr>
<td>$\text{table([ ])}$</td>
<td>(12.24)</td>
</tr>
<tr>
<td>$4.7$</td>
<td>(12.25)</td>
</tr>
<tr>
<td>$T[2] := 4.8$</td>
<td></td>
</tr>
<tr>
<td>$4.8$</td>
<td>(12.26)</td>
</tr>
<tr>
<td>$4.9$</td>
<td>(12.27)</td>
</tr>
<tr>
<td>$T[3]$</td>
<td></td>
</tr>
<tr>
<td>$4.9$</td>
<td>(12.28)</td>
</tr>
</tbody>
</table>
### Troubleshooting tables: checking for extra keys

We assign \(i, j, \text{ and } k\) the values of 2, 3.0 and 4. We expect that this will result in creating or changing certain entries of the table.

<table>
<thead>
<tr>
<th>(i := 2)</th>
<th>(k \leftarrow i + 1.0)</th>
<th>(j \leftarrow 6 - i)</th>
<th>(T[k] \leftarrow 5.0)</th>
<th>(T[j] \leftarrow 5.1)</th>
<th>(T[1] := 4.75)</th>
<th>(T[1], T[2], T[3], T[4])</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.0</td>
<td>4</td>
<td>5.0</td>
<td>5.1</td>
<td>4.75</td>
<td>4.75, 4.8, 4.9, 5.1</td>
</tr>
</tbody>
</table>

(12.29) (12.30) (12.31) (12.32) (12.33) (12.34)

Where did the 5.0 go? We thought we did \(T[3.0] := 5.0\) with the use of \(k\).

<table>
<thead>
<tr>
<th>(indices(T))</th>
<th>(entries(T))</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1], [2], [3], [4], [3.0])</td>
<td>([4.75], [4.8], [4.9], [5.1], [5.0])</td>
</tr>
</tbody>
</table>

(12.35) (12.36) (12.37)

We use the \(indices\) and \(entries\) functions to tell us what is going on in \(T\). Evidently the keys used for \(T\) include both 3 and 3.0. Dimly, light comes on where we recall that 3 and 3.0 are regarded as different in Maple. "3" is an exact integer and "3.0" is a floating point number.

<table>
<thead>
<tr>
<th>(T[3.0])</th>
<th>(T[3])</th>
<th>(T[3] := 5.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>4.9</td>
<td>5.0</td>
</tr>
</tbody>
</table>

(12.38) (12.39) (12.40)

We see that there are two distinct entries in the table. Reassigning one of them does not erase the other entry, although the value of the first one does change.

<table>
<thead>
<tr>
<th>(indices(T))</th>
<th>(entries(T))</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1], [2], [3], [4], [3.0])</td>
<td>([4.75], [4.8], [5.0], [5.1], [5.0])</td>
</tr>
</tbody>
</table>

(12.41) (12.42)
Troubleshooting tables: checking for extra keys

\[ T[3.0] := 'T[3.0]' \]
\[ T_{3.0} \]  
(12.43)

As indicated in the demonstration of tables (page 185), unassigning a table entry will erase it from the table.

\[ indices(T) \]
\[ [1], [2], [3], [4] \]  
(12.44)

\[ entries(T) \]
\[ [4.75], [4.8], [5.0], [5.1] \]  
(12.45)

\[ T[3.0] \]
\[ T_{3.0} \]  
(12.46)

12.5 Summary of Chapter 12

<table>
<thead>
<tr>
<th>Type of container</th>
<th>Examples</th>
<th>How to create it</th>
<th>How to access a part of it</th>
<th>Operations that can be performed with container</th>
<th>Ways of changing data container after it's been created</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>T := table(); T[1] := 32.5; T[2] := 32.6; tblName := table();</td>
<td>T[1] := 32.7</td>
<td>T[key] retrieves the value stored under the key from within the table. Thus if you have done T[1] := 32.5; then subsequently doing T[1] will retrieve the 32.5. The key can be any value Maple expression. Thus x, or t=3..5 can be keys for a Table.</td>
<td>entries(T) provides a sequence of the values stored in the table (including duplicate values)</td>
<td>Key-value assignment: T[1] := 32.7</td>
</tr>
</tbody>
</table>
13 For Looping: repetitive execution

13.1 Chapter Overview

The computer can carry out thousands or millions of instructions in Maple per second. While a "powerful" instruction such as *solve* or *plot* to Maple is a way of having a few lines of code tap into this power, we need a way of specifying calculations that perform a lot of work without writing a commensurate amount of code. Maple's *for...do...end do* and *while...do...end do* instructions are a way of specifying that lines of code be repeated a definite number of times, or indefinitely until some condition is no longer true. This chapter describes the syntax -- how to say it -- and some example calculations that use this capability.

Such programming is called *looping* because the computer is told to loop back and repeat some instructions. Loop programming is a basic skill all programmers need to know, since most programming languages have similar ways of doing repetition.

13.2 Repetition with *to...do...end do*

In programming it's common to want to repeat a segment of a code region a few times. We could do that by copying and pasting the code that many times, but this gets tedious. We might also want to place the repetition factor under control of a script parameter, so that it would be easy to change the number of times things could be repeated.

Maple provides a convenient way to control the repetition of a code segment through something called a *loop*. The segment of code that is repeated is called the *loop body*. Its general form looks like this:

to \( n \) do
segment of code
end do;

where \( n \) is an expression describing how many times the segment should be repeated.

Table 13.1: Examples of repetitive looping with *to...do...end do*

<table>
<thead>
<tr>
<th>Examples of repetitive looping with <em>to...do...end do</em></th>
<th>The '*' at the end of the printf is necessary or else all the output will appear on the same line.</th>
</tr>
</thead>
<tbody>
<tr>
<td>to 5 do</td>
<td>Recall that the execution trace contains both the results of each line execution, and the output of any print or printf statements that are being executed. Since the ?? which displays as &quot;nothing&quot;, this execution trace contains just the side effect of displaying the various messages.</td>
</tr>
<tr>
<td>printf(&quot;I will not burp in class.\n&quot;);</td>
<td>(For these and other repetitive sayings by Bart Simpson, see <a href="http://www.snpp.com/guides/chalkboard.openings.html">http://www.snpp.com/guides/chalkboard.openings.html</a>.)</td>
</tr>
<tr>
<td>end do; printf(&quot;The repetition is over.\n&quot;);</td>
<td>This script makes it easy to change the number of times the loop repeats itself, through the standard</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>I will not not burp in class.</td>
<td></td>
</tr>
<tr>
<td>The repetition is over.</td>
<td></td>
</tr>
<tr>
<td>n := 7;</td>
<td></td>
</tr>
<tr>
<td>to n do</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;I will write %d times, &quot;Underwear should be worn on the inside.\n&quot;, n);</td>
<td></td>
</tr>
<tr>
<td>end do; printf(&quot;The repetition is over.\n&quot;);</td>
<td></td>
</tr>
</tbody>
</table>
7

I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
I will write 7 times, "Underwear should be worn on the inside."
The repetition is over.

n := 7;
to n do
  printf("I will write %d times, "Underwear should be worn on the
inside.\n", n);
  printf("The repetition is over.\n");
end do;

Warning, premature end of input, use <Shift> + <Enter> to avoid
this message.

It's stylistic convention to indent the loop body code although not required by the grammatical rules of Maple. This is because it makes it easier to determine the extent of loops particularly in more complicated codes where there are multiple loops or even nested loops -- loops within loops.

## 13.3 Repetition with for ... to ... do ... end do and for ... to ... by .... do ... end do

We can introduce a "stepper variable" into the repetition through the use

for stepperVariable from initialValue to finalValue do .... end do;

Sometimes the stepper variable is referred to as an index variable.

The first time the code segment is executed, the value of the stepper variable is initialValue.
The second time, it is set to one more, initialValue + 1. etc. The last repetition is typically final value.

Typically, initialValue and finalValue are expressions that evaluate to integers (whole numbers). If initialValue is omitted, then by default it is 1.

**Table 13.2: Example of repetition with a stepper/index variable: for ... from .. to ... do ... end do**

<table>
<thead>
<tr>
<th>Example of repetition with a stepper/index variable: for ... from .. to ... do ... end do</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Set parameters</td>
</tr>
<tr>
<td>n := 5;</td>
</tr>
<tr>
<td>#Repeat an action with i stepping through the values</td>
</tr>
<tr>
<td>#1, 2, 3, 4, ... n</td>
</tr>
<tr>
<td>for i to n do</td>
</tr>
<tr>
<td>printf(&quot;Repetition %d: \sin(%d) = %f.\n&quot;, i, i, evalf(sin(i)))</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>printf(&quot;The repetition is over.\n&quot;);</td>
</tr>
</tbody>
</table>

Since the initial value is omitted, the default value of 1 for initialValue is used here.

Maple, having Algol 68 in its ancestry, allows loop bodies to end with "od" (do spelled backwards) as well as the more English like "end do". Either one is equally acceptable.
We can make the `stepperVariable` take on different values between `initialValue` and `finalValue` through the use of a `stepValue`.

```
for stepperVariable from initialValue to finalValue by stepValue do ...
end do
```

will repeat the loop.

**Table 13.3: Example of repetition with a stepper/index variable: for ... from .. to .. by ... do ... end do**

```
#Final value parameter
fv := 1.2;

#Repeat an action with i stepping through the values
#0, .1, .2, .3, ....
for x from 0 by .1 to fv do
  printf("sin(%f) = %f.\n", x, evalf(sin(x))) end do;

printf("The repetition is over.\n"):  
```

This kind of loop can be used to print out tables of values, where each row depends on the value of the stepper variable.

What would we have to change in order to print out values of since for 0.0, 0.05, 0.10, 0.15?

```
#Final value parameter
fv := 1.2;

#Repeat an action with i stepping through the values
#0, .1, .2, .3, ....
for x from fv to 0 by -.1 do
  printf("exp(%f) = %f.\n", x, evalf(exp(x))) end do;
```

We can make the stepper value decrease in values with each repetition by having the starting value (in this case, 1.2) larger than the final value (in this case, 0), and making the step value negative (-.1).
exp(1.200000) = 3.320117.
exp(1.100000) = 3.004166.
exp(1.000000) = 2.718282.
exp(0.900000) = 2.459603.
exp(0.800000) = 2.225541.
exp(0.700000) = 2.013753.
exp(0.600000) = 1.822119.
exp(0.500000) = 1.648721.
exp(0.400000) = 1.491825.
exp(0.300000) = 1.349859.
exp(0.200000) = 1.221403.
exp(0.100000) = 1.105171.
exp(0.000000) = 1.000000.

13.4 Repetition with for... from ... to ... by .... do ... end do

General form:

for stepperVariable from initialValue to finalValue by stepValue do
  loop body;
end do;

We can make the stepperVariable take on different values between initialValue and finalValue through the use of a stepValue. We can even make the stepperVariable decrease rather than increase by giving a negative value for the stepValue.

For loop with a different step size

for x from 0 by .1 to 1 do
  printf("sin(\%f) = \%f.\n", x, evalf(sin(x)));
end do;

sin(0.000000) = 0.000000.
sin(0.100000) = 0.099833.
sin(0.200000) = 0.198669.
sin(0.300000) = 0.295520.
sin(0.400000) = 0.389418.
sin(0.500000) = 0.479426.
sin(0.600000) = 0.564642.
sin(0.700000) = 0.644218.
sin(0.800000) = 0.717356.
sin(0.900000) = 0.783327.
sin(1.000000) = 0.841471.

The from, to, and by portions of a for are all optional. If that portion is skipped, then 1 is used for the initialValue, finalValue, or stepValue. We can give the information in any order and it will mean the same thing. Thus "for i from 1 to 10 by 3..." is as acceptable as "for i to 10 by 3 from 1 do".

For loop with a decrementing rather than incrementing steps

for x by -.3 to -1 do
  printf("sqrt(\%f) = \%a\n", x, evalf(sqrt(x)));
end do;
We are using %a format rather than %e or %f format because some of the answers are imaginary numbers, which are not floating point so don't work with the other formats.

Since there is no "from" in the for statement, the initial value of x is 1. Since the step size is -.3, then the finalValue should be less than initialValue of 1.

13.5 A pattern of looping: accumulation of results

So far we have seen examples of loops where what happens inside the loop is based on the current value of the index. However, by doing a little extra work before and during the loop, we can calculate accumulative results.

For example, suppose we want to add together the first 100 numbers.

We could start typing in the single expression:

result := 1 + 2 + 3 + 4 + 5 +

but we can see that this is going to be a lot of work. We can do this with a loop by using the ability of variables to add onto their present value. What is required to initialize the variable, and then repetitively accumulate.

Table 13.4: An example of an accumulation loop

<table>
<thead>
<tr>
<th>An example of an accumulation loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Set parameters of script</td>
</tr>
<tr>
<td>n := 10;</td>
</tr>
<tr>
<td>#initialize accumulation variable</td>
</tr>
<tr>
<td>result := 0;</td>
</tr>
<tr>
<td>#repeat: add i to the accumulated sum</td>
</tr>
<tr>
<td>for i to n do</td>
</tr>
<tr>
<td>result := result+i;</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>printf(&quot;The sum of the numbers from 1 to %d is: %d&quot;, n, result);</td>
</tr>
</tbody>
</table>
The sum of the numbers from 1 to 10 is: 55

We see that the execution trace shows \( n \) and \( \text{result} \) being assigned its pre-loop initial value. With each repetition of the loop, result is changed, with the current value of the stepper variable \( i \) added onto it. In the first repetition, \( i \)'s initial value is added onto result. In the next repetition, \( i \) has a value of 2, so 2 is added onto result (producing 3). This continues until the last repetition, where \( i \)'s final value of 10 is added onto result.

