Scientific Programming

Scientific Programming:

Numeric, Symbolic, and Graphical Computing with Maxima

By
Jorge Alberto Calvo

Cambridge
Scholars
Publishing

Scientific Programming:
Numeric, Symbolic, and Graphical Computing with Maxima

By Jorge Alberto Calvo

This book first published 2018

Cambridge Scholars Publishing

Lady Stephenson Library, Newcastle upon Tyne, NE6 2PA, UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN (10): 1-5275-1117-0
ISBN (13): 978-1-5275-1117-0

Cover Photograph: *Antarctic Ice* by Christine A. Wilkins (2011).

The beautifully intricate structures formed by ice crystals off the Antarctic coastline serve as a reminder both of the order required when designing computer programs and of the infinite possibilities that result when these are organized by a well-disciplined method.

All figures were rendered by the author using *Maxima*'s `draw` package and the `tikz` package for the LaTeX typesetting language. The following photographs are included with the copyright owner's permission:

- *Lofting Ducks* (page 361) by Mathew Emerick (2012).
- *Oratory at Sunrise* (page 455) by Tyler Neil Photography (2015).

To my father

Table of Contents

Foreword ..... **ix**

Chapter 1. An Introduction To Programming **1**

1.1 Getting Started with *Maxima* 3
1.2 Symbolic Computation 21
1.3 User-Defined Functions 41
1.4 Repetition and Iteration 61
1.5 Leibniz's Series 73
1.6 Archimedes' Sequence 87
1.7 Computational Error 101

Chapter 2. Computation in Mathematics **117**

2.1 Fibonacci's Rabbits 119
2.2 The Babylonian Algorithm 133
2.3 Detecting Prime Numbers 147
2.4 Lists and Other Data Structures 161
2.5 Gaussian Elimination 177
2.6 Partial Pivoting 191
2.7 Binary Codes 207
2.8 Floating-Point Numbers 225

Chapter 3. Graphics and Visualization **241**

3.1 Celestial Mechanics 243
3.2 An Epitaph for Archimedes 265
3.3 Rabbits Revisited 283
3.4 Projectiles in Motion 303
3.5 Fourier Likes It Hot 317
3.6 Plotting Curves by Sampling 329
3.7 Plotting Lines with Pixels 347

Chapter 4. Interpolation and Approximation **359**

4.1 Lagrange's Formula 361
4.2 Piecewise Linear Interpolation 375
4.3 Interpolation using Splines 385
4.4 Smooth Splines 401

4.5 Approximation using Splines 413
4.6 Interpolating Curves 427
4.7 Interpolating Surfaces 445

Chapter 5. Numerical Integration 457

5.1 Riemann Sums 459
5.2 Trapezoids and Parabolas 475
5.3 Smooth Splines Revisited 491
5.4 Gaussian Quadrature 505
5.5 Monte Carlo Simulation 519

Epilogue 533

References 537

Index 539

Foreword

> "If one approaches a problem with order and method, there should be no difficulty in solving it; none whatever."
>
> — Hercule Poirot
>
> as quoted in *Death in the Clouds*
> by Agatha Christie (1935)

This book was developed for an undergraduate course in scientific programming, offering an introduction to computer programming, numerical analysis, and other mathematical ideas that extend the basic topics learned in calculus. It is designed for students who have finished their calculus sequence but have not yet taken a proofs course. The primary goal is to teach students how to write computer programs, and covers both the general building blocks of programming languages (such as conditional statements, recursion, and iteration) as well as a description of how these concepts are put together to allow computers to produce the results they do. In particular, careful attention is given to binary arithmetic, the IEEE standard for floating-point numbers, and algorithms for rendering graphics. The content builds on the topics covered in an introductory calculus course, including the numerical solution of both ordinary and partial differential equations, the smooth interpolation of discrete data, and the numerical approximation of non-elementary integrals.

