2

Maple for APPL

Maple is a computer algebra system and programming language that can be used for numerical computations, solving equations, manipulating symbolic expressions, plotting, and programming, just to name a few of the basics. APPL is, simply, a set of supplementary Maple commands and procedures that augments the existing computer algebra system. In effect, APPL takes the capabilities of Maple and turns it into a computer algebra system for computational probability. This chapter contains guidelines for using Maple, and discusses the Maple commands that are used in APPL programming. After reading this chapter, an APPL user will have the knowledge necessary to modify the APPL code to meet his or her particular needs. We will start with a discussion of basic numeric computation, then advance to defining variables, symbolic computations, functions, data types, solving equations, calculus and graphing. Then we will discuss the programming features of Maple that facilitate building the APPL language: loops, conditions, and procedures.

2.1 Numerical Computations

Numerical computations in Maple give it the functionality of a hand-held calculator. The command prompt is `>`. To execute an arithmetic expression in Maple, the expression must be terminated with a semicolon or colon. The `#` symbol is used for commenting in a Maple worksheet. Below are several examples of numerical computations with their corresponding outputs.

J.H. Drew et al., *Computational Probability*, International Series
in Operations Research & Management Science 246,
DOI 10.1007/978-3-319-43323-3_2

```
> 2 + 2;
```

$$4$$

```
> 1 + 1 / 2;
```

$$\frac{3}{2}$$

```
> 1 + 0.5;
```

$$1.5$$

```
> sqrt(2);                    # sqrt() takes the square root
```

$$\sqrt{2}$$

```
> Pi;
```

$$\pi$$

```
> evalf(Pi);
```

$$3.141592654$$

```
> 2 * 2.5:
> % + 1 / 2;
```

$$5.500000000$$

From these few examples, it is important to note the following:

- Spaces between symbols are optional, so `2 + 2;` and `2+2;` are equivalent. We include spaces between operators for readability, consistent with good programming practice.
- Maple performs exact calculations with rational numbers and approximate calculations with decimals. To Maple, the rational number 3/2 and the floating-point approximation 1.5 are different objects. Using both decimals and rational numbers in a statement produces a decimal output.
- Maple interprets irrational numbers as exact quantities. Maple also recognizes standard mathematical constants, such as π and e, and works with them as exact quantities.
- The `evalf` command converts an exact numerical expression to a floating-point number. By default, Maple calculates the result using ten digits of accuracy, but any number of digits can be specified. The optional second argument of `evalf` controls the number of floating-point digits for that particular calculation.
- For particularly precise numerical expressions, a call to `evalhf` evaluates an expression to a numerical value using the hardware floating-point precision of the underlying system. The evaluation is done in double precision. The `evalhf` function computes only with real floating-point arguments.

- If a statement ends with a colon, instead of a semicolon, then Maple suppresses the output, although it will store the result (if assigned to a variable) internally.
- The ditto operator `%` refers to your last calculated result, even if that result is not on the line preceding the `%`.

2.2 Variables

It is convenient to assign variable names to expressions that are referred to one or more times in a Maple session. Maple's syntax for assigning a variable name is `name :=` *expression*. Almost any expression, including numbers, equations, sets, lists, and plots, can be given a name, but it is helpful to choose a name that describes the expression. The `restart` command makes Maple act (almost) as if just started, and it clears the values of all Maple variables. Some guidelines for variable names are:

- Maple is case sensitive, so the names `X` and `x` denote unique variables.
- A variable name can contain alphanumeric characters and underscores, but it *cannot* start with a number.
- Once a variable is assigned to an expression, it remains that expression until changed or cleared.
- A variable name can be cleared by assigning `variable := 'variable';` or executing the statement `unassign('variable');`
- Maple has some predefined and reserved names, such as `Sum`, `sqrt`, and `length`, that are not available for variable assignment. Maple will not allow an expression to be assigned to a predefined variable name.
- When you close a Maple session, variable names assigned during that session will be forgotten. When a Maple worksheet is re-opened, variable names must be reactivated.
- The `restart` command at the top of the worksheet, followed by the sequence of keystrokes `Alt`, `E`, `E`, `W` (Edit, Execute, Worksheet) restarts the memory of the variables and then executes the entire worksheet in order. Often one will be in the middle of a series of commands, fix an error and need to re-execute all commands in order once again.