We don't have to accumulate results in sums. For example, we can accumulate results in a table, or multiple tables.

**Table 13.5: Accumulating results in two tables and plotting them**

<table>
<thead>
<tr>
<th>Accumulating results in two tables and plotting them</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Set parameters of script</td>
</tr>
<tr>
<td>( n := 5; )</td>
</tr>
<tr>
<td>#initialize accumulation variables (empty tables)</td>
</tr>
<tr>
<td>( yVals := \text{table}(); )</td>
</tr>
<tr>
<td>( tVals := \text{table}(); )</td>
</tr>
<tr>
<td>#repeat: calculate ( 10^i ) and its algorithm. Store in table</td>
</tr>
<tr>
<td>#do this for ( i = 1, 2, 3, 4, 5 ).</td>
</tr>
<tr>
<td>for ( i ) to ( n ) do #1 will be used for the starting value since</td>
</tr>
<tr>
<td>#there is no &quot;from&quot; given. It's optional.</td>
</tr>
<tr>
<td>( tVals[i] := 10^i; )</td>
</tr>
<tr>
<td>( yVals[i] := \text{evalf}(\log(tVals[i])); )</td>
</tr>
<tr>
<td>printf(&quot;Logarithm of %f is %f\n&quot;, tVals[i], yVals[i]);</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>#After the loop is finished, convert tables into a list</td>
</tr>
<tr>
<td>( tList := \text{convert}(tVals, \text{list}); )</td>
</tr>
<tr>
<td>( yList := \text{convert}(yVals, \text{list}); )</td>
</tr>
</tbody>
</table>
#Do a data plot
plot(tList, yList, style=point, symbol=diamond, color=red, labels=["x","log(x)"]);
Logarithm of 10.000000 is 2.302585

100
4.605170186
Logarithm of 100.000000 is 4.605170

1000
6.907755279
Logarithm of 1000.000000 is 6.907755

10000
9.210340372
Logarithm of 10000.000000 is 9.210340

100000
11.51292546
Logarithm of 100000.000000 is 11.512925

[10, 100, 1000, 10000, 100000]
[2.302585093, 4.605170186, 6.907755279, 9.210340372, 11.51292546]
13.6 Suppressing execution trace output inside loops

Note that we can make the script add the numbers from 1 to 1000 by changing the one line defining the value of $n$. However, the output grows somewhat tedious to view. We can suppress the part of the execution trace by putting a colon at the end of the "end do".

**Table 13.6: Accumulation of two results in the same loop; suppression of output**

<table>
<thead>
<tr>
<th>Accumulation of two results in the same loop; suppression of output</th>
<th>What the execution trace of this loop looks like without output suppression. Since the loop computes two things per repetition, and repeats 10 times, there are 10 lines of output to look at. Imagine what would happen if there $n$ were 100.</th>
</tr>
</thead>
<tbody>
<tr>
<td>n := 5; result := 0; result2 := 1; for i to n do result := result+i; result2 := result2*i^2; end do; printf(&quot;The sum of the numbers from 1 to %d is: %d.\n&quot;, n, result); printf(&quot;The product of squares i^2 from i=1 to %d is: %d.\n&quot;, n, result2);</td>
<td></td>
</tr>
</tbody>
</table>
The sum of the numbers from 1 to 5 is: 15.
The product of squares $i^2$ from $i=1$ to 5 is 14400.

We suppressed output from the execution trace by placing a colon at the end do, which suppresses the execution trace of everything in the loop body. This is something to do after you have gotten the loop to work correctly.
13.7 Application of accumulative looping: making plots and movies with lists and tables

Table 13.7: Plot the position of a particle moving under Henon's rules of motion

| #Henon attractor
| #Example of a "chaotic dynamical system"
| #Governed by the equations of motion where
| #next value of x = present value of y + 1 - 1.3*(present value of x)^2
| #next value of y = 0.3*(present value of x)
| #We start with x and y initially 0.
| #We store successive values in a table for x and y values.
| #The keys (indices) are the steps i = 0, 1, 2, 3....

#Initialize x and y tables
x := table():
y := table():
x[0] := 0;
y[0] := 0;

#Take 5000 time steps through the system
for i from 0 to 4999 do
    x[i+1] := y[i] + 1 - 1.3*(x[i])^2;
y[i+1] := 0.3*x[i];
end do: #Suppress output of loop (because we've already tested it)

xl := convert(x,list):
yl := convert(y,list):
printf("Size of list :\n",nops(xl));
plot(xl,yl, color=red);
The accumulation variables are the two tables \( x \) and \( y \). As the loop repeats the tables fill with successive computed values. Even though the motion is circumscribed, the motion of the particle cannot be described by a simple mathematical formula such as the models we used for trajectories of a Human cannonball.

**Example: make a movie of the position of a particle moving under Henon's rules of motion**

```plaintext
# Henon attractor
# Example of a "chaotic dynamical system"

# Governed by the equations of motion where
# next value of \( x \) = present value of \( y \) + 1 - 1.3*(present value of \( x \))^2
# next value of \( y \) = 0.3*(present value of \( x \))

# We start with \( x \) and \( y \) initially 0.
# We store successive values in a table for \( x \) and \( y \) values.
# The keys (indices) are the steps \( i = 0, 1, 2, 3... \)

# Initialize \( x \) and \( y \) tables
x := table():
y := table():
p := table();

x[0] := 0;
y[0] := 0;
p[0] := plot([x[0]], [y[0]], style=point, color=red);

# Take 5000 time steps through the system
for i from 0 to 49 do
```

For Looping: repetitive execution
\[
x[i+1] := y[i] + 1 - 1.3*(x[i])^2;
y[i+1] := .3*x[i];
p[i+1] := plot([x[i+1]], [y[i+1]], style=point, color=red);
\]
end do: # Suppress output of loop (because we've already tested it)

pL := convert(p, list):
printf("Size of list : %d\n", nops(pL));

with(plots):
display(pL, insequence=true);

table([ ])

0
0

\textit{PLOT(\ldots)}

Size of list : 51
The accumulation variables are the two tables x and y as in the previous example, along with the table p that contains a point plot for each value of the index \( i \). At the end, instead of printing or plotting the values of x and y, the \textit{insequence} option of display is used to take the list of point plots and turn them into an animation. Compared to the graph, the animation is another way of gaining insight into how the particle moves over the time.

13.8 Troubleshooting with loops

Some of the basic problems come from forgetting the "end do" or putting too many in. Examples of this were given \textit{left out} (page 190).

Another problem with a loop is that the values provided for its control cause a different effect than you expected. Discover the difference between what the rules say will happen and what you thought would happen by observing the execution trace, possibly putting in print statements.

Finally, in accumulation loops, the initial value of the accumulator variable may be wrong, or the accumulation is not happening as planned. The control of the loop may stop the accumulation too early, or too late.

\begin{verbatim}
Troubleshooting loops: a script that doesn't work

s := 0;
x := .2;
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
  #We changed the order of to and from, but that's not an error
  s := s + x^i/i!;
end do;

printf("%f compared to expected %f", s, exp(x));
\end{verbatim}
The execution trace seems to indicate that the right number of terms is being computed, but the answer we computed by our own programming is different than what Maple's built-in \texttt{exp} function produces. The computed answer seems to be one less than the answer we expect. This is different than the other examples previously shown where the programmed results agreed quite closely with Maple's built-in computation.

\begin{verbatim}
Rerunning the script with more information generated by the execution trace
s := 0;
x := .2;
#Approximate \texttt{exp(x)} = 1 + x + x^2/2! + x^3/3! + ...
for i to 10 from 1 by 1 do
    printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);
s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));
\end{verbatim}

We enhance the execution trace by having it print out the values of all the variables with each repetition. It's work to enter in this printf statement, but it pays off in helping us find the mistake more quickly. Comparing the actual value of \texttt{s} with what the program comment tells us to expect term by term, we see that while \texttt{s} starts off at zero in the first iteration, the first term (\( \frac{x^1}{1!} \)) is .2, not 1 as we expect. Looking at the second and third terms, we realize that we have forgotten to add in the first term to \texttt{s}.

There are two ways to do this -- either to start at \texttt{i=0} (since \( \frac{x^0}{0!} \) is 1) or to keep the loop as it is and just initialize \texttt{s} to 1 instead of zero before the loop starts. We opt for the latter since it doesn't require remembering facts from high school days such as that 0!=1.

\begin{verbatim}
Rerunning the script with repairs
s := 1;
x := .2;
\end{verbatim}
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
    printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);
    s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));

1
0.2
i= 1, x = 0.200000, s = 1.000000, x^i/i! = 0.200000

1.2
i= 2, x = 0.200000, s = 1.200000, x^i/i! = 0.020000

1.220000000
i= 3, x = 0.200000, s = 1.220000, x^i/i! = 0.001333

1.221333333
i= 4, x = 0.200000, s = 1.221333, x^i/i! = 0.000067

1.221400000
i= 5, x = 0.200000, s = 1.221400, x^i/i! = 0.000003

1.221402667
i= 6, x = 0.200000, s = 1.221403, x^i/i! = 0.000000

1.221402756
i= 7, x = 0.200000, s = 1.221403, x^i/i! = 0.000000

1.221402759
i= 8, x = 0.200000, s = 1.221403, x^i/i! = 0.000000

1.221402759
i= 9, x = 0.200000, s = 1.221403, x^i/i! = 0.000000

1.221402759
i= 10, x = 0.200000, s = 1.221403, x^i/i! = 0.000000

1.221402759
1.221403 compared to expected 1.221403

Making this change produces the expected agreement. At this point, we would "comment out" the printf that we put in to enhance the execution trace, and could change the value of x to compute an approximation to exp(x) for other values of x.

More pernicious are loops which do not give error messages but either stop with the wrong answer, or do not stop.

Runaway loops can be stopped by hitting the red "stop hand". But the result is not really satisfactory, as illustrated in the example below.
Table 13.8: A runaway "infinite" loop

<table>
<thead>
<tr>
<th>A runaway &quot;infinite&quot; loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for x from 3.0 by .1 do</td>
</tr>
<tr>
<td>print(&quot;sin(x)&quot;, sin(x));</td>
</tr>
<tr>
<td>end do;</td>
</tr>
</tbody>
</table>

"sin(x)", 3.0, 0.1411200081
"sin(x)", 3.1, 0.04158066243
"sin(x)", 3.2, -0.05837414343
"sin(x)", 3.3, -0.1577456941
"sin(x)", 3.4, -0.2555411020
"sin(x)", 3.5, -0.3507832277
"sin(x)", 3.6, -0.4425204433
"sin(x)", 3.7, -0.5298361409
etc., etc.

The problem with this loop is that it will never stop. Because we mistakenly left off the "to finalValue" portion of the loop (which is legal, even if dangerous), the loop will just keep repeating, adding .1 onto the value of x.

While in principle the "red stop hand" is useful for computations that don't do a lot of printing, in practice it is **bad news** if you make a mistake and execute this kind of infinite loop. Like the Sorcerer's Apprentice (see http://en.wikipedia.org/wiki/The_Sorcerer%27s_Apprentice if you don't know the cultural reference), you will see seemingly endless lines of output. Clicking on the red stop hand will eventually stop the output, but not until after a long, long wait while all the output that resulted from before you used the hand scrolls by. This is because Maple typically generates thousands of repetitions of the loop in less than a second. It takes a long time for it to print all the output from the repetitions, and you won't be able shut off the printing with the red hand.

About the only way to make the output stop is to force quit out of the Maple application. This will cause you to lose the current state of the worksheet!

