Monaco

A program to answer probability questions, especially from dice and card games.

A Tutorial

Introduction

When playing games that include chance, usually using dice or cards, questions often arise of the forms:

- How *likely* is that event?
- What is the *average value* of that result?
- What is the *distribution* of that result?

This is a short tutorial that is intended to enable you to start using a program called **Monaco** that can be used to answer questions such as these that can arise in a wide range of games.

Getting Started

This document describes how to get started using the program to answer some questions of those sorts. The examples given are mainly to solve problems that can arise in real games. You should try out each example solution, and maybe try modifying it to see how to extend or improve it. By the time you have finished, you should be able to solve a lot of real problems.

Before you can start, you need to get two things out of the way:

- You need an *executable* version of the program. Maybe you were given it, or maybe you had to build it. This tutorial will assume that you have handled that.
- Monaco is a *command line* program. This tutorial assumes you know what that means and how to run such a program. In this tutorial it is assumed that you use `monaco` to run the program. You should make whatever changes you need to make to that and other parameters, such as whether they need to be put in quotes or otherwise modified.

Exact and Approximate Answers

There are two ways to run the program:

- To get *exact answers*. These are better, so most of this tutorial is about them.
- To get *approximate answers*. These are sometimes needed, see near the end of this tutorial for some reasons why.

Dice and Cards

There are two main sorts of problems the program can handle:

- Problems using *dice*.
- Problems using *cards*.

Dice are easier to handle than cards, so this tutorial starts with dice problems.

Program Parameters

For exact answers, following `monaco` we need two types of program *parameters*, in this order:

- One or more parameters called *options*. These start with a `-` sign. For exact answers the first option is `-exact`. Other options control the program's output.
- A single parameter called the *expression*.

Output Parameters

The main *output options* correspond to the three questions that started this tutorial:

- How likely is that event? Use `-probability -statistics`.
- What is the average value of that result? Use `-statistics`.
- What is the distribution of that result? Use `-all`, but we may not need all of the output, which also includes the `-statistics` output. Later we will see an alternative to this.

The Expression

The expression sets up the game situation and also extracts the information we want from it. To do this, the expression can be *evaluated* to produce a *result* that we are interested in. This represents a *random* process that can have a different result each time. When answers are approximate, each result is random; when answers are exact all possible results are created. For the first question the result must be *true* or *false* and we want to know how often it is true. For the other questions the result is what we want the average value of or the distribution of.

First Problem: Totalling Dice

We start with the problem of rolling three *standard dice*, with faces numbered from 1 to 6, and adding them up. We use `-all` to give us both the average value and the distribution.

The expression we will use is `sum(sorted3d6)`. It is explained below, after showing its use.

We can now run the program using the command line:

```
monaco -exact -all sum(sorted3d6)
```

First Problem – Average Output

The output from `-all` is in four parts, separated by blank lines.

First we get the `-statistics` output that gives us the average. For this example this is:

```
Number of evaluations      = 56
Number of results          = 216
Mean                       = 10.5 = 21/2
Standard deviation         = 2.95804
Minimum result             = 3
Maximum result             = 18
```

The average, or (arithmetic) *mean* is on the third line. It is 10.5, or as a fraction 21/2. For this simple example we can work that out without using the program and thus we can verify it.

The first two lines are explained after explaining `sum(sorted3d6)`. The last two lines should be clear. The *standard deviation* is considered later when describing approximate answers.

First Problem – Distribution Output

The second piece of output from `-all` gives us the distribution of the results:

```
 3 -  1 ~ 0.00462963 = 1/216
 4 -  3 ~ 0.0138889  = 1/72
 5 -  6 ~ 0.0277778  = 1/36
 6 - 10 ~ 0.0462963  = 5/108
 7 - 15 ~ 0.0694444  = 5/72
```

```
 8 - 21 ~ 0.0972222  = 7/72
 9 - 25 ~ 0.115741   = 25/216
10 - 27 ~ 0.125      = 1/8
11 - 27 ~ 0.125      = 1/8
12 - 25 ~ 0.115741   = 25/216
13 - 21 ~ 0.0972222  = 7/72
14 - 15 ~ 0.0694444  = 5/72
15 - 10 ~ 0.0462963  = 5/108
16 -  6 ~ 0.0277778  = 1/36
17 -  3 ~ 0.0138889  = 1/72
18 -  1 ~ 0.00462963 = 1/216
```

To understand this output, consider the row that starts `8`. It says that the *probability* of a result of 8, exactly, is 7/72 or 0.0972222 (to six *significant figures*). Probabilities are in the range from 0 (never) to 1 (always). As a *percentage* that is (multiplying by 100) 9.72222%. There is a way to make the program report percentages rather than probabilities; this is described later.