More importantly, however, this book will teach students about "order" and "method," particularly when it comes to organizing data and solving problems. The guiding principle throughout is the belief that, for a novice mathematician, writing computer programs can be a useful exercise in developing the intuition for abstract concepts necessary to make the transition towards writing proofs. If nothing else, when you write an incorrect program, you get immediate feedback that something went wrong, even if the error message itself might

seem cryptic at first. This gives you a chance to find mistakes in a way that writing an incorrect proof never can.

All of the programming in this book is done in an open-source Computer Algebra System (CAS) called *Maxima*. There are several reasons for this choice of software. First of all, instead of tackling a high level language like *C*, *Java*, or *Python*, we prefer a CAS that provides students a friendly, unified front-end which allows them to perform more familiar mathematical tasks, such as graphing functions or solving equations, as well as write new programs. Secondly, in contrast to other well-known proprietary systems, *Maxima* is freely available for Macs, Windows, and Unix machines. Not only is *Maxima* supported by a large network of volunteer developers and used by researchers throughout the world, but it provides an excellent environment in which students can learn the basic structures of programming before moving on to other popular platforms. Unlike mastering a foreign language, which sometimes involves learning new and alien grammatical concepts like noun declensions (in Latin) or adjectival nouns (in Japanese), making the transition from one programming language to another often involves only small adjustments in syntax. The epilogue at the end of this book provides some simple examples of how this works in practice.

To install *Maxima* on your computer, visit the Maxima Project's website at `maxima.sourceforge.net`, download the installation files for your operating system, and follow the documentation included with the installer. For Unix and Windows machines, this involves running a single script. For Macs, more care is required as new fonts must be loaded and three different applications must be installed and configured to work together; for more detailed instructions, consult the excellent article at:

`themaximalist.org/about/my-mac-os-installation`.

Any additional questions can be directed to the Maxima discussion email list, which can be accessed from:

`maxima.sourceforge.net/maximalist.html`.

I am indebted to Michael Marsalli, who entrusted me with the development of this course, and to Patrick Kelly, Ricardo Rodriguez, and my other colleagues at Ave Maria University for their continual advice. My gratitude also goes to all of my Math 270 students who were subjected to one iteration of course notes after another; their

feedback, both implicit and explicit, has made this a better book than it otherwise would have been.

My beautiful wife and children deserve a special word of mention since they have tolerated many a late night, especially as I put the manuscript in its final form.

Finally, I wish to dedicate this book to my father, to whom I owe more than I could ever repay. I have many fond memories of going as a child to see his enormous Burroughs B1700 mainframe computer at work. The world has changed a lot since the time of punch cards and tape reels. His countless sacrifices throughout these years have made me into the man I am today.

Framingham, Massachusetts
March 2018

CHAPTER 1

An Introduction to Programming

"Method, you comprehend! Method! Arrange your facts. Arrange your ideas. And if some little fact will not fit in—do not reject it but consider it closely. Though its significance escapes you, be sure that it *is* significant."

— Hercule Poirot

as quoted in *The Murder on the Links*
by Agatha Christie (1923)

1.1. Getting Started with *Maxima*

In pure mathematics, we often study numbers, formulas, and geometrical shapes for their own sake, with no intention of ever finding an application for our newfound knowledge. For instance, in a previous calculus course, you may have learned that Archimedes of Syracuse (c. 287 – 212 BC) showed that the infinite sequence

$$2\sqrt{2},\ \ 4\sqrt{2-\sqrt{2}},\ \ 8\sqrt{2-\sqrt{2+\sqrt{2}}},\ \ 16\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{2}}}},\ \ldots$$

converges to π. You may have also learned that Gottfried Wilhelm Leibniz (1646 – 1716) proved that the infinite series

$$4-\frac{4}{3}+\frac{4}{5}-\frac{4}{7}+\frac{4}{9}-\frac{4}{11}+\frac{4}{13}-\frac{4}{15}+\cdots$$

is also equal to π. From a purely theoretical vantage point, these two statements stand side by side as equally valid facts.