If you've saved your worksheet while you were working, then you can recover something close to the state before the "infinite loop" made a mockery of your session. Alternatively, you can recover the worksheet using File->Recent Documents -> Restore Restore Backup menu item. **This will fill your Maple with copies of all the autosaved worksheets. You may then delete them or save them to permanent file space as you wish. (page 20)** as described in Restore Backup menu item. **This will fill your Maple with copies of all the autosaved worksheets. You may then delete them or save them to permanent file space as you wish. (page 20).**
## 13.9 Summary of Chapter 13

<table>
<thead>
<tr>
<th>Repetition type</th>
<th>Example</th>
</tr>
</thead>
</table>
| Repetition with `to ... do ... end do`| `to 5 do
  printf("I will not not burp in class.\n");
edo;
  printf("The repetition is over.\n");` |
| Repetition with `for ... to ... do ... end do` | `#Set parameters
  n := 5;

  #Repeat an action with i stepping through the values
  #1, 2, 3, 4, ... n
  for i to n do
    printf("Repetition %d: sin(%d) = %f.\n", i, i, evalf(sin(i)))
edo;
  printf("The repetition is over.\n");` |
| Repetition with `for ... to ... by ... do ... end do` | `#Final value parameter
  fv := 1.2;

  #Repeat an action with i stepping through the values
  #0, .1, .2, .3, ....
  for x from 0 by .1 to fv do
    printf("sin(%f) = %f.\n", x, evalf(sin(x))) end do;
  printf("The repetition is over.\n");` |
| Repetition (incrementing) with `for... from ... to ... by ... do ... end do` | `for x from 0 by .1 to 1 do
  printf("sin(%f) = %f.\n", x, evalf(sin(x))); end do;` |
| Repetition (decrementing) with `for ... to ... do ... end do` | `for x by -.3 to -1 do
  printf("sqrt(%f) = %a\n", x, evalf(sqrt(x))); end do;` |
| Accumulation loop type                     | Example                                                                 |
| Prints result                             | `#Set parameters of script
  n := 10;

  #initialize accumulation variable
  result := 0;

  #repeat: add i to the accumulated sum
  for i to n do
    result := result+i;
edo;
  printf("The sum of the numbers from 1 to %d is: %d", `
Storing results in a table

```maple
# Set parameters of script
n := 5;

# Initialize accumulation variables (empty tables)
yVals := table();
tVals := table();

# Repeat: calculate 10^i and its algorithm. Store in table
# do this for i = 1, 2, 3, 4, 5.

for i to n do # 1 will be used for the starting value since
    # there is no "from" given. It's optional.
    tVals[i] := 10^i;
    yVals[i] := evalf(log(tVals[i]));
    printf("Logarithm of %f is %f\n", tVals[i], yVals[i]);
end do;

# After the loop is finished, convert tables into a list

tList := convert(tVals, list);
yList := convert(yVals, list);

# Do a data plot
plot(tList, yList, style=point, symbol=diamond, color=red, labels=["x","log(x)"]);
```

Accumulating two results in the same loop, suppressing output

```maple
n := 5;
result := 0;
result2 := 1;
for i to n do
    result := result+i;
    result2 := result2*i^2;
end do;

printf("The sum of the numbers from 1 to %d is: %d.\n", n, result);
printf("The product of squares i^2 from i=1 to %d is %d.\n", n, result2);
```

Troubleshooting tips

Print out values of variables using execution trace

```maple
s := 1;
x := .2;

# Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...

for i to 10 from 1 by 1 do
    printf("i=$i, x=$x, s=$s, x^i/i! = $\%f\n", i, x, s, x^i/i!);
    s := s + x^i/i!;
end do;
```
printf("%f compared to expected %f", s, exp(x));

Do not create an infinite loop

# Do not run this
# We do not have a "to finalValue" portion in our for loop

for x from 3.0 by .1 do
    print("sin(x)", sin(x));
end do;


**14 More on repetition and looping**

**14.1 Chapter Overview**

In the previous chapter we have seen how `for...do...end do` can be used to repeat a segment of code, possibly with a stepping variable that is automatically managed by Maple during the execution. In this chapter we look at other ways of getting repetition to occur: through some of the built-in shortcuts such as `seq`, `sum`, and addition of lists.

We also look at `while .. do ... end do` which is a way of having repetition occur while a condition holds.

`for ... while ... do .... end do` is a way of having both a stepping variable with an upper limit, and an additional condition that can end the repetition early if it becomes false.

**14.2 Implicit looping: seq, list addition, sum**

In chapter 11, we learned about `for` and its ability to loop with an index (stepping) variable. While this is a marvelous abbreviation, it still takes work to set up. Certain repetitious actions occur so commonly that many programming languages provided even shorter ways of performing looping idioms.

For example, suppose we wanted to build a list consisting of the numbers 1, 4, 9, 25, ... 169. We could type such a list in, but from what we know of looping we suspect that there's an easier way. Using the ideas we've seen so far, we could use a table `valueTab` that has an accumulation variable `L`. With `i` as the stepping variable going from 1 to 100, we would fill up the table with $i^2$ values. After the loop finishes, we could convert the table into a list to get our result:

<table>
<thead>
<tr>
<th>Table 14.1: Making a list through a loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Making a list through a loop</strong></td>
</tr>
<tr>
<td>#initialize table</td>
</tr>
<tr>
<td>valueTab := table();</td>
</tr>
<tr>
<td>#fill up the table</td>
</tr>
<tr>
<td>for i from 1 to 13 do</td>
</tr>
<tr>
<td>valueTab[i] := i^2;</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>#Produce list as result</td>
</tr>
<tr>
<td>L := convert(valueTab,list):</td>
</tr>
<tr>
<td>printf(&quot;L is: %a&quot;, L);</td>
</tr>
</tbody>
</table>
The nice thing about this loop is that it is no more work for us to create a list with 100 items 1, 4, 9, ... 10,000. All we have to do is change the "13".

However, there is an even shorter way to create this list. Maple has a function seq whose syntax borrows heavily from for. The seq function creates a sequence as indicated by the values of the stepping variable. To get a list, we convert the sequence into a list, as explained in Chapter 12 (page 185).

### Table 14.2: Making a list with seq

<table>
<thead>
<tr>
<th>Making a list with seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>#initialize table</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>#Create a sequence</td>
</tr>
<tr>
<td>s := seq(i^2, i=1..13);</td>
</tr>
<tr>
<td>#Convert it to a list</td>
</tr>
<tr>
<td>L := [s];</td>
</tr>
<tr>
<td>printf(&quot;L is: %a&quot;, L);</td>
</tr>
</tbody>
</table>

L is: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169]

With this kind of facility, it is easy to quickly solve many problems that are solved with Maple functions that take lists.

### Table 14.3: Creating a list of values with seq

<table>
<thead>
<tr>
<th>Creating a list of values with seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list of the approximations to $\sqrt{1}$, $\sqrt{2}$, ..., $\sqrt{20}$.</td>
</tr>
</tbody>
</table>
Creating a list of values with seq

\[
\text{seq}(\text{evalf}(\sqrt{i}), i = 1..20)
\]

\[
[1., 1.414213562, 1.732050808, 2., 2.236067977, 2.449489743, 2.645751311, 2.828427124, 3.,
3.162277660, 3.316624790, 3.464101616, 3.605551275, 3.741657387, 3.872983346, 4.,
4.123105626, 4.242640686, 4.358898944, 4.472135954]
\]

(14.2)

The general forms of \texttt{seq} are

\[
\text{seq}(\text{expression involving } \text{steppingVariable}, \text{steppingVariable}=\text{range}) \quad \text{and} \quad \text{seq}(\text{expression involving } \text{steppingVariable}, \text{steppingVariable}=\text{range}, \text{stepValue}).
\]

The second form behaves like the \texttt{for...from...to...by...do...end do} described in the previous chapter.

Table 14.4: Giving lists to Maple functions using seq with a step value

Adding together the elements of two lists is another common operation that has a built-in shortcut.

Table 14.5: Adding together two lists with +

Adding a similar list of values for \( \cos \left( \frac{n \pi}{10} \right) \)

\[
L_{\cos} := \text{evalf} \left( \text{seq}(\cos \left( \frac{i \cdot \pi}{10} \right), i = 1..10) \right)
\]

\[
[0.9510565163, 0.8909169943, 0.5877852522, 0.3090169938, 0.0, 0.0, 0.5877852522, 0.3090169938, 0., -0.8909169943, -0.9510565163, -1.]
\]

(14.5)

Add them together using "+".

\[
L_{\text{sum}} := L_{\text{sin}} + L_{\cos}
\]

\[
[1.260073511, 1.396802247, 1.396802247, 1.260073510, 1., 0.6420395227, 0.2212317422,
-0.2212317419, -0.6420395219, -1.]
\]

(14.6)
Adding together two lists with +

We don't get component-wise "automatic multiplication" with "*". Maple doesn't give an error, but it doesn't do the multiplication, either.

\[ L_{\text{wrong}} := L\sin - L\cos \]

\[
\begin{bmatrix}
0.3090169944, & 0.8877852524, & 0.8090169944, & 0.9510565165, & 1., & 0.9510565165, & 0.8090169944, \\
0.8877852524, & 0.3090169944, & 0.9510565165, & 0.8090169944, & 0.5877852522, & 0.3090169938, & 0., \\
-0.3090169938, & -0.8877852522, & -0.8090169944, & -0.9510565163, & -1.
\end{bmatrix}
\]

(14.7)

We get this done without writing a loop to do it with the use of \textit{seq}.

\[ L_{\text{right}} := \{ \text{seq}(L\sin[i] \cdot L\cos[i], i = 1..\text{nops}(L\sin)) \} \]

\[
\begin{bmatrix}
0.2938926262, & 0.4755282582, & 0.4755282581, & 0.2938926257, & 0., & -0.2938926257, & -0.4755282581, \\
& -0.4755282582, & -0.2938926262, & 0.,
\end{bmatrix}
\]

(14.8)

"Automatic addition" works with numbers, expressions, lists, sequences, ranges, and equations, but does not work with tables, plots, or sets. "Automatic multiplication" only works with numbers and expressions. Maple does have additional data containers with use with linear algebra, vectors, and matrices for which both addition and multiplication work -- but that should wait until you need to do "linear algebra".

### 14.3 More implicit looping with \textit{sum}

Maple has another function which will use looping when it is expedient: summation. This is available textually through the \textit{sum} operation, and through the Expression Palette as $\sum_{i=k}^{n} f$. The general form of \textit{sum} is

\[ \text{sum}(\text{expression}, \text{indexVariable}=\text{range}) \]

When the range is numerical and contains a manageable number of values, Maple will automatically do the summation through a loop.

#### Table 14.6: Automatic summation

<table>
<thead>
<tr>
<th>Automatic summation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum_{i=1}^{100} i !$ #This does a summation using the Expression Palette</td>
</tr>
<tr>
<td>942690016837099792608598341244735398720707226139826724429383593056246782234795060234:</td>
</tr>
<tr>
<td>00294093599136466986609124347432647622826870038220556442336528920420940313</td>
</tr>
<tr>
<td>$\sum(\sqrt{i}, i = 1..10)$ #This does summation using the textual form of the operation</td>
</tr>
<tr>
<td>$1 + \sqrt{2} + \sqrt{3} + \sqrt{4} + \sqrt{5} + \sqrt{6} + \sqrt{7} + \sqrt{8} + \sqrt{9} + \sqrt{10}$</td>
</tr>
<tr>
<td>at 5 digits</td>
</tr>
<tr>
<td>22.468</td>
</tr>
</tbody>
</table>

(14.9)

(14.10)

(14.11)

However, there is more to the \textit{sum} function than just an abbreviation of a \texttt{for} loop. One can give non-numeric value as part of the range, and Maple will use mathematical analysis to give a formula for the summation if it can. Furthermore, it will this analysis to provide a short-cut to the looping if it turns out to be faster.
Maple knows the formula for the sum of 4th powers of integers.

\[
\frac{1}{4} (n + 1)^{5} - \frac{1}{2} (n + 1)^{4} + \frac{1}{3} (n + 1)^{3} - \frac{1}{30} n - \frac{1}{30}
\]  

(14.12)

If we did the following with a for loop, it would take literally centuries for even a supercomputer to complete. Rather than do that, Maple figures out the formula for the sum of squares and plugs in \(n = 10^{30}\).

\[
\text{sum}(i^{2}, i = 1..10^{30})
\]

One annoying aspect of using sum is that it expects the index variable to be unassigned (have no value). If you discover that you've used the index variable previously in the session, then you have to unassign it. This is similar to the situation that arose when solve broke because the variable being solved for had been previously assigned a value, or the variable was already the name of a built-in function.

Table 14.7: sum and non-looping behavior

<table>
<thead>
<tr>
<th>( \sum_{i=1}^{n} i^{4} )</th>
<th>Maple knows the formula for the sum of 4th powers of integers.</th>
</tr>
</thead>
</table>
|                         | \[
|                         | \frac{1}{4} (n + 1)^{5} - \frac{1}{2} (n + 1)^{4} + \frac{1}{3} (n + 1)^{3} - \frac{1}{30} n - \frac{1}{30}
|                         | \]                                                           |

(14.13)

One annoying aspect of using sum is that it expects the index variable to be unassigned (have no value). If you discover that you've used the index variable previously in the session, then you have to unassign it. This is similar to the situation that arose when solve broke because the variable being solved for had been previously assigned a value, or the variable was already the name of a built-in function.

```maple
restart;
L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];
total := sum(L[i], i=1..nops(L));
avg := total/nops(L);
for i from 1 to 10 do
    printf("Grade = %f, deviation= %f \n", L[i], L[i]-avg);
end do;
stddev := sum((L[i]-avg)^2,i=1..nops(L))/avg; #this is broken
i := 'i'; #unassign the index variable in the sum.
stddev := sum((L[i]-avg)^2,i=1..nops(L))/avg; #this works
```

\[
[3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9]
\]

30.6

2.781818182

Grade = 3.600000.2, deviation= 0.818182.2
Grade = 3.900000.2, deviation= 1.118182.2
Grade = 2.100000.2, deviation= -0.681818.2
Grade = 3.900000.2, deviation= 1.118182.2
Grade = 2.100000.2, deviation= -0.681818.2
Grade = 0.700000.2, deviation= -2.081818.2
Grade = 1.300000.2, deviation= -1.481818.2
Grade = 2.800000.2, deviation= -0.258182.2
Grade = 3.600000.2, deviation= 0.818182.2
Grade = 3.700000.2, deviation= 0.918182.2

Error, (in sum) summation variable previously assigned, second argument evaluates to 11 - 1 .. 11

\[
4.369934642
\]  

(14.14)
The error message comes when we try to compute the standard deviation. While the use of $i$ in the computation of total had no problems, the use of $i$ in the computation of the standard deviation occurred after it had been assigned values during the for loop. The way to fix this is to either use another variable that is known not to have been assigned a value (e.g. $k$), or to unassign $i$ before using it in the second sum.

### 14.4 Conditional repetition through while... do... end do

**for ... do ... end do** allows you to repeat a segment of code under control of an index variable. Typically, the starting, ending and stepping values of the index variable are set at the start of execution of the for. This means that the number of times that the loop is repeated is known in advance of the execution.

