A tutorial introduction to AXIOM with TEXMACS

The AXIOM computer algebra system has been acclaimed as a milestone in its field but, to date, has tended to be used mainly by serious mathematical researchers.

This tutorial is an attempt to demystify AXIOM and to bring its unique advantages to a wider audience. It deliberately avoids using advanced mathematical techniques, choosing its examples from topics normally encountered at school, or just beyond. As well as making AXIOM accessible to both non-mathematicians and those at an early stage of their mathematical careers, it is hoped that this approach will also be acceptable to those mathematicians who might otherwise have been intimidated by the *computer* in computer algebra – by allowing them to learn "the AXIOM way of doing things" in a context of totally familiar results.

The primary purpose of the tutorial is to allow new users to become at ease with the AXIOM style of working. All of the examples in the text are provided with AXIOM and can be evaluated within the AXIOM environment.

The AXIOM output in this tutorial was produced automatically by TEXMACS using the AXIOM interface under Linux.

Because of the difficulty in drawing a dividing line between computer input and the mathematical objects to which it refers, we have tended to use the typewriter font rather than the more usual mathematical italic for the names of variables and the like: i.e. `x` rather than x.

Up

Chapter 1, First steps

AXIOM belongs to the general class of programs known as "computer algebra" or "symbolic manipulation" packages. Such packages are generally capable of a wide range of mathematical operations, from simple arithmetic to (in AXIOM's case) abstract algebra. This does not mean that the user needs to know about such a wide range of mathematical subjects – indeed, many practicing mathematicians would be familiar with only a subset of the topics which AXIOM can handle.

The first rule in getting familiar with AXIOM is *do not be intimidated*. AXIOM may provide an unusually rich choice of features but you do not need to use them all – you will soon become familiar with those you need by actually using them. Like any computer package, AXIOM has its own basic rules governing how you communicate with it and, whilst these may appear strange at first sight, they are not particularly complicated and will soon become familiar.

Finally, you may, as you progress with AXIOM, begin to feel that it is unduly fussy at times. It is fussy, but *not* unduly so. AXIOM insists on performing any operation only on objects which are (or can become) of the type for which that operation is defined; it also makes a point of "considering all the possibilities". AXIOM's careful approach ensures that it avoids many pitfalls which are present in less formal computer algebra systems and which can, at times, lead to completely spurious results.

Chapter 2

Up

1.1. Beginning, ending ...

When you start AXIOM with TEXMACS the first thing you see is a TEXMACS window. TEXMACS is used as an interface to the AXIOM system. Some of the commands you type in TEXMACS are sent to AXIOM for processing, and TEXMACS is responsible for printing the results, including any typeset mathematics.

To start an AXIOM session within TEXMACS select Session Axiom from the Text pull-down menu. The symol → (called a prompt) appears on the right indicating that Axiom is ready for input.

→

A command is sent to AXIOM by pressing *Enter*. A new AXIOM input-area appears immediately following the output.

Some commands, known as *system* commands, are used to perform non-mathematical operations within AXIOM. These commands are preceded by a right parenthesis `)`. An example of a system command is `)show` which is used to gain information about types in AXIOM. (The system command `)help` gives further information on available system commands.)

AXIOM keeps an internal record of the *command history*. To see the commands typed, enter `)history )show` at any stage. What you get is a numbered list of AXIOM commands as they were typed in. System commands are not recorded in the command history and cannot be refered to using the %% syntax. (See section 1.6 for an explanation of this syntax.) Note also that commands which return no output do not print the command number, although they are stored in the command history.

To end an AXIOM session simply select Close TeXmacs from the File pull-down menu.

Next

1.2. Simple arithmetic

Now let us try some simple calculations.(1)

```
→ 1+1
```

$$2 \qquad (1)$$

Type: PositiveInteger

→

The user input is typed after the → symbol. If you are using TEXMACS to view this tutorial and if you have Axiom installed, then you may try entering Axiom commands your self right now. Just position the cursor after the promt and trying typing a command, e.g. $1+2$.

Note that as well as the answer AXIOM gives its *type*, for future reference. The notion of types of expressions is fundamental to the way AXIOM works, as will become apparent.

One of the characteristics of computer algebra systems is their ability to handle numbers of *arbitrary precision*, unlike conventional programming languages which are restricted to fixed levels of precision, such as ten or fewer digits in integers.

AXIOM is a typical computer algebra system in this respect – it does not restrict the number of digits allowed:

```
→ 123^45
```

$$\begin{array}{c} 1111040818513195628591079058717645191855915321226802182362907319986611100124274328396612704804 3 \end{array} \qquad (2)$$

Type: PositiveInteger

→

but provides all the 95 digits necessary to express this result. (The ^ represents exponentiation and may be read as "to the power". As in other languages, ** may also be used to represent exponentiation.

More complicated expressions may use parentheses to indicate the order in which they should be evaluated:

```
→ 2^(3+4)
```

$$128 \qquad (3)$$

Type: PositiveInteger

→

and, as in most languages, may sometimes be typed more simply using some built in rules of *precedence*:

exponentiation (^)	has higher precedence than (is done before)
multiplication (*) and division (/)	which have the same precedence, higher than
addition (+) and subtraction (-)	which again have the same precedence.

Operations with the same precedence are performed in the order in which they are input – that is, from left to right.

So far, the results of our operations have all been of type `PositiveInteger`, however, if we use division in our expression this is no longer the case:

```
→ 4/3
```

$$\frac{4}{3} \tag{4}$$

Type: Fraction Integer

→ 2/2

$$1 \tag{5}$$

Type: Fraction Integer

→

– although the second result is displayed as 1, AXIOM is telling us that it is really a fraction (whose components are integers). This kind of distinction can be important, since we can do things with fractions which are impossible if we only have integers (division being a case in point). In this simple case we are unlikely to be misled but, in more complicated situations, we could be.

Next Previous

Up

1.3. Assignments

So far, we have only evaluated expressions. Generally, we want to set up *variables* to hold the values of the expressions, to reuse later. The operation of giving a value to a variable is known as *assignment*. The simplest type of assignment in AXIOM is *immediate assignment*, which has the form *variable* `:=` *expression*. In this, the value of the expression is calculated immediately and this becomes the value of the variable. For instance, if we have

```
→ a := 2
```

```
   2                                                                    (7)
                                                        Type: PositiveInteger
```

```
→ b := a
```

```
   2                                                                    (8)
                                                        Type: PositiveInteger
```

```
→
```

both `a` and `b` now have the value 2 and each will continue to do so until its value is explicitly changed by another assignment. Changing the value of `a` does not affect `b`:

```
→ a := 3
```

```
   3                                                                    (9)
                                                        Type: PositiveInteger
```

```
→ b
```

```
   2                                                                   (10)
                                                        Type: PositiveInteger
```

```
→
```

In some computer algebra systems, although assignments equivalent to these will have the same effect, reversing the order of commands `(6)` and `(7)` would cause `b` to depend on `a`, so that the final result at `(9)` would have been 3. This can be very confusing, even to experienced users. In AXIOM, these two types of assignment are clearly distinguished. The second, known as *deferred assignment*, uses the operator `==` to establish a permanent relationship between a variable and an expression. We shall return to this in section 2.7.

Note that variable names (and other *symbols*) usually begin with an alphabetic character and continue with alphabetic and numeric characters. Upper and lower case letters are distinct, so that `AB`, `Ab`, `aB` and `ab` are four different names. The characters `%` and `!` are also allowed in names, with `?` allowed in non-initial positions; these three characters can, however, have a special significance when they appear in certain positions. In AXIOM, the underline or underscore character `_` is an escape character, so that `A_B` is the same as `AB`.

Next Previous

Up

1.4. Setting types

Although AXIOM will give the variable used in an assignment a type appropriate to the value of the expression, it is also possible for the user to fix the type of a variable.

For example:

```
→ i : Integer
                                                                    Type: Void
→
```

– remember that the name of the type must be appropriately capitalised. (The act of assigning a type to `i` does not return a value, and so the result is of type `Void`.)

Once the type of a variable has been fixed in this way, only expressions which can be converted to the chosen type may be assigned to the variable:

```
→ i := 2/3

      Cannot convert right-hand side of assignment
      2
      -
      3
         to an object of the type Integer of the left-hand side.

→ i := 4/2
   2                                                                    (12)
                                                                 Type: Integer
→
```

(Note that the command number did not increase after the unsuccessful assignment, indicating that nothing has been changed by it.)

It is also possible to combine a type declaration and an assignment into a single statement:

```
→ c : PositiveInteger := 3
   3                                                                    (13)
                                                          Type: PositiveInteger
→
```

This is exactly equivalent to giving the type declaration and assignment separately.

If several variables are to be declared to have the same type, this can be accomplished in a single statement, by separating each pair of names with a comma and enclosing them in parentheses, as in

```
→ (j, k, l) : Integer
                                                                    Type: Void
→
```

– the parentheses are necessary to tell AXIOM that the type declaration applies to all of `j`, `k` and `l`, not just to `l`.

Whilst discussing types, we should note the use of the word *domain*, which occurs widely in the AXIOM literature. A domain may be thought of abstractly as the collection of all the possible objects of a particular type, so that the expressions "has type *Type1*" and "belongs to the domain *Type1*" are equivalent. It is also used to mean the collection of code in the AXIOM system which defines an abstract domain.

Next Previous

Up

1.5. Keeping track

If an AXIOM session has run for a while, you may lose track of the names of the objects which you have created. If this happens, try

```
→ )display names
   Names of User-Defined Objects in the Workspace:
   %    a    b    c    i    j    k    l
   Names of System-Defined Objects in the Workspace:
   %e                %i                %infinity         %minusInfinity
   %pi               %plusInfinity     SF

→
```

Don't be alarmed at the first item, `%` – this is a variable used to hold the result of your most recent calculation (see section 1.6). The "System-Defined Objects" are system *macros* (see section 2.7) put there for you by AXIOM. To find out more about any object, say `SF`, type

```
→ )display properties SF
   Properties of SF :
      This is a system-defined macro.
      macro SF () == DoubleFloat()

→
```

You will find that it is a *macro* – in effect, an abbreviation – for `DoubleFloat`, which is an AXIOM type used to represent real numbers approximately in "floating point" form, using the "double precision" representation available on the type of computer on which AXIOM is running. On most computers this means a representation with about 16 decimal digits of accuracy. `SF` originally stood for `SmallFloat`, in contrast to AXIOM's own `Floats`, which can be defined to have as many digits of accuracy as you wish; see section 2.7 for more details.

The system macros beginning with `%` correspond to the mathematical objects whose names follow the `%` sign.

Having discovered the names of the variables in use, you may decide that some of these – say `c` and `i` – are no longer necessary. They can be most simply removed as follows

```
→ )clear properties c i
→
```

Recall that we declared `i` to have type `Integer` and gave it the value 2. The type declaration and value could have been cancelled separately, with `)clear mode i` and `)clear value i`, respectively. The keywords `properties`, `mode` and `value` may be abbreviated to their initial letters.

Beware of typing commands such as `)clear a` or `)clear c` if you really mean `)clear value a` etc. In particular, `)clear c` is taken to mean `)clear completely` and is quite thorough in doing so: try it once and see (then remember not to, in future).

You can return to a completely "clean slate" without having to reload AXIOM, by using the command `)clear all` or `)clear a`. However, code which has been loaded by AXIOM is not removed in this process and so will not be reloaded if previously used commands are repeated.

Next Previous

Up

1.6. Accessing previous results, recalling input

Suppose that we have just calculated a value for some (possibly quite complicated) expression, then realise that we are likely to need this value in other calculations. This is easily remedied: the command `keepit := %` will assign the most recent output to the variable `keepit` (or to whatever other variable you have chosen to use). `%` refers to the most recently calculated result. Other features of the *history* mechanism are provided by the function `%%` – the result of instruction number n is referred to as `%%(`n`)`, so:

```
→ %%(1)
```

```
   2                                                                  (15)
                                                        Type: PositiveInteger
```

```
→
```

displays the result of the very first command, `1 + 1`. (If it doesn't do so for you, this is probably because you tried `)clear c`. Type in a few expressions and try again.) It is also possible to count back from the present line number, by using a negative value for n; thus:

```
→ 1 + %%(-3)
```

```
   4                                                                  (16)
                                                        Type: PositiveInteger
```

```
→
```

adds one to the result of step `(15 - 3)`, that is, step `(12)`.

If you simply want to see the last few (k say) commands which you have issued, you can use the `)history` command itself, in the form `)history )show` k. This will replay your input to the last k distinct prompts. If you omit the k, AXIOM uses a default value of 20.

Adding `both` to the end of the `)history )show` command causes AXIOM to display both input and output; in this case, the default for k is reduced to 5.

Next Previous

1.7. Continuation lines

Occasionally, you may wish to type a command which will not fit on a single line. On many computer systems you can simply continue typing without pressing *Enter* or *Return* and the line will "wrap around" on your screen; however, there is usually a limit to the length allowed for such extended lines.

To continue a command within AXIOM with TEXMACS, prese *Shift+Enter*. For example,

```
→ 1
  +
  1

  2                                                              (17)
                                                  Type: PositiveInteger

→ 1
  +
  2

  3                                                              (18)
                                                  Type: PositiveInteger

→
```

Next Previous

1.8. Comments

AXIOM commands can be stored in *input* files. Until $\TeX_{MACS}$ files, input files are simple AXII format text files. This facility allows you to build up a complicated series of commands in a file (or several files) and have AXIOM process them in order. To help users make sense of these files, AXIOM has a facility for adding comments.

The basic method for a user to add comments in AXIOM is to use the character string `--` to introduce them. Anything between this flag and the end of the (possibly continued) line is ignored. If you wish to make comments which are more than one line long, either continue the line with `_` or use `--` before each individual line of the comment. Flagging each individual line as a comment is clearer.

(You may at some point encounter AXIOM and meet a second type of comment which begins with the string `++`. This is used in the automatic generation of on-line documentation.)

Here is an example of the use of a comment:

```
-- Commenting alone doesn't increment the command count.
```

Chapter 2 Previous

Up

Chapter 2, Solving some simple problems in algebra

Chapter 1 Chapter 3

2.1. Solving polynomial equations

Elementary algebra consists mainly of solving problems which can be represented as low order polynomial equations, perhaps with rational polynomial equations included. Let us see how AXIOM handles some typical problems at this level.

Typing: `solve(3*x=x+2)` The first time that you issue a command like this within a session there is a slight delay. This is because AXIOM contains very large amounts of code – too much to fit into the memory of most present day computers. Furthermore, it will attempt to grow to accommodate the size of problems which it encounters. Thus, it can potentially be too large for any computer. For these reasons, the version which starts up provides only the central, non-algebraic "kernel" of the system, together with a very few other facilities some of which we used in chapter 1 First Steps; other subsystems are loaded as needed. The subsystems required depend on the type of problem being solved: you will gradually learn which *domains* and *packages* you regularly use (a package in AXIOM is simply a collection of related functions). Next, AXIOM reports the actual answer

```
→ )clear all

   All user variables and function definitions have been cleared.

→ solve(3*x=x+2)
```

$$[x = 1] \tag{1}$$

```
                                   Type: List Equation Fraction Polynomial Integer
→
```

or, to be more precise, the list of answers. Lists in AXIOM are enclosed in square brackets (`[]`) and have their components separated by commas (`,`), as we shall see shortly. In this case, the list contains only one solution to the problem posed.

Note that the AXIOM variable `x` does not acquire a value in the solution process:

```
→ x
```

$$x \tag{2}$$

```
                                                          Type: Variable x
→
```

Continuing with our simple examples:

```
→ solve(3*x - 1 = 0)
```

$$\left[x = \frac{1}{3}\right] \tag{3}$$

```
                                   Type: List Equation Fraction Polynomial Integer
→ solve(3*x - 1)
```

$$\left[x = \frac{1}{3}\right] \tag{4}$$

```
                                   Type: List Equation Fraction Polynomial Integer
→
```

Here, AXIOM followed the common convention that, if only one side of an equation is specified to `solve`, this is assumed to be completed by the addition of `= 0`.

```
→ solve(3*x^2 - 7*x + 2)
```

$$\left[x=2,\ x=\frac{1}{3}\right] \tag{5}$$

Type: List Equation Fraction Polynomial Integer

```
→ solve(x^2 -2)
```

$$\left[x^2-2=0\right] \tag{6}$$

Type: List Equation Fraction Polynomial Integer

→

The solution, in each case, is a list of equations which can be expressed in terms of ratios of polynomials with integer coefficients. If we start from such an equation (which we did in these cases), the solution can always be expressed in this way and this is what AXIOM does, by default.

In the last example, the list which AXIOM returned simply contained the completed equation, since this particular equation cannot be solved in terms of the rational numbers. We shall return to this shortly but first let us look at some higher degree polynomial equations.

```
→ solve(x^4 - 8*x^3 + 23*x^2 - 28*x + 12)
```

$$[x=3,\ x=2,\ x=1] \tag{7}$$

Type: List Equation Fraction Polynomial Integer

→

We have, of course, followed tradition in choosing an equation which can be solved exactly. However, a fourth degree equation ought to have four solutions, so one of the roots must be repeated. The usual method of solving such an equation by hand is to attempt to factor it, so

```
→ factor(x^4 - 8*x^3 + 23*x^2 - 28*x + 12)
```

$$(x-3)(x-2)^2(x-1) \tag{8}$$

Type: Factored Polynomial Integer

→

– sure enough, the second factor is squared, so we have a repeated root at 2.

Returning to the solution of `x^2 - 2 = 0`, we can obtain solutions in terms of `x` if we allow square roots to appear in them. AXIOM provides a command `radicalSolve` which will return solutions in terms of radicals (or nth roots).

```
→ radicalSolve(x^2 -2)
```

$$\left[x=\sqrt{2},\ x=-\sqrt{2}\right] \tag{9}$$

Type: List Equation Expression Integer

→

Note that `radicalSolve` only returns the solutions which can be expressed in this way and ignores all others. For example:

```
→ radicalSolve(x^5+x^2+1)
```

$$[] \tag{10}$$

Type: List Equation Expression Integer

→

– you may, therefore, wish to use `solve` or `factor` first.