If we want the probability of a result of e.g. 8 or less, the third piece of output gives us that:

```
<=  3 -   1 ~ 0.00462963 = 1/216
<=  4 -   4 ~ 0.0185185  = 1/54
<=  5 -  10 ~ 0.0462963  = 5/108
<=  6 -  20 ~ 0.0925926  = 5/54
<=  7 -  35 ~ 0.162037   = 35/216
<=  8 -  56 ~ 0.259259   = 7/27
<=  9 -  81 ~ 0.375      = 3/8
<= 10 - 108 ~ 0.5        = 1/2
<= 11 - 135 ~ 0.625      = 5/8
<= 12 - 160 ~ 0.740741   = 20/27
<= 13 - 181 ~ 0.837963   = 181/216
<= 14 - 196 ~ 0.907407   = 49/54
<= 15 - 206 ~ 0.953704   = 103/108
<= 16 - 212 ~ 0.981481   = 53/54
<= 17 - 215 ~ 0.99537    = 215/216
<= 18 - 216 ~ 1
```

The line starting `<= 8` tells us that the probability of a result of 8 or less is 7/27 or 0.259259.

The final output reports probabilities such as of a result of 15 or more (5/54 or 0.0925926):

```
>=  3 - 216 ~ 1
>=  4 - 215 ~ 0.99537    = 215/216
>=  5 - 212 ~ 0.981481   = 53/54
>=  6 - 206 ~ 0.953704   = 103/108
>=  7 - 196 ~ 0.907407   = 49/54
>=  8 - 181 ~ 0.837963   = 181/216
>=  9 - 160 ~ 0.740741   = 20/27
>= 10 - 135 ~ 0.625      = 5/8
>= 11 - 108 ~ 0.5        = 1/2
>= 12 -  81 ~ 0.375      = 3/8
>= 13 -  56 ~ 0.259259   = 7/27
>= 14 -  35 ~ 0.162037   = 35/216
>= 15 -  20 ~ 0.0925926  = 5/54
>= 16 -  10 ~ 0.0462963  = 5/108
>= 17 -   4 ~ 0.0185185  = 1/54
>= 18 -   1 ~ 0.00462963 = 1/216
```

First Problem – The Expression

The expression `sum(sorted3d6)` produces a result, a number that in this case is from 3 to 18. Results are always *integers*, or whole numbers. Those particular numbers are all greater than zero, but we can also have numbers, including results, that are zero or less than zero.

Every expression result is a single number. But within an expression we can also use *lists* that each contain one or more numbers. Lists have the following properties:

- They are *ordered*. That means that the list containing 2 then 3 then 6 is a different list to the list containing 3 then 6 then 2. We can write those lists as `{2,3,6}` and `{3,6,2}`. The numbers in a list are called its *elements*; we number them from zero.
- They each have a fixed *length*. For example, the list `{2,3,6}` has a length of 3.

`sum(sorted3d6)` is made up of two parts: `sum` and `sorted3d6`. We consider it from the inside, `sorted3d6`, to the outside, `sum`. Later examples are also considered from left to right.

`sorted3d6` is a list containing the rolls of 3 standard dice, each with 6 sides. We could create all possible lists of these dice rolls, 6×6×6 = 216 of them. But when we total dice rolls we do not care which die is which; lists of the same dice rolls in different orders, such as `{2,3,6}`, `{3,6,2}` and four others, would have the same total. So instead we only create one of those lists; we choose the one *sorted* in order, i.e. `{2,3,6}`. There are 56 such sorted lists, in sets of 6, 3 and 1. We now have two numbers: the number of possible lists, 216, and the number of lists we are considering, 56. Those are the numbers in the first two lines of the first output.

The 56 lists are not equally likely, but the program handles that for us. The time that is needed depends on the 56, not the 216, a saving by a factor of about 4; factors can be much bigger.

`sum` is a *function*, one that converts a list to a number, in this case by adding up its elements. We use it by following its name with its *argument*, the list `sorted3d6`, in *parentheses* `()`.