There are other situations where the number of repetitions is not known in advance. The programmer wants "keep on repeating this until a condition is reached", or alternatively "keep on repeating this while a condition is true". Maple has a way of doing this, with syntax similar to *for*.

Table 14.8: The general form of the while statement

<table>
<thead>
<tr>
<th>The general form of the while statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>while condition do</td>
</tr>
<tr>
<td>loop body;</td>
</tr>
<tr>
<td>end do;</td>
</tr>
</tbody>
</table>

The *condition* is evaluated only once per repetition, at the start before any of the code in the *loop body* is executed. If the *condition* is true, then the entire *loop body* is executed. If not, it decides that the repetition is finished and moves onto the next instruction after the end do.

The *condition* is evaluated at the start of the loop iteration. It does not matter if the *condition* changes in the middle of the *loop body*; this won't terminate execution of the loop until the execution reaches the end of the *loop body* and the *condition* looked at again before the start of the next repetition of the loop.

The continuation *condition* is usually an inequality such as $x<.1$ or $abs(y-z)> tol$. It always must be an expression that can be evaluated to be true or false. If the inequality is numeric such as $x<.1$, variables in the *condition* such as $x$ must have been assigned a numeric value earlier in the Maple session.

Table 14.9: Repeating while a condition is true

Example: Repeating *while a condition is true*: accumulate terms of a sum until next term is very small.

The code region below computes the sum

$$x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots$$

until it finds a term that is tiny (less than the value of the parameter $tol$). It does this through repeated addition to the accumulation variable $s$. The continuation condition repeats the loop body while the value of $t \left(\frac{x}{i!}\right) > tol$. This condition is evaluated only once per repetition, just before all the code in the loop body is executed. The particular term calculated during a repetition is $(-1)^{\frac{(i-1)}{2}} \cdot \frac{x^i}{i!}$ for $i = 1, 3, 5, \ldots$.

Some of you who have taken calculus may recognize this sum as an approximation to $\sin(x)$. Floating point calculations in Maple, Matlab, or calculators such as the TI-89 use sums or similar means of approximations to provide values for $\sin$ and other mathematical functions.

```plaintext
# Initialize script parameters
#tol is the size of the stopping criterion.
tol := 10e-7;
#:x is the value of sin(x) that we are trying to approximate
x := .3;
```

---

*The error message comes when we try to compute the standard deviation. While the use of $i$ in the computation of total had no problems, the use of $i$ in the computation of the standard deviation occurred after it had been assigned values during the for loop. The way to fix this is to either use another variable that is known not to have been assigned a value (e.g. $k$), or to unassign $i$ before using it in the second sum.*
# Initialize accumulation variables
i := 1; # i will be 1, 3, 5, 7, 9 as the repetition continues
term := x;
s := term; # s accumulates the sum, we initialize it to the first term.

# Add on successive terms
while abs(term) >= tol do
  i := i+2;
  term := (-1)^((i-1)/2)*(x^i)/i!;
  s := s+term;
  printf("Term is %e, sum is %f.\n", term, s);
end do: # Suppress execution trace inside the loop (but printfs still work).

# Print some summary messages.
printf("Sum is: %e compared to %e.\n", s, sin(x));
printf("Stopped when i=%d, term=%e.\n", i, term);

0.0000010
  0.3
  1
  0.3
  0.3

Term is -4.500000e-03, sum is 0.295500.
Term is 2.0250000e-05, sum is 0.295520.
Term is -4.339286e-08, sum is 0.295520.
Sum is: 2.955202e-01 compared to 2.955202e-01.
Stopped when i=7, term=-4.339286e-08.

Notes
The terms being computed are sometimes positive and sometimes negative. This is why the continuation condition uses the absolute value of the term rather than just the value of the term. It says "do another repetition while the term is bigger than tol. The last iteration of the loop computes a term that's -4.3392857140^-8. After that, the condition is not true, so the repetition stops. The final result agrees to ten decimal places with Maple's built-in sin function's.

The textual form of inequalities such as "<" and ">" are easy since there is a symbol on the keyboard for them. The following table describes the textual version of inequalities such as ≤ for which there is no single symbol on the keyboard. The symbols for inequalities are sometimes referred to as relational operators.

Table 14.10: Textual entry of relational operators

<table>
<thead>
<tr>
<th>Mathematical operator</th>
<th>Name</th>
<th>Textual entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
<td>=</td>
</tr>
<tr>
<td>≠</td>
<td>Not equals</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equals</td>
<td>&lt;=</td>
</tr>
</tbody>
</table>
Another example of accumulation

This example is similar to the previous one. It computes successive terms of a sum for the natural logarithm:

\[ \ln(x) = x - 1 - \frac{1}{2} (x-1)^2 + \frac{1}{3} (x-1)^3 - \frac{1}{4} (x-1)^4 + \frac{1}{5} (x-1)^5 + \ldots \]

but stops when the term becomes smaller than the tolerance.

```
# Computation of logarithms through sums.
# Initialize script parameters
# tol is the size of the stopping criterion.
tol := 10e-4;
# x is the value of sin(x) that we are trying to approximate
x := .2;
xm1 := x-1.0: # "x minus 1", since we use it a lot

# Initialize accumulation variables
i := 1; # i will be 1, 2, 3, 4, 5, ... as the repetition continues
term := xm1;
s := term; # s accumulates the sum, we initialize it to the first term.
xm1power := xm1: # to be used for powers of x minus 1, initially just xm1

printf("x is %f, tol is %e\n",x,tol);
# print out all values on one line as a check before we start.

# add on successive terms
while abs(term) >= tol do
  i := i+1; # compute next value of the denominator
  xm1power := -xm1power*xm1; # Next power of x-1, adjusted by +- sign.
term := xm1power/i; # Next term of the sum
  s := s + term; # Add term to sum.
  printf("term is %e, sum is %f\n", term, s);
end do;
printf("sum is: %e compared to: %e, ln(x));
print(i, "terms used");
```
The approximation does not produce ten decimal places of agreement with the logarithm as calculated by the Maple library, but it's closing in on that when the repetition stops. This is because \textit{tol} is set to $10^{-4}$, so the repetition stops as soon as a \textit{term} is computed that is less than that in absolute value. If we really wanted ten places of accuracy, we might want to set \textit{tol} to $10^{-10}$. It would make the repetition go on longer, but would produce more decimal places of agreement.

It would be naive to presume that just because we set \textit{tol} to $10^{-10}$ that the result will agree with the correct result to 10 decimal places. (What actually happens? Try it.) Mathematical analysis can be used to provide guarantees or proofs that things will work out. The area of mathematics that studies this is called \textit{numerical analysis}. Many undergraduate mathematics or computer science students interested in devising efficient accurate programs for numerical computation take a numerical analysis course as a junior or senior.
14.5 Patterns of looping

We have introduced quite a lot of machinery. It can be easy for beginners to be uncertain about the connection between \textit{intent} (what you want to achieve) and \textit{syntax} (how you say it in the programming language). Here as a short table summarizing the connections we have introduced so far

<table>
<thead>
<tr>
<th>Patterns of looping</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat exactly the same thing ( n ) times</td>
<td>to 5 do</td>
</tr>
<tr>
<td></td>
<td>\hspace{1cm} print(&quot;I will not teach others to fly&quot;);</td>
</tr>
<tr>
<td></td>
<td>\hspace{1cm} end do;</td>
</tr>
<tr>
<td></td>
<td>&quot;I will not teach others to fly&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;I will not teach others to fly&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;I will not teach others to fly&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;I will not teach others to fly&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;I will not teach others to fly&quot;</td>
</tr>
<tr>
<td></td>
<td>\hspace{1cm} (14.17)</td>
</tr>
</tbody>
</table>

The action within the loop depends on the value of variable stepping from an initial to a final value, adding a fixed amount to the stepper variable between each step

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( t ) from 0 by .1 to 1 do</td>
</tr>
<tr>
<td>\hspace{1cm} printf(&quot;arccos(%f) = %f\n&quot;, t, arccos(t))</td>
</tr>
<tr>
<td>\hspace{1cm} end do;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Accumulation: Variables within the loop accumulate results. Each repetition incrementally changes onto the partial result. When the repetition is finished, the entire result has been computed.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := .3: #Parameter</td>
</tr>
<tr>
<td>prod := 1: #Accumulation variable</td>
</tr>
<tr>
<td>#&quot;sum&quot; and &quot;product&quot; are reserved Maple names</td>
</tr>
<tr>
<td>#So we should use something else like &quot;s&quot; or &quot;prod&quot;</td>
</tr>
<tr>
<td>#for variables.</td>
</tr>
<tr>
<td>#Compute the product of ((x+2)<em>(x+3)</em>(x+4)<em>...</em>(x+15))</td>
</tr>
<tr>
<td>for ( i ) by 2 to 15 do</td>
</tr>
<tr>
<td>\hspace{1cm} prod := (x+i)*prod;</td>
</tr>
<tr>
<td>\hspace{1cm} end do: #Suppress execution trace inside the loop</td>
</tr>
<tr>
<td>print(prod);</td>
</tr>
<tr>
<td>3.54944528310^6</td>
</tr>
</tbody>
</table>
This computes an approximation to
\[ e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots. \]

#script parameters
\[ x := 0.2; \]
\[ tol := 10^{-6}; \]

#accumulation variables, initialized
\[ i := 1; \]
\[ s := 1; \] #First term of sum
\[ term := 1; \] #Next term of sum

while (abs(term) >= tol) do
  term := \(\frac{x^i}{i!}\);
  s := s + term;
  i := i + 1;
end do:

printf("compare %e to %e\n", s, exp(x));

compare 1.221403e+00 to 1.221403e+00

14.6 Combining for and while

Maple's syntax allows the use of an stepper variable and a continuation condition (other than that of the stepper variable reaching a final value.

\[\text{for} \ \text{stepperVariable from initialVariable to finalVariable by stepValue while condition do ... end do}\]

promotes a repetition which stops when either the stepperVariable reaches the finalValue, or the condition becomes false. If the to initialValue is omitted, then the while condition is the sole criterion used to decide whether to continue the repetition. This can abbreviate the coding in many standard cases.

Table 14.13: Example of for and while repetition

Example of for and while repetition: accumulate terms of a sum until next term is very small.

The code region below computes the sum
\[ x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots \] until it finds a term that is tiny (less than the value of the parameter tol). Compared to the previousExample (page 214), the main difference is that we can skip the initialization and increment of the stepper variable \(i\) since the for loop handles it for us automatically. Since there is no finalValue, the loop will not stop because the stepper variable reached a final value. However, at the start of each repetition, the condition abs(term) >= tol will also be checked. This is what stops the repetition after a few trips through the loop.

# Initialize script parameters
#tol is the size of the stopping criterion.
\[ tol := 10^{-7}; \]
#x is the value of sin(x) that we are trying to approximate
\[ x := 0.2; \]

#Initialize accumulation variables

\[ x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \ldots \]
term := x;
s := term; # s accumulates the sum, we initialize it to the first term.

# add on successive terms
for i from 1 by 2 while abs(term) > -tol do
  i := i+2;
term := (-1)^((i-1)/2) * (x^i)/i!;
printf("term is %e\n", term);
s := s + term;
end do;

# Print out the final sum, and compare it to what it's supposed to be.
printf("sum is: %e compared to %e.", s, sin(x));

0.0000010

0.3

0.3

0.3

3

-0.004500000000

term is -4.500000e-03

0.2955000000

7

-4.33928571410^-8

term is -4.339286e-08

0.2954999566

sum is: 2.955000e-01 compared to 2.955202e-01.

We can get more decimal places of agreement between our approximation and the reference result by changing the value of tol. This will cause more repetitions of the loop.

This kind of repetition is often used in practice in situations where there is no built-in function that will compute the result for you. You'd want to ask a numerical analyst to provide assurances that it's going to work well.
14.7 Troubleshooting while loops: loops that don’t stop

We've already reviewed some of the basic problems with loops that come from forgetting to put in a end do or do, or putting too many in. These typically give some kind of Error, message and were discussed left out (page 190).

More pernicious are loops which do not give error messages but either stop with the wrong answer, or do not stop.

You can tell that this is happening because you are either flooded with a lot of output, or the status window in the lower left hand corner of the worksheet says Evaluating... but never seems to return to Ready.

Table 14.14: A runaway infinite loop

<table>
<thead>
<tr>
<th>A runaway &quot;infinite&quot; loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x := 3.0;</code></td>
</tr>
<tr>
<td><code>while (x&lt;10) do</code></td>
</tr>
<tr>
<td><code>print(&quot;sin(x)&quot;, sin(x));</code></td>
</tr>
<tr>
<td><code>end do;</code></td>
</tr>
<tr>
<td>3.0</td>
</tr>
<tr>
<td>&quot;sin(x)&quot;, 0.1411200081</td>
</tr>
<tr>
<td>&quot;sin(x)&quot;, 0.1411200081</td>
</tr>
<tr>
<td>&quot;sin(x)&quot;, 0.1411200081</td>
</tr>
<tr>
<td>&quot;sin(x)&quot;, 0.1411200081</td>
</tr>
<tr>
<td>etc., etc.</td>
</tr>
</tbody>
</table>

The problem with this loop is that it will never stop. The continuation condition is x<10, which is true initially. Because the loop body never changes the value of x, it will continue to be true for every successive repetition. Presumably the programmer made a mistake and forgot to put in lines of code into the loop body that changed the value of x and (with enough repetition) made it eventually grow to 10 or larger.