The expressions produced by `radicalSolve` soon become unmanageable, however: try typing `radicalSolve(x^4+x+1)`. For some applications, we may be satisfied with a numeric approximation of the solution. AXIOM will provide this to within a specified tolerance if that tolerance is given as the second parameter of `solve`:

```
→ solve(x^2 - 2, 0.00001)
```

$$[x = -1.414211273193359375,\ x = 1.414211273193359375] \tag{11}$$

Type: List Equation Polynomial Float

→

Note that, although AXIOM has returned the results to 19 digits, only the first six figures are reliable, given the accuracy which we requested. This may be irritating to numerical analysts but is a fairly standard approach in computer algebra, where it is generally assumed that users, unless they indicate otherwise, want to see precisely what they have calculated. By default, AXIOM provides `Floats` to 20 digits of accuracy. If you wish to change this, to 30 say, issue the command `digits(30)`.

It is possible to control, independently, the number of significant digits printed, using the `outputGeneral` command:

```
→ outputGeneral 6
```

Type: Void

```
→ %%(11)
```

$$[x = -1.41421,\ x = 1.41421] \tag{13}$$

Type: List Equation Polynomial Float

→

`outputGeneral` is simply another AXIOM function; although it returns nothing, it achieves what we wanted as a *side effect*. In using it, we omitted the parentheses around its argument. This is permissible when a function has a single argument and there is no ambiguity in where that argument ends. If you are not familiar with this convention, you should note that function application has a higher precedence than arithmetic operators, so that `sin 2*a` means `(sin 2)*a` and not `sin(2*a)`. One effect of this is that we cannot omit the parentheses around arguments which begin with a minus – for instance `abs(-1)` represents `1`, the absolute value of `-1`, but `abs -1` is taken to mean the polynomial `abs - 1`, in which `abs` is the name of a variable. This occurs because an attempt to apply the function `abs` before the arithmetic operator would result in `abs` being applied to the minus sign alone, which is meaningless, so that AXIOM must look for another interpretation and decides that *this* `abs` is a variable.

We can return to using AXIOM's default setting for significant digits by issuing the command `outputGeneral()`. However, for legibility, we shall retain the present setting for the rest of this chapter. It is possible for the user to see `Floats` in floating (scientific) notation, e.g. $0.141421e1$, or in fixed notation, e.g. 1.41421. By default, AXIOM chooses whichever representation is more appropriate, the so-called general notation.

It is similarly possible to ask AXIOM to produce an approximate rational solution, by using a rational tolerance:

```
→ solve(x^2 - 2, 1/100000)
```

$$\left[x = -\frac{370727}{262144},\ x = \frac{370727}{262144}\right] \tag{14}$$

Type: List Equation Polynomial Fraction Integer

→

This may be less immediately informative for the reader but can simplify later operations which use the result – in general, AXIOM is happier with exact rationals than with floating point approximations. We shall see an example of this in chapter 3 Calculus. Here, as before, the exact solutions will differ from those displayed by at most the requested tolerance.

When `solve` is used with a tolerance parameter, if the coefficients in the equation are real, only real solutions will be returned. To obtain complex solutions in this situation, we must use `complexSolve` instead:

```
→ solve(x^2-2*x+3,0.00001)
```

$$[] \tag{15}$$

Type: List Equation Polynomial Float

```
→ complexSolve(x^2-2*x+3,0.00001)
```

$$[x = 1.0 - 1.41421i,\ x = 1.0 + 1.41421i] \tag{16}$$

Type: List Equation Polynomial Complex Float

→

– in this case, of course, the solutions represent the complex numbers $1 \pm \sqrt{2}i$.

If the coefficients in your equation are expressed as decimal fractions, as in `x^2 - 1.21 = 0`, you can use the techniques of this section, after first converting the coefficients to rational fractions, using `::` (AXIOM's *type conversion* operator). Rather than convert each individual coefficient, we can usually save some keystrokes by converting the entire polynomial (or equation). For example:

```
→ solve((x^2 - 1.21) :: Polynomial Fraction Integer, 0.00001)
```

$$[x = -1.1,\ x = 1.1] \tag{17}$$

Type: List Equation Polynomial Float

→

(if the coefficients were complex, we should use `Complex Integer` in place of `Integer`). We shall return to type conversion in section 2.4 Rearranging Expressions.

AXIOM will also solve equations involving several variables, expressing the one we specify in terms of the others. To obtain a familiar result we could use

```
→ radicalSolve(a*x^2 + b*x + c, x)
```

$$\left[x = \frac{-\sqrt{-4ac+b^2}-b}{2a},\ x = \frac{\sqrt{-4ac+b^2}-b}{2a}\right] \tag{18}$$

Type: List Equation Expression Integer

→

Next

2.2. Digression – taking things apart

We might wish to manipulate the individual solutions which AXIOM has been returning to us as components of lists. This facility is provided by the operator . which returns individual components of lists: *list*.*n* is the *n*th component of *list*.

Continuing the previous example:

```
→ qs := %; -- the semicolon (;) inhibits AXIOM's output display
                                        Type: List Equation Expression Integer
→
```

– but not the type information.

```
→ qs1 := qs.1
```

$$x = \frac{-\sqrt{-4a\,c+b^2}-b}{2a} \tag{20}$$

```
                                        Type: Equation Expression Integer
→
```

We can use the functions `lhs` and `rhs` to access the two sides of the equation:

```
→ x1 := rhs %
```

$$\frac{-\sqrt{-4a\,c+b^2}-b}{2a} \tag{21}$$

```
                                        Type: Expression Integer
→
```

With these facilities, we could have obtained a numeric value for `x` directly from `(9)`:

```
→ numeric rhs %%(9).1
```

$$1.41421 \tag{22}$$

```
                                        Type: Float
→
```

We can save ourselves some effort by using the `map` function, which applies a function to each top level component of a structure:

```
→ xs := map(rhs, qs)
```

$$\left[\frac{-\sqrt{-4a\,c+b^2}-b}{2a},\ \frac{\sqrt{-4a\,c+b^2}-b}{2a}\right] \tag{23}$$

```
                                        Type: List Expression Integer
→
```

Top level components include the elements of a list, the entries in a matrix, the two sides of an equation, the numerator and denominator of a fraction and many other instances: if you have an object with, in some sense, constituent parts then there is probably a version of `map` which applies to it – try it and see, or see if you can use `map` to help factor the fraction `15015/32768`.

We can then obtain some more familiar results:

```
→ xs.1 + xs.2
```

$$-\frac{b}{a}$$

(24)

Type: Expression Integer

→ `xs.1 * xs.2`

$$\frac{c}{a}$$

(25)

Type: Expression Integer

→

You might like to try using the same method to obtain the corresponding relationships for third and fourth degree polynomial equations. The individual solutions are very long and complicated so you probably do not want to see these – remember to use a final semicolon to prevent AXIOM from printing them.

Next Previous

2.4. Rearranging expressions

One of the most basic things we learn from algebra is a facility for reorganising symbolic expressions into more convenient forms. However, many computer algebra systems will either stubbornly refuse to express a result in the form we want or will only do so after a selection of obscure switches has been set, which usually results in some other expression taking an undesired form.

AXIOM's type system provides a much more sensitive mechanism for converting the forms of expressions, since it allows us to control the type of individual objects. What is more, it is much easier to learn than an arbitrary collection of switches, having a simple, consistent and mnemonic approach.

We saw in section 1.4 Setting Types how we could fix the type of a variable with the `:` operator. It is also possible to manipulate the type of an expression with the `::` operator:

```
→ a := (x+ y)/2
```

$$\frac{1}{2}y+\frac{1}{2}x \tag{31}$$

```
                                   Type: Polynomial Fraction Integer
```

```
→
```

AXIOM has obligingly separated the coefficients of `x` and `y` for us and the type reflects this – the object is a *polynomial* whose coefficients are *fraction*s of *integers*; if we want to insist on the form which we first input, this can easily be achieved – what we want is a *fraction* of *polynomial*s whose coefficients are *integers*:

```
→ a:: Fraction Polynomial Integer
```

$$\frac{y+x}{2} \tag{32}$$

```
                                   Type: Fraction Polynomial Integer
```

```
→
```

However, note that we have not changed `a` since our instruction did not involve an assignment:

```
→ a
```

$$\frac{1}{2}y+\frac{1}{2}x \tag{33}$$

```
                                   Type: Polynomial Fraction Integer
```

```
→
```

– to change the type of `a` by this method, we must assign the result to `a`:

```
→ a := a :: Fraction Polynomial Integer
```

$$\frac{y+x}{2} \tag{34}$$

```
                                   Type: Fraction Polynomial Integer
```

```
→ a
```

$$\frac{y+x}{2} \tag{35}$$

```
                                   Type: Fraction Polynomial Integer
```

```
→
```

(We shall shortly see how to control the order of the variables `x` and `y`.)

We have met a more compact method of setting the type of a variable. First resetting `a`:

```
→ a := (x+y)/2;
                                                    Type: Polynomial Fraction Integer
→
```

we can assign its value to a variable of the desired type:

```
→ b : Fraction Polynomial Integer := a
```

$$\frac{y+x}{2} \tag{37}$$

```
                                                    Type: Fraction Polynomial Integer
→
```

but we cannot use this to redefine the type of `a`:

```
→ a : Fraction Polynomial Integer := a

      You cannot declare a to be of type Fraction Polynomial Integer
         because either the declared type of a or the type of the value of
         a is different from Fraction Polynomial Integer .

→
```

`Fraction` and `Polynomial` are examples of *type constructors*, which are used in AXIOM to build new types from old. `Fraction` *type1* specifies fractions whose numerator and denominator are of type *type1*; `Polynomial` *type2* specifies polynomials whose coefficients are of type *type2*. Another simple type constructor is `Factored`, which does exactly what its name suggests:

```
→ y := x^2 + 3*x + 2
```

$$x^2+3x+2 \tag{38}$$

```
                                                    Type: Polynomial Integer
→ y := y :: Factored Polynomial Integer
```

$$(x+1)(x+2) \tag{39}$$

```
                                                    Type: Factored Polynomial Integer
→
```

We have also met the type constructors `Complex` and `Equation` and the type `Float`.

As type names can become rather unwieldy, we are permitted to abbreviate their components:

`INT`	for	`Integer`	`FRAC`	for	`Fraction`
`NNI`	for	`NonNegativeInteger`	`FR`	for	`Factored`
`PI`	for	`PositiveInteger`	`EQ`	for	`Equation`
`POLY`	for	`Polynomial`	`EXPR`	for	`Expression`

(For some of the shorter type names the abbreviation is merely the same word all in upper case, eg. `FLOAT` for `Float`.)

If our expressions involve polynomials in more than one variable (multivariate polynomials), AXIOM will choose which to treat as the "main variable", with the others appearing in its coefficients.

```
→ )clear p y -- since y has a value
→ P := (y + z)*x^2 + z*x + c
```

$$(x^2+x)z+x^2y+c \tag{40}$$

Type: Polynomial Integer

→

Here, AXIOM chose the alphabetically last variable, `z`, although we carefully input the expression for `P` as a polynomial in `x`.

We can control the choice of main variable by using the type `UnivariatePolynomial`, abbreviated to `UP`, which takes two parameters, the first specifying the variable of choice and the second the type of the coefficients. To display `P` in the form we chose for input, we could try:

```
→ P :: UP(x,POLY INT)
```

$$(z+y)x^2+zx+c \tag{41}$$

Type: UnivariatePolynomial(x,Polynomial Integer)

→

which almost gives the form we want. We can, of course, also use `UnivariatePolynomial` to control the form of the coefficients (specifying the "next most important" variable and so on):

```
→ P :: UP(x, UP(y, POLY INT))
```

$$(y+z)x^2+zx+c \tag{42}$$

Type: UnivariatePolynomial(x,UnivariatePolynomial(y,Polynomial Integer))

→

Note that in this example, `P` has retained its original type as provided by AXIOM; we have, so far, merely displayed it in different forms. We could, if we wished, completely specify the ordering of the variables in `P`:

```
→ P := P :: UP(x, UP(y, UP(z, UP(c, INT))))
```

$$(y+z)x^2+zx+c \tag{43}$$

Type:
UnivariatePolynomial(x,UnivariatePolynomial(y,UnivariatePolynomial(z,UnivariatePolynomial(c,Intege

→

In practice, if we are interested in the order of the variables in a multi-variate polynomial we would use the AXIOM type whose first argument is an ordered list of symbols, but we shall not go in to such details here.

Various other useful forms of polynomials are discussed in section 1.9 of the AXIOM manual.

Next Previous

2.5. Summation of series

We can use AXIOM to sum a variety of series,such as we might encounter in a basic algebra course. For instance, to find the sum of the first n terms of the series:

$$\frac{1}{1\times 4\times 7}+\frac{1}{4\times 7\times 10}+\ldots+\frac{1}{(3r-2)(3r+1)(3r+4)}+\ldots$$

```
→ )clear p all
→
```

axiom]

```
→ sum(1/((3*r-2)*(3*r+1)*(3*r+4)), r=1..n)
```

$$\frac{3n^2+5n}{72n^2+120n+32} \tag{44}$$

Type: Union(Fraction Polynomial Integer,...)

→

To find the sum to infinity of the same series:

```
→ limit(%, n=%plusInfinity)
```

$$\frac{1}{24} \tag{45}$$

Type: Union(OrderedCompletion Fraction Polynomial Integer,...)

→

The type returned by `sum` was `Union(Fraction Polynomial Integer,Expression Integer)` – an object is in a `Union` of types if it is guaranteed to belong to one of the branches. Users may, of course assign this to a variable of a type not involving `Union`. The `OrderedCompletion` mentioned in the `limit`'s type is the result of joining $-\infty$ and ∞ to the type `Fraction Polynomial Integer`. Finally, the "failed" branch of the `Union` represents the circumstance where no result may be returned, i.e. the limit does not exist.

Having encountered the notion of series summation, we may turn to some commonly encountered series, such as arithmetic progressions (APs). If the initial term of an AP is `a` and the increment is `b` then the `r`th term is $a+(r-1)*b$ and the sum of the first `n` terms can be found by:

```
→ SA := sum(a + (r-1)*b, r = 1..n)
```

$$\frac{b\,n^2+(-b+2a)n}{2} \tag{46}$$

Type: Fraction Polynomial Integer

→

This is probably not the most familiar form of this result. However, we can, if we wish, adjust its form by using some of the methods met in the previous section:

```
→ SA :: UP(a, Polynomial Fraction Integer)
```

$$n\,a+\frac{1}{2}b\,n^2-\frac{1}{2}b\,n \tag{47}$$

Type: UnivariatePolynomial(a,Polynomial Fraction Integer)

```
→ SA :: UP(a, UP(b, FRAC FR POLY INT))
```

$$n\,a + \frac{(n-1)n}{2}b \tag{48}$$

```
Type: UnivariatePolynomial(a,UnivariatePolynomial(b,Fraction Factored
Polynomial Integer))
```

→

We can also sum a geometric progression:

→ `SG := sum(a*b^(r-1), r=1..n)`

$$\frac{a\,b\,b^{(n-1)} - a}{b-1} \tag{49}$$

```
                                                    Type: Expression Integer
```

→

This is of type `Expression Integer`. Expressions are more complicated than polynomials in that they can involve fractions and various kinds of functions. We shall meet them again, later on.

Next Previous

2.6. Developing series

At a slightly more advanced level we encounter the notion of developing a series expansion of some expression – for instance, the binomial expansion. AXIOM can generate the general binomial series for us:

```
→)set stream calculate 5 ⟨bgroup⟩⟨egroup⟩
```

```
→series((1 + x)^n, x=0)
```

$$1 + n\,x + \frac{n^2-n}{2}x^2 + \frac{n^3-3n^2+2n}{6}x^3 + \frac{n^4-6n^3+11n^2-6n}{24}x^4 + \frac{n^5-10n^4+35n^3-50n^2+24n}{120}x^5 + O\left(x^6\right) \tag{50}$$

Type: UnivariatePuiseuxSeries(Expression Integer,x,0)

The command)set stream calculate n determines the number of terms explicitly displayed in the series expansion. AXIOM's default value for n is 10.

The parameter x=0 in series means that we want an expansion (valid in some region around 0) expressed in powers of x. If we had used x=v, with some other value of v, we would have obtained a series in terms of powers of (x - v) and valid in some region around v.

A *Puiseux* series is one expressed in terms of rational powers. If, as in this case, we know that integer powers will suffice, we may use taylor instead of series:

```
→taylor((1 + x)^n, x=0)
```

$$1 + n\,x + \frac{n^2-n}{2}x^2 + \frac{n^3-3n^2+2n}{6}x^3 + \frac{n^4-6n^3+11n^2-6n}{24}x^4 + \frac{n^5-10n^4+35n^3-50n^2+24n}{120}x^5 + O\left(x^6\right) \tag{51}$$

Type: UnivariateTaylorSeries(Expression Integer,x,0)

(If you would like to see AXIOM generate a Puiseux series which *isn't* a Taylor series, try expanding a root of a trigonometric or hyperbolic function.)

Only the terms displayed are initially calculated: we can, however, ask for any term we wish – or, more precisely, for the coefficient of any particular power. The next coefficient, for instance, is:

```
→%.6
```

$$\frac{n^6-15n^5+85n^4-225n^3+274n^2-120n}{720} \tag{52}$$

Type: Expression Integer

Next Previous

Up

2.7. Digression – introducing deferred assignment and functions

As mentioned in section 1.3 Assignments, a deferred assignment has the form *variable* `==` *expression* and has the effect of permanently linking *variable* to the value of *expression*.

The expression can be of any type: in the following example it is `Boolean` (that is, has possible values `true` and `false`).

```
→xPositive? == (x :: Float > 0)
```

Type: `Void`

```
→x := 17-sqrt(300);
```

Type: `AlgebraicNumber`

```
→xPositive?
```

$$false \tag{55}$$

Type: `Boolean`

```
→x := 18-sqrt(300);
```

Type: `AlgebraicNumber`

```
→xPositive?
```

$$true \tag{57}$$

Type: `Boolean`

It is conventional, in AXIOM, to end the names of Boolean objects with `?`.

When a deferred assignment (or a function) is first used in a particular domain, AXIOM it to Lisp (or possibly to machine code) so that it may be executed more quickly. This compiled code is kept as long as it remains appropriate, as we shall see shortly.

Note that we defined `xPositive?` as the value of the expression `x :: Float > 0` which means "turn `x` into a floating point number and check whether this is greater than zero" – the conversion to `Float` is a precaution in case we should, at some later stage, accidentally rely on the value of `xPositive?` when `x` is not numeric, since the symbol `>` is used to test the relative order of elements of any domain where an "order" is defined. In particular, this means that, when `x` is a variable, the expression `x > 0` always has the value `true`. Conversion to `Float` protects us against this:

```
→)clear p x
```

Compiled code for xPositive? has been cleared.