Below are several examples of defining variable names.

```
> restart:
> EventA := 0.3:
> EventB := 0.3:
> EventC := 0.4:
> S := EventA + EventB + EventC;
```

$$S := 1.0$$

```
> Sum := EventA + EventB;
Error, attempting to assign to `Sum` which is protected
```

```
> prob := p * (1 - p) ^ 2:
> p := 1 / 4:
> prob;
```

$$\frac{9}{64}$$

```
> unassign('p'):                # You could also write p := 'p'
> newprob := 3 * p ^ 2 * (1 - p);
```

$$newprob := 3p^2(1-p)$$

Once a Maple expression is given a name, it can be evaluated at different values using `subs()` or `eval()`. The command `subs(p = 1 / 2, newprob)` or `eval(newprob, p = 1 / 2)` yields the value $\frac{3}{8}$. Since `newprob` is a variable, and not a function, Maple does not understand the syntax `newprob(1 / 3)`, which the user may incorrectly try to use to determine the value of `newprob` for $p = 1/3$. If an expression is intended to actually define a function, then the function must be formally defined, and a technique for doing so is in Sect. 2.4.

Sometimes assumptions must be made on variables in order to set variable properties or relationships. A common use of the `assume` function is to assume a constant is positive, i.e., `assume(p > 0)`. Making such assumptions allow Maple routines to use this information to simplify expressions, for example, $\sqrt{p^2}$. When an assumption is made about a variable, thereafter the variable is displayed with an appended tilde `~` to indicate that it carries assumptions. The `additionally` function adds additional assumptions without removing previous assumptions. For example, we could further restrict p to be less than one with the command `additionally(p < 1)`.

2.3 Symbolic Computations

One of Maple's main strengths is its ability to manipulate symbolic expressions. Symbols can be treated in the same way that numbers were in the previous section and can do much more. Entering a constant or variable followed by a variable, e.g., `2x` or `ab`, does not imply multiplication in Maple. In fact, `ab` would be treated as a new two-letter variable instead of the product of two single-letter variables. You may *not* omit the multiplication symbol ($*$) in expressions with more than one factor. Below are a few examples of Maple's symbolic abilities using three commands, `combine`, `expand`, and `simplify`, that appear in the APPL code.

```
> exp(-t ^ 2) * exp(-s ^ 2);
```

$$e^{(-t^2)}e^{(-s^2)}$$

```
> combine(%);
```

$$e^{(-t^2-s^2)}$$

```
> mgf_1 := (1 / 3) * exp(t) + (2 / 3) * exp(2 * t):
> mgf_2 := (1 / 4) + (3 / 4) * exp(-t):
> mgf_1 * mgf_2;
```

$$\left(\frac{1}{3}e^{t}+\frac{2}{3}e^{(2t)}\right)\left(\frac{1}{4}+\frac{3}{4}e^{-t}\right)$$

```
> expand(%);
```

$$\frac{7}{12}e^{t}+\frac{1}{4}+\frac{1}{6}(e^{t})^{2}$$

```
> mgf_1 / mgf_2;
```

$$\frac{\frac{1}{3}e^{t}+\frac{2}{3}e^{(2t)}}{\frac{1}{4}+\frac{3}{4}e^{(-t)}}$$

```
> simplify(%);
```

$$\frac{4}{3}\,\frac{e^{(2t)}\,(1+2e^{t})}{e^{t}+3}$$

2.4 Functions

Functions are defined in Maple by using the arrow notation `->` or the `unapply()` command. The assignment operator `:=` associates a function name with a function definition. Two equivalent ways of defining a function f are

```
> f := x -> exp(-2) * 2 ^ x / x!:
> f := unapply(exp(-2) * 2 ^ x / x!, x);
```