Second Problem: *Dungeons and Dragons*

The sum of 3 standard dice was the original way to create a characteristic playing the game *Dungeons and Dragons* (*D&D*). Players wanted better characters, and so other ways to create a characteristic were devised. One of these was to roll 4 dice and total the 3 largest.

We start with a list of 4 dice `sorted4d6`. We can always change numbers in *terms* like that if it makes sense and they are not too large. Because that list is sorted, the 3 dice values we want are its last 3 elements. We use a new kind of function, one that converts a list to another list, called `tail3`. There is a `head3` for the first 3 elements, and other lengths in each case.

Our 3 dice are now `tail3(sorted4d6)` and their total is `sum(tail3(sorted4d6))`.

To save space here we just look at the `-statistics` output for this expression, which is:

```
Number of evaluations      = 126
Number of results          = 1296
Mean                       = 12.2446 = 15869/1296
Standard deviation         = 2.84684
Minimum result             = 3
Maximum result             = 18
```

The average value of a characteristic has gone up from 10.5 to 12.2446.

Third Problem: *Yspahan*

In the game *Yspahan*, players usually roll 9 standard dice, as `sorted9d6`. We might want to know how many different values were rolled and the greatest number of the same value rolled.

How many different numbers were rolled is given by the function `count_diff`. To save space and give an example of the "how likely" question, we just ask how likely are we to see all 6 numbers. We can do this with `count_diff(sorted9d6)==6`. Using == to test "is equal to" is taken from some computer languages. We use the options `-probability -statistics`.

The output in this case, giving us the probability of 0.189043, is:

```
Number of evaluations      = 2002
Number of results          = 10077696
Number of false results    = 8172576
Number of true results     = 1905120
Probability                = 0.189043 = 245/1296
```

In the game a player sometimes rolls 12 dice instead of 9. To show that and also something new, you might prefer percentages to probabilities. For this we add the option `-percent`. Doing that, and changing 9 to 12, gives us the following output and an answer of 43.7816%:

```
Number of evaluations      = 6188
Number of results          = 2176782336
Number of false results    = 1223752896
Number of true results     = 953029440
Probability                = 43.7816% = 1654565/3779136
```

The difference between 6188 and 2176782336 (the work we would need without sorting dice) is the difference between minutes and milliseconds in how long the program takes to run.

We can also get percentages in distribution output. We show that while answering our second question, how many of the most common die value did we roll? There is a function for that, `count_mode`, and hence for 9 dice we use `count_mode(sorted9d6)`. To save space, here is just the second output from `-all`, which with `-percent` is now:

```
2 - 1587600 ~ 15.7536%     = 1225/7776
3 - 5628000 ~ 55.8461%     = 58625/104976
4 - 2320920 ~ 23.0303%     = 10745/46656
5 -  472500 ~ 4.68857%     = 4375/93312
6 -   63000 ~ 0.625143%    = 875/139968
7 -    5400 ~ 0.0535837%   = 25/46656
8 -     270 ~ 0.00267918%  = 5/186624
9 -       6 ~ 5.95374e-05% = 1/1679616
```

Fourth Problem: Dice Combinations

In several games a player rolls five standard dice, in this problem just rolling once. We then look for *combinations* such as *long straight*, {1,2,3,4,5} or {2,3,4,5,6}, *short straight*, four dice are {1,2,3,4}, {2,3,4,5} or {3,4,5,6}, and *full house*, three of one value and two of another value.

We can check for a long straight by comparing the dice rolled, `sorted5d6`, with the two possible lists. That needs some new features. To check for the list {1,2,3,4,5} we use a new function as `list_eq(sorted5d6,{1,2,3,4,5})`; do not try to use ==. `list_eq` is a new kind of function, it has two arguments, separated by a comma. We could then check the other list, and then try to combine the tests using the logical or *operator* `|`, more on which later.

But writing this the obvious way does not work because it would use `sorted5d6` twice, and each such use creates a different list. We need to compare the same list in each case. We do that by storing the list in a *variable* and then comparing that variable with each straight, as:

```
v0:=sorted5d6;list_eq(v0,{1,2,3,4,5})|list_eq(v0,{2,3,4,5,6})
```

There are two new things here:

- The variable `v0`, which we set to our list of dice with `v0:=sorted5d6` and use as `v0`.
- We *sequence* setting `v0` and then testing `v0` with `;` which means do this, then do that, left to right. The result of the sequence is the result of what comes after the `;`.