From a computational point of view, however, we may consider how each of these infinite limits may be used to provide a suitable approximation for the number π. For example, to determine a partial sum in Leibniz's series, we simply add finitely many fractions together; if one finite sum is deemed inadequate, we can improve it by adding more fractions. Unfortunately, as we shall soon find out, Leibniz's series converges painfully slowly, so it will take literally hundreds of additions before we arrive at a reasonable approximation for π. On the other hand, computing a sufficiently complicated term in Archimedes' sequence involves working out one square root after another, starting from the innermost part of the formula and making our way to the outermost power of two. If we are dissatisfied with one approximation, then we need to start the next approximation essentially from scratch. Nevertheless, if we survive the computations, the convergence for this sequence is quite fast and provides accurate estimates of π in short order. This is essentially what Archimedes discovered some twenty two hundred years ago, when he determined that $\pi \approx \frac{22}{7} \approx 3.14$. It is all the more impressive to remember that he completed this excruciatingly delicate calculation entirely by hand and without the benefit of our decimal number system.

The advent of the computer has put large-scale numerical experimentation within our reach. For instance, back in 1949, ENIAC (the first general-purpose computer) determined the first 2 037 places of π in about 90 hours; nowadays, a desktop computer can do the same

in just a couple of seconds. However, we are still faced with the task of explaining to our computers exactly how to perform the calculations that we need them to perform. This is where programming languages come into play. In the course of this book, we will start to learn a programming language called *Maxima*. Just as our everyday thoughts are expressed in English (or perhaps Latin or Spanish) and our descriptions of quantitative phenomena are expressed using mathematical notation, our computational instructions will be expressed in the language of *Maxima*.

Before successfully using computers to study a mathematical problem, we need to understand three key areas:

① the *mathematical theory* that underlies the problem at hand,

② the *implementation*, or the way that we choose to represent mathematical objects inside the computer, and

③ the *syntax*, or the grammar that we use to communicate our instructions to the computer.

For instance, in order to compare the computational efficacy of Leibniz's series and Archimedes' sequence in our discussion above, we need to remember the mathematical concepts of infinite sequences and series, as well as the definition of convergence. These definitions form part of the mathematical theory that gives shape and context to the problem. Next, we need to decide whether these abstract objects will be represented numerically, symbolically, or graphically in our computer. This is the realm of implementation. Finally, we need to know what instructions to type into our computer to bring about this implementation. This is a question of syntax; it is here that we shall start our study of programming in general and of *Maxima* specifically.

In order to begin, you will need to sit down in front of a computer terminal and start up a session of *Maxima*.[1] You can do this by clicking on the `wxMaxima` icon from the *Applications* folder in a Macintosh or the *Start* menu in a Windows PC. Depending on the version of the software installed in your computer, the `wxMaxima` icon will look something like this:

After you click on this icon, your computer will open up a new *Maxima* window for you. When the *Maxima* session first starts up, you will see

[1] For instructions on how to install *Maxima* on your computer, consult page x.

the words "Ready for user input" at the bottom of the window. This is *Maxima*'s way of telling you that she is ready for your instructions.

Maxima consists of three distinct components running simultaneously on your computer:

① an external user interface or "front end" called the ***iris***,

② an internal computational engine called the ***kernel***, and

③ an extensive library of functions bundled in ***packages***.

The *Maxima* window is just the visible part of the iris, which is run by the application `wxMaxima`. The kernel is maintained by a separate application that runs in the background of your computer. The various packages are stored in a hidden directory in your computer's hard drive. For the sake of simplicity, we will use the term "*Maxima* session" to refer to our interaction with all three of these components.

The main content of a *Maxima* session is organized as a sequence of ***cells***, each one of which is made up of some text or of some *Maxima* instructions and their corresponding results. Every time that you type an instruction into the iris, hold down the *Shift* key on your keyboard, and press *Enter*, the iris passes this instruction to the kernel. The kernel may need look up one or more functions from a package in the library to execute this instruction, but when it is finished, it passes the result back to the iris to display. Then the whole process is repeated with a new cell.