$$f := x \rightarrow \frac{e^{(-2)}2^{x}}{x!}$$

This notation allows a function to be evaluated in the "usual" way, i.e., `f(0)`, when it appears in Maple expressions. Such functions are an integral part of APPL, and are used to create PDFs, CDFs, and transformations of random variables. Unassigning a function is done in the same way that a variable is unassigned to a value, `f := 'f'`. As shown in the examples below, piecewise functions may also be defined in Maple, and we certainly take advantage of this in APPL, e.g., the triangular distribution.

```
> g := unapply(exp(-lambda) * lambda ^ x / x!, lambda, x);
```

$$g := (\lambda, x) \rightarrow \frac{e^{(-\lambda)}\lambda^x}{x!}$$

```
> g(2, 0);
```

$$e^{(-2)}$$

```
> h := x -> piecewise(x <= 1, 0, x > 1, 2):
> h(x);
```

$$\begin{cases} 0 & x \leq 1 \\ 2 & x > 1 \end{cases}$$

In addition to facilitating the creation of custom functions, Maple has built-in mathematical functions, e.g., trigonometric functions such as `tan`, inverse trigonometric functions such as `arccos`, absolute value (`abs`), the exponential function (`exp`), the gamma function (`GAMMA`), and binomial coefficients (`binomial`). As illustrated earlier in the chapter, the exponential function in Maple is written as `exp(x)`. In order to compute e^3, enter `exp(3)`, *not* `e ^ 3`. The binomial function `binomial(n, x)` determines binomial coefficients for selecting x items from n items without replacement. Examples of calling Maple functions that might arise in the APPL code are given below.

```
> abs(ln(0.4) + arcsin(0.65));
```

0.2087062952

```
> binomial(4, 2);
```

6

```
> GAMMA(5);
```

24

```
> evalf(erf(2));
```

0.9953222650

Maple has packages of functions available with specialized commands to perform tasks from an extensive variety of disciplines. The commands in a package can be activated using the `with` command. For example, `with(networks)` adds the `networks` package, which supplies tools for constructing, drawing, and analyzing combinatorial networks. Likewise, `with(plots)` enables more options with plotting functions.

When a package's functions are accessed with `with`, occasionally a previous definition or a function, either built-in or defined by the user, will be overridden. For example, `trace` is by default defined to be the program-tracing function. However, after invoking `with(linalg)`, `trace` is redefined to mean the trace of a matrix. The `with` function prints out a warning message whenever it overrides a previous definition or function.

2.5 Data Types

This section examines some basic types of Maple objects, including lists, sets, arrays, and strings.

A Maple *list* can be created by enclosing any number of expressions (separated by commas) in brackets. Maple preserves the order and repetition in a list. Thus, `[a,b,c]` and `[b,c,a]` are different lists. The `nops` command determines the total number of elements in a list. The membership of an element in a list can be checked with `member()`, which returns a *true* or *false* Boolean value.

```
> probs := [0.3, 0.5, 0.2]:
> nops(probs);
```

$$3$$

```
> member(0.3, probs);
```

$$true$$

Because order is preserved in a list, a particular element can be extracted from a list without searching for it. Use `L[i]` to access the ith element of a list `L`. The `op` command applied to a list removes the brackets of the list, and `sort` preserves the list structure while sorting the elements into ascending order.

```
> probs := [0.3, 0.5, 0.2]:
> op(probs);
```

$$0.3, 0.5, 0.2$$

```
> sort(probs);
```

$$[0.2, 0.3, 0.5]$$

In APPL, we utilize the functionality of Maple's list structure to represent a random variable as a "list-of-sublists." A list-of-sublists is just a list, each of whose elements is itself a list. For example, we create a random variable as a list of three sublists, where the first sublist contains the functional form that describes the probability distribution, the second sublist contains its support, and the third sublist uses strings to describe the type of the random variable, i.e., continuous or discrete, and the type of function, e.g., the PDF, CDF or HF. When APPL receives a random variable for manipulation, it first checks the type of the random variable before continuing; that is, is the random variable continuous or discrete? Depending on the answer, APPL treats that random variable accordingly, for example, integrating to determine expected value versus summing. Additional information about APPL's list-of-sublists data structure for random variables is contained in Chaps. 3 and 7. Elements in a list-of-sublists can be extracted using indices. For example,

```
> x := [[1, 2, 3], [4, 5, 6], [7, 8]];
```