In theory, you can stop long-running computations such as infinite loops with the "red stop hand" found on the Maple tool bar. Clicking on the red stop hand will eventually stop the output, but not until after a long, long wait while all the output that resulted before you told the computer to stop looping, is printed. It typically takes a long time for it to print all the output from the repetitions, and you won't be able shut off the printing with the red hand.

So in practice, if you have a runaway loop you often will have to force-quit your Maple session and recover the most recent version of your worksheet through File -> Recent Backups- > Recover Backup, as described in chapter 2. If you absolutely must execute a runaway loop again, first change it to print very little so that it will stop before you get old waiting for it.
14.8 Troubleshooting loops: loops that give the wrong answer

Loops that return the wrong answer

Loops that give the wrong answer need to be handled analytically -- you need to figure out what should be happening and why. Then compare what should be happening with what the execution trace is saying does happen to diagnose your problem. "Sick loops" take a fair bit of thought to work through, since you are looking a code that gets repeatedly executed, possibly many times.

Some tips for handling loops include:

a) Develop incrementally. Set your loops up for a few repetitions (two or three) initially. Verify that things work before you try to make the repetitions go on longer.

b) Use print/printf statements within the loop to make it clearer what is going on inside the repeated instructions.

Another problem with a loop is that the values provided for its control cause a different effect than you expected. Discover the difference between what the rules say will happen and what you thought would happen by observing the execution trace, possibly putting in print statements.

Finally, in accumulation loops, the initial value of the accumulator variable may be wrong, or the accumulation is not happening as planned. The control of the loop may stop the accumulation too early, or too late.
Table 14.16: Troubleshooting loops: a script that doesn't work

<table>
<thead>
<tr>
<th>Troubleshooting loops: a script that doesn't work</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := 0;</td>
</tr>
<tr>
<td>x := .2;</td>
</tr>
<tr>
<td>#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ...</td>
</tr>
<tr>
<td>for i to 10 from 1 by 1 do</td>
</tr>
<tr>
<td>#We changed the order of to and from, but that's not an error</td>
</tr>
<tr>
<td>s := s + x^i/i!;</td>
</tr>
<tr>
<td>end do;</td>
</tr>
<tr>
<td>printf(&quot;%f compared to expected %f&quot;, s, exp(x));</td>
</tr>
</tbody>
</table>

0.221403 compared to expected 1.221403

The execution trace seems to indicate that the right number of terms is being computed, but the answer we computed by our own programming is different than what Maple's built-in exp function produces. The computed answer seems to be one less than the answer we expect. This is different than the other examples previously shown where the programmed results agreed quite closely with Maple's built-in computation.

Rerunning the script with more information generated by the execution trace

| s := 0;                                          |
| x := .2;                                         |
| #Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ... |
| for i to 10 from 1 by 1 do                       |
|   printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!); |
|   s := s + x^i/i!;                               |
| end do;                                          |
| printf("%e compared to expected %e", s, exp(x)); |

References
We enhance the execution trace by having it print out the values of all the variables with each repetition. It's work to enter in this printf statement, but it pays off in helping us find the mistake more quickly. Comparing the actual value of s with what the program comment tells us to expect term by term, we see that while s starts off at zero in the first iteration, the first term \( \frac{x}{i!} \) is .2, not 1 as we expect. Looking at the second and third terms, we realize that we have forgotten to add in the first term to s.

There are two ways to do this -- either to start at \( i=0 \) (since \( \frac{x}{0!} \) is 1) or to keep the loop as it is and just initialize s to 1 instead of zero before the loop starts. We opt for the latter since it doesn't require remembering facts from high school days such as that \( 0!=1 \).

\[
\begin{array}{l}
\text{Table 14.17: Rerunning the script with repairs} \\
\hline
s := 1; \\
x := .2; \\
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ... \\
for i to 10 from 1 by 1 do \\
\quad printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!); \\
\quad s := s + x^i/i!; \\
end do; \\
printf("%e compared to expected %e", s, exp(x));
\end{array}
\]
Making this change produces the expected agreement. At this point, we would "comment out" the printf that we put in to enhance the execution trace, and could change the value of x to compute an approximation to exp(x) for other values of x.

### 14.9 Summary of Chapter 14

<table>
<thead>
<tr>
<th>Implicit iteration</th>
<th>Example</th>
</tr>
</thead>
</table>
| Using `seq` and `sum` | #Set parameters of script  
n := 5;  
#Unassign i just in case it's been used in the session  
#previously  
i := 'i';  
#compute consecutive powers of 10, in a list.  
tList := [seq(10^i,i=1..n)];  
#Compute logarithms of the powers you have  
yList := [seq(evalf(log(tList[i])),i=1..n)]; |
# Do a data plot
plot(tList, yList, style=point, symbol=diamond, color=red,
     labels=["x","log(x)"], symbolsize=30);

# Add together the logarithms
s := sum(yList[i], i=1..n):
printf("Sum of logarithms is:%e\n", s);

Automatic addition

L := [.3, .5];
velocity := .1;
angle := convert(55, units, degrees, radians);
dv := evalf([velocity*cos(angle), velocity*sin(angle)]);
# add dv onto L
newL := L + dv;
### Examples of while loops

**Example of while...do**

```plaintext
#Add on successive terms
while abs(term) >= tol do
  i := i+2;
  term := (-1)^((i-1)/2)*(x^i)/i!;
  s := s+term;
  printf("Term is %e, sum is %f.\n", term, s);
end do: #Suppress execution trace inside the loop #(but printfs still work).

#Print some summary messages.
printf("Sum is: %e compared to %e.\n", s, sin(x));
printf("Stopped when i=%d, term=%e.\n", i, term);
```

Sum is: 2.214028e-01 compared to 1.986693e-01.
Stopped when i=11, term=8.888889e-08.

**Example of for...to...do**

```plaintext
for i to 10 do
  printf("i= %d, ", i);
end do;
i= 1, i= 2, i= 3, i= 4, i= 5, i= 6, i= 7, i= 8, i= 9, i= 10,
```

**Example of for...to...by...do**

```plaintext
for i to 10 by 2 do
  printf("i= %d, ", i);
end do;
i= 1, i= 3, i= 5, i= 7, i= 9,
```

**Example of for...to...from...by...do**

```plaintext
for i to 10 from 5 by 2 do
  printf("i= %d, ", i);
end do;
i= 5, i= 7, i= 9,
```

### Troubleshooting tips

**Example**

```plaintext
s := 1;
x := .2;
#Approximate exp(x) = 1 + x + x^2/2! + x^3/3! + ... 
for i to 10 from 1 by 1 do
```
Have the foresight to avoid running an infinite loop. If despite your best efforts, this happens and hitting the "red stop hand" doesn't work fast enough, kill the Maple session and start again. Hopefully you have saved your work or can use the File->Open Recent->Recover Backup.

```plaintext
printf("i= %d, x = %f, s = %f, x^i/i! = %f\n", i, x, s, x^i/i!);
s := s + x^i/i!;
end do;
printf("%f compared to expected %f", s, exp(x));
```

```plaintext
x := 1.0;
while abs(x)<2 do
    print("sin(x)", sin(x));
    x := x/2;
end do;
```
15 Conditional Execution with if... end if

15.1 Chapter Overview

We introduce the ability to selectively execute a segment of code through if ... then ... end if. We can select from two alternatives through if...then....else...end if. We can chose code to execute from three or more cases through if...then...elif....elif....else...end if.

Loops and conditional execution are found in most programming languages in a similar form. Learning how to control execution with them is a skill that usually transfers easily, with practice and opportunity, to working with many other computer tools.

15.2 Choosing alternatives: if...then...end if, if... then... else...end if

The if statement allows a programmer to selectively execute code only if a condition is true.

if condition then
    code to executed if the condition is true
end if

Table 15.1: If-then-else-end if: one possible outcome

<table>
<thead>
<tr>
<th>If-then-else-end if: one possible outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade := 3.7: #This value will cause execution of the &quot;then&quot; code</td>
</tr>
<tr>
<td>if (grade &gt;= 3.5)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>printf(&quot;An excellent grade: %.2f\n&quot;, grade);</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>printf(&quot;A Grade: %.2f\n&quot;, grade);</td>
</tr>
<tr>
<td>end if;</td>
</tr>
<tr>
<td>An excellent grade: 3.70</td>
</tr>
</tbody>
</table>

If an "else" is added, then the computer will chose which of two segments of code it will execute, depending on whether the condition is true or false.

if condition then
    code to executed if the condition is true
else
    code to be executed if the condition is false
end if

Table 15.2: If-then-else-end if: another possible outcome

<table>
<thead>
<tr>
<th>If-then-else-end if: another possible outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>grade := 3.2: #This value will cause execution of the &quot;else&quot; code</td>
</tr>
<tr>
<td>if (grade &gt;= 3.5)</td>
</tr>
<tr>
<td>then</td>
</tr>
</tbody>
</table>
printf("An excellent grade: %.2f\n", grade);
else
    printf("A grade: %.2f\n", grade);
end if;

A grade: 3.20

Obviously, if you are planning to do something once, you can probably just look at what's being computed and decide yourself what to do. The value of if is when you plan to use a script many times, or if you need to make many conditional decisions within a loop in a single execution of a script. If is helpful because you won't have to stop to think about what happens each time. You can just set things up ahead of time so that the right actions occur in either case.

Table 15.3: If-then-else-end if in a loop

<table>
<thead>
<tr>
<th>If-then-else-end if in a loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Initialize a list with some numbers, some grade point</td>
</tr>
<tr>
<td># averages</td>
</tr>
<tr>
<td>L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9];</td>
</tr>
<tr>
<td>#Initialize some accumulation variables. We will use them</td>
</tr>
<tr>
<td>#to count how many grades are 3.5 or above, and</td>
</tr>
<tr>
<td>#how many are below.</td>
</tr>
<tr>
<td>countHigh := 0;</td>
</tr>
<tr>
<td>countLow := 0;</td>
</tr>
<tr>
<td>#This goes first, before a loop, as a table header</td>
</tr>
<tr>
<td>printf(&quot;Grades at least 3.5:\n&quot;);</td>
</tr>
<tr>
<td>#i steps through the possible index values for L.</td>
</tr>
<tr>
<td>#Recall that nops(L) is the length of the list.</td>
</tr>
<tr>
<td>for i from 1 to nops(L) do</td>
</tr>
<tr>
<td>if (L[i] &gt;= 3.5)</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>printf(&quot;%.2f\n&quot;, L[i]); #print out the grade with 2 decimals</td>
</tr>
<tr>
<td>countHigh := countHigh+1; #add one to the &quot;high&quot; counter</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>countLow := countLow+1; #add one to the &quot;low&quot; counter</td>
</tr>
<tr>
<td>end if;</td>
</tr>
<tr>
<td>end do:</td>
</tr>
<tr>
<td>printf(&quot;\nThere were %d grades at or above, and %d grades below 3.5.\n&quot;,countHigh, countLow);</td>
</tr>
</tbody>
</table>
Grades at least 3.5:
3.60
3.90
3.90
3.60
3.70

There were 5 grades at or above, and 6 grades below 3.5.

A fairly common occurrence is when the else alternative would specify to do nothing at all. Instead of having an empty else branch, you should write the code without the else as if...then...endif.

Table 15.4: If-then-end-if

<table>
<thead>
<tr>
<th>Example of if-then-end-if</th>
</tr>
</thead>
<tbody>
<tr>
<td>In University of West Dakota at Hoople, the Dean's list is determined by computing the average of all student's grade point averages, and selecting those whose gpa is at least .5 point higher than the average.</td>
</tr>
</tbody>
</table>

```
restart; #unassign any previous values
L := [3.6, 3.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 3.8, 2.0]:
deanCount := 0:
#Compute the average for all students by
#using sum to add up all the grades, and dividing by the
#number of grades
avg := sum(L[i],i=1..nops(L))/nops(L):
#Print out table header.
printf("Grade average of %d students is: %f.\n", nops(L), avg);
printf("Grades at least one point above average (Dean's List):\n");
for i from 1 to nops(L) do
if (L[i] >= (avg+1) )
then
printf("%.2f\n", L[i]);
deanCount := deanCount+1;
end if;
end do:
printf("\nThere were %d students making the Dean's List.",deanCount);
```

Grade average of 12 students is: 2.791667.
Grades at least one point above average (Dean's List):
3.90
3.90
3.80
There were 3 students making the Dean's List.
15.3 Coding situations with three or more cases using if-then-elif-elif...else-end if

The *if* statements we have seen so far can be described as being useful when:

1. The *condition* was used to select between one series, and doing nothing as the alternative. (*if*.. *then*... *endif*).
2. The *condition* was used to select between two alternative series of actions (*if*...*then*..*else*..*endif*).

Another situation that arises in practice is to select between three or more alternatives. This can be thought of as "case by case" situation.

**Table 15.5: if-then-elif-else-endif**

<table>
<thead>
<tr>
<th>if-then-elif-else-endif</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General form</strong></td>
<td></td>
</tr>
<tr>
<td><em>if (condition 1) then</em></td>
<td></td>
</tr>
<tr>
<td>statements separated by ; or :</td>
<td></td>
</tr>
<tr>
<td><em>elif (condition 2) then</em></td>
<td></td>
</tr>
<tr>
<td>statements separated by ; or :</td>
<td></td>
</tr>
<tr>
<td>(more alternatives specified by more elifs)</td>
<td></td>
</tr>
<tr>
<td><em>else</em></td>
<td></td>
</tr>
<tr>
<td>statements separated by ; or :</td>
<td></td>
</tr>
<tr>
<td><em>end if</em></td>
<td></td>
</tr>
</tbody>
</table>

*If condition 1 is true when the statement is executed, then the first series of statements is executed.*

*Otherwise, if condition 1 is not true but condition 2 is true, then the second series as executed.*

*Additional conditions will be checked if none of the previous ones turn out to be true. The sequence of actions corresponding to the first true condition will be executed.*

*If none of the conditions are true, then last series of statements are done.*

The *else* section can be omitted. This is equivalent to doing nothing if none of the other conditions apply.