To start, let us suppose that you want to make a header for your *Maxima* session. You may wish to include such information as your name, today's date, a title, and whatever other relevant comments you think are appropriate. For example, if you were working on the homework problems at the end of this section, you could include a phrase like "Section 1.1 homework" as part of your header. To do this, select

Cell ▷ *Insert Text Cell*

from the drop-down menu at the top of the screen. Immediately, a cell marker in the shape of a red square bracket will appear on the left hand side of the window. You can then start typing your header information.

When you have finished, you can create a new cell by going to the drop-down menu and selecting

Cell ▷ *Insert Input Cell*

Again, there will be a cell marker on the left-hand side of the window,

but this time you will also see an arrow-shaped input prompt that looks like this:

```
-->
```

The simplest kind of expression you might type at the input prompt is a number:

```
--> 485
```

When you press *Shift-Enter*, *Maxima* will respond by displaying the same number back to you:

```
(%i1) 485
(%o1) 485
```

You will observe that the arrow prompt was replaced with the input label `(%i1)`, and that the corresponding output is also given an output label `(%o1)`. For the sake of simplicity, we will ignore input and output labels throughout this book. Instead, we will indicate *Maxima* input and output by using a cell marker, with the instructions given in `typewriter` font and the corresponding output in normal Roman font. Thus, the short interaction above would be rendered as follows:

```
485;
485
```

Take special note of the use of the semicolon in the expression above. This is our first encounter with *Maxima*'s syntax, which requires that every expression end in either a semicolon or a dollar sign. Ending an expression with a semicolon instructs *Maxima* to display the result of evaluating the expression. In our example above, this result was the number 485. In contrast, ending an expression with a dollar sign instructs *Maxima* not to display the result and to just move on to the next expression. If you do not have the appropriate punctuation mark, then *Maxima* will insert a semicolon automatically for you, but sometimes this can result in an error message.

Numbers may be combined with ***arithmetic operators*** into more complicated expressions, such as:

```
139 + 346;
485
```

TABLE 1-1. *Maxima*'s five arithmetic operators, in order of decreasing precedence.

operator	operation	associativity
`^`	exponentiation	right to left
`/`	division	left to right
`*`	multiplication	left to right
`-`	subtraction	left to right
`+`	addition	left to right

```
5 * 97;
```
485

Maxima recognizes the five arithmetic operators listed in Table 1-1. In addition, parentheses provide a straightforward way to create even more complex expressions out of simpler ones by means of nesting. For instance, you might enter:

```
(3 * 5) / (10 - 6);
```
$$\frac{15}{4}$$

```
(3 * ((2 * 4) + (3 + 5))) + ((10 - 7) + 6);
```
57

Observe that as soon as you type an open parenthesis, *Maxima* will display it along with its matching close parenthesis, highlighting both. The cursor will then be placed in between the parentheses to allow you to enter a nested expression. When you reach the end of the expression, simply type a close parenthesis or use the right arrow key to move the cursor over the one that is already there. At first, you may find *Maxima*'s automatic parenthesis matching a little awkward and it might take you some time to get used to it. However, in time you will come to appreciate its utility, especially when you are entering much more complicated expressions than the ones above.

In principle, there is no limit to the depth of nesting or to the overall complexity of the expressions that *Maxima* can evaluate. In

fact, when presented with even the most complicated of expressions, *Maxima* always operates in the same basic ***read-evaluate-display cycle***:

① The iris reads an expression from the terminal.

② The kernel evaluates the expression.

③ The iris displays the result and awaits a new input.