sets `x` to a list of three elements, each of which is a list. Sublists or individual elements can be extracted as follows:

```
> x[2];
```

$$[4, 5, 6]$$

```
> x[2, 1];
```

$$4$$

```
> x[2][1];
```

$$4$$

A Maple *set* is constructed by enclosing any number of Maple objects (separated by commas) in braces. The braces identify the object as a set. Maple does not preserve order or repetition in a set. Thus the sets {`a,b,c`}, {`b,c,a`}, and {`c,c,b,a`} are identical. The `member` command validates membership in a set, and to choose an item from a set, the subscript notation `[i]` is used, where `i` identifies the position of the desired element in the set. Also, the `nops` function counts the number of elements in a set.

The `seq` command is used to generate sequences, which can be contained in a list or a set if desired. For example, a sequence of binomial coefficients can be obtained by using `seq` as follows:

```
> seq(binomial(4, i), i = 0 .. 4);
```

$$1, 4, 6, 4, 1$$

The Maple *array* data structure is an extension of the Maple list data structure. Each element is still associated with an index (which can be negative or zero), but an array is not restricted to one dimension.

A *string* is a Maple object created by enclosing a sequence of characters in double quotes. We use strings in APPL in the third sublist of the list-of-sublists that define a random variable to help identify its functional form, such as `"Continuous"`, `"Discrete"`, `"PDF"`, `"CDF"`, etc.

2.6 Solving Equations

Maple can find analytic solutions for a large class of algebraic equations. The `solve` command is a general-purpose equation solver. It takes a set of one or more equations and attempts to solve it exactly for the specified set of unknowns. The `fsolve` command is the numeric equivalent of `solve`; `fsolve` calculates the solution(s) to equations using a variation on Newton's

method, producing approximate (floating-point) solutions. For a general equation, `fsolve` searches for a single root and often returns the first root it finds, which may not be the desired root. For a polynomial, it looks for all real roots. Fortunately, there is a way to make `fsolve` look for a specific solution by specifying an interval over which `fsolve` may search, as illustrated in the examples to follow. Some special notes about solving equations are:

- If the variable(s) being solved for are not specified, Maple solves for all variables.
- If no right-hand side to an equation is given (for example, `solve(2 * x - 8)`), the right-hand side is assumed to be 0.
- The `fsolve` command often only returns one real solution.
- If there are multiple solutions to an equation, they can be extracted by setting the solution set to a variable, then using subscripts `[1], [2],` etc.

Here are several examples of solving one or more equations at a time.

```
> solve(x ^ 3 = 2 * x, x);
```

$$0, \sqrt{2}, -\sqrt{2}$$

```
> fsolve(x ^ 3 = 2 * x, x);
```

$$-1.414213562, 0., 1.414213562$$

```
> eqns := {x + 2 * y = 3, y + 1 / x = 1}:
> solns := solve(eqns, {x, y});
```

$$solns := \{y = 2, x = -1\}, \left\{x = 2, y = \frac{1}{2}\right\}$$

```
> solns[2];
```

$$\left\{x = 2, y = \frac{1}{2}\right\}$$

```
> solve(cos(x) - x = 0);
```

$$\mathrm{RootOf}(\_Z - \cos(\_Z))$$

```
> fsolve(cos(x) - x = 0);
```

$$0.7390851332$$

```
> express1 := (1 / x) + (1 / x ^ 2) - (3 / x ^ 3):
> solve(express1);
```

$$-\frac{1}{2} + \frac{\sqrt{13}}{2}, -\frac{1}{2} - \frac{\sqrt{13}}{2}$$

```
> fsolve(express1);
```

$$1.30277563$$

```
> fsolve(express1, x = -3 .. 0);
```

$$-2.302775638$$

Maple occasionally returns solutions in terms of the `RootOf` function, which is a place holder for representing all the roots of an equation in one variable. The prefix $\_Z$ on the variable in a `RootOf` solution indicates that it has integer values. The `fsolve` command is able to approximate these roots (at most one at a time for non-polynomial equations, and a specific root if a search interval is specified). Below is also an indication of the range of a variable using two periods in a row. The range delimiter "`..`" denotes endpoints of intervals in plots, integrations, and other situations.