**Table 15.6: Example of if-then-elif-elif-...-else-endif**

**Example of If-then-elif-elif-...-else-end if**

```
#Initialize L to be a list of grades
L := [3.6, 3.9, 0.9, 2.1, 3.9, 2.1, 0.7, 1.3, 2.8, 3.6, 3.7, 2.9, 1.3];

#Initialize counters for the number of As, Bs, etc. (silently)
countA := 0;
countB := 0;
countC := 0;
countD := 0;
countF := 0;

#Process each element of the list, with the following cases:
#As (3.3 and above)
#Bs (at or above 2.7 and below an A)
#Cs (at or above 1.7 and
#Ds (at or above 1.0 and below a C)
#Fs (below 1.0).

for i from 1 to nops(L) do
  x := L[i]; #Use "x" instead of typing L[i] several times
  if (x >= 3.3) then
    countA := countA + 1;
  elif (x >= 2.7) then
    countB := countB + 1;
  elif (x >= 1.7) then
    countC := countC + 1;
  elif (x >= 1.0) then
    countD := countD + 1;
  else
    countF := countF + 1;
  end if;
end for;
```
countA := countA +1;  
if (x >=2.7)  #this is tried only if first condition false  
then  
  countB := countB +1;  
endif;  
else  #we will be here only if everything else hasn't worked  
  # so we know x<1.0.  
  countF := countF +1;  
end if;  
end do:

printf("There were %d As, %d Bs, %d Cs, %d Ds, and %d Fs.", countA,countB,countC, countD,countF);

There were 5 As, 2 Bs, 2 Cs, 2 Ds, and 2 Fs.

It is possible to include if statements as one of the actions inside another if statement. This is useful where a case may have subcases. We will discussion this situation in more detail in later chapters.

15.4 For the curious: comparing control statements in conventional programming languages

Almost all conventional programming languages (C, Mathematica, C#, Java, Matlab, Maple, C++, NXT, Python, Perl, Visual Basic, etc.) have the "loop" idea and the "selection/case" ideas in them. The main difference in how they specify them through the use of different words or symbols. Java for example, uses { and } rather than "then" or "end if" to delimit the begin and end of segments of actions.

Table 15.7: Portion of a Java program with a conditional

<table>
<thead>
<tr>
<th>Portion of a Java program with a conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>int countThree, countSubThree;</td>
</tr>
<tr>
<td>int [] L= {1,5,9,2,6,7,-2,1,7};</td>
</tr>
<tr>
<td>// Set up counters</td>
</tr>
<tr>
<td>countThree=0;</td>
</tr>
<tr>
<td>countSubThree=0;</td>
</tr>
<tr>
<td>// Loop through L counting how many are above three</td>
</tr>
<tr>
<td>for (int i=0; i&lt;L.length; i++)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if (L[i]&gt;3)</td>
</tr>
<tr>
<td>{ countThree++;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>{countSubThree++;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>System.out.println(&quot;three and above:&quot;+countThree+</td>
</tr>
<tr>
<td>&quot; below three: &quot;+ countSubThree);</td>
</tr>
</tbody>
</table>
While this portion of the Java program is quite intelligible to someone who is more experienced in Maple, a full program to do this would require additional statements that we don't should here. Note that the printing statement \texttt{println} uses \	exttt{"+"} not for numerical addition but to combine the output into a single string that \texttt{println} can handle. Getting used to these language-specific idiosyncrasies are part of the overhead of working with more than one tool on a computer.

The Matlab program follow with a loop and a conditional may look similar to a Maple program, although it uses different functions to extract size ("size" rather than "nops") and there is different syntax (no \texttt{do}, just "end", \	exttt{"="} instead of \texttt{from} and \	exttt{"."} instead of \texttt{to}.

Table 15.8: Portion of a Matlab M-file with a loop and a conditional

<table>
<thead>
<tr>
<th>Portion of a Matlab M-file with a loop and a conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
| %Program to count how many elements of a vector/list are greater than and less than three  
| countThree=0;  
| countSubThree=0;  
| L=[1,5,9,2,6,7,-2,1,7];  
| [r,c]=size(L);  
| %Loop for i from 1 to c, where c is the number of items in L (length of L)  
| for i=1:c  
|     if L(i)>3  
|         countThree=countThree+1;  
|     else  
|         countSubThree=countSubThree+1;  
|     end;  
| end;  
| %Use formatted printing to print out things.  
| sprintf('Three and Above: %d, Below three: %d', [countThree, countSubThree]) |

There may be other ways of doing the same task in a language that may be easier in that particular language. For example, summing together numbers in Java involve writing a loop that gathers the sum of the numbers, whereas in Maple one could use "sum" and get the job done in one line. C, Java and C++ have a "?" operator which one could use to write the \texttt{if} of the above example more succinctly even though it would work the same as the kind we have worked about. No language will ever be "the most convenient" for all tasks, which is one of the reasons why so many languages endure.

Regardless of what you think about the way other languages handle loops and conditional execution, learning how to do it in Maple transfers to most other languages you may choose to work with in the future.

15.5 Troubleshooting if statements

While, \texttt{for}, and \texttt{if} statements require you to have words such as "while" "do", "if", "else", etc. written in exactly the right sequence or else the language processor will reject them. Sixty years of advances in programming language technology haven't made the situation for programmers much easier. It's still necessary to learn how to do it correctly, and how to troubleshoot your way out of problems when you make a mistake.

You can use the fact that the computer will let you experiment as much as you want to help you learn. Now that you have learned how to enter code into worksheets and code edit regions, you can start off by learning new syntax by exactly reproducing an example that contains output as well as input, and seeing whether you can get the same results. Next, you can try modifying the examples to do slightly different things, and seeing whether you are successful at the modifications. Finally, you can try to enter something of your own devising, from scratch.

Getting expert help can be a useful way of quickly seeing how to notice and fix the basics. However, the point from getting such help is not to only to get your mistake fixed, but also to learn from the experience so that you can avoid and/or recognize how to fix similar things in the future on your own.
# Table 15.9: Error messages from if statements

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| x := 3; 
countThree := 0; 
countSubThree := 0; 
if x>3 
then 
  countThree := countThree +1; 
else 
  countSubThree := countSubThree +1; 
end if; 
printf("three and above:%d, below three: %d", 
countThree, countSubThree); | No missing pieces here. |

```
3
0
0
1
three and above:0, below three: 1
```

In this example, we forgot to put in the word "then". After the error, the cursor is positioned in the midst of the line after the "if", which is the first time that Maple’s input processor notices that there's something wrong. To handle this error, one looks backwards from the position of the cursor to see if there's anything missing. Admittedly there's not a lot of information given that the cause is a missing then, but this is fairly typical in working with programming languages. You’ll get an indication that there's something wrong, and the point at which the wrongness was first noticed, but you won’t be able to depend on the error message to tell you exact what was wrong. If computer technology could do that reliably, systems would just fix the errors automatically. But doing that well seems to require artificial intelligence beyond present conventional technology.

```
Error, missing operator or `;`
```

We forgot to end the statement with end if (or fi).
printf("three and above:%d, below three: %d",
    countThree, countSubThree);

3
0
0

Warning, premature end of input, use <Shift> + <Enter> to avoid this message.

x := 3;
countThree := 0;
countSubThree := 0;

if x>3
then
    countThree := countThree +1;
else
    countSubThree := countSubThree +1;
end do;

printf("three and above:%d, below three: %d",
    countThree, countSubThree);

3
0
0

Error, reserved word 'do' unexpected

We said "end do" rather than "end if".

Actually, this is not a mistake. Evidently "end if" and "end" are acceptable here. "fi" is another way of ending an if statement.
x := 3;
countThree := 0;
countSubThree := 0;

if x>3
then
  countThree := countThree +1;
else
  countSubThree := countSubThree +1;
end if

printf("three and above:%d, below three: %d",
  countThree, countSubThree);

3
0
0

Error, missing operator or `;`

x := 3;
countThree := 0;
countSubThree := 0;

IF x>3
THEN
  countThree := countThree +1;
ELSE
  countSubThree := countSubThree +1;
END IF;

printf("three and above:%d, below three: %d",
  countThree, countSubThree);

3
0
0

Error, missing operator or `;`

We get the same error message as in the second item in this example. However, this time the problem is that we left off the semi-colon after the end if. We need it if there's an instruction after the if statement in the script.

Programmers need to consider that there could be more than one kind of mistake that would provoke an error message. This kind of situation can be encountered with almost any sort of programming system (e.g. C, Matlab, Java, etc.) due to the limitations of conventional language-processing technology.

The mistake here is that "IF" does not mean the same thing as "if" to Maple -- case matters. Executing this script gives an error message with the cursor positioned right after the IF. The error message is saying that Maple doesn't know what to do with the IF. It is not particularly informative about the cause. You could discover the cause by comparing what is done with examples that are known to work which all use "if" rather than "IF".
### 15.6 Chapter summary for Chapter 15

<table>
<thead>
<tr>
<th>General form</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
</table>
| **If** \(\text{condition}\) \(\text{then}\) \(\text{action}\); \(\text{... action}\); \(\text{endif}\) | If \(\text{condition}\) is true when the statement is executed, then the actions are performed, otherwise they aren’t. \(\text{fi}\) can be used instead of \(\text{endif}\). They mean the same thing. | \[
\text{if } (x \geq 3.5) \\
\text{then} \\
\text{printf}(\"%.2f\n", x); \\
\text{countHigh} := \text{countHigh}+1; \\
\text{end if;}
\] |
| **If** \(\text{condition}\) \(\text{then}\) \(\text{action}\); \(\text{... action}\); \(\text{else}\) \(\text{other action}\); \(\text{... other action}\); \(\text{endif}\) | If \(\text{condition}\) is true when the statement is executed, then the actions are performed, otherwise the alternative actions are performed. \(\text{fi}\) can be used instead of \(\text{endif}\). They mean the same thing. | \[
\text{if } (x \geq 3.5) \\
\text{then} \\
\text{printf}(\"%.2f\n", x); \\
\text{countHigh} := \text{countHigh}+1; \\
\text{else} \\
\text{countLow} := \text{countLow}+1; \\
\text{end if;}
\] |
| **If** \(\text{condition1}\) \(\text{then}\) \(\text{action}\); \(\text{... action}\); \text{elif} \(\text{condition2}\) \(\text{actions2}\); \(\text{... actions2}\); \text{elif} \(\text{condition3}\) \(\text{actions3}\); \(\text{... actions3}\); \text{endif}\) | If \(\text{condition1}\) is true when the statement is executed, then the actions are performed. If not, then \(\text{condition2}\) is true, then the second group of actions is performed, etc. If none of the conditions are true, then nothing is done. This is like a "case by case" description of what to do, where the various conditions describe the cases. | \# Leaves weather variable unchanged except for two cases where there's change. 
\[
\text{if } \text{barometer}="\text{falling}" \text{ and } \text{weather}="\text{It's sunny}" \\
\text{then} \\
\text{weather} := "\text{It's cloudy}"; \\
\text{elif} \\
\text{barometer}="\text{rising}" \text{ and } \text{weather}="\text{It's cloudy}" \\
\text{then} \\
\text{weather}="\text{It's sunny}"; \\
\text{end if;}
\] |
| **If** \(\text{condition1}\) \(\text{then}\) \(\text{action}\); \(\text{... action}\); \text{elif} \(\text{condition2}\) \(\text{actions2}\); \(\text{... actions2}\); \text{elif} \(\text{condition3}\) \(\text{actions3}\); \(\text{... actions3}\); \text{else}\) \(\text{actions otherwise}; \(\text{... actions otherwise}\). | Like the previous situation, only there is an additional set of actions that are done if none of the conditions are true. This is like a "case by case" description of what to do, where the various conditions describe the cases, and the last series of actions describe what happens otherwise if none of the other cases hold. | \#Determine the location of an \((x,y)\) point. 
\# We assume that \(x\) and \(y\) already have values from \# assignments made in the previous point in execution. 
\[
\text{if } x\geq0 \text{ and } y\geq0 \\
\text{then} \\
\text{location} := "\text{quadrant 1}"; \\
\text{elif} \\
x\geq0 \text{ and } y<0 \\
\text{then} \\
\text{location} := "\text{quadrant 4}"; \\
\text{elif } x<0 \text{ and } y\geq0 \\
\text{then} \\
\text{location} := "\text{quadrant 2}"; \\
\text{elif } x<0 \text{ and } y<0 \\
\text{then} \\
\text{location} := "\text{quadrant 3}";
\] |
| end if | else     |
|        | location := "unknown"; |
|        | end if;         |
16 Thinking about iteration

16.1 Chapter Overview

Looping taps into the power of the computer to do many calculations rapidly. It is not easy to learn but must be mastered in order to be able to handle situations creatively. Knowing standard patterns and roles helps.

Many languages also have shortcuts in common situations that avoid the need to write loops. Using the shortcuts available in Maple (seq, sum, and map, for example) can save programmer time and increase reliability. Other programming languages such as Matlab or Mathematica will have similar shortcuts available.