When it comes to evaluating complex expressions, the second step in this cycle consists of first evaluating each subexpression separately, and then combining the results by applying the operator in question. For example, in order to evaluate the expression

```
(3 * ((2 * 4) + (3 + 5))) + ((10 - 7) + 6),
```

Maxima first evaluates each one of the subexpressions

```
3 * ((2 * 4) + (3 + 5))
```
and
```
(10 - 7) + 6
```

separately, and once it has those values in hand, it adds them together, as indicated by the operator `+`. Of course, to evaluate the first subexpression above, *Maxima* must evaluate

```
3
```
and
```
(2 * 4) + (3 + 5),
```

and to evaluate the second of these, *Maxima* must evaluate

```
2 * 4
```
and
```
3 + 5.
```

The process continues in this way until our original expression has been broken down into its simplest ***atomic*** constituents, at which point these are combined, two at a time, to make up the final result.

Fig. 1-1 shows a tree-like schematic of this evaluation process. The computation starts at the node at the top of the tree and moves down the branch on the left side as the first subexpression is evaluated. When this value is determined to be 48, the process returns to the top and follows the branch on the right as it evaluates the second subexpression. Then, once this value is found to be 9, the two values are combined into the final result of 57.

In practice, you can avoid using unnecessary parentheses in *Maxima* expressions. Therefore, instead of typing the long expression above, you might enter:

```
(%i1) 3 * (2 * 4 + 3 + 5) + 10 - 7 + 6;
(%o1) 57
```

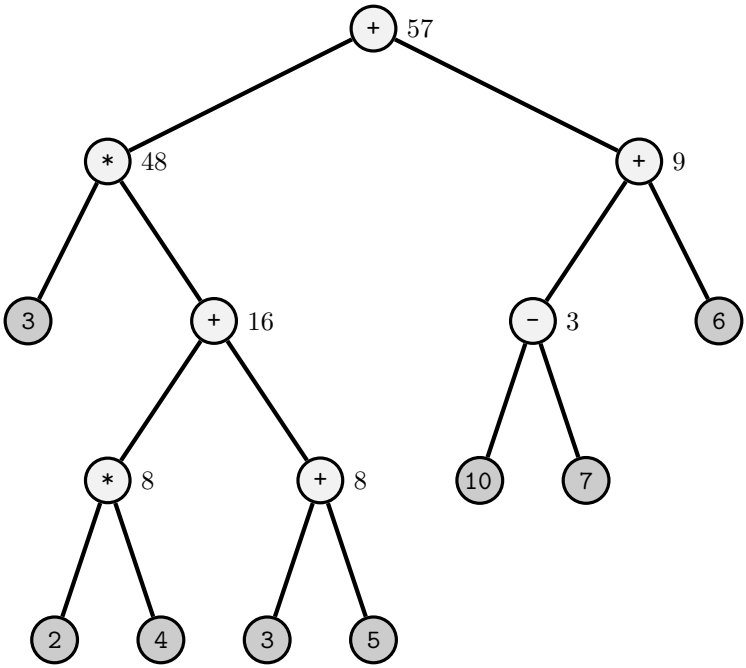

FIG. 1-1. Evaluating `(3*((2*4)+(3+5)))+((10-7)+6)`.

In either case, *Maxima* will return precisely the same value. In the absence of any parentheses, complex expressions are evaluated by following a strict precedence rule in which higher precedence operators are applied before lower precedence ones. The five arithmetic operators that appear in Table 1-1 are listed from highest to lowest precedence. This means that, in an expression with no parentheses, exponentiation is done first, followed by division, multiplication, subtraction, and finally addition. If the same operator appears more than once in an expression, then it will typically be applied from left to right; this is known as ***left-to-right associativity***. For instance, when evaluating the expression

```
3 * (2 * 4 + 3 + 5) + 10 - 7 + 6,
```

Maxima will first tackle the highest precedence operation, which in this case happens to be the product

```
3 * (2 * 4 + 3 + 5).
```

```
3 * (2 * 4 + 3 + 5) + 10 - 7 + 6
                ↓
  3 * (8 + 3 + 5) + 10 - 7 + 6
                ↓
    3 * (11 + 5) + 10 - 7 + 6
                ↓
      3 * 16 + 10 - 7 + 6
                ↓
        48 + 10 - 7 + 6
                ↓
          48 + 3 + 6
                ↓
            51 + 6
                ↓
              57
```

FIG. 1-2. Evaluating `3*(2*4+3+5)+10-7+6`.