Two other commands of interest for obtaining either the left-hand side or right-hand side of an expression are `lhs` and `rhs`, respectively. Especially useful to APPL is the ability of these commands to extract the first and last term of a sequence. This functionality allows APPL to read off the bounds on the support of a probability function, which is displayed in the second sublist for a random variable. Both uses of `lhs` and `rhs` are displayed below.

```
> eqn1 := x + y = z + 3:
> lhs(eqn1);
```

$$x + y$$

```
> rhs(eqn1);
```

$$z + 3$$

```
> support := 1 .. infinity:
> lhs(support);
```

$$1$$

```
> rhs(support);
```

$$\infty$$

2.7 Graphing

Another capability of Maple is graphing. The explicit function $y = f(x)$ and its domain must be specified for a Maple plot. Plotting the standard normal distribution on the interval $-3 \leq x \leq 3$, for example, can be done with the Maple statements shown below. We will use the function form `->` to define the PDF of the standard normal distribution. The plot is displayed in Fig. 2.1.

```
> f := x -> 1 / sqrt(2 * Pi) * exp(-x ^ 2 / 2):
> plot(f(x), x = -3 .. 3);
```

In order to plot more than one function on a single graph, the functions must be contained in brackets or braces. The use of brackets means that order is important and Maple should graph them in the given order; the use of braces means that order is not important and Maple may graph them in any order. Also, it is sometimes useful to specify the range of the y-values in order to display important features of a graph. Figure 2.2 contains a plot of two normal PDFs. The functions are defined with two different variables, but used as arguments in `plot` with a third variable, as one would expect.

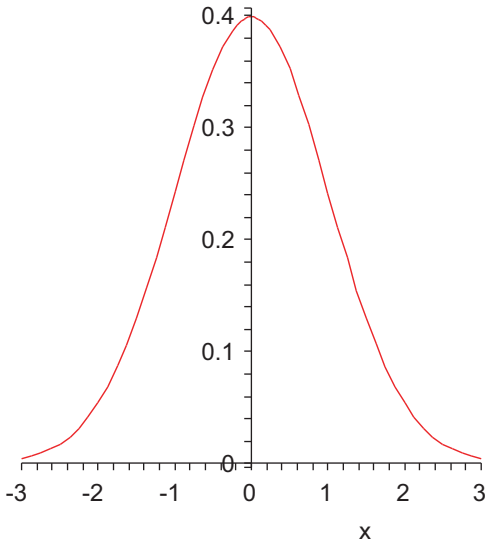

Fig. 2.1. Plot of a standard normal PDF between $x = -3$ and $x = 3$

```
> f1 := t -> 1 / sqrt(2 * Pi) * exp(-t ^ 2 / 2):
> f2 := z -> 1 / (2 * sqrt(2 * Pi)) * exp(-(z - 1) ^ 2 / 8):
> plot({f1(x), f2(x)}, x = -7 .. 9, color = [red, blue]);
```

2.8 Calculus

Maple provides many powerful tools for solving calculus problems. This section describes Maple's ability to determine limits, symbolically compute derivatives and definite integrals, and determine summations. The need for calculus operations in APPL is significant because many probability theorems rely on calculus to create new distribution functions. A few comments about determining limits, derivatives, and definite integrals in Maple follow:

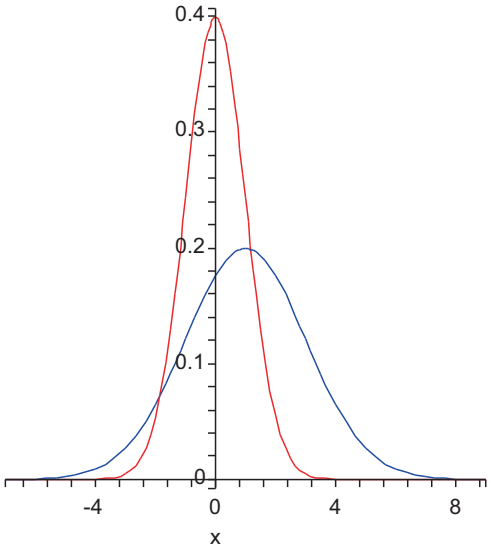

Fig. 2.2. Plot of a standard normal PDF and a normal PDF with mean $\mu = 1$ and standard deviation $\sigma = 2$ between $x = -7$ and $x = 9$

- Maple will compute both two-sided limits, e.g., $\lim_{x \to 5} \left(\frac{x+2}{x-1} \right)$, and one-sided limits, e.g., $\lim_{x \uparrow 1} \left(\frac{x+2}{x-1} \right)$, with the `limit` procedure.
- Maple can determine limits at ∞, such as the limit of an exponential CDF at infinity.

Examples of the `limit` command are

```
> limit((3 ^ (x + h) - 3 ^ x) / h, h = 0);
```

$$3^x \ln(3)$$

```
> assume(lambda > 0):
> limit(1 - exp(-lambda * x), x = infinity);
```

$$1$$

The `diff` procedure computes the derivative of an expression. Maple differentiates with respect to the variable placed in the second argument of the `diff()` procedure. Clearly, `diff` is used often in APPL for changing from one functional form of a continuous random variable's distribution to another, e.g., deriving a PDF from a CDF. One such example of the `diff()` command follows; however it is important to note that APPL will do such conversions in a more automated manner. Also note the use of the `unapply` command to turn the final expression into a function.

```
> F := x -> 1 - exp(-2 * x);
```

$$F := x \to 1 - e^{(-2x)}$$

```
> f := unapply(diff(F(x), x), x);
```

$$f := x \to 2\, e^{(-2x)}$$

The capabilities of Maple include a robust set of integration operations. Symbolic integrals (often used in APPL) can be calculated as in the following example:

```
> f := y -> 2 * exp(-2 * y):
> F := unapply(int(f(y), y = 0 .. x), x);
```

$$F := x \to 1 - e^{(-2x)}$$

Maple's ability to integrate also includes the numerical evaluation of integrals for which no closed-form solution is available, such as the case with the standard normal random variable:

```
> g := 1 / sqrt(2 * Pi) * exp(-x ^ 2 / 2):
> int(g, x = -infinity .. 1.96);
```

$$0.9750021049$$

However, if we integrate the standard normal PDF over exact values, rather than floating point values, Maple responds with `erf`. For example, integrating a standard normal PDF from $x = -1$ to 1 yields the following:

```
> int(g, x = -1 .. 1);
```

$$\mathrm{erf}\left(\frac{\sqrt{2}}{2}\right)$$

The function `erf(x)` is the error function and it is defined for real and complex x by $\mathrm{erf}(x) = \frac{2}{\pi}\int_0^x e^{-t^2} dt$. Maple can numerically evaluate an `erf(x)` output with the `evalf` command.

```
> evalf(int(g, x = -1 .. 1));
```

$$0.6826894920$$

Maple can also compute finite sums and infinite series, which are necessary capabilities for working with discrete random variables. The next few examples demonstrate the use of `sum`. The first example verifies that a geometric distribution with $p = 1/4$ has a PDF that sums to one.

```
> geometric := (1 / 4) * (3 / 4) ^ (x - 1);
```

$$geometric := \frac{\left(\frac{3}{4}\right)^{(x-1)}}{4}$$

```
> sum(geometric, x = 1 .. infinity);
```

$$1$$

The second example determines the expected value of a binomial distribution with parameters n and p.

```
> assume(p > 0):
> additionally(p < 1):
> pdf := binomial(n, x) * p ^ x * (1 - p) ^ (n - x):
> simplify(sum(x * pdf, x = 0 .. n));
```