16.2 Constructing iterations and the roles of variables

Setting up iterations is one of the most prized skills for a programmer, and one of the most difficult for beginners to learn. To construct an iteration from scratch, we recommend the following process:

Without trying to create an iteration, just write down the series of directions for the computational process you want to program. If you keep on writing down steps, eventually you will notice similarities between actions in the series. Often times the similarity allows you to discover that rather than giving a new variable name to every result you compute, you can reuse certain variables because you no longer need them beyond a certain point in the series.

Finding the pattern is key. Once you have it, then the repeating pattern becomes the loop body. Steps before the repetition kicks in become initialization before the loop. Usually there are summary actions (e.g. making a plot or animation, of all the results computed by the loop) which become part of the post-loop actions.

Only through experience do you get to the point where you can construct loops without explicitly finding the repetition pattern is. Don't expect it to come without practice and thoughtful reflection on experience.

In trying to construct a code for a successful loop, it can be good practice to decide upon the role of each variable you are using. If you are unclear about a variable's role or purpose, it may mean that the variable is superfluous. It could also mean that you have skipped over a key computational action which you need to develop code for. Here are typical roles variables can play in a loop:

<table>
<thead>
<tr>
<th>Table 16.1: Roles of variables in a loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables that are fixed values. They are assigned during the initialization but don't change afterwards.</td>
</tr>
<tr>
<td>Variables that are steppers. Stepper variables change value methodically with each iteration of the loop.</td>
</tr>
<tr>
<td>Variables that gather the final result. Each repetition of the loop leaves the accumulation variable one step closer to its final objective (e.g. adding together all the terms of a sum).</td>
</tr>
<tr>
<td>Variables that hold the most recent value going through a succession. Usually (but not always) the most recent value is computed from previous values.</td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Consider for example this starter code from Lab 2, that computed chemical concentrations

<table>
<thead>
<tr>
<th>Table 16.2: Lab 2 CS 122 Problem 2.1 starter code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 2 CS 122 Chemical problem starter code</td>
</tr>
<tr>
<td>#Lab 2 Problem 2.1</td>
</tr>
<tr>
<td>restart; #just in case there are unwanted assignments hanging around from #previous runs.</td>
</tr>
<tr>
<td># Chemical reaction modelled by Lotka-Volterra Model #Discrete approximation</td>
</tr>
</tbody>
</table>
numTimeSteps := 50;

#Initial concentrations
A0 := 1000;
B0 := 0;
X0 := 10;
Y0 := 50;
#Chemical constants
k1 := .001;
k2 := .001;
k3 := .1;
A := A0; B := B0; X := X0; Y := Y0;

for i from 1 to numTimeSteps do
    newA := A - k1*A*X;
    # newX :=
    # newY :=
    # newB :=
    # Update all four chemicals
    A := newA; #B, X, Y updated similarly.
    indexTab[i] := i;
    Atab[i] := A; #Each entry is a (time, concentration)
end do:

with(plots): with(plottools):

#Given a table of x coordinates and a table of y coordinates,
#Create a point plot.
Fplot := (itab,vtab,COLOR) -> plot(convert(itab,list),convert(vtab,list), color=COLOR, symbol=diamond, style=point);

#Display this point plot, along with a title and labels.
display([Fplot(indexTab,Atab,"Green")], title="graph of A", labels=["time", "concentration"]);

The analysis of the roles of variables in this code is as follows:

| Variables that are fixed values. They are assigned during the initialization but don't change afterwards. | k1, k2, k3 are fixed values. So are A0, B0, X0, and Y0. So is num-TimeSteps. |
| Variables that are stepper. Stepper variables change value methodically with each iteration of the loop. | i is a stepper variable. It begins at 1. One is added onto its value each trip through the loop, so it steps through the values 1, 2, 3, .... |
| Variables that gather the final result. | While there are no variables that sum or count in this example, indexTab and Atab are tables that store one value per iteration. The tables gather the whole collection of values generated during the simulation, which are needed to produce the plot which is the final result. |
| Variables that hold the most recent value going through a succession. Usually (but not always) the most recent value is computed from previous values. | newA is holds the most recent value of the concentration of chemical A. newB, newX, and newY also take on this kind of role. |
| 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 | At the beginning of the loop, A (and as the code was developed, B, X, and Y) took on the role of the previous value. It was used to compute... |
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.

Here's a classic example of a task accomplished through a loop: finding the smallest value of a list of numbers

**Table 16.4: An example using a variable that holds the previous value**

```markdown
An example using a variable that holds the previous value

```

```markdown
L := [3, 6, -4, 2, 1, 9];

minSoFar := infinity;

for i from 1 to nops(L) do
  printf("Comparing %a to %d\n", minSoFar, L[i]);
  if minSoFar > L[i] then
    minSoFar := L[i];
  end if;
end do;

printf("The minimum is: %a.\n", minSoFar);
```

```
[3, 6, -4, 2, 1, 9]

∞

Comparing infinity to 3
Comparing 3 to 6
Comparing 3 to -4
Comparing -4 to 2
Comparing -4 to 1
Comparing -4 to 9
The minimum is: -4.
```

The analysis of the roles of variables in this code is as follows:

**Table 16.5: Roles of variables in a loop**

| Variables that are fixed values. They are assigned during the initialization but don't change afterwards. | L is the only fixed variable. |
| Variables that are steppers. Stepper variables change value methodically with each iteration of the loop. | i is a stepper variable. It begins at 1. One is added onto its value each trip through the loop, so it steps through the values 1, 2, 3, ... |
| Variables that gather the final result. Each repetition of the loop leaves the accumulation variable one step closer to its final objective (e.g. adding together all the terms of a sum). | There aren't any "gatherers" in this short bit of code. |
**Variables that hold the most recent value** going through a succession. Usually (but not always) the most recent value is computed from previous values.

**minSoFar** hold the most recent value of the minimum value seen in L so far. Before the loop starts, it is initialized at infinity, which guarantees that after the first element of L is inspected it will become the value of **minSoFar**. Note that infinity will be printed as the minimum at the end of the code if L := [ ] (the list with nothing in it).

**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.

**minSoFar** also plays this role too. As the i-th execution of the loop body begins, the value of **minSoFar** is the minimum of L[1] through L[i-1] -- the minimum computed from all the previous repetitions of the loop. As the i-th execution of the loop body ends, **minSoFar** has changed to be the minimum of L[1] through L[i]. The code works because if you know the minimum of a collection of numbers, and add a new number to the collection, the overall minimum is the smaller of the new number and the minimum of all the other numbers.

### 16.3 Reminder: you can skip writing some looping code through use of built-in operations

As we mentioned in chapter 14 (page 209), Maple has built-in operations such as *sum* or *seq* that can help users avoid the work of writing looping code because they have looping built into them. For example, `min([3, 6, -4, 2, 1, 9]);` will compute the minimum without having to write any loops at all. Thus, Maple users would probably not find minimums using the "classic example" of the previous section. Using `min` would be faster.

Similarly Maple users would use `sum(L[i], i=1..nops(L))` to add together the elements of L without a loop.

**Table 16.6: Generating a multi-plot with seq**

<table>
<thead>
<tr>
<th>Plot</th>
<th>( \sin\left(\frac{i \cdot x}{x}\right) ) for ( i = 1..4 ) altogether on the same plot.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i := 'i' )</td>
<td>( i := i ) (16.2)</td>
</tr>
<tr>
<td>( Lcolors := [&quot;Red&quot;, &quot;DarkGreen&quot;, &quot;Blue&quot;, &quot;Magenta&quot;] )</td>
<td>We unassign i so that it will work with the <em>seq</em> that we're about to do. Alternatively, we could use the instruction <em>restart</em> if we didn't mind losing all the assignments we've already made in the Maple session.</td>
</tr>
<tr>
<td>( Lcolors := [&quot;Red&quot;, &quot;DarkGreen&quot;, &quot;Blue&quot;, &quot;Magenta&quot;] ) (16.3)</td>
<td>We create a list of colors taken from the official list of Maple colors (see on-line help for <em>colornames</em>).</td>
</tr>
<tr>
<td>( Lplots := {seq\left( \text{plot}\left( \frac{1}{i} \cdot \sin(i \cdot x), x = 0..10, \text{color} = Lcolors(i) \right), i = 1..4 \right) } )</td>
<td></td>
</tr>
<tr>
<td>( Lplots := {PLOT(...), PLOT(...), PLOT(...), PLOT[...]) ) (16.4)</td>
<td></td>
</tr>
</tbody>
</table>
Here is another programming feature which you can use to cut down on the amount of work that you would normally do with a loop. A common situation in programming is when you want to perform the same operation on each component of a structure. We could do this with a loop or with seq, but there's an even shorter way to do it in Maple, map.

\[
\text{map}(\text{function or function name}, \text{structure})
\]

will apply the function to each element of the structure, producing a new structure where the i-th element of the new structure is the result of applying the function to the i-th element of the old structure.

The advantage of \text{map} over loops or seq is that the programmer does not need to set up any stepper variables, nor need to figure out how to construct the same kind of structure that the input has. The programmer does not need to use \text{nops} or other techniques to figure out the length of the structure, as \text{map} will figure out the length automatically.

The typical use of \text{map} is to a list or set. \text{map} does not work with sequences because the use of a sequence does not present map with a single structure to map onto.

\text{map} does not work on tables. It will however, work, if you convert a table into a list and then apply map to the list.

**Table 16.7: Generating a multi-plot with seq**

\[
L := \begin{bmatrix} -1.2, 3, \sin\left(\frac{\pi}{3}\right) \\ -1.2, 3, \frac{1}{2} \sqrt{3} \end{bmatrix}
\]

We find the absolute value of every number in L by having map apply abs to every element of L. We create a new list of the absolute values and assign it to L2.
We find the natural log of every number in set1. Since they are all floating point numbers, applying \( \ln \) to them results in approximations to the logarithms.

\[
L2 := map(abs, L) \\
\begin{bmatrix} 1.2, 3, \frac{1}{2} \sqrt{3} \end{bmatrix}
\]  
(16.6)

\[
set1 := \{1,0,2,2,9,5\} \\
\begin{bmatrix} 1.0, 2.2, 9.5 \end{bmatrix}
\]  
(16.7)

We get a list of left hand sides of the equations in list3.

\[
list3 := \left[x^2 - 1 = 0, x^3 - 1 = 0\right]
\]  
(16.9)

Mapping plot onto these expressions produces a list of plot structures.

\[
list4 := \left[x^2 - 1, x^3 - 1\right]
\]  
(16.10)

We can plot them together using display.

\[
list5 := \left[PLOT(...), PLOT(...)\right]
\]  
(16.11)

If we're feeling confident, we can daisy-chain (compose) the functions together into a one line of coding that takes list6 and produces a multi-plot from it. You'd probably develop the code for this working inside out. First you'd convince yourself that the \( lhs \) map worked. Then you'd add on the plot. Finally, you'd add on the display and the parameters that control the range of horizontal and vertical views. It would take multiple edits, but you would only have to execute a single line of code repeatedly, rather than executing an entire worksheet or selecting a segment to re-execute.

\[
list6 := \left[x^2 - 1 = 0, x^3 - 1 = 0, x^4 - 1 = 0, x^5 - 1 = 0\right]
\]  
(16.12)

\[
with(plots):
\]  

\[
\begin{align*}
\text{display(list5)}
\end{align*}
\]
map's terse way of performing a lot of computation can take some getting used to. But experienced programmers often find it to be less error-prone to use map to get a result than a loop since less coding (albeit more thinking) is needed. In using map, the programmer turns more of the job setting up loops over to the supporting internal programming, in the same way that using plot or solve saves the programmer from having to write graphics code or mathematical equation solvers.

A variety of programming languages (e.g. Lisp, Matlab, Python, Mathematica) have map or a similar technique of computation. For example, Matlab has ARRAYFUN(f, A) that applies a function f to every element of a Matlab array (a kind of data container) A. map is a key component of Google's "easy supercomputing" package (see http://code.google.com/edu/parallel/mapreduce-tutorial.html).

16.5 The advantages and limitations of shortcuts

Shortcuts are good because:

1. They save programmer time, once the programmer masters the shortcut.

2. In Maple they typically are more efficient because more of the computation is handled by internal programming which typically has been optimized for speed.

3. One theory of software engineering is that long programs can be harder to debug, and have more bugs in them. To the extent that shortcuts reduce the length of programs, they can save programmer time because there's less time spent debugging.