This is followed by a subtraction and two additions, which are performed from left to right. Of course, before even the first multiplication can take place, the subexpression

```
2 * 4 + 3 + 5
```

must be evaluated. This, too, consists of a multiplication followed by two additions. Fig. 1-2 gives a step-by-step schematic of the entire evaluation process. Observe that, even though the final result is the same as before, the order in which the individual subexpressions were evaluated and combined is, in fact, different than the one shown in Fig. 1-1.

The only exception to the left-to-right associativity rule is exponentiation, which has ***right-to-left associativity***, as indicated in the last column of Table 1-1. We can see this principle at work in an expression like

```
4^3^2;
262144
```

which gives the same answer as the nested expression

```
4^(3^2);
```
262144

rather than the much smaller value of the expression

```
(4^3)^2;
```
4096

Besides the five arithmetic operators discussed above, *Maxima* is equipped with literally hundreds of ***primitive functions*** at our disposal. For example, we can call on the function `sqrt()`, which computes the square root of any number we give it:

```
sqrt(45);
```
$3\sqrt{5}$

A function call, like the one above, is always composed of the name of the function followed by one or more inputs surrounded by a pair of parentheses. As before, the use of parentheses allows nesting of functions, or indeed any combination of functions and operators, inside one another. For instance, suppose that we ask *Maxima* to determine the value of the following compound expression:

```
2 + sqrt(5^2 + (3 * 4)^2);
```
15

In this case, *Maxima* first evaluates the subexpression

```
5^2 + (3 * 4)^2
```

resulting in a value of 169. This value is then passed as the input to `sqrt()`, which returns an output value of 13. Finally, this result is added to 2 to produce the final answer displayed above.

Table 1-2 lists some useful primitive functions of which you should take note. As their names indicate, these functions compute absolute values, natural exponentials and logarithms, as well as the standard and inverse trigonometric functions. You should take particular care to remember that `log()` computes the natural logarithm (base e) and not the common logarithm (base 10), as you might otherwise expect:

```
log(2.718281828459045);
```
1.0

TABLE 1-2. Sixteen primitive functions of note.

`abs()`	`exp()`	`log()`	`sqrt()`
`sin()`	`cos()`	`asin()`	`acos()`
`tan()`	`cot()`	`atan()`	`acot()`
`sec()`	`csc()`	`asec()`	`acsc()`

You can store a computational object, like a number or a formula, in the kernel's memory by using the ***variable assignment operator*** denoted by a colon (`:`). For instance, the command

```
pi : 3.14;
```
3.14

makes the variable name `pi` indistinguishable from the numerical value 3.14 in any expression you might type. For example:

```
pi;
```
3.14

```
2 * pi;
```
6.28

```
pi^2 - 0.14 * pi - 6.70172;
```
2.71828

```
2 * cos(pi/5);
```
1.618408361976065

You can even use the value of one variable when you assign values to other variables. For instance, you might enter:

```
circumference : 2 * pi * radius;
```
6.28 radius

```
area : pi * radius^2;
```
$3.14\,\text{radius}^2$

This associates each of the variables `circumference` and `area` with a ***formula*** that depends on the variables `pi` and `radius`. In each case, `pi` was replaced with its value. On the other hand, `radius`, which is a ***free variable*** that has not yet been given a value, appears by name only. We can replace it with a value, say with the radius of the earth (in kilometers), by calling the substitution command `subst()` as follows:

```
rad_of_earth : 6371;
```
6371

```
subst(radius = rad_of_earth, circumference);
```
40009.88

```
subst(radius = rad_of_earth, area);
```
1.274514727410^8