$$np\sim$$

2.9 Loops and Conditions

Like all programming languages, loops in Maple allow the user to execute a sequence of statements repeatedly, either for a prescribed number of times or until a condition is satisfied (or not satisfied). This is a useful tool in APPL when working with random variables. Loops are used in APPL to do many things: cycle through segments of PDFs, verify inverse transformations, and error-checking, to name a few. For example, if a discrete random variable's PDF consists of a finite number of numeric probabilities in the first sublist in a list-of-sublists, we require these probabilities to be non-negative. Using a `for` loop allows APPL to cycle through the items in this sublist (`from` item one `to nops(sublist)`) until the list is complete or a negative value is identified. Similarly, if numeric probabilities in the first sublist are supposed to be equally-likely probabilities, then APPL can repeatedly check each probability value `while` the probability values remain the same. The expression in the `while` clause is a Boolean expression which must evaluate to *true* or *false* upon each cycle.

Conditional statements in APPL have a variety of purposes including error checking, branching in logic, and conditional plotting. If an APPL procedure requires input arguments from the user, then an `if` statement can be used to verify that the input is valid. If a procedure does not receive the correct type of arguments, e.g., lists, an error message is sent back to the APPL user. Also, in the third sublist is the "form" of the random variable. The first element of the third sublist tells APPL whether the random variable is continuous or discrete. If the random variable is continuous, then continuous mathematics (e.g., integration) is used to verify the validity of its PDF or compute its mean, for example. If discrete random variables are indicated by the field, then discrete mathematical techniques (e.g., summation) are used.

Boolean expressions in the conditional statement are most often formed by using `<`, `<=`, `>`, `>=`, `=`, `<>`, `or`, `and`, or `not`. For example, if the "form" of the random variable is *not* one of the approved formats (e.g., PDF, CDF, SF, HF, CHF, IDF), then an error message is reported to the user.

A few examples of small loops and conditionals are presented here to amplify the above explanations:

```
> probs := [0.1, 0.05, 0.2, 0.25, 0.15, 0.10, 0.15]:
> totprob := 0:
> for i from 1 to nops(probs) do
>   totprob := totprob + probs[i]:
> end do;          # 'od' can also be used to end a 'do' loop
```

$$totprob := 0.1$$
$$totprob := 0.15$$
$$totprob := 0.35$$
$$totprob := 0.60$$
$$totprob := 0.75$$
$$totprob := 0.85$$
$$totprob := 1.00$$

```
> die := rand(1 .. 6):          # random integer from 1 to 6
> Roll1 := die():
> Roll2 := die():
> if (Roll1 >= Roll2) then
>   print("Roll 1 is greater than or equal to Roll 2");
> else
>   print("Roll 1 is less than Roll 2");
> end if;    # 'fi' can also be used to end an 'if' conditional
```

To conclude this section, a `while` loop is created in which the user is encouraged to experiment with various values for the probability `prob`.

```
> i := 1:
> totprob := 0:
> prob := 1 / 2:
> while (totprob <= 1) and (i <= 1000) do
>   totprob := prob + totprob:
>   prob := prob ^ 2 + 0.001:
>   i := i + 1:
>   print(i):
>   print(totprob):
> end do:
```

When a `break` statement is used in a procedure, a break is executed and the result is to exit from the innermost repetition (`for/while/do`) statement within which it occurs.

2.10 Procedures

The capability to create new procedures in Maple is a key element to enabling APPL programming. In fact, APPL is a text file of new Maple procedures that is imported into a worksheet session with the `read` command. When a file is read into Maple, each line in the file is treated as if it had been typed in order. The APPL procedures are available during the Maple session, but the commands are not displayed on the Maple worksheet. When the user reads in the APPL commands, it's as if they used the `with` command to add extra capability. The syntax for reading APPL into Maple from the ASCII source code file named `APPL.txt` is `read('APPL.txt')`.