→ `x`

$$x \tag{58}$$

Type: `Variable x`

→ `xPositive?`

Cannot convert from type `Variable x` to `Float` for value

$$x$$

As `x`'s type is no longer known, the compiled code for `xPositive?` is no longer applicable and is removed.

AXIOM finds that it cannot now compile `xPositive?`, so attempts to *interpret* it without compilation. A major advantage of this is that it allows the actual cause of the problem to be pinpointed more accurately, as we see in the message

```
Cannot convert from type Variable x to Float for value
x
```

Functions may be written in AXIOM by a method very closely related to deferred assignment, the difference being that the left hand side is followed by a parenthesised list of variables, called *parameters*, which may be used in defining the expression on the right, thus:

→ `halfSum(x, y) == (x + y)/2`

Type: `Void`

→ `halfSum(1, 3)`

$$2 \tag{60}$$

Type: `Fraction Integer`

→ `halfSum(1.5, 2.5)`

$$2.0 \tag{61}$$

Type: `Float`

→ `halfSum(2, 4)`

$$3 \tag{62}$$

Type: `Fraction Integer`

AXIOM first compiled a version of `halfSum` for `PositiveIntegers` and, later, one for `Floats`; however, it still kept the first version and so did not need to recompile when we again applied the function to `PositiveIntegers`.

It is possible to write much more complicated functions in AXIOM by using a *block* for the expression on the right. A block is a sequence of expressions, separated by semicolons, the whole being enclosed in parentheses.

Suppose we are interested in the number of digits in various numbers of the form `2^n` (we shall assume that `n` is a positive integer). We can proceed as follows:

```
→f(n)==#((2^n)::String)
```

Type: `Void`

```
→f(20)
```

7 (64)

Type: `PositiveInteger`

This function works by first converting `2^n` to a string (of characters) and then applying the length function `#` to the result.

Suppose further that we are concerned with printing such numbers with a line width of, say, 120 characters or less. In the case of longer strings we are interested only in knowing that they are too long, not in the actual length

```
→f(n) == (local length; length := #((2^n)::String);
            if length > 120 then "Too long!" else length)
```

Type: `Void`

```
→f 100
```

31 (66)

Type: `NonNegativeInteger`

```
→f 1000
```

"Too long!" (67)

Type: `String`

This time, we have made the function definition into a block, to allow it to utilise a series of commands. In this way, we could first calculate the length, then proceed differently, according to whether or not the length exceeds 120. The value of a block is the value of the last statement executed – in this example, the last statement is always

```
if length > 120 then "Too long!" else length
```

which may return a string or a positive integer (as we saw at line (66), AXIOM refers to such indeterminate types as being of type `Any`). We have declared the variable `length` to be `local` – this means that it is distinct from any variable with the same name used elsewhere. The alternative to this is `free`, meaning a variable in the general AXIOM environment (usually referred to as a *global* variable). A function's parameters are always local – any other variable should always be declared as `local` or `free`. AXIOM has a set of rules for determining whether undeclared variables are local or global – but relying on these can easily lead to mistakes.

Note that we had to keep the entire definition on a single "line" by using a final `_` for continuation.

When we return a string, the quotation marks (`" "`) appear in the output from the call, as in

```
"Too long!"
```

This is because we are looking at the object. If we wish to display it as a message we can use the `output` command.

We began our discussion of the function by assuming that `n` was a positive integer. This can be built into the definition:

```
→f(n : PositiveInteger) : Any ==
          (local length; length := #((2^n)::String);
          if length > 120 then "Too long!" else length)
                                                    Type: Void
```

```
→f 0
```

Cannot convert from type `Integer` to `PositiveInteger` for value

0

We declared the type of `n` where it appeared in the parameter list. In the present version of AXIOM, if we declare any parameter's type, we must declare them all and must also declare the type to be returned, immediately after the parameter list; from our earlier experiment, we know the type `Any` will suffice for the returned type; however, as a general rule it is good practice to specify a more precise type than this. In our case, such a precise type could be `Union(PositiveInteger,String)`., You may like to try using this type in the definition and see how AXIOM displays the *branch* it has chosen in the type of the results of the function.

If a function unexpectedly fails to return anything, this usually indicates that its return type has not been specified by the user but has been set to `Void` by AXIOM – for example, because the definition ends with a call to a function (such as `print`) whose own return type is `Void`.

Finally, note that functions, like variables, may be deleted by using the system command `)clear properties`.

Depth of evaluation

At the beginning of section 2.7, we saw that `==` associates the *value* of the expression on the right with the variable or function form on the left. It is important to realise that only one evaluation takes place. Thus, if we say

```
→g1(x) == 2*x
```

Type: Void

```
→g2(x) == %
```

Type: Void

```
→G := 2*x
```

$$2x \tag{71}$$

Type: Polynomial Integer

```
→g3(x) == G
```

Type: Void

```
→g1(1)
```

$$2 \tag{73}$$

Type: PositiveInteger

we have defined three functions with quite different effects:

```
→g2(2)
```

$$2 \tag{74}$$

Type: PositiveInteger

```
→g3(3)
```

$$2x \tag{75}$$

Type: Polynomial Integer

The first evaluates `2*x` with the given value of `x`; the second evaluates `%`, giving the result of the immediately previous calculation (note that the expression `%` does not involve `x` – its *value* might, but this is irrelevant); the third evaluates `G`, whose value is always the polynomial `2*x` (again, we do not obtain the value of this polynomial).

As in most languages, AXIOM functions evaluate their arguments and this provides a means for users to define functions which return the value of the value of an expression (equivalent to `g2(2)` and `g3(3)` above returning `4` and `6`, respectively): the function called `function` takes an expression as its first argument and returns an equivalent function. As `function` is a function, it evaluates this first argument, so that `G`, say, evaluates to `2*x` at this stage. When the user's function is called, assuming that `x` is one of its arguments, this in turn will be evaluated giving, say, `2*2` so that `4` is returned.

Readers may wonder why the simple method of defining a function using `==` does not behave in this way – surely it would be friendlier if as many levels of evaluation as possible were carried out at the time of the definition, so that, in our case, `g1`, `g2` and `g3` were all equivalent. Whilst that might be helpful for simple cases like this, allowing an unevaluated expression to be used by `==` makes this type of assignment much more versatile, adding considerable power to AXIOM. To take a trivial example, suppose we are working with polynomials involving `%i`: AXIOM normally expresses these with complex coefficients – say with the type `Polynomial Complex Integer`; converting this to the type `Complex Polynomial Integer` causes the real and imaginary terms to be grouped separately. Making the definition `cpi == % :: Complex Polynomial Integer` allows us to to perform this conversion on any polynomial which occurs as a result in future, simply by typing `cpi`.

Anonymous functions

An alternative way to define a function is to use the infix operator `+->`, which defines an *anonymous function*, mapping its left hand side into its right hand side: for instance `x +-> x^2` is the square function. This can be useful when calling an AXIOM function which requires a function as one of its arguments – the argument function can be defined *in situ*:

```
→l1 := [1,2,3,4,5]
```

$$[1, 2, 3, 4, 5] \qquad (76)$$

Type: `List PositiveInteger`

```
→l2 := map(x +-> x^2,l1)
```

$$[1, 4, 9, 16, 25] \qquad (77)$$

Type: `List PositiveInteger`

Note that any variable names used in the function definition are local to the function – any variable of the same name occurring outside the function definition is ignored.

Macros

A much broader discussion of functions can be found in the AXIOM manual. A closely related topic is *macros*.

A macro in AXIOM is a rule for replacing one character string with another. It can have either of two forms:

`macro` *string1* `==` *string2*

or

string1 `==>` *string2*

and has the effect that, whenever *string1* is encountered as a separate token, *string2*is substituted for it. The phrase "as a separate token" here means that *string1* does not form part of a larger string; thus, defining `sine ==> sin` allows us to obtain a value for `sine(0)` but not for `asine(0)`, although AXIOM has functions `sin` and its inverse `asin`.

It is also possible to define macros with parameters, in the manner of functions. However, beware of regarding parameterised macros as functions: as far as AXIOM is concerned they are not and will cause an error if used where functions are required (for instance, as the first parameter of `map`).

A user defined macro with the same name as a system macro, such as `%i`, will effectively hide the system macro. User defined macros can be removed by using `)clear properties` *names*.

Next Previous

2.8. Back to series

We can now set up the binomial expansion as a function of `n`

```
→ BE(n) == taylor((1+x)^n, x=0)
```

Type: `Void`

```
→ BE(5)
```

$$1 + 5x + 10x^2 + 10x^3 + 5x^4 + x^5 \tag{79}$$

Type: `UnivariateTaylorSeries(Expression Integer,x,0)`

and use this to display the expansion for any power we wish, remembering that we have `)set stream calculate 5`:

```
→ BE(6)
```

$$1 + 6x + 15x^2 + 20x^3 + 15x^4 + 6x^5 + O\left(x^6\right) \tag{80}$$

Type: `UnivariateTaylorSeries(Expression Integer,x,0)`

AXIOM notices that the series for $(1+x)^5$ terminates after six terms.

Chapter 3 Previous

Up

Chapter 3, Introductory calculus

In introductory differential calculus, the main areas covered are techniques for obtaining the derivatives of (real) expressions involving algebraic and transcendental functions and the use of derivatives in locating maxima, minima, points of inflection and limits. Integral calculus is usually taught as a collection of techniques for obtaining the antiderivative of a function, with various applications of definite integrals to physical situations.

Chapter 2 Chapter 4

3.1. Differentiation

In AXIOM, the operator `D` is used to obtain the derivative of its first argument with respect to its second:

```
→D(x^2,x)
```

$$2x \tag{1}$$

Type: `Polynomial Integer`

```
→D(sin x,x)
```

$$\cos(x) \tag{2}$$

Type: `Expression Integer`

```
→D(sin(log(x/tan(x))),x)
```

$$\frac{\left(-x\tan(x)^2+\tan(x)-x\right)\cos\left(\log\left(\frac{x}{\tan(x)}\right)\right)}{x\tan(x)} \tag{3}$$

Type: `Expression Integer`

Higher order derivatives may be obtained by giving the order as a third argument. For instance, the second and third derivatives of $\tan x$ are:

```
→D(tan x,x,2)
```

$$2\tan(x)^3+2\tan(x) \tag{4}$$

Type: `Expression Integer`

```
→D(tan x,x,3)
```

$$6\tan(x)^4+8\tan(x)^2+2 \tag{5}$$

Type: `Expression Integer`

If the expression given by the first parameter depends on more than one variable, AXIOM calculates the partial derivative with respect to the variable identified in the second parameter. Thus, to obtain

$$\frac{\partial \sin(x\,y)}{\partial x}$$

→ `D(sin(x*y),x)`

$$y\cos(x\,y) \tag{6}$$

Type: `Expression Integer`

However, for mixed partial derivatives, we must specify a list of the variables of differentiation, in the order in which they are to be applied. For instance, to calculate

$$\frac{\partial^3 \sin x\, y}{\partial x^2 \partial y}$$

→ `D(sin(x*y),[y,x,x])`

$$-2y\sin(x\,y) - x\,y^2\cos(x\,y) \tag{7}$$

Type: `Expression Integer`

(The call `D(sin(x*y),[y,x],[1,2])` provides an alternative way of obtaining the same effect.)

We can also experiment with differentiating more complex forms, involving, for instance, the product and quotient of functions ($f(x)$ and $g(x)$, say) and their composition, $f(g(x))$. To explore this area in a general setting, we first need to meet AXIOM's representation of arbitrary functions as *operators*.

Next

3.3. Back to differentiation

Now that we know the representation of arbitrary functions as operators, we can return to demonstrating that AXIOM is able to differentiate various abstract forms. In particular, we can ask it to display the product, quotient and chain rules:

```
→f := operator 'f; g := operator 'g;
```

Type: BasicOperator

```
→D(f(x)/g(x),x)
```

$$\frac{-f(x)g'(x)+g(x)f'(x)}{g(x)^2} \tag{14}$$

Type: Expression Integer

```
→D(f(g(x)),x)
```

$$f'(g(x))g'(x) \tag{15}$$

Type: Expression Integer

At step (12), we included two commands on one line. Multiple commands are allowed in AXIOM, provided that a semicolon (;) separates each pair.

In displaying the results, AXIOM used a raised comma (') to approximate the "primed" notation for differentiation.

AXIOM can also be used for the reliable calculation of total derivatives (derivatives of functions whose arguments are themselves functions of some parameter, with respect to that parameter), which can quickly become very cumbersome by hand. For instance, suppose that we wish to transform the time-dependent locus of a point from polar to Cartesian coordinates:

```
→r := operator 'r; theta := operator 'theta ;
```

Type: BasicOperator

```
→x(t) == r(t)*cos theta t
```

Type: Void

```
→y(t) == r(t)*sin theta t
```

Type: Void

As well as omitting the parentheses in the expressions on the right of the assignments, we relied on the fact that AXIOM groups function calls to the right, so that `cos theta t` means `cos(theta(t))`.

We can now obtain the expressions for the transformed components of velocity, by taking the total derivatives:

```
→D(y(t),t)
```

$$r(t)\cos(theta(t))theta'(t) + \sin(theta(t))r'(t) \tag{20}$$

Type: Expression Integer

It is, in fact, possible to carry this through without the explicit use of functions. Instead, we shall use that technique to obtain the second derivatives, in solving a rather more interesting problem: to find expressions for the radial and transverse accelerations, in terms of the time derivatives of r and θ. We can obtain these expressions by simply finding the second time derivatives of our transformations from polar to Cartesian coordinates and then taking the case $\theta = 0$, to make the radial and transverse directions coincide with the x and y directions, respectively.

```
→)clear all
```

```
All user variables and function definitions have been cleared.
```

```
→r := operator 'r; theta := operator 'theta;
```

Type: BasicOperator

```
→r := r(t); theta := theta(t);
```

Type: Expression Integer

In the second pair of definitions we are using the operators defined in the first pair. In redefining `r` and `theta` at step (2), we lose the definitions from step (1) – but we no longer need these. Now, when we give the transformations to Cartesian coordinates, the dependence on `t` is no longer explicitly visible:

```
→x == r*cos theta; y == r*sin theta;
```

Type: Void

– in fact, as we see shortly, AXIOM regards `x` and `y`, when defined in this way, as *rules* rather than functions. We shall return to the topic of rules in section 3.7.

Having seen the form of the first derivatives in the previous example, we may suspect that the second derivatives are likely to be quite complicated and decide to inhibit their display:

```
→ax := D(x,t,2); ay := D(y,t,2);
```

Type: Expression Integer

We can now evaluate these expressions at $\theta = 0$, as explained above:

```
→eval(ax,theta=0)
```

$$r''(t) - r(t)theta'(t)^2 \tag{5}$$

Type: Expression Integer

→ `eval(ay,theta=0)`

$$r(t)theta'(t) + 2r'(t)theta'(t) \tag{6}$$

Type: Expression Integer

In a very few steps we have obtained the standard results that the radial acceleration is $\ddot{r}$ minus the *centripetal acceleration* $r\omega^2$ (where ω is $\dot{\theta}$, the time derivative of θ) and the transverse acceleration is $r\ddot{\theta}$ plus the *Coriolis acceleration* $2\dot{r}\dot{\theta}$. One of the most satisfying uses of computer algebra systems such as AXIOM is to highlight the essential steps in a calculation whilst being relieved of such mechanical tasks as keeping track of the derivatives.

One other point to observe is the notation used by AXIOM itself for partial derivatives. Retaining the previous compact notation for `r` and `theta` we have:

→ `f := operator 'f`

$$f \tag{7}$$

Type: BasicOperator

→ `D(f(r,theta),t)`

$$theta'(t)f_{,2}(r(t), theta(t)) + r'(t)f_{,1}(r(t), theta(t)) \tag{8}$$

Type: Expression Integer

in which the subscript $,i$ means differentiation with respect to the ith parameter so, in other words, we have

$$\frac{df(r,\theta)}{dt} = \frac{d\theta}{dt}\frac{\partial f}{\partial \theta} + \frac{dr}{dt}\frac{\partial f}{\partial r}$$

The second derivative is given by

→ `D(f(r,theta),t,2)`

$$theta'(t)^2 f_{,2,2}(r(t), theta(t)) + r'(t)^2 f_{,1,1}(r(t), theta(t)) + f_{,2}(r(t), theta(t))theta''(t) + f_{,1}(r(t), theta(t))r''(t) + r'(t)theta'(t)f_{,2,1}(r(t), theta(t)) + r'(t)theta'(t)f_{,1,2}(r(t), theta(t)) \tag{9}$$

Type: Expression Integer

In mixed derivatives, the order of the subscripts is the same as the order in which the differentiations are performed.

Next Previous

3.5. Mathematical experimentation – an example

AXIOM can also integrate more general expressions, involving variables other than the variable of integration. For example:

```
→I(x^n,x)
```

$$\frac{x\,e^{(n\log(x))}}{n+1} \tag{17}$$

Type: `Union(Expression Integer,List Expression Integer)`

The form of the results from AXIOM's integrations may sometimes be a little unexpected – but they have the advantage of generality, being valid (at least in the limit) for any values of the variables at which the result and the concept of an indefinite integral are defined.

With `x` restricted to well-behaved values (those not lying at the origin or on the negative real axis) and `n` a real number other than `-1`, the last result simply reduces to the familiar `x^(n+1)/(n+1)`. To investigate its behaviour when `n` is `-1`, first subtract `1/(n+1)`, which is permissible since the integral implicitly includes an arbitrary constant:

```
→% - 1/(n + 1)
```

$$\frac{x\,e^{(n\log(x))}-1}{n+1} \tag{18}$$

Type: `Expression Integer`

We cannot evaluate this if `n` is `-1` – we can, however, take its limit at that point:

```
→limit(%,n=-1)
```

$$\log(x) \tag{19}$$

Type: `Union(OrderedCompletion Expression Integer`, Record(leftHandLimit: `Union(OrderedCompletion Expression Integer,failed)`, rightHandLimit: `Union(OrderedCompletion Expression Integer,failed)`),`failed`)

so that the integral of `1/x`, which is usually introduced as a special case, is just a limiting case of the general result.