Shortcuts such as seq or map have limitations, however. Not all computational tasks are shortened by using such techniques, particularly when you have many variables with "previous value" roles and don't have enough memory or time to construct lists or tables that hold all the previously computed results. There will still be many situations where a for/while loop is the best way of doing the computation.
16.6 Summary of chapter

<table>
<thead>
<tr>
<th>Variable types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed value</td>
<td>These variables are set before the loop begins, and do not change while the loop is being executed.</td>
</tr>
<tr>
<td>Steppers</td>
<td>These variables change some fixed amount every time a loop iterates. They hold the current step a loop is on. For example, the index variable in a for loop is a stepper variable.</td>
</tr>
<tr>
<td>Variables that gather the final result</td>
<td>These variables accumulate a result that is returned at the end of a loop.</td>
</tr>
<tr>
<td>Variables that hold the most recent value in a loop</td>
<td>These variables hold some value that was computed during the current iteration of a loop. They are often compared to a previous value to compute a change.</td>
</tr>
<tr>
<td>Variables that hold a previous value in a loop</td>
<td>These variables hold some value computed in the previous iteration of a loop.</td>
</tr>
</tbody>
</table>

Looping without using built-in operations

Plot \( \frac{\sin(i \cdot x)}{i} \) for \( i = 1..4 \) altogether on the same plot.

```maple
i := 'i'

Lcolors := ["Red", "Blue", "Pink", "Black"]

Lplots := map(plot(\( \frac{1}{i} \cdot \sin(i \cdot x), x = 0..10, color = Lcolors[i], i = 1..4 \))

display(Lplots)
```

Using the map function
Performing an operation on every item in a set.

\[ set1 := \{1,0,2,2,9.5\} \]

\[
\begin{align*}
&\{1.0, 2.2, 9.5\} \\
&\text{map(ln, set1)} \\
&\{0.0, 0.7884573604, 2.251291799\}
\end{align*}
\] (16.16) (16.17)

Solving a list of equations.

\[ \text{list3 := \left[ x^2 - 1 = 0, x^3 - 1 = 0 \right]} \]

\[
\begin{align*}
&[x^2 - 1 = 0, x^3 - 1 = 0] \\
&\text{map(solve, list3)} \\
&\left[1, -1, \frac{-1}{2} + \frac{1}{2} \sqrt[3]{3}, -\frac{1}{2} \sqrt[3]{3}\right]
\end{align*}
\] (16.18) (16.19)

Solving a set of equations. Note that duplicates are removed.

\[ \text{set3 := convert(list3, set)} \]

\[
\begin{align*}
&\{x^2 - 1 = 0, x^3 - 1 = 0\} \\
&\text{map(solve, set3)} \\
&\left[1, -1, -\frac{1}{2} - \frac{1}{2} \sqrt[3]{3}, \frac{1}{2} + \frac{1}{2} \sqrt[3]{3}\right]
\end{align*}
\] (16.20) (16.21)

Using map to display a set of plots.

\[ \text{list4 := map(lhs, list3)} \]

\[
\begin{align*}
&[x^2 - 1, x^3 - 1] \\
&\text{list5 := map(plot, list4)} \\
&\text{with(plots):}
\end{align*}
\] (16.22) (16.23)
display(list5)
Index

Symbols
! (exclamation point, factorial), 13
! icon (execute group), 161
# (program comment), 162
%d, 156, 167
%f, 156, 167
(, 12
), 12
- (minus or dash), 12
: (colon), 161
:=, 40
; (semi-colon), 161
< (less than), 215
<= (less than or equals), 215
<default>, 190, 215, 221
= (equals), 215
= (equation), 21
> (greater than), 215
>= (greater than or equals), 216
\317\200 (symbolic constant), 28, 29
\317\200 (symbolic constant), 25
^ (caret), 12
_ (underscore), 39

A
abs (absolute value), 90
accumulation, 218, 222
actual parameter, 65
acute accent marks, 71
adding lists, 211
addition (+), 12
Algol 68 (programming language), 190
altitude, 143
animate, 149
animation controls, 130
animation speed, 130
animations
   exporting to GIF format, 151
apostrophe, 112
apostrophes, 71
arc cosine, 25
arc sine, 25
arc tangent, 25
arccos (inverse of sine), 90
argument, 65
arguments
   of function, 89
arithmetic, 11
   approximate, 26
   exact, 14
   operations, 11
arrow, 95, 99
assignment, 22
   mental model of, 41
   of a plot structure, 121
assistant
   Unit Converter, 115
asterisk, 12
autosaving, 20
average
   numbers in a list, 79
Axiom, 5

B
backquotes, 71
backspace key, 16
backup, 180, 221
backups
   retrieving, 20
ball, 143
base of natural logarithm (e), 28
Blackberry, 78
blue, 124
brackets, 52
by, 192

C
C, 233
C (programming language), 46
C (the programming language), 5, 7, 159, 233
C# (the programming language), 233
C++ (the programming language), 5, 233
Canadian, 78
Canadian roots, 125
Canvas (Maple word processor feature), 36
caret (power), 12
chemical concentration, 242
circle, 125, 128, 145
   function, 125
circle-X, 9
clickable interface, 68
code edit region, 155
coding, 165
colon, 161
Colorado, 53
colornames (plot), 138
colors
   plotting, 77
comma, 52
command-=, 35
command-C (copy operation), 16, 55
cmd-ALT (copy operation), 16
command-e (execute region), 161
command-R, 34, 35
command-T, 34
command-V (paste operation), 16, 55
command-X (cut operation), 16
comment
   program (#), 162
comments
   header, 162
   program, 171
completion
   command, 92
composition
   of functions, 108, 109
computing
   algebraic, 1
   graphical, 1
   high performance, 3
   logical, 1
   numerical, 1
constant
   symbolic, 48
containers
   data, 183
control-=, 35
control-C (copy operation), 16, 55
control-click, 21
control-e (execute region), 161
control-R, 34, 35
control-T, 34, 35
control-V (paste operation), 16, 55
control-X (cut operation), 16
convert, 102, 113, 144, 185
   sequence into list, 210
convert a list into a sequence, 79, 86
convert a list into a string, 79, 86
copy (editing operation), 16
copy and paste, 55
crash
   Maple, 20
curly braces, 52
currency, 111
cut (editing operation), 16
digits, 26
disk (plottools function), 138
display, 123, 148
   function, 122, 125
divide and conquer, 57
division (/), 12
do, 189
documentation
   need for, 1
dot
   centered, 13
double-quotes, 71
down the road
   just, 78
E
e, 36
   from keyboard not a symbolic constant, 28
   symbolic constant, 28, 29
e (natural logarithm), 25
Edit->Execute->Selection, 55
Edit->Execute>Worksheet., 55
e1f, 232
e11lipse, 128
e11se, 229, 232
e11nd do, 189
e11nd if, 229, 232
ENGR101, 57
enter, 55
entries, 185, 187
equals, 215
equation, 21, 51, 183
   heating, 172
error
   invalid fraction, 16
   invalid left hand side in assignment, 186
   invalid matrix/vector product, 15
   invalid operator parameter name, 98
   invalid power, 66
   invalid sequence, 16
   invalid sum/difference, 14
   logic, 18
   missing operation, 15
   missing operator or `;`, 235
   numeric exception: division by zero, 15
   unable to match delimiters, 15, 66
   unexpected option, 71
   vocabulary, 18
Error
   attempting to assign to `solve` which is protected, 48
   controlling variable of for loop must be a name, 48
   error message, 14
   escape key, 36
escape key (ESC), 92
Euro, 108
eval, 101
evalf, 102
evaluate
   at a point, 55
evaluating
   preventing, 112
state, 20
evaluating a formula, 127
Evaluating..., 221
evaluation, 40
   of expressions, 11
execute region, 161
execution trace, 197
exp (exponential function), 91
exponentiation, 17
Exporting animations, 151
expression, 183
expressions
   algebraic, 21
   from palette, 24
F
factorial (!), 13
fence post error, 28
final result variable, 241, 242, 243
final value, 190
find something similar, 57
fixed value variable, 241, 242, 243
for, 190, 219
Format -> Create Document Block, 37
formatted output (%a %f %d etc.), 157
Formatting
   output
      as currency, 111
Fortran, 5
freeze frame
   animation, 130
from, 192
fruit flies, 94
csolve, 99, 100, 183
ft, 111
function
   idea of, 89
function definition, 98
function definition alternative, 99
functions
   define, 94
      designing from word problems, 107
      thousands of built-in, 92
fuzzy notions, 57
G
gallon, 112
gallons, 102
   Imperial, 78
gathering variable, 241, 242, 243
gigaflop, 3
Google, 247
graphical user interface (GUI), 5
Graphics Exchange Format (GIF), 151
greater than, 215
greater than or equals, 216
green, 124
H
header comments, 162
help
   on-line, 74, 85, 93
   quick, 9
   using on-line, 94
Help (Maple menu), 36
horsepower, 112
hour, 102, 112
hypotenuse, 117
I
I (square root of -1), 48
Ideal Gas Law, 105
if, 231, 232
Image (Maple word processor feature), 36
immutable, 183
inch, 111
inches, 102
index, 51, 125, 184, 214
index variable, 190
indices, 184, 187
inequalities, 51
infin (shortcut for infinity in math mode), 36
infinite loop, 205
infinity, 243
initial value, 190
Insert (Maple menu item), 36
Insert->2-D Math, 34
Insert->Text, 34
interactive, 5
interrupting Maple, 221
inverse trig functions, 25
J
Java, 5, 46
Java (the programming language), 233
justification of formatted output, 156
K
Kessel Run, 112
key, 184
kilometer, 112
kilopascals, 106
kilowatt, 112
km, 111
L
label, 11
labels for axes
plotting, 77
left arrow key, 16
left hand side, 51
left hand side (right-click menu), 27, 30
less than, 215
less than or equals, 215
lhs, 101, 183
line
function, 124
linear algebra, 15, 212
Lisp, 247
list, 52, 68, 183
add together items from, 79
count number of items in, 79
empty, 183
one item from, 79
operations on, 78
part of, 78
liters, 102
log10 (base 10 logarithm), 90
logarithm
base e, 25
logic error, 18, 28
loop, 189
loop body, 189
looping, 189
Los Alamos National Laboratory, 3
M
Macsyma, 5
map, 245
Maple, 5, 233
as a word processor, 33
document mode, 9
mapreduce, 247
math entry mode, 33
Mathematica, 5, 6, 233, 247
mathematical model, 1
mathematical models, 126
Matlab, 5, 46, 159, 233, 247
Symbolic Toolkit, 6
matrices, 212
max (maximum of arguments), 90, 100
maxdigits, 18
mental model, 41
meter, 111
meters, 102
micrometer, 112
mile, 111
millimeter, 111
min, 244
min (minimum of arguments), 90, 100
minimum, 243, 244
minus, 12
minute, 112
mistake
typographical
how to fix, 16
most recent value variable, 241, 242, 244
multiple systems, 7
multiplication (*), 12
N
name
assignment to, 39
National Forest Service, 55
negation (-), 12
nops, 183
not equals, 215
NULL, 155, 156
NXT (the programming language), 233
O
Oak Ridge National Laboratory, 3
Octave, 5
omega (Greek letter), 36
Omega (upper case Greek letter), 36
on-line documentation, 77
op, 183
orange, 124
oscilloscope, 57
P
palette, 24
Expression, 67, 89, 94, 96, 100, 212
square root, 25
palettes, 36
parameter, 56
parameterized plots, 127
parentheses, 12, 65
parsec, 112
paste (editing operation), 16
patterns of looping, 218
Perl (the programming language), 233
petaflop, 3
Pi, 48
Pi (symbolic constant), 48
pick names, 56
pitfalls
  common
    in Maple, 89
playing
  animation, 130
plot, 101, 183
  builder, 74
  colors, 71
  labels, 71
  options, 77
  structure, 121
PlotBuilder, 23
plots
  package, 123
plotting, 22
plotting a function, 127
plottools, 124
plus, 12
pop-up, 99
power (^), 12
pressure, 116
preventing
  evaluation, 112
previous value variable, 241, 242, 244
print, 155
printf, 156
program code, 165
Python, 5, 6, 233, 247

Q
Quick Reference (help menu item), 36
quotation, 112

R
radians, 102
range, 183
ready
  Maple, 20
Ready, 221
rectangle
  dashed (entry), 10
red, 124
red stop hand, 221
refactor
  code, 110
relational operators, 215
repeat n times, 218
replay
  animation, 130
restart, 47, 57
retrieving
  a Maple session, 19
reuse
  of software, 1
rhs, 101, 183
right hand side, 51
right hand side (of equation), 26
right hand side (right-click menu), 27, 30
right-click, 21
roles of variables in a loop, 241

S
SAGE, 5
Saving
  a Maple document, 19
scaling=constrained, 128
scaling=constrained (option to plot), 128
scripting, 55
search field
  for help, 74
select
  entry, 51
  item from a set, 27
select entry (right-click menu), 27, 30
semi-colon, 161
seq, 210, 244
sequence, 16, 52, 117, 183
  empty, 183
set, 52, 183
  empty, 63, 183
sheep, 53
shortcut, 247
shortcuts
  keyboard, 36
Show Al Palettes, 36
side effect, 155
simplify - assuming real (clickable operation), 91
simulation, 1
single-quotes, 71
slash, 12
small steps, 57
Solo
  Han, 112
solve, 51, 66, 85, 100, 183
  a system of equations, 79
  by right-clicking or control-clicking, 21
numerical, 26
solving an equation, 127
sprintf, 157
sqrt (square root), 90
square root
  from palette, 25
standard deviation, 214
T

table, 185
Table (Maple word processor feature), 36
temperature, 116
template
textual, 92
teraflop, 3
terminating
Maple computation, 221
test, 57
text entry mode, 34
text mode to math mode
can't switch back, 37
textual entry, 65
then, 229, 232
to, 189
tol, 214
Tools->Spellcheck, 37
trace
execution, 174
triangle, 117
troubleshooting, 57
assignment, 47
function definitions, 98
if statements, 237
loops, 205
tables, 187
textual input, 85
unit conversion, 112
word processing, 37

U
unassign, 41, 42, 47
underscore, 35
underscore (_), 39
Unit Converter assistant, 115
units, 144
Units Calculator, 78
Use of printf, 156, 167

V

value
of an unassigned name, 40
variable
programming, 39
vectors, 212
version
number, 180
vertical range, 67
View (Maple menu), 36
Visual Basic (the programming language), 233
visualization, 1

W

Warning
premature end of input use <Shift> + <Enter> to avoid this message, 190
what you see is what you get, 33
whattype, 158
while, 217, 219
WYSIWYG, 33

Y

yard, 112

Z

zeroes
leading or trailing, 156