The syntax for creating a procedure in Maple is

`proc`(⟨*argseq*⟩) [local ⟨*nseq*⟩ ; global ⟨*nseq*⟩ ;]

⟨*statement sequence*⟩

end;

where [ ] indicates optional parameters. Then

- ⟨*argseq*⟩ is a sequence of variable names, separated by commas. These variables are the arguments to the procedure.
- ⟨*nseq*⟩ is a sequence of variable names, separated by commas.
- local ⟨*nseq*⟩ is a list of variables local to the procedure. Any assignments of these variables will only have a scope of the procedure. The values of these variables will be unassigned when the procedure starts, regardless of the value of the variable outside the procedure.
- global ⟨*nseq*⟩ is a list of global variables used by the procedure. Any assignments made in the procedure will be global in scope and the initial values of the variables will be as they are in the existing Maple session.
- ⟨*statement sequence*⟩ is the body of the procedure and may consist of any valid sequence of Maple statements.

The `proc` command is usually used in conjunction with the `RETURN` keyword. The syntax is

`RETURN` (*expression*) ;

This command will terminate the execution of the procedure and return the value of *expression*.

Since APPL's random variables often have several arguments, it's helpful that Maple provides an easy way to do data typing in procedures. The arguments passed to a procedure may be given a specific data type. If any of these variables is then assigned an incorrect type, Maple generates an error message. The syntax for declaring a variable to be of a given data type is the declaration is

variable name :: *data type*

where Maple accepts a wide range of data types. The types most frequently encountered in APPL are `list` and `listlist` (because of the list-of-sublists structure of APPL random variables), `constant`, `posint`, `int`, and `array`. Another type used frequently in APPL is `symbol`. Infinity (∞) is of type `symbol`, and this is important information for APPL procedures to know when evaluating random variables over supports that are not finite. The type of an expression can be determined in Maple with the command `type`. The syntax is

`type` (*expression*, *type*)

which returns a Boolean value, either *true* or *false*.

Another useful error-checking command that can used with procedures is `nargs`, which is the number of arguments passed to a procedure. We use `if` statements in APPL to determine if optional arguments are passed to a procedure and how the procedure should proceed given this number of arguments. For example, the `OrderStat` procedure in APPL will perform differently based on whether or not it is given an optional fourth argument that specifies whether sampling is done without replacement.

To summarize the last few sections on programming in Maple, the following excerpt from the APPL source code is noteworthy. The procedure is called `ReduceList` and it is a small sub-procedure in APPL that looks for redundant support entries in a random variable list-of-sublist's second sublist. One sees how the `proc` command begins the procedure as well as some argument checking, local variable declarations, a `for` loop and some conditional branching. The `RETURN` and `end` commands complete the procedure.

```
#
# ReduceList is a procedure that eliminates floating point
# redundancies (e.g., 3 vs. 3.0) from a sorted Maple list.
#
ReduceList := proc(LST :: list)
local i, size, delt, deltamin, ListIn:
deltamin := 0.0000001:
ListIn := LST:
size := nops(ListIn):
for i from (size - 1) by -1 to 1 do
  if (ListIn[i] <> -infinity and ListIn[i + 1] <> infinity) then
    delt := evalf(ListIn[i + 1]) - evalf(ListIn[i]):
    if (delt < deltamin) then
      if (whattype(ListIn[i]) <> float) then
        ListIn := subsop((i + 1) = NULL, ListIn):
      else
        ListIn := subsop(i = NULL, ListIn):
      fi:
    fi:
  fi:
```

```
od:
RETURN(ListIn):
end:
```

This concludes our brief introduction to the Maple computer algebra system. Additional help on a specific Maple topic can be accessed from the Maple worksheet by entering `?topic` or `help(topic)` at the prompt. Also, help can be obtained directly by going to the "Help" menu in Maple, scrolling down to "Topic Search," and entering the topic of interest in the "Topic" line. We now turn to the main topic of the monograph, defining random variables in APPL and the associated algorithms to manipulate them. We begin with continuous random variables.

http://www.springer.com/978-3-319-43321-9

Computational Probability
Algorithms and Applications in the Mathematical Sciences

Drew, J.H.; Evans, D.L.; Glen, A.G.; Leemis, L.M.

2017, XI, 336 p. 85 illus., 25 illus. in color., Hardcover
ISBN: 978-3-319-43321-9