In the above development, we followed the usual mathematics textbook approach of pulling the necessary transformation out of a hat, the only justification being that it worked. One of the great advantages of computer algebra systems is that they allow us to explore and experiment with mathematical objects relatively quickly: the following reconstruction of the steps which led us to subtract `1/(n+1)` illustrates this process.

```
→In := %% 17
```

$$\frac{x\,e^{(n\log(x))}}{n+1} \tag{20}$$

Type: Union(Expression Integer,List Expression Integer)

→ limit(%,n=-1)

$$[leftHandLimit = -infinity, rightHandLimit = +infinity] \tag{21}$$

Type: Union(OrderedCompletion Expression Integer, Record(leftHandLimit: Union (OrderedCompletion Expression Integer,failed), rightHandLimit: Union(OrderedCompletion Expression Integer,failed)),failed)

Looking at the series expansion can show where the infinity comes from:

→)set stream calculate 5
black

→ series(In,n=-1) -- expand In in powers of (n+1)

$$x\,e^{(-\log(x))}(n+1)^{(-1)} + x\log(x)e^{(-\log(x))} + \frac{x\log(x)^2e^{(-\log(x))}}{2}(n+1) + \frac{x\log(x)^3e^{(-\log(x))}}{6}(n+1)^2 + \frac{x\log(x)^4e^{(-\log(x))}}{24}(n+1)^3 + \frac{x\log(x)^5e^{(-\log(x))}}{120}(n+1)^4 + O\left((n+1)^5\right) \tag{22}$$

Type: UnivariatePuiseuxSeries(Expression Integer,n,-1)

The first term of the series is the only one involving a negative power of (n + 1) and so the only one to become infinite at n = -1. For well-behaved values of x, the expression

```
- log(x)
  x %e
```

reduces to 1 and the term depends only on n. These facts suggest that subtracting the "simplified" version of this term, 1/(n + 1), as an adjustment of the constant of integration, might be a useful approach (and, in fact, we already saw that it works).

On the other hand, if we only noticed, initially, that the first term was the probable cause of the infinite limit, we might try subtracting it in the form it appeared in the series:

→ In2 := In - x*%e^(-log(x))*(n+1)^(-1)

$$\frac{x\,e^{(n\log(x))} - x\,e^{(-\log(x))}}{n+1} \tag{23}$$

Type: Expression Integer

→ limit(In2,n=-1)

$$x\log(x)e^{(-\log(x))} \tag{24}$$

Type: Union(OrderedCompletion Expression Integer, Record(leftHandLimit: Union (OrderedCompletion Expression Integer,failed), rightHandLimit: Union(OrderedCompletion Expression Integer,failed)),failed)

Now, of course, having seen that this subtraction has the desired effect we would be motivated to examine its form more closely and no doubt try the effect of the simpler expression to which it usually reduces.

Incidentally, does the same trick work with the standard result?

→ `limit(x^(n+1)/(n+1),n=-1)`

$$[leftHandLimit = -infinity, rightHandLimit = +infinity] \tag{25}$$

Type: Union(OrderedCompletion Expression Integer, Record(leftHandLimit: Union (OrderedCompletion Expression Integer,failed), rightHandLimit: Union(OrderedCompletion Expression Integer,failed)),failed)

→ `limit(x^(n+1)/(n+1)-1/(n+1),n=-1)`

$$\log(x) \tag{26}$$

Type: Union(OrderedCompletion Expression Integer, Record(leftHandLimit: Union (OrderedCompletion Expression Integer,failed), rightHandLimit: Union(OrderedCompletion Expression Integer,failed)),failed)

Yes, it does. Some texts, in fact, introduce a function ln (on the positive reals) defined as this limit, then show that it coincides with $\log_e$.

Next Previous

3.6. More about integration

In the last section's example, we were fortunate enough that AXIOM could find a single expression for the integral, regardless of the value of n. However (still restricting ourselves to the real domain), the form of the integral can sometimes depend on the values of parameters (such as `n` in that example):

```
→ I(1/(a+x^2),x)
```

$$\left[\frac{\log\left(\frac{(x^2-a)\sqrt{-a}+2a\,x}{x^2+a}\right)}{2\sqrt{-a}},\ \frac{\arctan\left(\frac{x\sqrt{a}}{a}\right)}{\sqrt{a}}\right] \tag{27}$$

Type: `Union(Expression Integer,List Expression Integer)`

AXIOM does not indicate the range of parameter values for which each form is applicable. However, the presence of the square root of `a` or `-a` makes it fairly obvious that the first form applies for `a` <0 and the second for `a` >0. The `a` $=0$ case is, again, a limit, as can be seen by taking the series expansion of, say, the second form:

```
→ series(second %, a=0)
```

$$\frac{\pi}{2}a^{\left(-\frac{1}{2}\right)}-\frac{1}{x}+\frac{1}{3x^3}a-\frac{1}{5x^5}a^2+O\left(a^{\frac{5}{2}}\right) \tag{28}$$

Type: `UnivariatePuiseuxSeries(Expression Integer,a,0)`

If the first (constant) term is subtracted from the integral, the corresponding series would begin with the term

$$\frac{-1}{x}$$

and all of the remaining terms would involve positive powers of `a`, and so vanish in the limit at `a = 0`.

Next Previous

3.7. Digression – rules

There are many relationships among mathematical functions which allow the same expression to be represented in a wide variety of ways. For instance, if we have an expression involving

$$\frac{1}{cos\,A}$$

we may prefer to represent this in terms of `sec A`. Similarly, if

$$\%e^x + \%e^{-x}$$

appears, we may decide to choose a representation in terms of `cosh x`.

AXIOM allows us to control the form of individual expressions in at least two ways – by applying operations from those packages such as `TrigonometricManipulations`, which are briefly discussed in chapter 5 The AXIOM browser, and by applying *rewrite rules* (henceforth simply called *rules*).

A rule takes the form `rule` *expression1* `==` *expression2* and indicates that expressions of the form *expression1* should be transformed into that of *expression2* Note that, if an opening parenthesis appears immediately after the word `rule`, it must be matched by a closing parenthesis *at the end of the rule*.

Here again, AXIOM gives much greater control than some other packages, which apply rules either universally or not at all: in AXIOM the user applies a rule to a specific expression; the result remains in the transformed form thereafter. The simplest way to apply a rule to a single expression is to use the rule, enclosed in parentheses, as if it were a function.

For instance, to convert the `atan` in the second component of the previous integral to `acot`:

```
→second %% 27
```

$$\frac{\arctan\left(\frac{x\sqrt{a}}{a}\right)}{\sqrt{a}} \tag{29}$$

Type: `Expression Integer`

```
→(rule atan A == acot(1/A)) %
```

$$\frac{acot\left(\frac{a}{x\sqrt{a}}\right)}{\sqrt{a}} \tag{30}$$

Type: `Expression Integer`

(We can re-rationalise this, if desired, by applying the function `ratDenom`.)

In this example, `second` was used to obtain the second component of the solution list; `%.2` would have had the same effect, as would `% 2` and the equivalent `%(2)`Analogous functions `first` and `third` exist, although, in general, it is simpler just to use whichever of the other forms you prefer.

If you wish to check that your recollection of an identity is correct, before applying it as a rule, first rewrite it in the form *expression* `= 0`; it can be shown that there is no *general* procedure for checking the validity of such an assertion – however, a trick which usually works for checking transcendental identities in AXIOM is to integrate the left hand side: if the identity is correct, the integral should be a constant (possibly zero). In awkward cases, differentiating to obtain zero can provide a way of verifying that a complicated expression is indeed constant – but this is rarely necessary. In the case of our `atan` relation we obtain zero immediately on integration, confirming the identity:

```
→I(atan x - acot(1/x),x)
```

$$0 \tag{31}$$

Type: `Union(Expression Integer,List Expression Integer)`

If a rule is to be used repeatedly, it may be assigned to a variable which can then be used like a function:

```
→atanRule := rule atan(A) == acot(1/A)
```

$$\arctan(A) == acot\left(\frac{1}{A}\right) \tag{32}$$

Type: `RewriteRule(Integer,Integer,Expression Integer)`

```
→atanRule atan x
```

$$acot\left(\frac{1}{x}\right) \tag{33}$$

Type: `Expression Integer`

Sometimes it is necessary to restrict the applicability of a rule. For example, suppose we wished to define a rule (for real quantities) which would simplify square roots of even powers. Provided that n itself is even, we can write $\sqrt{x^{2n}}$ as x^n. (If n is odd, applying this rule when x is negative will result in a negative quantity but, for reals, $\sqrt{\ }$ is defined as the positive square root.)

A rule to accomplish this could be defined as:

```
→rSimp := rule(sqrt(x^(2*(n|even? n))) == x^n)
```

$$\sqrt{x^{(2n)}} == x^n \tag{34}$$

Type: `RewriteRule(Integer,Integer,Expression Integer)`

The vertical bar may be read as "such that" and introduces a *predicate*.

We can now apply this rule:

```
→rSimp(sqrt(x^4))
```

$$x^2 \tag{35}$$

Type: Expression Integer

→ `rSimp(sqrt(x^6))`

$$\sqrt{x^6} \tag{36}$$

Type: Expression Integer

Rules in AXIOM can be surprisingly powerful: a rule applied to an expression involving an operator will be applied to the parameters of that operator and the result of the operation, as applied to the modified parameters, will form part of the final result, even if the presence of the operator is not apparent in the output form of the original expression.

For instance, in section 3.3 Back to differentiation we obtained the standard result for differentiating a product of functions. If we wanted to apply that result to specific functions, we could do so by means of rules; (as we long since `)cleared` that result, we now need to set it up again):

→ `f := operator 'f; g := operator 'g; dprod := D(f(x)*g(x),x)`

$$f(x)g'(x) + g(x)f'(x) \tag{37}$$

Type: Expression Integer

→ `(rule f x == sin x)%`

$$\sin(x)g'(x) + g(x)\cos(x) \tag{38}$$

Type: Expression Integer

→ `(rule g x == exp x)%`

$$e^x \sin(x) + \cos(x)e^x \tag{39}$$

Type: Expression Integer

AXIOM handled the differentiation of the substituted functions correctly, even though these only appeared in the output form as "primed" symbols, not explicitly showing the differential operator `D`.

We can make a collection of rules into a single object (called a *ruleset*) if we follow the word `rule` by a block of rules (indicated here by enclosing them in parentheses):

→ `(rule (f x == sin x; g x == cos x))dprod`

$$-\sin(x)^2 + \cos(x)^2 \tag{40}$$

Type: Expression Integer

A rule or ruleset can, of course, be given a name; this may then be applied to an expression:

```
→substitutions := (rule (f x == sec x; g x == csc x))
```

$$\{f(x) == \sec(x),\ g(x) == \csc(x)\} \tag{41}$$

Type: `Ruleset(Integer,Integer,Expression Integer)`

```
→substitutions dprod
```

$$\csc(x)\sec(x)\tan(x) - \cot(x)\csc(x)\sec(x) \tag{42}$$

Type: `Expression Integer`

Next Previous

Up

3.8. More integrals

With AXIOM's integrator available, it is no longer necessary to resort to the miscellaneous selection of tricks usually learnt for handling awkward integrals. It would, perhaps, be reassuring to see how well AXIOM does on some of these (classified by their possible methods of manual solution).

Logarithmic integrals

```
→I(cot x, x)
```

$$\frac{2\log\left(\frac{\sin(2x)}{\cos(2x)+1}\right) - \log\left(\frac{2}{\cos(2x)+1}\right)}{2} \tag{43}$$

Type: `Union(Expression Integer,List Expression Integer)`

The generality of AXIOM's integrator does, at times, cause it to produce rather inelegant results, such as this. However, it does provide us with the tools to simplify the expression, if we so desire. (For many applications, of course, the detailed form of the expression is unimportant – what matters is that it is correct.)

As we are dealing with real integrals, we can use the familiar transformations without compunction, when simplifying this result. There are many possible routes which we could follow, involving various logarithmic and trigonometric identities. Perhaps the most obvious starting point is to get rid of the double angles – the function `normalize` will do this:

```
→normalize %
```

$$\frac{-\log\left(\tan(x)^2+1\right) + 2\log(\tan(x))}{2} \tag{44}$$

Type: `Expression Integer`

Previous experience led to the choice of `normalize` in this case. The best way to find out what operations are available is to use the browser, described in chapter 5 The AXIOM browser and in chapter 14 of the on-line guide to AXIOM. Note that the quickest way to access the browser is to click on the return type. This will take you to the top browser page for that type.

Noticing that this involves both tan x itself and $1 + \tan^2 x$, which is, of course, $\sec^2 x$, suggests that expressing it in terms of sin and cos may be helpful. This can be achieved with `simplify`:

```
→ simplify %
```

$$\frac{2\log\left(\frac{\sin(x)}{\cos(x)}\right) - \log\left(\frac{1}{\cos(x)^2}\right)}{2} \tag{45}$$

Type: `Expression Integer`

Applying two of the standard rules for real logarithms:

```
→ (rule N*log A + M*log B == log(A^N*B^M)) %
```

$$\frac{\log\left(\sin(x)^2\right)}{2} \tag{46}$$

Type: `Expression Integer`

```
→ (rule log(A^N) == N*log A)%
```

$$\log(\sin(x)) \tag{47}$$

Type: `Expression Integer`

we end up with the usual "book form" of this result. (Note that this, like the original result, would require adjustment by a complex constant, such as log(-1), to produce a real form when $\sin(x)$ is negative.)

Next Previous

Up

3.9. Algebraic numbers in integrals

The need to factor denominators of rational functions can result in expressions for integrals which involve the roots of polynomials. For polynomials of degree greater than 4, there is no general closed form for such *algebraic numbers* and, for degree greater than 2, use of the closed form would usually result in an integral too complicated to be easily understood. AXIOM, therefore, leaves the algebraic numbers in such cases as symbols – for instance, `%%T0` in the following example

```
→ I(1/(x^3 + x + 1),x)
```

$$\left(\sqrt{\frac{-93\%\%T0^2+12}{31}} - \%\%T0\right)\log\left((62\%\%T0+31)\sqrt{\frac{-93\%\%T0^2+12}{31}} + 62\%\%T0^2 - 31\%\%T0 + 18x - 4\right) + \left(-\sqrt{\frac{-93\%\%}{31}}\right.$$

Type: Union(Expression Integer,List Expression Integer)

In the previous expression, %%T0 is a root of a polynomial – we can discover what polynomial as follows:

→ definingPolynomial %%T0

$$\frac{31\%\%T0^3 - 3\%\%T0 - 1}{31} \tag{49}$$

Type: Expression Integer

(This, of course, has three roots – however, the form of the integral is such that the particular choice of root is immaterial.) In this case we could obtain a complicated closed form using radicalSolve – however, let us instead obtain a numerical estimate of the real root (there happens to be only one):

→ outputGeneral 5

Type: Void

→ solve((numerator %% 49) :: POLY INT,0.00001)

$$[\%\%T0 = 0.41724] \tag{51}$$

Type: List Equation Polynomial Float

(The solve command will not handle expressions of such a general type as Expression Integer, so we constrained the numerator of our expression to a polynomial type. We could equally well have used the entire expression, constrained to type Fraction Polynomial Integer.)

We can, if we wish, plug this value back into the integral, using eval, which evaluates expressions when one or more variables are replaced by, possibly constant, expressions.

Although eval can find a complex numeric value for an Expression Integer, simply applying it to %% 48 results in the following expression:

$$\begin{aligned}&\left(0.5\sqrt{-0.13517} - 0.20862\right)\log\left(56.869\sqrt{-0.13517} + 18.0x - 6.1409\right)\\ &\qquad + 0.41724\log\left(9.0x + 6.1409\right)\\ &+ \left(-0.5\sqrt{-0.13517} - 0.20862\right)\log\left(-56.869\sqrt{-0.13517} + 18.0x - 6.1409\right)\end{aligned}$$

Type: Expression Float

whose form is not immediately obvious, involving as it does square roots of negative numbers. We can avoid this difficulty by changing the type of %% 48 to Expression Complex Float, which provides a form with individual complex coefficients.

→ eval(%% 48 :: EXPR COMPLEX FLOAT,%%T0= rhs first %)

$$(-0.20862 + 0.18383i)\log(18.0x - 6.1409 + 20.908i) + (-0.20862 - 0.18383i)\log(18.0x - 6.1409 - 20.908i) + 0.41724\log(9.0x + 6.1409) \tag{52}$$

Type: Expression Complex Float

This still looks more complicated than it really is and might be thought to be an intrinsically complex function of x. In fact, its imaginary part can be shown to be zero.

AXIOM has a function called complexForm which will separate the real and imaginary parts of an expression – that is, turn an Expression Complex into a Complex Expression. However, this is not applicable to an Expression Complex Float. To avoid the difficulty with the presence of Floats, we could go back to the step where they first occur and try working with rationals instead, as suggested in chapter 2:

```
→ solve((numerator %% 49) :: POLY INT,1/100000)
```

$$\left[\%\%T0 = \frac{109377}{262144}\right] \tag{53}$$

Type: List Equation Polynomial Fraction Integer

```
→ eval(%% 48,%%T0=rhs first %) :: EXPR Complex Integer
```

$$\frac{\left(i\sqrt{\frac{287955795165}{31}} - 109377\right)\log\left(\frac{7453919i\sqrt{\frac{287955795165}{31}} + 618475290624x - 210999907937}{34359738368}\right) + 218754\log\left(\frac{309237645312x + 210999907937}{34359738368}\right.}{524288}$$

Type: Expression Complex Integer

Now we *can* use complexForm:

```
→ complexForm %
```

$$\frac{-109377\log\left(\frac{95627921278110315577344x^2 - 65249114691486659641344x + 14015512395892583830752 1}{295147905179352825856}\right) + 218754\log\left(\frac{309237645312x + 2109999}{34359738368}\right.}{524288}$$

Type: Complex Expression Integer

and, since the imaginary part has vanished, we can go straight to a (real) Expression Float:

```
→ % :: EXPR Float
```

$$-0.20862\log\left(324.0x^2 - 221.07x + 474.86\right) + 0.41724\log\left(9.0x + 6.1409\right) - 0.36766\arctan\left(\frac{1.1616}{x - 0.34116}\right) \tag{56}$$

Type: Expression Float

in which ...

- the argument of the first `log` is always positive,
- rewriting the second log with the coefficient halved and the argument squared gives a form which is real wherever it is defined

  and

- `atan` is a real-valued function of reals,

so our result is real wherever it is defined.