The `-probability -statistics` output to test the likelihood of a long straight is then:

```
Number of evaluations      = 252
Number of results          = 7776
Number of false results    = 7536
Number of true results     = 240
Probability                = 0.0308642 = 5/162
```

There is more about variables later. But we can also solve this problem without them. To do that we can use a function `runs` that is one of several that can *analyse* a list. `runs` creates a list of the same length that tells us how many *runs* of length 1, 2, … are found in the list that it analyses. Runs are not counted if they are part of a longer run, but all possible runs are counted, even when they overlap. Thus, for example, `runs({2,3,3,4,6})` is {1,0,2,0,0}.

So to check for a long straight we could use `list_eq(runs(sorted5d6),{0,0,0,0,1})`. However, we only need to check the last element of the resulting list, using the function `last`, as `last(runs(sorted5d6))==1`. There is also a function `first` for the first element.

The output is, as we would expect, exactly as the previous output.

For a short, but not long, straight, we must make two changes. First, we want the penultimate element of the result of `runs`. This is `get(3,runs(sorted5d6))`. `get` is a function that gets an element by number. Although this is for length 4, elements are numbered from 0, hence the `3`. Second, it is possible to have two short straights, e.g. in {2,3,3,4,5}, so we replace `==1` by `!=0`, where `!=` means not equal to. We could also use `!=0` testing for a long straight.

This is a new result, with corresponding `-probability -statistics` output:

```
Number of evaluations      = 252
Number of results          = 7776
Number of false results    = 6816
Number of true results     = 960
Probability                = 0.123457 = 10/81
```

For a full house, we have a function `groups` that is a companion to `runs`. It counts how many singletons, doubletons etc. are in a list. For example, `groups({2,3,3,3,5})` is {2,0,1,0,0}. We can thus check for a full house using `list_eq(groups(sorted5d6),{0,1,1,0,0})`.

The corresponding `-probability -statistics` output in this case is:

```
Number of evaluations      = 252
Number of results          = 7776
Number of false results    = 7476
Number of true results     = 300
Probability                = 0.0385802 = 25/648
```

Functions

Functions, plus operators described in the next section, are how to manipulate lists and numbers. The functions we have so far (we do not consider `sorted3d6` as a function) are:

- `sum`, which adds up the elements of a list to give a number.

- `count_diff`, which counts how many different values there are in a list.
- `count_mode`, which counts how many of the most common value there is in a list.
- `first` and `last`, which extract the first or last element from a list, respectively.
- `get`, which extracts a single element from a list. Specifically, `get(`*n*`,`*list*`)` extracts element number *n*, counting from zero, from *list*. Here and in later examples, *n* (and later *m*) is anything that results in a number, and *list* is anything that results in a list.
- `head£` and `tail£`, where `£` is replaced by a number greater than zero, e.g. `head4` or `tail3`. These functions convert a list of any length greater than `£` to length `£` by taking the first or last `£` elements of that list, respectively.
- `list_eq`, which is true if two lists are the same, false if they are not. There is also a function `list_ne` that is the opposite of that.
- `runs`, which determines the numbers of runs of lengths 1, 2, … in a list, as a list.
- `groups`, which determines the numbers of groups of equal elements of sizes 1, 2, … in a list, as a list.

There are many more functions. Here are some of the most useful ones:

- `min` and `max` extract the minimum and maximum element, respectively, from a list. So `min({2,3,3,4,4,6})` is 2 and `max({2,3,3,4,4,6})` is 6.
- `mode_min` and `mode_max` extract the most common number among the elements of a list. If there is more than one most common number, `mode_min` extracts the smallest such number and `mode_max` extracts the largest such number. So `mode_min({2,3,3,4,4,6})` is 3 and `mode_max({2,3,3,4,4,6})` is 4.
- `same` and `different`. These determine if the elements of a list are all the same or all different, respectively. So `same({1,1,1,1})` and `different({2,3,5,6})` are both true, but `same({2,3,3,5})` and `different({2,3,3,5})` are both false.
- `count_eq(`*list*`,`*n*`)` is the number of elements of *list* that are equal to *n*. So `count_eq({2,3,3,5,6},3)` is 2. There are also functions `count_ne`, `count_lt`, `count_le`, `count_gt` and `count_ge` that count the numbers of elements in a list that are not equal to, less than, less than or equal to, greater than, and greater than or equal to, respectively, a number. For example, `count_gt({2,3,3,5,6},3)` is 2 and `count_ge({2,3,3,5,6},3)` is 4.
- `sort(`*list*`)` is *list* sorted in ascending order. So `sort({2,5,3,1})` is {1,2,3,5}. `rsort(`*list*`)` is *list* sorted in descending order. So `rsort({2,5,3,1})` is {5,3,2,1}.
- `reverse(`*list*`)` is *list* reversed in order. So `reverse({2,5,3,1})` is {1,3,5,2}.
- `lower` and `upper`. These take the element by element minimum and maximum, respectively, of two equal length lists. So `lower({1,2,4,5},{2,3,3,5})` is {1,2,3,5} and `upper({1,2,4,5},{2,3,3,5})` is {2,3,4,5}.
- `counts(`*list*`)` is a list that replaces each element of *list* by how many times that number appears in *list*. So `counts({2,3,3,3,5,5})` is {1,3,3,3,2,2}.

Operators

Operators are like functions but are written using symbols. There are *binary operators* that can combine two numbers or two lists, before and after the symbol, and *unary operators* that apply to one following number or list. These numbers and lists are called the *operands*. The result is also a number or a list. All lists, operands and the result, must have the same length.

The most useful binary operators for numbers are:

- The *arithmetic* operators + (addition), - (subtraction), * (multiplication), / (division) and % (modulus or remainder). / rounds towards zero; the result of % is always ≥ 0.
- The *comparison* operators == (equal to), != (not equal to), < (less than), <= (less than or equal to), > (greater than) and >= (greater than or equal to). The result is true or false as appropriate.
- The *logical* operators & (and), ^ (exclusive or) and | (inclusive or). & and | only evaluate the second operand if needed, if the first operand is true or false, respectively.

The two most useful unary operators for numbers, one arithmetic, one logical, are:

- Negative -. The effect of subtracting a number from zero.
- Not !. Turns all non-zero numbers to false and turns zero to true.

For lists, we usually only need the arithmetic operators, working element by element. For example, `{1,2,2,4}+{2,3,4,5}` is {3,5,6,9}. We can combine a list and a simple number; the number is combined with each list element. For example, `{1,2,2,4}+2` is {3,4,4,6}.

Binary operator *precedence* – what is combined with what first – is as usual in computing, i.e. in the bullet order above. Not all operators under one bullet have the same precedence, in particular `* / %` are above `+ -`. Within other bullets, or if in any doubt, use parentheses `()`.

Variables

We used the variable `v0` to store a list. There are also list variables `v1` to `v9`, but no further. Each list variable has a fixed length. For example, if `v0` has stored a list of length 4 it cannot later store a list of length 3. That `v0` can be used in, for example, `v0+{2,3,3,4}` or `v0*2`.

There are also variables `r0` to `r9` that store numbers. We set them similarly using `:=`.

An *assignment* such as `r0:=r0+2` can be written as `r0+=2`. This works for binary arithmetic and logical operators, but not for comparison operators. We can also do this for list operators.

Before each evaluation of the expression, all used variables are set to either 0 or a list of 0s.

More Dice

A single die can be e.g. `d6`. For example, to set `r0` to the value of a 10-sided die use `r0:=d10`.

Some non-standard dice can be created by arithmetic. For example, `d3-2` can be used for a die with faces -1, 0 and 1. For a list of 4 such dice use `sorted4d3-2`. Some dice have face values that are not a sequence. We can put those values in a list such as `{2,3,3,4,4,5}` and use e.g. `sorted_select3from[{2,3,3,4,4,5}]` to produce a sorted list of 3 such dice values and `sum(sorted_select3from[{2,3,3,4,4,5}])` to sum those values. What is in the `[]`, in this and similar terms, must be *constant*, i.e. use no dice or variables.