Notice that `subst()` takes two inputs separated by a comma. The first consists of the substitution we wish to make (using an equal sign rather than a colon), and the second is the expression in which we wish to make the substitution. The resulting substitution is only temporary and does not affect the values stored in memory:

```
radius;
```
radius

```
circumference;
```
6.28 radius

```
area;
```
$3.14\,\text{radius}^2$

As you might imagine, *Maxima* maintains a record of all of the variables assigned during a session. You can see the names in this list by evaluating the variable `values`:

```
values;
```
[pi, circumference, area, rad_of_earth]

You can remove a variable from this list by invoking the command `kill()`:[2]

```
kill(rad_of_earth);
done
```

As expected, this command removes the variable in question from the list of assigned variables and disassociates it from its former value:

```
values;
[pi, circumference, area]
```

```
rad_of_earth;
rad_of_earth
```

Although you can use nearly any combination of letters or numerals and even some symbols like `%` or `\` to name a variable, there are a few rules you must abide by. First of all, you cannot start a variable name with a numeral. Secondly, you should remember that *Maxima* is ***case-sensitive***, so the names `Billy`, `BILLY`, and `billy` are all distinct. Finally, there are a few ***protected names*** that *Maxima* reserves for its own use. For example, you would get an error if you tried to store a value under the variable name `values`. You would get a similar error if you tried to assign a value to the names `%e`, `%phi`, or `%pi`, since these are permanently assigned to the exact values of the mathematical constants e, φ, and π. You can see their decimal (or floating-point) approximations by using the special function `float()` as follows:

```
float(%e);
2.718281828459045
```

```
float(%phi);
1.618033988749895
```

```
float(%pi);
3.141592653589793
```

[2] This aggressive-sounding command can also allow you to clear all of the contents in the kernel's memory and reinitialize the *Maxima* session. Simply enter `kill(all)`. Yikes!

Suppose that, inspired by the last result, you decide to use a better approximation for π in the circumference and area formulas that we defined above. To start, you would give the variable `pi` a new value by entering:

```
pi : float(%pi);
```
3.141592653589793

This change will affect all future interactions with *Maxima* during this session. For example, you can now define a new formula for the volume of a sphere by typing:

```
volume : 4/3 * pi * radius^3;
```
$4.188790204786391\,\mathrm{radius}^3$

As expected, the variable `pi` in this formula was replaced by its (new and improved) numerical value. On the other hand, the circumference and area formulas are not affected by the change in the value of `pi`. Instead, they stubbornly cling to their old (and now obsolete) values:

```
circumference;
```
$6.28\,\mathrm{radius}$

```
area;
```
$3.14\,\mathrm{radius}^2$

To understand the reason behind *Maxima*'s strange behavior here, we need to go back and remember exactly what happened in our interactions with *Maxima*, from the kernel's point of view. When we first assigned formulas to `circumference` and `area`, the variable `pi` was already assigned the value 3.14. Therefore, these formulas were immediately evaluated, respectively, as

$$6.28\,\texttt{radius} \qquad \text{and} \qquad 3.14\,\texttt{radius}^2.$$

These were the values that were originally stored in the kernel's memory. When the free variable `radius` was replaced with a numerical value, *Maxima* was able to evaluate both `circumference` and `area` as numbers, but their values in memory (in other words, the formulas above) never changed. Since the variable name `pi` does not appear in either of those formulas, they remained unaffected when the value of `pi` was changed later on, as we saw above. These observations might be summarized in the following fortune cookie mantra:

> The kernel remembers your instructions only in the order that you enter them.