Next Previous

3.10. Complex integration

If we wish to consider integration in the complex domain (so that the variables and any parameters are allowed to be complex) we may use the command `complexIntegrate`. This has the advantage of returning a single (complex) function, of which the various functions of the real case are special cases (possibly differing by an arbitrary complex constant, of course). The results tend to lean heavily on exponentials and logarithms – trigonometric and hyperbolic functions and their inverses can, of course, be expressed in terms of these – which may result in expressions for the integrals which seem unfamiliar, compared to the real forms.

Chapter 4 Previous

Up

Chapter 4, Ordinary differential equations

The AXIOM command has facilities for solving individual linear ordinary differential equations (ODEs) and certain nonlinear ODEs. For `solve` to be used in this way, its first parameter must be an ODE, in which the dependent variable appears as a function of the independent variable (and so must first be declared as an operator); the second and third parameters must name the dependent and independent variables, respectively.

We shall examine two simple examples.

Chapter 3 Chapter 5

Up

4.1. Simple harmonic motion

The differential equation describing simple harmonic motion is

$$\frac{d^2s}{dt^2} = -k^2s$$

which can be written in AXIOM notation as `D(s(t),t,2) = -k^2*s(t)`, so that the problem can be posed to AXIOM as follows

```
→ s := operator 's
```

$$s \tag{1}$$

Type: `BasicOperator`

```
→ solve(D(s(t),t,2) = -k^2*s(t), s, t)
```

$$[particular = 0, basis = [\cos(kt), \sin(kt)]] \tag{2}$$

Type: `Union( Record(particular: Expression Integer, basis: List Expression Integer), Expression Integer,failed)`

The general solution of an ODE is the particular solution added to an arbitrary linear combination of basis solutions, in this case giving $s = c_1\sin(k\,t) + c_2\cos(k\,t)$. If enough boundary conditions are known, these may be substituted into the general solution to produce a set of linear equations which can be solved by AXIOM. However, in the particular case of an initial value problem, in which the values of the dependent variable and its derivatives (up to the order of the equation) are known, a special form of `solve` allows for the complete specification of the problem in a single command. In this, the third parameter has the form *variable=value*, specifying the value of the independent variable at which the boundary conditions are known, and a fourth parameter is required, in the form of a list of values of the dependent variable and its derivatives at this point. In our case, suppose that, at time $t = 0$, the displacement s is A and the velocity $\dot{s}$ is 0; the problem may then be solved as follows:

```
→ solve(D(s(t),t,2) = -k^2*s(t), s, t=0, [A, 0])
```

$$A\cos(kt) \tag{3}$$

Type: `Union(Expression Integer,failed)`

Next

Up

4.2. Damped oscillations

If a damping force, proportional to velocity, is introduced into the previous equation, it becomes

$$\frac{d^2s}{dt^2} = -k^2 s - c\frac{ds}{dt}, \qquad c > 0$$

In AXIOM, an equation, like any other object, can be named by being assigned to a variable, so we can write `DE := D(s(t),t,2) = -k^2*s(t) - c*D(s(t),t)` and solve with the same initial conditions as before:

```
→DE := D(s(t),t,2) = -k^2*s(t) - c*D(s(t),t)
```

$$s''(t) = -c\,s'(t) - k^2 s(t) \tag{4}$$

Type: `Equation Expression Integer`

```
→S := solve(DE, s, t=0, [A, 0])
```

$$\frac{\left(A\sqrt{-4k^2+c^2}+A\,c\right)e^{\frac{t\sqrt{-4k^2+c^2}-ct}{2}} + \left(A\sqrt{-4k^2+c^2}-A\,c\right)e^{\frac{-t\sqrt{-4k^2+c^2}-ct}{2}}}{2\sqrt{-4k^2+c^2}} \tag{5}$$

Type: `Union(Expression Integer,failed)`

Now, provided that $c^2 > 4k^2$, both terms in the numerator are negative exponentials – the second obviously so, the first since $\sqrt{c^2-4k^2}$ is certainly less than c, and we have the standard "overdamped" situation of exponential decay.

Next Previous

Up

4.3. A simple graphical display

Given the result in the previous section, AXIOM will draw the curve of s against t for us, for a particular choice of the constants `A`, `k` and `c`. One approach is to first define an expression which only depends on `t`, by using `eval` to substitute the chosen values in `S`, then ask AXIOM to draw this, for a range of values of `t`:

```
→S1 == eval(S,[A=1,k=1,c=3])
```

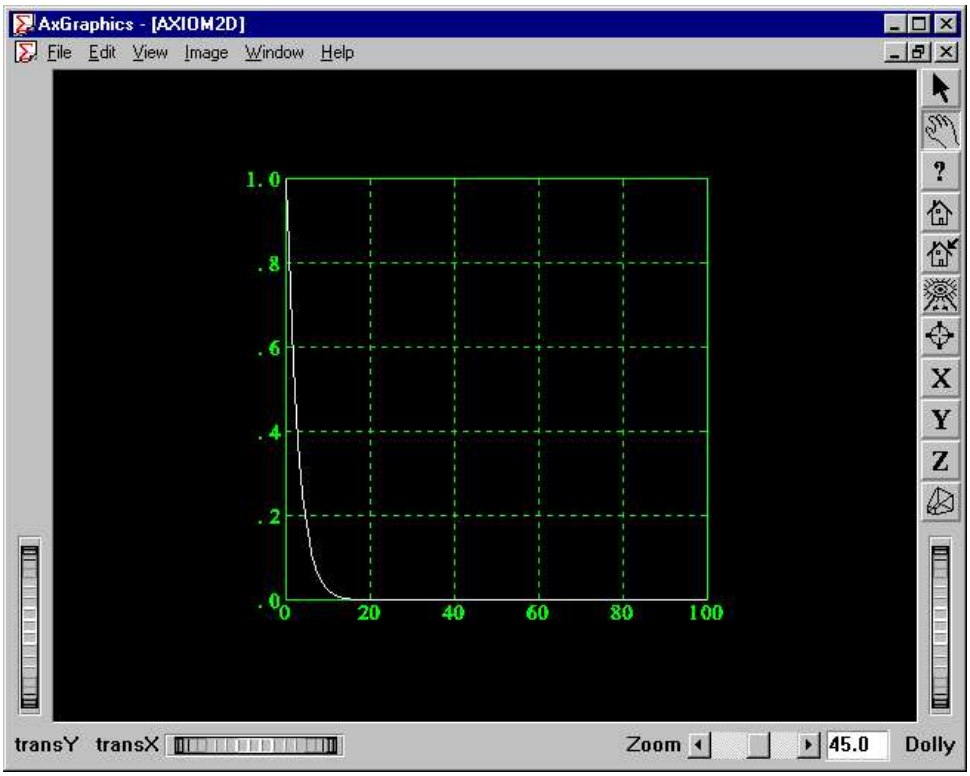
Type: `Void`

```
→draw(S1, t=0..100)
```

```
                "a twoDimensionalViewport"                    (7)
```

Type: `TwoDimensionalViewport`

This produces a window containing the graph shown in Axiom Graphics. The graph may be translated, edited and transformed using the pull-down menus and controls provided by Axiom Graphics.



Next Previous

4.4. Manipulating parts of an expression

Returning to the solution of the damped ODE:

→ `S`

$$\frac{\left(A\sqrt{-4k^2+c^2}+A\,c\right)e^{\frac{t\sqrt{-4k^2+c^2}-ct}{2}}+\left(A\sqrt{-4k^2+c^2}-A\,c\right)e^{\frac{-t\sqrt{-4k^2+c^2}-ct}{2}}}{2\sqrt{-4k^2+c^2}} \tag{8}$$

Type: `Union(Expression Integer,failed)`

if c^2-4k^2 is negative, the form of the result suggests the expansions of the sine and cosine functions in terms of exponentials:

$$\sin\theta=\frac{e^{i\theta}-e^{-i\theta}}{2i} \qquad \cos\theta=\frac{e^{i\theta}+e^{-i\theta}}{2}$$

so it may be worth trying to transform the expression for `S` into one involving `sin` and `cos`. We first have to persuade AXIOM to factor `i` out of the various square roots. To do this, we begin by finding out what AXIOM considers to be the component parts (or *kernels*) of the expression:

→ `kernels S`

$$\left[e^{\frac{t\sqrt{-4k^2+c^2}-ct}{2}},\ e^{\frac{-t\sqrt{-4k^2+c^2}-ct}{2}},\ \sqrt{-4k^2+c^2},\ c,\ A\right] \tag{9}$$

Type: `List Kernel Expression Integer`

With this information we can use `eval` to manipulate individual components in the expression:

→ `k3 := %.3`

$$\sqrt{-4k^2+c^2} \tag{10}$$

Type: `Kernel Expression Integer`

→ `eval(S,k3,%i*sqrt(4*k^2 - c^2))`

$$\frac{\left(A\sqrt{4k^2-c^2}-i\,A\,c\right)e^{\frac{it\sqrt{4k^2-c^2}-ct}{2}}+\left(A\sqrt{4k^2-c^2}+i\,A\,c\right)e^{\frac{-it\sqrt{4k^2-c^2}-ct}{2}}}{2\sqrt{4k^2-c^2}} \tag{11}$$

Type: `Expression Complex Integer`

The function `trigs` will convert from the complex exponential representation to sines and cosines

→ `ST := trigs %`

$$\frac{A\,c\,e^{\left(-\frac{c\,t}{2}\right)}\sin\left(\frac{t\sqrt{4k^2-c^2}}{2}\right)+A\,e^{\left(-\frac{c\,t}{2}\right)}\sqrt{4k^2-c^2}\cos\left(\frac{t\sqrt{4k^2-c^2}}{2}\right)}{\sqrt{4k^2-c^2}} \tag{12}$$

Type: `Expression Complex Integer`

Chapter 5 Previous

Chapter 5, The AXIOM browser

In the last chapter, we used the function `trigs` to convert an expression involving exponentials to one involving sines and cosines. AXIOM has several such functions for changing the form of expressions – but they can be quite difficult to locate. If we assume that the function we want may involve one of the strings `sin` and `trig` in its name, we can use the AXIOM, which forms part of the TEXMACS system, to track it down.

Chapter 3 of the on-line AXIOM manual provides a general introduction to TEXMACS and chapter 14 gives a fairly extensive discussion of the Browser. We shall not repeat this material here but simply explore this one example of the use of the Browser.

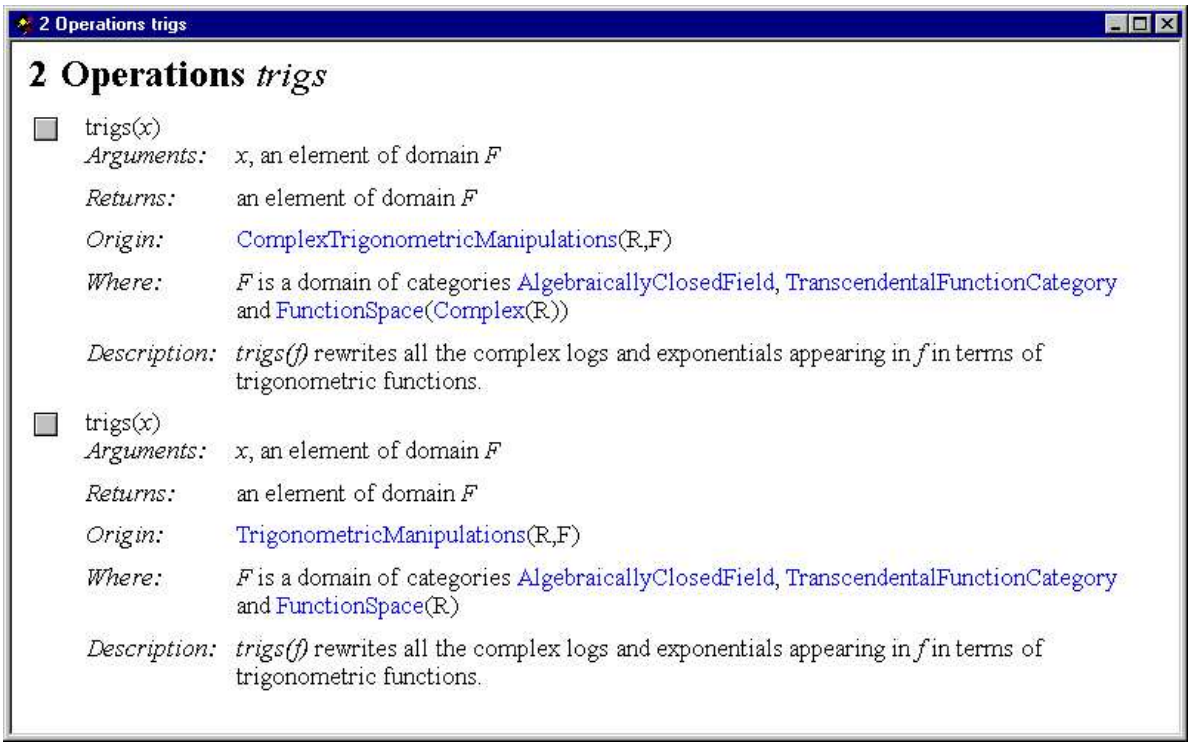
You can access the Browser from the first page that was opened when AXIOM started. Click on the button "AXIOM Browser". A new window will be generated which looks like this:

The input box contains the word `Integer` by default. Click on this and clear it using the delete key. Type `*sin* or *trig*` (the `*` means "allow any arbitrary substring here".)

Most of the Browser's functionality is accessed through a dynamic context menu under the right-hand mouse button. Try clicking the right-hand mouse button and selecting Operations with the left-hand button.

This will bring up a window which looks something like this:

This contains a list of all function (*operation*) names matching the pattern which we provided. None of those involving `sin` look very promising; some of those involving `trig` are *unexposed* – that is, not available to users without some special action – and, of the remainder, only `trigs` is not fairly obviously for some other purpose. If we click on the word `trigs`, we obtain the following window, containing a description of this function, which is indeed the one which we needed.



In entering our *search patterns* we used `*` to mean "any substring" and `or` to search for a choice of patterns. We could also have used `and` to search for names containing more than one particular substring and `not` to avoid names with a particular substring. Some limited use of parentheses is also allowed, to clarify search patterns.

Chapter 4 Chapter 6

Up

Chapter 6, Vectors and matrices

Continuing our tour of introductory mathematics brings us next to linear algebra, which is largely concerned with vectors and matrices.

Chapter 5 Chapter 7

6.1. Simple vectors

The simplest view of vectors is that they are merely linear arrays and we shall return briefly to this view later; however, most of the interesting uses of vectors apply only when the components are elements of a ring. (Roughly speaking, a ring is a set of objects on which operations analogous to addition and multiplication are defined and behave according to certain axioms.) We say the vector is defined *over* this *underlying* ring. Initially, we shall be dealing with vectors of numbers but, even here, difficulties could arise: for example, the positive integers do not form a ring, since zero is not a positive integer, breaking one of the ring axioms. When performing vector operations, AXIOM will sometimes attempt to help by treating elements of a subdomain – say `PositiveInteger` – as belonging to the full domain (`Integer`) but, for clarity, let us ensure that our first examples are vectors of ring elements:

```
→ (vecA,vecB) : Vector Integer
```

Type: `Void`

Remember that we have to use parentheses in multiple declarations – the comma (`,`), regarded as an operator, has a lower precedence than the colon (`:`).

Vectors may be created directly in AXIOM by means of the function `vector`, which takes a list as its argument:

```
→ vecA := vector [3,0,4]
```

$$[3,\ 0,\ 4] \tag{2}$$

Type: `Vector Integer`

We can add vectors of the same length:

```
→ vecB := vector [2,4,-2]
```

$$[2,\ 4,\ -2] \tag{3}$$

Type: `Vector Integer`

```
→ vecA + vecB
```

$$[5,\ 4,\ 2] \tag{4}$$

Type: `Vector Integer`

or subtract them:

```
→ vecA - vecB
```

$$[1,\ -4,\ 6] \tag{5}$$

Type: `Vector Integer`

multiply by a scalar (that is, a ring element):

→ `5*vecA`

$$[15,\ 0,\ 20] \tag{6}$$

Type: `Vector Integer`

→ `vecB*7`

$$[14,\ 28,\ -14] \tag{7}$$

Type: `Vector Integer`

but not divide by one:

→ `vecB/2`

Cannot find a definition or applicable library operation named / with argument type(s) `Vector Integer PositiveInteger`

However, if division is defined in the ring (making it a "division ring") we can multiply by the inverse of a scalar:

→ `1/2 * vecB`

$$[1,\ 2,\ -1] \tag{8}$$

Type: `Vector Fraction Integer`

→ `vecA * (1/2)`

$$\left[\frac{3}{2},\ 0,\ 2\right] \tag{9}$$

Type: `Vector Fraction Integer`

We may not, however, write this as `vecA*1/2`, since this would be interpreted as `(vecA*1)/2` which is `vecA/2`.

We can also form the inner product ("dot product") of two such vectors:

→ `dot(vecA,vecB)`

$$-2 \tag{10}$$

Type: `Integer`

The length of a vector is the square root of its inner product with itself:

```
→sqrt dot(vecA,vecA)
```

$$5 \tag{11}$$

Type: `AlgebraicNumber`

but AXIOM provides a built in `magnitude` function to do this:

```
→magnitude vecA
```

$$5 \tag{12}$$

Type: `AlgebraicNumber`

Note that the type here is `AlgebraicNumber`, since AXIOM had to compute a root to find this value.

The direction of a vector is conventionally defined by its "direction cosines", formed when its components are divided by its magnitude. Provided that we have a division ring, we can form the set of direction cosines as a vector:

```
→direction x == 1/magnitude x * x
```

Type: `Void`

```
→direction vecA
```

$$\left[\frac{3}{5}, 0, \frac{4}{5}\right] \tag{14}$$

Type: `Vector AlgebraicNumber`

The magnitude of a direction vector is always one:

```
→magnitude direction vecB
```

$$1 \tag{15}$$

Type: `AlgebraicNumber`

We can form vectors of expressions; for example, the position of a particle moving freely under gravity could be described by:

```
→p := vector [u1*t,u2*t,u3*t-1/2*g*t^2]
```

$$\left[t\,u1, t\,u2, t\,u3 - \frac{1}{2}g\,t^2\right] \tag{16}$$

Type: `Vector Polynomial Fraction Integer`

AXIOM does not provide a derivative function for vectors:

```
→D(p,t)
```
black
Cannot find a definition or applicable library operation named `D` with argument type(s) `Vector Polynomial Fraction Integer Variable t`

but we can easily define one by mapping `D` onto each component:

```
→DV(s,t) == map(x+->D(x,t),s)
```
Type: `Void`

```
→v := DV(p,t) -- the velocity,
```
$$[u1,\ u2,\ u3 - g\,t] \tag{18}$$
Type: `Vector Polynomial Fraction Integer`

```
→a := DV(v,t) -- acceleration
```
$$[0,\ 0,\ -g] \tag{19}$$
Type: `Vector Polynomial Fraction Integer`

```
→j := DV(a,t) -- and jerk
```
$$[0,\ 0,\ 0] \tag{20}$$
Type: `Vector Polynomial Fraction Integer`

While we are working abstractly, AXIOM does not generally need to know that a particular symbol represents a vector valued quantity. In the following, the symbols `s` (position), `v` (velocity), `u` (initial velocity), `aF` (acceleration due to a constant field, such as gravity) and `a` (total acceleration) will all represent vectors – but this will not be apparent to AXIOM until we assign vector values to them. In addition to the above, let `m` be the mass of our particle and suppose that the frictional force on it is `-r v`, so that (switching briefly to mathematical notation) we have $a = a_F - (r/m)v$, which we can rewrite as $a - a_F + (r/m)v = 0$.

Since a is $\ddot{s}$ and v is $\dot{s}$ our equation of motion is $\ddot{s} - a_F + (r/m)\dot{s} = 0$.

As usual, `t` will represent time; we shall assume that `s=0` at `t=0`.

Just to avoid any possible confusion, let us remove any existing definitions:

```
→)clear properties all
```
```
Compiled code for DV has been cleared.
Compiled code for direction has been cleared.
```

Now we can use the techniques of chapter 4 to integrate our equation:

```
→s := operator 's;
```

Type: BasicOperator

```
→sSol := solve(D(s(t),t,2) - aF + r/m*D(s(t),t),s,t=0,[0,u])
```

$$\frac{\left(-m\,r\,u+a\,F m^2\right)e^{\left(-\frac{r\,t}{m}\right)}+m\,r\,u+a\,F m\,r\,t-a\,F m^2}{r^2} \tag{22}$$

Type: Union(Expression Integer,failed)

(In what follows, we shall not be concerned with v, the velocity, but we could, if we wished obtain it by applying D, the derivative operator, to this result, since the expression does not explicitly refer to vectors.)

Using a two-dimensional coordinate system for simplicity, we can now let:

```
→u := vector [ux,uy]
```

$$[u\,x,\ u\,y] \tag{23}$$

Type: Vector OrderedVariableList (ux,uy)

```
→aF := vector [0,-g]
```

$$[0,\ -g] \tag{24}$$

Type: Vector Polynomial Integer

and try to turn our expression for s into a vector-valued function:

```
→function(sSol ,'s,'t)
```

$$s \tag{25}$$

Type: Symbol

AXIOM accepted that, but what has it made of it? Let us look at this function:

```
→s t
```

Cannot find a definition or applicable library operation named / with argument type(s) Vector Expression Integer Polynomial Integer

The numerator of our expression has become vector valued and, as we already discovered, we cannot divide vectors in AXIOM – but we can multiply by the inverse of a scalar, so perhaps we should handle the numerator and denominator separately. Let us try making the numerator into a function then multiplying that by the inverse of the denominator:

```
→function(numerator sSol,'n,'t)
```

$$n \tag{26}$$

Type: `Symbol`

```
→function(1/denominator sSol*n(t),'s,'t)
```

$$s \tag{27}$$

Type: `Symbol`

```
→s t
```

$$\left[\frac{-m\,u\,x\,e^{\left(-\frac{r\,t}{m}\right)}+m\,u\,x}{r},\ \frac{\left(-m\,r\,u\,y-g\,m^2\right)e^{\left(-\frac{r\,t}{m}\right)}+m\,r\,u\,y-g\,m\,r\,t+g\,m^2}{r^2}\right] \tag{28}$$

Type: `Vector Expression Integer`

(We cannot simply use `1/denominator sSol * numerator sSol` in the function definition, as AXIOM simplifies this back to the original expression).

We might, perhaps, like to see the trajectory of our particle (for particular values of the constants `m`, `ux`, `uy`, `r` and `g`). By way of illustration, let us assume that `m = 1, ux = 20, uy = 10, r = 0.1` and `g = 9.8` and, as in chapter 2, improve legibility with

```
→outputGeneral 6
```

Type: `Void`

```
→map(x+->eval(x,[m=1,ux=20,uy=10,r=0.1,g=9.8]),s t)
```

$$\left[-200.0e^{(-0.1t)}+200.0,\ -1080.0e^{(-0.1t)}-98.0t+1080.0\right] \tag{30}$$

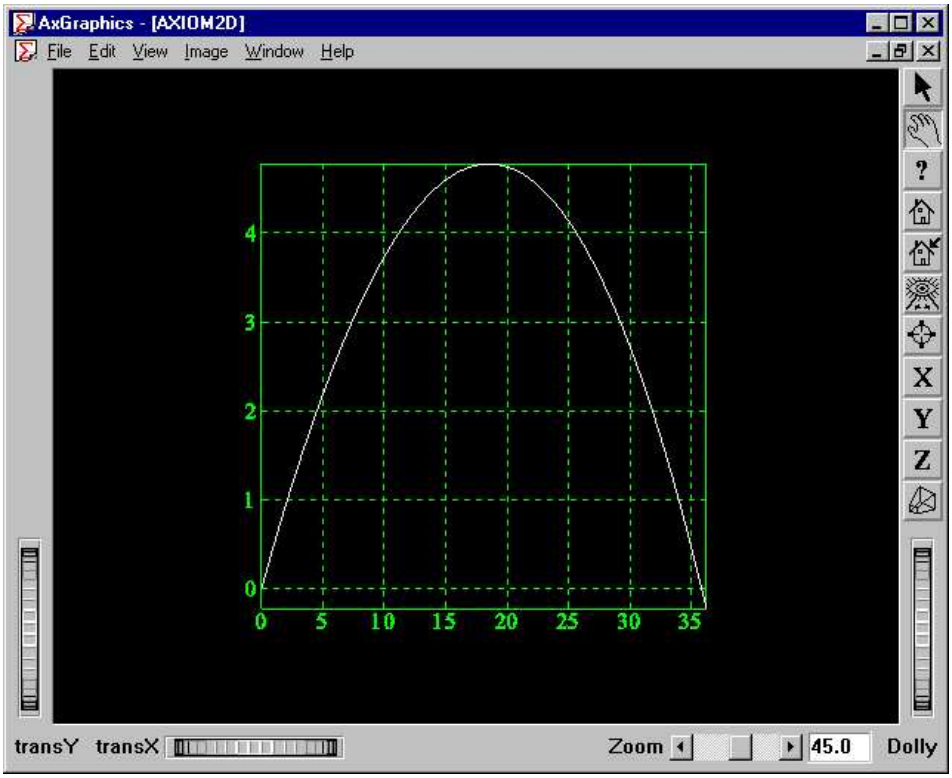
Type: `Vector Expression Float`

AXIOM provides the function `curve` as an aid to plotting parametric curves. We can use it as follows:

```
→draw(curve(%.1,%.2),t=0..2)
```

```
"a twoDimensionalViewport"                                    (31)
```

Type: `TwoDimensionalViewport`



We can compare this with the case where there is no friction:

```
→map(x+->eval(x,[m=1,ux=20,uy=10,r=0,g=9.8]),s t)
black
```

```
                                        >> Error detected within library code:
                                                    catdef: division by zero
```

We forgot the `r`s in the denominators. If we take the limit, we can get rid of those, then try again:

```
→map(x+->limit(x,r=0),s t)
```

$$\left[t\,u\,x,\ \frac{2t\,u\,y - g\,t^2}{2}\right] \tag{32}$$

Type: `Vector Union(OrderedCompletion Expression Integer, Record(leftHandLimit: Union(OrderedCompletion Expression Integer,failed), rightHandLimit: Union(OrderedCompletion Expression Integer,failed)),failed)`

With such a complicated type, AXIOM may not be able to find the operations we want; as the limits did not fail, we can simplify it considerably:

```
→map(x+->eval(x,[m=1,ux=20,uy=10,g=9.8]),%::Vector Expression Integer)
```

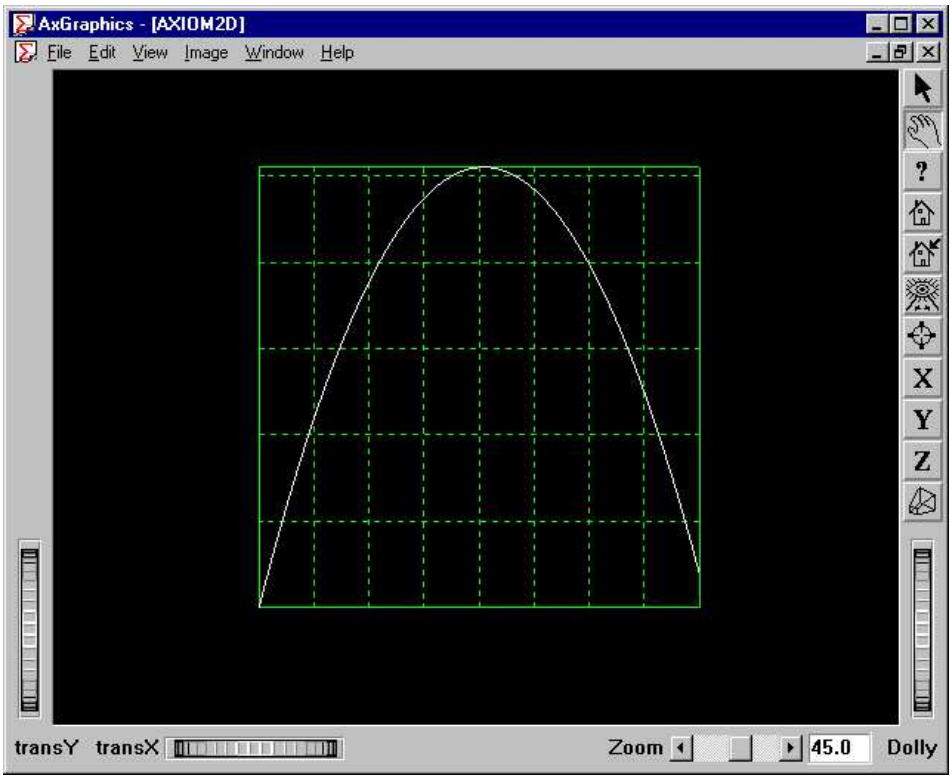
$$\left[20.0t,\ -4.9t^2 + 10.0t\right] \tag{33}$$

Type: Vector Expression Float

```
→draw(curve(%.1,%.2),t=0..2)
```

"a twoDimensionalViewport" (34)

Type: TwoDimensionalViewport



Next

6.3. Matrices (and more vectors)

The components of a vector may be of any `Type` (`Type` is the simplest category, and all domains belong to it):

```
→vecC := vector [0,x +-> x,[a,b,c]]
```

$$[0,(x \mapsto x),[a,b,c]] \tag{42}$$

Type: `Vector Any`

However, the elements of a matrix must come from a `Ring`. The function `matrix`, analogous to `vector`, may be used to create matrices directly. It takes a list of lists as its argument:

```
→matA := matrix [[x,0],[7.3,%i]]
```

$$\begin{bmatrix} x & 0.0 \\ 7.3 & i \end{bmatrix} \tag{43}$$

Type: `Matrix Polynomial Complex Float`

In forming vectors and matrices, AXIOM looks for a common type for the elements. For the vector `vecC`, the components we used were so varied that the only common type was `Any`, a special type to which any AXIOM object may belong, whilst also retaining its own, individual type.

To see this:

```
→vecC(1)
```

$$0 \tag{44}$$

Type: `NonNegativeInteger`

```
→vecC.2
```

$$x \mapsto x \tag{45}$$

Type: `AnonymousFunction`

```
→vecC 3
```

$$[a,b,c] \tag{46}$$

Type: `List OrderedVariableList` a,b,c

(Like lists and other linear structures, vectors can be *applied* to an index to yield one of their elements.)

AXIOM vectors can, in fact, serve as general data structures as well as being analogues of mathematical vectors.

If a variable has a vector or matrix type, AXIOM will convert a list or list of lists, respectively, to an object of that type, on assignment to the variable:

```
→vecD : Vector Integer := [1,2,3,4,5,6]
```

$$[1, 2, 3, 4, 5, 6] \tag{47}$$

Type: `Vector Integer`

```
→matB : Matrix Integer := [[1,2,3],[4,5,6]]
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{48}$$

Type: `Matrix Integer`

In AXIOM, the same representation is used for row and column vectors. Which of these is intended is usually discovered from the context, as we shall see shortly; where the context does not provide the necessary information (and a decision is necessary) a column vector is assumed. For example, we can coerce a vector into a matrix:

```
→vector [1,2,3] :: Matrix Integer
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{49}$$

Type: `Matrix Integer`

Turning to our original matrix, the common type of the elements was `Polynomial Complex Float`, that is, polynomials whose coefficients are complex floating point numbers. If we look at of one of its elements, we shall see that it has been converted to this type:

```
→matA(1,2)
```

$$0.0 \tag{50}$$

Type: `Polynomial Complex Float`

As matrices are two-dimensional structures, their elements are selected by applying the matrix to a pair of indices. The simplest notation for this is `M(i,j)` which returns the element located in the ith row and jth column of the matrix `M`.

For the moment, the elements of our matrices will usually be numbers of some description or variables (in the guise of polynomials) – but they could also be, for example, algebraic expressions, equations, series or some classes of operators, as well as square matrices (of a fixed size) whose elements are any of these.

Since our matrix `matA` is square, we might expect to be able to define another matrix with `matA` as its only element:

```
→matrix [[matA]]
```

$$matrix\begin{bmatrix}\begin{bmatrix} x & 0.0 \\ 7.3 & i \end{bmatrix}\end{bmatrix} \tag{51}$$

Type: `Symbol`

Instead, we obtained a `symbol` consisting of the word `matrix` with a list, containing the matrix `matA`, as a subscript. As described in section 3.2, we have supplied the function `matrix` with the wrong kind of list and obtained a subscripted symbol. We needed a list of lists of ring elements but the type of `matA` is `Matrix Polynomial Complex Float` – and arbitrary matrices do not form a ring, since they cannot, in general, be multiplied: square matrices of a fixed size are required. To obtain the desired effect, we can tell AXIOM that `matA` should be considered as a square matrix by specifying a `SquareMatrix` type for it or by applying the function `squareMatrix`.

The `SquareMatrix` type constructor takes two arguments, the first specifying the size of the matrices and the second the underlying ring. Thus we can obtain our desired effect by:

```
→matrix [[matA::SquareMatrix(2,Polynomial Complex Float)]]
```

$$\begin{bmatrix}\begin{bmatrix} x & 0.0 \\ 7.3 & i \end{bmatrix}\end{bmatrix} \tag{52}$$

Type: `Matrix SquareMatrix(2,Polynomial Complex Float)`

The `squareMatrix` function takes a single argument (the name of the matrix), so its use requires rather less effort:

```
→matrix [[squareMatrix matA]]
```

$$\begin{bmatrix}\begin{bmatrix} x & 0.0 \\ 7.3 & i \end{bmatrix}\end{bmatrix} \tag{53}$$

Type: `Matrix SquareMatrix(2,Polynomial Complex Float)`

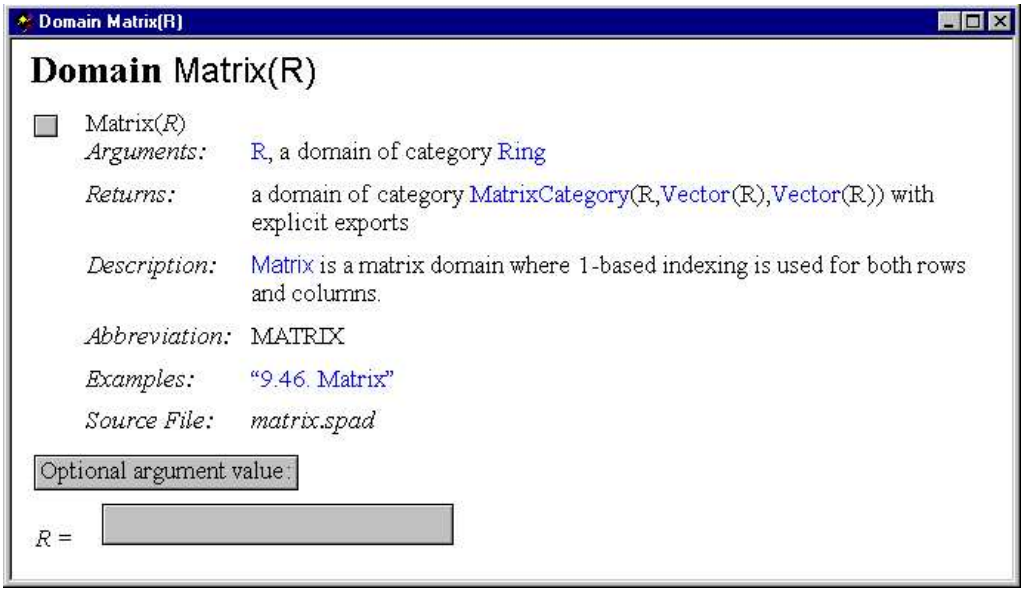
although it takes AXIOM a little longer to process, since it has to work out what kind of `SquareMatrix` you want.

Next Previous

6.4. Exploring operations on vectors and matrices

What else can we do with AXIOM vectors and matrices? One way to find out is to make use of the Browser.