Suppose that, in an attempt to remedy the situation, you decide to move the cursor back to the cell where you first gave `pi` a value, and change the contents of that cell to:

```
pi : float(%pi);
3.141592653589793
```

You can then bring the cursor back to bottom of the *Maxima* session and evaluate `circumference` and `area` once again. In this case, you will see:

```
circumference;
6.28 radius
```

```
area;
3.14 radius^2
```

Note that nothing has changed in the kernel's memory! Even though the iris shows the new definition of `pi` occurring *before* the definitions of `circumference` and `area`, as far as the kernel is concerned, `pi` was given its new value *after* these two formulas were defined. In fact, you can see that this is the case by taking a closer look at the numbers in the input and output labels for the cells containing the respective definitions. Once again, you will find that *the kernel remembers your instructions only in the order that you enter them.*

In order to fix the two formulas in your computer's memory, you will need to enter the original definitions for `circumference` and `area` a second time. Luckily, *Maxima* keeps track of all of the instructions that you have entered so far in your session, saving you the effort of typing the definitions from scratch. You can see these instructions by holding down the *Alt* key while pressing the up arrow key several times. You will see the last commands that you typed appear in reverse order in a new cell. You can also scroll forward through your commands by holding down the *Alt* key while pressing the down arrow key. For now, continue pressing *Alt*-⇑ until you see the command with which you first defined `circumference`, and then, without changing a thing, press *Shift-Enter*:

```
circumference : 2 * pi * radius;
```
$$6.283185307179586\,\text{radius}$$

Repeating the same process, press *Alt-*⇑ until you find the command with which you defined `area`. Once again, without changing anything, press *Shift-Enter*:

```
area : pi * radius^2;
```
$$3.141592653589793\,\text{radius}^2$$

You have finally changed the contents of the kernel's memory as desired!

Using the mouse to move back and forth between the various cells in a *Maxima* session is a convenient way to correct small errors; however you should do so only with extreme caution. In particular, since the order in which the contents of the *Maxima* session appear in the iris might not reflect the true contents of the kernel's memory, it is also an excellent way to propagate chaos, mayhem, and confusion. A safer (though arguably less efficient) strategy is to make use of the *Alt-*⇑ and *Alt-*⇓ shortcuts to scroll through your past commands and make the appropriate changes in order. Regardless of the approach you choose, you should always keep in mind that *the kernel remembers your instructions only in the order that you enter them*!

"*Nobody* expects the Spanish Inquisition!"[3] And nobody expects to encounter critical errors in their computations. However, every once in a while things may go awry and you will be required to restart *Maxima*. To prevent losing your data in such an event, you should save your work often. This can be done by selecting

File ▷ *Save*

from the drop-down menu at the top of the screen, by clicking the *Save* icon at the top of the *Maxima* window, or by using the keyboard shortcut *Command-S*.

The first time that you save a *Maxima* session, you will be asked to choose a name, a location for your file, and a document format. There are two file format options from which to choose. One of these options is a "*wxMaxima* document" ending with the file extension `.wxm`. This format saves all of your input during the session (including text comments), but none of the corresponding output produced by *Maxima*.

[3] *Monty Python's Flying Circus*, Series 2, Episode 2 (1970)

To recover the output, you would have to evaluate all of the session's input again, perhaps by choosing the drop-down menu selection

Cell ▷ *Evaluate All Cells*

or by pressing *Command-R*.

The second option is to save your *Maxima* session as a "*wxMaxima* XML document" ending in the extension `.wxmx`. This format saves both the input and output from a session, but at the cost of producing larger files. In general, the choice of format is entirely up to you, but if you are planning on turning in a *Maxima* session as a homework assignment, you should always use the XML format so your instructor can see the same results you saw before you saved your work. This is particularly important if you ignored our advice (which you are always free to do) and the contents of the iris do not reflect the order in which you evaluated them.

Finally, to close the *Maxima* session and exit, you can either go to the drop-down menu and select

wxMaxima ▷ *Quit wxMaxima*

or press *Command-Q*. If you have not already done so, you will be offered one last chance to save your work. Then, after a few moments, the iris will close down until you summon *Maxima* once again at a later time.