Now, try typing either `vector` or `matrix` in the Browser's input box and click on `domains`. The new window which appears in the `matrix` case looks like this:

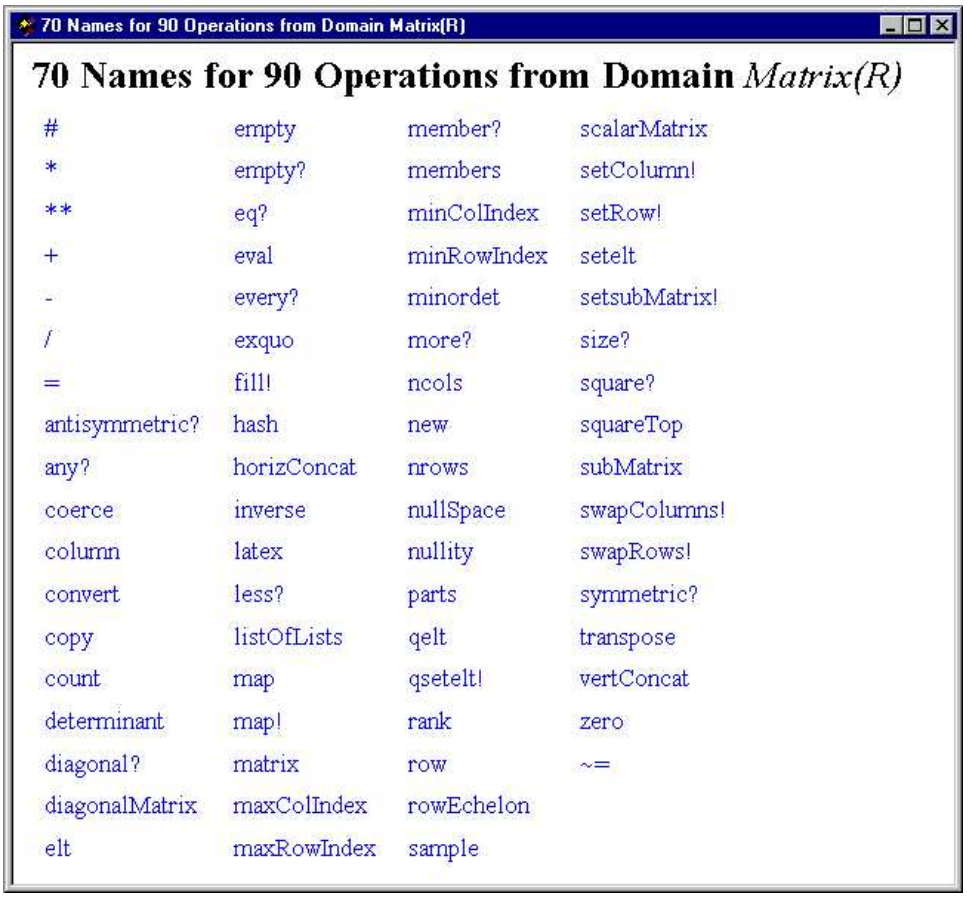


We could specialise to matrices with elements from a particular ring (say `Integer`) by typing the name of the ring in the input box of this window; however, for the moment, let us find out about all kinds of matrices.

To discover the names of all functions which come from the matrix domain (are *exported* by it), click on `Operations` in the right-hand mouse button menu.

The meanings of some of these functions are obvious, others less so – detailed descriptions can be obtained by clicking on the particular function name.

In the descriptions of the functions, you will find frequent references to "an element of domain %" and "an element of domain R" – what are these? % is the standard AXIOM syntax for "the type being defined"; when this appears in the Browser it may be thought of as shorthand for "the present domain", so "an element of domain %" is just a matrix (of whatever specific type we may be considering). One clue to R is in the title bar of the Browser window, which mentions "*Matrix(R)*": any matrix is a matrix of ring elements and R is simply the underlying ring. Thus, if we are considering, say, objects of type `Matrix Polynomial Integer` – which could also be written `Matrix(Polynomial Integer)` – the R in *Matrix(R)* means `Polynomial Integer`.



We should, however, note that for some operations the R is not necessarily just a ring – to find out exactly what it is, we may need to look more carefully at the description of the operation. For example, the second argument of the operation `exquo` is "an element of domain R" and the description includes the line

Conditions: R has `IntegralDomain`

so that the divisor must belong to an *integral domain* – that is, a ring in which the product of non-zero elements is never zero.

To take a more commonplace example of matrix functions, clicking on `*` reveals that six different product operations are defined:

matrix multiplication of pairs of matrices (with compatible dimensions),

left and right scalar multiplication (multiplication on the left or right of every element of the matrix by a *scalar* – that is, a member of the underlying ring),

left multiplication by an integer,

and

left and right multiplication by an appropriate vector.

Multiplication by an integer may differ from multiplication by a scalar, since the scalars may not be numbers – they could, for example, be 2×2 matrices. Multiplication by the integer n may be thought of as equivalent to addition of n copies of the matrix. Addition, of course, is defined by application of the `+` operation of the underlying ring to corresponding elements of two matrices.

Looking at vector-matrix multiplication provides an example of AXIOM using context to decide whether a vector is a column- or row-vector,

```
→matC := matrix [[a,b,c],[d,e,f]]
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \tag{54}$$

Type: `Matrix Polynomial Integer`

```
→lvec := vector [2,3]
```

$$[2,\ 3] \tag{55}$$

Type: `Vector PositiveInteger`

```
→rvec := vector [4,5,6]
```

$$[4,\ 5,\ 6] \tag{56}$$

Type: `Vector PositiveInteger`

```
→lvec * matC
```

$$[3d+2a,\ 3e+2b,\ 3f+2c] \tag{57}$$

Type: `Vector Polynomial Integer`

→`matC * rvec`

$$[6c+5b+4a,\ 6f+5e+4d] \tag{58}$$

Type: `Vector Polynomial Integer`

so `lvec` was taken to be a row vector and `rvec` a column vector.

In fact, it is even possible for the same vector to be treated as a column- and a row-vector in different parts of the same expression:

→`lrvec := vector [1,2]`

$$[1,\ 2] \tag{59}$$

Type: `Vector PositiveInteger`

→`lrvec * ((matrix [[a,b],[c,d]] * lrvec) :: Matrix Polynomial Integer)`

$$[4d+2c+2b+a] \tag{60}$$

Type: `Vector Polynomial Integer`

We coerced the result of the right hand multiplication to be a (column) matrix because AXIOM does not apply matrix multiplication to a product of vectors since, in that case, it would have no means of deciding which of the vectors was a row and which a column.

The `Matrix` domain is by no means the only source of functions applicable to matrices in AXIOM. A Browser search with the search string `*matrix*` reveals thirteen packages and eight other domains concerned with some aspect of matrices; several other packages may be discovered by using the string `*eigen*`.

Next Previous

6.5. Other ways of defining vectors and matrices

(Although this section refers to vectors and matrices, the same techniques may be used to define other structures such as lists and arrays.)

Using new

The AXIOM function is a useful way of defining structures where most of the elements have the same value. It has the form `new(`*size*`,`*value*`)` for one-dimensional structures and `new(`*column-size*`,`*rowsize*`,`*value*`)` for two-dimensional structures.

When we use `new`, AXIOM needs to know what kind of object we are trying to create, otherwise it will, justifiably, complain. The most direct way of doing this is to declare the type of the new object before assigning to it. As `vecD` has already been declared to be of type `Vector Integer`, we can, for example, define a vector of five zeros using:

```
→vecD := new(5,0)
```

$$[0, 0, 0, 0, 0] \tag{61}$$

Type: `Vector Integer`

Suppose, however, that we wanted to use a structure defined by `new` without assigning it to a variable. (It might, for example, be part of a larger expression of a totally different type.)

When functions of the same name exist for different domains, AXIOM provides a means to distinguish the one we want: we simply have to add a dollar sign (`$`) followed by the name of the appropriate domain, after the function call.

Thus, we could set up a 3×3 zero matrix, considered to be a matrix of integers, by specifying that we want to use the version of `new` which comes from the domain `Matrix Integer`:

```
→new(3,3,0)$Matrix Integer
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{62}$$

Type: `Matrix Integer`

This technique is called a *package call*. The effect of the `$` is to specify the source of the function, *not* the type of the result.

Why "package"? Recall that a package in AXIOM is a collection of related functions; another way of thinking of it is as a domain which does not define any types. Thus, package calling is a way of specifying the domain *or package* which we wish to provide the function.

Suppose, now, that we want to define a unit matrix of order 3 (that is, a 3×3 matrix with `1`s on the principal diagonal, `0`s elsewhere). One approach is to modify the elements of the previous matrix, one by one:

```
→Z3 := %;
```

Type: `Matrix Integer`

```
→I3 := Z3; I3(1,1) := 1; I3(2,2) := 1; I3(3,3) := 1;
```

Type: `PositiveInteger`

```
→I3
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{65}$$

Type: `Matrix Integer`

That works, but soon becomes tiresome. Does AXIOM provide a loop mechanism for cases like this?

Next Previous

6.6. Digression – introducing for loops and segments

AXIOM has a variety of looping constructs, which are extensively described in the AXIOM manual; perhaps the simplest to use in defining our unit matrix is a `for` loop:

```
→I3 := Z3; for k in [1,2,3] repeat I3(k,k) := 1
```

Type: `Void`

```
→I3
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{67}$$

Type: `Matrix Integer`

Note that the loop variable (`k` here), as in most modern computer languages, is *local* to the loop – it is distinct from any variable of the same name occurring outside the loop and only has a value within the loop.

As an alternative to the list `[1,2,3]` we could have used a *segment*, written as verb+1..3+, to control the loop:

```
→I3 := Z3; for k in 1..3 repeat I3(k,k) := 1
```

Type: `Void`

```
→I3
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{69}$$

Type: `Matrix Integer`

In this case, that was only slightly easier – but it is obviously much less work to use a segment when a long sequence of numbers is required.

Segments do not always consist of finite sequences of consecutive positive integers – they may be generalised in several ways:

> either endpoint may be negative;
>
> a step other than 1 between (real) elements may be specified by adding the clause `by` n (where n is an integer); for example, `1..9 by 2` consists of the odd, single digit integers;
>
> omitting the second endpoint produces an open-ended sequence, called a `UniversalSegment`.
>
> For example: `1..` are the natural numbers.

finally, it is possible to define segments of any AXIOM type; however, only segments of integers may be used in loop controls.

Note that the `by` n clause may be applied to any kind of segment. It may be regarded as a *filter*, whose effect is to take every nth element of the segment, starting at the first.

It is possible to convert a finite segment (of a real numeric type) into a list, using the `expand` function:

```
→expand(-7/2..7/3)
```

$$\left[-\frac{7}{2}, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}\right] \tag{70}$$

Type: `List Fraction Integer`

Segments with `by 0` generate an error.

Experimenting with `expand` will reveal that final element produced from a segment is the largest number which is less than or equal to the second endpoint and which can be formed by adding zero or more copies of the step to the first endpoint (and so, of course, the final element is not always equal to the second endpoint, as in the above example).

Applying `expand` to a `UniversalSegment` produces a *stream*, the open-ended analogue of a list.

AXIOM provides two function, `lo` and `hi`, which deliver the endpoints of segments; `lo` returns the first endpoint of any segment and `hi` the second endpoint of a finite segment (even if the first endpoint is larger than the second).

In the above examples, the *loop body*, which follows `repeat`, consisted of a single instruction; it could equally well have been a block:

```
→for i in 11..20 repeat
       ( print i;
              if prime? i then messagePrint(" (That was prime.)")$OutputForm )
```

11

(That was prime.)

12

13

(That was prime.)

14

15

16

17

(That was prime.)

18

19

(That was prime.)

20

Type: Void

Using messagePrint, rather than print, allows us to avoid printing the annoying quotation marks (" ") enclosing the string; previously, in the example in section 2.7, we could not achieve this. The messagePrint function comes from the domain OutputForm which is normally unexposed – that is, its functions are not immediately available to users. It is possible to explicitly)expose domains but this can have undesirable side-effects: for example, in the present case, we do not want the rules which govern the printing of output objects to be generally applicable, since they are not appropriate for manipulating algebraic expressions. Package calling function avoids the need to expose the domain. (For a fuller discussion of exposure see the AXIOM manual.)

It is possible to nest loops, by using a second loop as the body of the first; more unusual is AXIOM's ability to loop over two (or more) structures in parallel, by using several *iterators* (in our case, for clauses) before repeat and the loop body:

```
→poly := 0;
```

Type: NonNegativeInteger

```
→for i in [1,2,3,4] for c in ['a,'b,'c,'d] repeat poly := poly + i*c
```

Type: Void

```
→poly
```

$$4d + 3c + 2b + a \tag{74}$$

Type: Polynomial Integer

The iterators do not need to produce the same number of items: the loop will terminate when any one of them is exhausted.

This construct provides one motivation for the presence of the repeat keyword in the for loop – it shows where a set of parallel iterators ends, distinguishing parallel from nested iteration.

In defining a for loop, alternatives to a list or a finite segment after the in keyword are an open-ended segment and a stream. Loops using these constructs are potentially non-terminating.

Next Previous

6.7. More matrix definition

Direct iteration

Lists, including those from which vectors and matrices are normally defined, may be created by enclosing an expression, followed by iterators defining its variables, within list brackets. For example:

```
→vecC := vector [n^2 for n in 1..3]
```

$$[1, 4, 9] \tag{75}$$

Type: `Vector PositiveInteger`

A notorious matrix in numerical analysis is the Hilbert matrix, a slight variant of which has elements defined by the reciprocal of the sum of their row and column indices. This provides an example of an iterator occurring outside of an inner list, so that it defines a list of lists:

```
→hilbert3 := matrix [[1/(i+j) for i in 1..3] for j in 1..3]
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix} \tag{76}$$

Type: `Matrix Fraction Integer`

We can now define our unit matrix by combining this style of *direct iteration* with, say, an anonymous function:

```
→I3 := matrix [[((m,n)+->if m=n then 1 else 0)(i,j) _
                                                for i in 1..3] for j in 1..3]
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{77}$$

Type: `Matrix Integer`

For clarity, the variables used in the iterators were different to the local, dummy variables in the anonymous function. As the variables in the iterators, like those in loops, are also local, we could, if we wished, have used the same pair of variables in both contexts. A single equals sign (=) is used in tests for equality.

The diagonalMatrix function

We have used unit matrices as a way of exploring various AXIOM features; however, it is worth pointing out that there is a much simpler way of defining them, with the `diagonalMatrix` function:

→ `diagonalMatrix [1,1,1]`

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{78}$$

Type: `Matrix Integer`

Next Previous

6.8. Manipulating matrices

As we have already seen, there are many matrix functions in AXIOM. These include the standard operations encountered in any basic course on matrices, which are described in chapter 9 of the AXIOM manual. We shall explore only a few of these here.

The inverse

Many square matrices have an inverse – that is, a matrix which, multiplied on either side by the original, results in a unit matrix with the same dimensions. When the underlying ring is a *field* – that is, if division by non-zero elements is defined and multiplication is commutative – AXIOM provides an `inverse` operation. For instance, considering the Hilbert-like matrix of order three, which we defined earlier, we can obtain its inverse by:

```
→inverse hilbert3
```

$$\begin{bmatrix} 72 & -240 & 180 \\ -240 & 900 & -720 \\ 180 & -720 & 600 \end{bmatrix} \tag{79}$$

Type: `Union(Matrix Fraction Integer,failed)`

and multiplying this by its inverse does, indeed, give a unit matrix:

```
→% * hilbert3
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{80}$$

Type: `Matrix Fraction Integer`

AXIOM can also find the inverse of a symbolic matrix. To take a very simple example:

```
→matC := matrix [[a,b],[c,d]]
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{81}$$

Type: `Matrix Polynomial Integer`

```
→inverse matC
```

$$\begin{bmatrix} \dfrac{d}{a\,d-b\,c} & -\dfrac{b}{a\,d-b\,c} \\ -\dfrac{c}{a\,d-b\,c} & \dfrac{a}{a\,d-b\,c} \end{bmatrix} \tag{82}$$

Type: `Union(Matrix Fraction Polynomial Integer,failed)`

Clearly, the elements of this inverse exist only if `(a d - b c)` is non-zero. This quantity, of course, is the determinant of the original matrix, so, for the 2×2 case, we have verified that the inverse of a matrix exists precisely when its determinant is non-zero; in other words, when the matrix is *non-singular*.

The determinant

AXIOM provides a `determinant` function:

```
→determinant matC
```

$$a\,d - b\,c \tag{83}$$

Type: `Polynomial Integer`

which we shall use extensively in the next chapter to investigate the accuracy of some floating point calculations.

Here, we shall explore the use of the determinant in determining the equation of known form which passes through a set of points. If we can write the general equation in a form which is linear in a set of unknown coefficients, the form we require is obtained by equating to zero the determinant of the matrix formed as follows: [1]

> the first row consists of a list of the new coefficients obtained by considering the unknown coefficients in the equation as the variables; thus, its elements are generally functions of the actual variables in the equation;
>
> the other rows of the matrix are formed by substituting the values of the coordinates at the known points into the expressions forming the first row.

A simple example may make this clearer: suppose we wish to determine the equation of a circle which passes through the three points $(1, 0)$, $(0, 1)$ and $(-1, -1)$. If we write the equation of a circle in its general form

```
→(x-a)^2 + (y-b)^2 - r^2 = 0
```

$$y^2 - 2b\,y + x^2 - 2a\,x - r^2 + b^2 + a^2 = 0 \tag{84}$$

Type: `Equation Polynomial Integer`

we find that AXIOM has conveniently expanded it for us into a form which is linear in a set of unknown coefficients, since, collecting terms, we could rewrite this as

$$A + B\,x + C\,y + (x^2 + y^2) = 0$$

For simplicity in referencing its rows, we will build up our matrix as a list of lists.

```
→row1 := [1,x,y,x^2+y^2]
```

$$[1,\ x,\ y,\ y^2 + x^2] \tag{85}$$

Type: `List Polynomial Integer`

1. For more detail see L E Fuller, Linear Algebra with applications, Dickenson, 1966.

and the others are formed by substituting the appropriate values in this:

```
→row2 := [eval(row1.i,[x,y],[0,1]) for i in 1..4]
```

$$[1,\ 0,\ 1,\ 1] \tag{86}$$

Type: `List Polynomial Integer`

```
→row3 := [eval(row1.i,[x,y],[1,0]) for i in 1..4];
```

Type: `List Polynomial Integer`

```
→row4 := [eval(row1.i,[x,y],[-1,-1]) for i in 1..4];
```

Type: `List Polynomial Integer`

We can now build up our matrix and obtain the desired equation from its determinant:

```
→determinant [row1,row2,row3,row4] = 0
```

$$3y^2 + y + 3x^2 + x - 4 = 0 \tag{89}$$

Type: `Equation Polynomial Integer`

A new form of `eval` was used in instructions `(86)` to `(88)`; this has a list of variables as its second parameter and a list of values to be substituted for these as its third.

Solving matrix equations

If `A` is a matrix of coefficients, `x` is a vector of variables and `c` is a vector of constants, the equation `Ax = c` is equivalent to a set of linear equations of the form

$$\sum_j a_{ij} x_j = c_i$$

Can AXIOM solve the equations in matrix form?

Not all forms of `solve` work for matrices, although if we write the equation as `Ax - c = 0` and use the "left hand side" form of `solve`, as in `solve(Ax - c)`, it will produce a solution.

However, it is not necessary to mention `x` to obtain the solution of such a set of equations: `solve` has another form, `solve(A,c)`, designed specifically for matrix equations. In using this, it is not even necessary to explicitly give matrices and vectors as the parameters – appropriate lists will suffice. Thus, for example, we could solve an equation using our matrix `hilbert3`, say

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} 3 & 5 & 7 \end{bmatrix}$$

as follows:

```
→solve([[1/(i+j) for i in 1..3] for j in 1..3],[3,5,7])
```

$$[particular = [276,\ -1260,\ 1140],\ basis = [[0,\ 0,\ 0]]] \tag{90}$$

Type: Record(particular: Union(Vector Fraction Integer,failed), basis: List Vector Fraction Integer)

The result is provided as a particular solution and a basis, in case the set of linear equations is *deficient* – that is, has fewer independent equations than the number of variables: in this case, any linear combination of the basis vectors (represented here as lists) may be added to the particular solution to give a valid solution. An example of such a system is provided by the matrix

```
→matD := matrix [[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7]]
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix} \tag{91}$$

Type: Matrix Integer

and the right hand side vector

```
→c := vector [5,6,7,8]
```

$$[5,\ 6,\ 7,\ 8] \tag{92}$$

Type: Vector PositiveInteger

```
→solve(matD,c)
```

$$[particular = [-3,\ 4,\ 0,\ 0],\ basis = [[1,\ -2,\ 1,\ 0],\ [2,\ -3,\ 0,\ 1]]] \tag{93}$$

Type: Record(particular: Union(Vector Fraction Integer,failed), basis: List Vector Fraction Integer)

A set of equations is dependent if one of them can be formed from a linear combination of the others. In matrix terms, this means that the rows of the *augmented matrix*, formed by horizontally concatenating the coefficient matrix and the right hand side vector, are dependent. The augmented matrix may be formed using the horizConcat function, which operates on two matrices or equivalent structures:

```
→horizConcat(matD,c)
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \tag{94}$$

Type: Matrix Integer

(There is also a vertConcat.)

The number of rows which are independent is the *rank* of the matrix:

→ `rank %`

$$2 \tag{95}$$

Type: `PositiveInteger`

and in this case, this is the same as the rank of `matD`:

→ `rank matD`

$$2 \tag{96}$$

Type: `PositiveInteger`

The matrices used in `solve` do not need to be square. For instance, the set of equations in the last example would be essentially the same if one or two of them were omitted:

→ `solve([[2,3,4,5],[3,4,5,6],[4,5,6,7]],[5,6,7])`

$$[particular = [-2,\ 3,\ 0,\ 0],\ basis = [[1,\ -2,\ 1,\ 0],\ [2,\ -3,\ 0,\ 1]]] \tag{97}$$

Type: Record(particular: `Union(Vector Fraction Integer,failed)`, basis: `List Vector Fraction Integer`)

(The submatrix used in the above command could also have been formed from `matD`, using the `subMatrix` function:

→ `subMatrix(matD,1,3,1,4)`

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix} \tag{98}$$

Type: `Matrix Integer`

whose parameters are the name of the original matrix, the first and last row indices and the first and last column indices required, in that order. The subvector, of course, can be constructed using the form `[c.i for i in 1..3]`.)

If there are more independent rows than the number of variables, the system is said to be *overdetermined* and has no solution. Overdetermination is an example of *inconsistency*. If the coefficient vector of one of the equations can be formed from a linear combination of some of the others, for consistency the right hand sides must stand in the same relationship. If they do not, the rank of the augmented matrix will be greater than that of the coefficient matrix (in fact, it will be one greater).

For example, if we change one of the elements of `c`, the rank of the augmented matrix becomes 3:

→ `rank horizConcat(matD,vector [5,6,7,9])`

$$3 \tag{99}$$

Type: PositiveInteger

→ `solve(matD,[5,6,7,9])`

$$[particular = \texttt{"failed"},\ basis = [[1,\ -2,\ 1,\ 0],\ [2,\ -3,\ 0,\ 1]]] \tag{100}$$

Type: Record(particular: `Union(Vector Fraction Integer,failed)`, basis: `List Vector Fraction Integer`)

With inconsistent equations, this form of `solve` returns `"failed"` for the particular solution. The `solve(Ax - c)` form would return an empty list.

Chapter 7 Previous

Chapter 7, A little error analysis

Why were the Hilbert-like matrices described in the previous chapter as "notorious in numerical analysis"? Let us explore what happens if we represent them using floating point numbers. In fact, we can simulate the most common situation in numerical analysis and use the double precision floating point numbers provided by the computer hardware.

This representation of floating point numbers is obtained in AXIOM by means of the `DoubleFloat` type. Note that AXIOM displays these numbers with 16 decimal digits, the last of which is unreliable. (As is the case with any other computer program using hardware floating-point numbers.)

```
→hilbert3 :: Matrix DoubleFloat -- continuing the previous session
```

$$\begin{bmatrix} 0.5 & 0.3333333333333333 & 0.25 \\ 0.3333333333333333 & 0.25 & 0.2 \\ 0.25 & 0.2 & 0.1666666666666666 \end{bmatrix} \tag{101}$$

Type: `Matrix DoubleFloat`

```
→% * inverse %
```

$$\begin{bmatrix} 0.9999999999999928 & -2.8421709430404e-014 & 0.0 \\ -1.4210854715202e-014 & 1.000000000000028 & -2.8421709430404e-014 \\ -7.105427357601001e-015 & -1.4210854715202e-014 & 1.0 \end{bmatrix} \tag{102}$$

Type: `Matrix DoubleFloat`

Instead of being 0, the off-diagonal elements are numbers of order of magnitude 10^{-14}; the diagonal elements approximate 1 with an accuracy of only 14 digits. This does not indicate an error in AXIOM's working; it is a consequence of the inherent inaccuracy of floating point computation, which is rapidly compounded in this calculation.

As the dimension of our matrix increases, the situation gets worse: for a 7×7 matrix only about half of the digits in the diagonal elements of the final product are accurate, for the 11×11 case, none of these digits is reliable. (As a space saving measure in this chapter, the displays of a number of results are inhibited by using a final semicolon (`;`) to terminate the input.)

```
→matrix [[1/(i+j) for i in 1..11] for j in 1..11]::Matrix DoubleFloat;
```

Type: `Matrix DoubleFloat`

```
→badUnit := % * inverse %;
```

Type: `Matrix DoubleFloat`

```
→diagEls := set [%(i,i) for i in 1..11];
```

Type: `Set DoubleFloat`

```
→min diagEls
```

$$0.822265625 \qquad (106)$$

Type: DoubleFloat

```
→max diagEls
```

$$1.152587890625 \qquad (107)$$

Type: DoubleFloat

The `min` functions operate on sets so, to determine the range of values in the diagonal, we collected these in a list and then converted it to a set, using the function `set`.

The off-diagonal elements are no longer closer to zero:

```
→offDiags := empty()$Set DoubleFloat
```

$$\{\} \qquad (108)$$

Type: Set DoubleFloat

```
→for i in 1..11 repeat _
                                        for j in 1..11 | i ~= j repeat _
                                  offDiags := union(offDiags,badUnit(i,j))
```

Type: Void

```
→min offDiags
```

$$-0.35546875 \qquad (110)$$

Type: DoubleFloat

```
→max offDiags
```

$$0.291015625 \qquad (111)$$

Type: DoubleFloat

Here, nested iteration was required so we used a `for` loop.

In constructing the set of off-diagonal elements we used a new form of `for` loop control, `for j in 1..11 | i ~= j`, in which the vertical bar (`|`) serves to introduce a filter to be applied to the preceding `for` iterator; in other words, it selects only those elements produced by the iterator which satisfy the condition following the `|`, which, in this case, is the condition that `i` is not equal (`~=`) to `j`. As in rules, the vertical bar is usually read as "such that".

The normal approach in computer algebra, of course, is to use exact representations – in this case rational numbers. Returning to this form, we can verify that AXIOM still computes the solution exactly:

```
→hilbert11 := matrix [[1/(i+j) for i in 1..11] for j in 1..11];
                                                    Type: Matrix Fraction Integer
```

```
→% * inverse %
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{113}$$

Type: Matrix Fraction Integer

7.1 The effect of the determinant

7.2 Perturbation analysis

Chapter 6 Chapter 8

Up

7.1. The effect of the determinant

Recalling that the determinant of a matrix appears in the denominators of the elements of its inverse, it may be informative to look more closely at this. The determinant of our matrix `hilbert3` is:

```
→detHilbert3 := determinant hilbert3
```

$$\frac{1}{43200} \tag{114}$$

Type: `Fraction Integer`

which is rather small. The inverse of a matrix only exists when the determinant is non-zero; in numerical analysis, calculations involving matrices whose determinants are close to zero tend to be inherently inaccurate – the matrices are said to be *ill-conditioned*.

The size of the determinant of Hilbert-like matrices shrinks very rapidly with increasing dimension. For the 11×11 case:

```
→detHilbert11 := determinant hilbert11
```

$$\frac{1}{2336567487454523809227262770311059420389631662424544935673856000000000} \tag{115}$$

Type: `Fraction Integer`

If we convert this to a `DoubleFloat`, we can compare it with the determinant of the `DoubleFloat` representation of the matrix:

```
→% :: DoubleFloat
```

$$4.279782224862714e-071 \tag{116}$$

Type: `DoubleFloat`

```
→determinant(hilbert11::Matrix DoubleFloat)
```

$$4.262628689674844e-071 \tag{117}$$

Type: `DoubleFloat`

so working with `DoubleFloats` throughout gives a determinant with only two significant figures of accuracy. As the determinant is intimately involved in the calculation of the inverse, it is not surprising that using `DoubleFloats` gave such a poor result for the inverse.

Next

7.2. Perturbation analysis

If we convert our matrix `hilbert3` to a matrix of polynomials we can add a symbolic perturbation to one of the elements and see how it affects the determinant.

Let us use a relative perturbation of `eps`:

```
→test3 := hilbert3 :: Matrix Polynomial Fraction Integer;
                                    Type: Matrix Polynomial Fraction Integer
```

```
→test3(1,1) := (1 + eps)/2;
                                    Type: Polynomial Fraction Integer
```

```
→determinant test3
```

$$\frac{1}{1200}eps + \frac{1}{43200} \tag{120}$$

Type: `Polynomial Fraction Integer`

```
→(% - detHilbert3)/detHilbert3
```

$$36eps \tag{121}$$

Type: `Polynomial Fraction Integer`

so the relative change in the determinant is 36 times that in the first element of the matrix. Inaccuracies in representing elements of the matrix can be multiplied many times over in the determinant; additional errors may be introduced by rounding at subsequent steps of a floating point computation.

As there are eight other elements in the matrix, we can expect the total effect of inaccuracies in representing the elements to be rather more than `36eps`. However, the element we perturbed was the largest – does this have a disproportionately large effect on the determinant? In fact, if we instead perturb the smallest element by eps, the relative change in the determinant is `100eps`, so it appears not. We could investigate this in more detail but it is perhaps more interesting to consider the total effect of all of the inaccuracies in the matrix elements. In a typical situation, roughly half of elements would be in error by a positive amount, the rest by a negative amount, so we might expect them, to some extent, to cancel each other. Depending on the pattern of "missing" bits and whether the last bit present is rounded or not, the magnitude of the errors could range from zero to 2^{-n}, where n is the number of bits in the representation.

Rather than worry about the exact error structure in individual elements, let us carry out an initial investigation, making the simplifying assumption that all elements are subject to an error of $\pm$`eps`, with the sign of the error being assigned at random among the elements. AXIOM provides a function `randnum(`n`)`, which generates pseudo-random integers in the range $[0, ..., n-1]$ with a rectangular distribution, i.e., all of the numbers in the range are equally likely to occur. Do not be concerned about the "pseudo" – for almost all purposes we can use them as random numbers. However, it is worth noting that the sequence of numbers generated is repeatable – that is, the same sequence will be generated on different runs of AXIOM. This is important since otherwise, behaviour of algorithms using random numbers could not be reproduced, so bugs could not be detected or fixed. It is possible to force varying behaviour using the `reseed` function to reset the random-number generator.

```
→for i in 1..3 repeat for j in 1..3 repeat _
                          test3(i,j) := hilbert3(i,j) + (2*randnum(2) - 1)*eps
                                                                    Type: Void
```

```
→test3
```

$$\begin{bmatrix} eps+\frac{1}{2} & -eps+\frac{1}{3} & -eps+\frac{1}{4} \\ -eps+\frac{1}{3} & -eps+\frac{1}{4} & eps+\frac{1}{5} \\ eps+\frac{1}{4} & eps+\frac{1}{5} & eps+\frac{1}{6} \end{bmatrix} \tag{123}$$

Type: Matrix Polynomial Fraction Integer

```
→(determinant test3 - detHilbert3)/detHilbert3
```

$$-172800eps^3-46080eps^2-1188eps \tag{124}$$

Type: Polynomial Fraction Integer

As we are assuming that eps is already very small, its higher powers may be disregarded in this expression and we see that the magnitude of the proportional error in the determinant is 741 times that in the individual elements. It looks rather as if the effects of the errors in individual elements are adding, since the effects we saw for the extreme elements were `36eps` and `100eps`. As we have only carried out one run, we have no measure of how typical this is. (In fact, it is fairly near the end of the range of values which occur – repeating the experiment 100 times gave a mean absolute error in the determinant of about `300eps`.) However, we do not need to resort to statistical methods to see how the errors interact; we can investigate this analytically by adding an "error matrix" to the original Hilbert-like matrix.

```
→error3 := matrix [[eps[i,j] for i in 1..3] for j in 1..3]
```

$$\begin{bmatrix} eps_{1,1} & eps_{2,1} & eps_{3,1} \\ eps_{1,2} & eps_{2,2} & eps_{3,2} \\ eps_{1,3} & eps_{2,3} & eps_{3,3} \end{bmatrix} \tag{125}$$

Type: Matrix Polynomial Integer

```
→test3 := hilbert3 + t*error3;
                                                    Type: Matrix Polynomial Fraction Integer
```

```
→detErr := (determinant test3 - detHilbert3)/detHilbert3;
                                                    Type: Polynomial Fraction Integer
```

Perturbing `hilbert3` by `t*error3` rather than by `error3` provides us with a simple means of eliminating the products of error components which, as before, we shall regard as negligible:

→ `detErrReduced := coefficient(%,'t,1)`

$$600e\ p\ s_{3,3} - 720e\ p\ s_{3,2} + 180e\ p\ s_{3,1} - 720e\ p\ s_{2,3} + 900e\ p\ s_{2,2} - 240e\ p\ s_{2,1} + 180e\ p\ s_{1,3} - 240e\ p\ s_{1,2} + 72e\ p\ s_{1,1} \tag{128}$$

Type: `Polynomial Fraction Integer`

where we used `coefficient(%,'t,1)` to produce the coefficient of the variable `t` to the power `1` in the polynomial `%` (or `detErr`); since each `eps` was multiplied by `t`, this gives the expression which we required with `eps`-products eliminated. We can verify that there is no constant term in `detErr`:

→ `coefficient(detErr,'t,0)`

$$0 \tag{129}$$

Type: `Polynomial Fraction Integer`

The quantity `detErrReduced` is symmetrical in the `eps`-subscripts, since `hilbert3` itself was symmetrical: if we had used the transpose of `test3` instead of `test3` itself in the calculation, we would have interchanged the indices but kept the same coefficients in the final result.

The `epss` in `detErrReduced` can all be varied independently, so any cancelling of their effects is purely fortuitous. As our initial experiments suggested, the error in the (1, 1) position has the least effect; however, the error with the most effect is in the $(2, 2)$ position, not as we might have guessed the $(3, 3)$ position.

This approach, of considering the effects of perturbing various components in an expression, can be useful in analysing a calculation or experiment to find where increased accuracy will have the greatest benefit on the accuracy of the final result. However, if we had a larger problem, we would not wish to look for the largest effect by eye; instead we could use `sort` to find this.

Applying this approach to the determinant of `hilbert3`, as a demonstration, we can use the two functions `variables` and `coefficients` to produce lists of the `epss` and their coefficients (in the same order, since `detErrReduced` consists entirely of first degree terms), sort a list of indices `[1,...,9]` according to the absolute sizes of the coefficients and, finally, use the index of the largest coefficient to pick the corresponding element of the list of `epss`.

Simply applying `sort` to a list returns a sorted version of the list; more useful to us is the version of `sort` which takes an additional (first) parameter, a function which indicates whether two elements are in the correct order. Such a function, in our case, is `((i,j)+->abs coefs.i > abs coefs.j)`, where `abs` is the absolute value function; this returns `true` (indicating that `i` and `j` are in the correct order) when `abs(coefs.i) > abs(coefs.j)` is true.

→ `epses := variables detErrReduced`

$$[eps_{3,3},\ eps_{3,2},\ eps_{3,1},\ eps_{2,3},\ eps_{2,2},\ eps_{2,1},\ eps_{1,3},\ eps_{1,2},\ eps_{1,1}] \tag{130}$$

Type: `List Symbol`

→ `coefs := coefficients detErrReduced`

$$[600,\ -720,\ 180,\ -720,\ 900,\ -240,\ 180,\ -240,\ 72] \tag{131}$$

Type: List Fraction Integer

→ index := first sort((i,j)+->abs coefs.i > abs coefs.j, expand(1..9))

$$5 \tag{132}$$

Type: PositiveInteger

→ epses.5

$$eps_{2,\,2} \tag{133}$$

Type: Symbol

→ coefs.5

$$900 \tag{134}$$

Type: Fraction Integer

We can check whether the coefficient of largest magnitude is unique by sorting the coefficients themselves and inspecting the second element:

→ sort((i,j)+->abs i > abs j, coefs).2

$$-720 \tag{135}$$

Type: Fraction Integer

Chapter 8 Previous

Chapter 8, Next steps

The purpose of this tutorial was to familiarise users with the AXIOM approach to mathematics. We have covered the basics of elementary mathematics and how AXIOM can be used to tackle these areas. To make full use of all of AXIOM's functionality we recommend that the user reads the accompanying on-line documentation via $\mathrm{TEX}_{\mathrm{MACS}}$.

The most useful part of the manual to read next is probably chapter 5, "Introduction to the AXIOM Interactive Language".

Index