An example from the game *Onward to Venus* replaces the 1 on a d6 with a 0, rolls 3 such dice and subtracts the minimum rolled value from the maximum rolled value. This can use:

```
v0:=sorted_select3from[{0,2,3,4,5,6}];last(v0)-first(v0)
```

Note that we can use `first` and `last` because `v0` is sorted. The `-statistics` output is:

```
Number of evaluations      = 56
Number of results          = 216
Mean                       = 3.33333 = 10/3
Standard deviation         = 1.63299
```

```
Minimum result              = 0
Maximum result              = 6
```

Cards

A standard *deck* (or *pack*) of 52 cards uses 4 named *suits* and 13 named and numbered *ranks*, each combination once. But for this program we have to use numbers. We can number the cards 0 to 51, with suits numbered 0 to 3 and ranks numbered 0 to 12. We let cards 0 to 3 be rank 0, cards 4 to 7 be rank 1 and so on, with each group of four in the same suit order.

There is a term `sequence52` that is a list of the numbers from 0 to 51 in that order; it can represent a standard deck. Now if we have some cards – the full deck or a smaller *hand* – in *list* then we can convert *list* to ranks using *list*`/4`, and we can convert *list* to suits using *list*`%4`.

Fifth Problem: Poker Hands

We consider the problem of drawing a 5 card *poker* hand from a standard deck and evaluating it, which we will limit to testing whether it is a full house or a *flush* – all cards of the same suit.

We do not care what order we draw the cards in, so like dice we can make that more efficient. Our 5 card hand is the list `combine5from[sequence52]`. There are 2598960 ways to draw that hand, which we can handle. However, we might find drawing more cards too slow to use.

Converting to ranks and using what we have done before, we could write the expression:

```
list_eq(groups(combine5from[sequence52]/4),{0,1,1,0,0})
```

This has the `-probability -statistics`: output:

```
Number of evaluations       = 2598960
Number of results           = 2598960
Number of false results     = 2595216
Number of true results      = 3744
Probability                 = 0.00144058 = 6/4165
```

However, this is inefficient. We should convert the cards to ranks before taking our hand, as:

```
list_eq(groups(combine5from[sequence52/4]),{0,1,1,0,0})
```

Now our `-probability -statistics` output, showing a greatly reduced workload, is:

```
Number of evaluations       = 6175
Number of results           = 2598960
Number of false results     = 2595216
Number of true results      = 3744
Probability                 = 0.00144058 = 6/4165
```

Testing for a flush can use `same(combine5from[sequence52%4])`, with the output:

```
Number of evaluations       = 56
Number of results           = 2598960
Number of false results     = 2593812
Number of true results      = 5148
Probability                 = 0.00198079 = 33/16660
```

However, to fully use both suits and ranks from the same hand we need the full 2598960 cases.

Sixth Problem: Bridge Hands

A *bridge* hand has 13 cards. If we care about ranks and suits, e.g. to try to fully assess the value of a hand, we would need to handle 635013559600 cases. We cannot do that exactly.

Instead, here we will just count the maximum number of cards in any suit using the expression `count_mode(combine13from[sequence52%4])`. We use an alternative `-table` to `-all` that can replace its second to fourth outputs. We can add `-statistics` for the first output.

```
 N      P(N)         P(<=N)      P(>=N)
 4   0.350805     0.350805   1
 5   0.443397     0.794202   0.649195
 6   0.165477     0.959679   0.205798
 7   0.0352664    0.994945   0.0403213
 8   0.00466761   0.999613   0.00505489
 9   0.000370445  0.999983   0.000387277
10   1.64642e-05  1          1.68315e-05
11   3.64074e-07  1          3.67274e-07
12   3.19363e-09  1          3.19993e-09
13   6.29908e-12  1          6.29908e-12
```

Additional Lists

So far, we have used random lists such as `sorted3d6`, lists such as `sequence52`, and lists such as `{2,3,5,7}`; we can also use non-constant lists such as `{2,d6,r0+1}`. We can use, for example, `[r0+1][3]` for the list `{r0+1,r0+1,r0+1}`. If we use `[d6][3]` then the d6 is only rolled once and is used for all 3 elements. We cannot use `[3][r0+1]`, because it does not have a fixed length. We can join lists using `#`, so that `{2,3}#[4][3]` is {2,3,4,4,4}. `#` is the highest precedence binary list operator; its operands can have different lengths, as here.

Loops

Expressions now look like a simple *programming language*. But so far they are missing the key programming ability of being able to do something more than once, in other words, a *loop*.

Loops in expressions are possible. They are implemented using functions. But these are not true mathematical functions, they evaluate their arguments more than once, using each value.

We consider two loops. The first is a *constant loop*, the number of times it is used is known beforehand. The second is a *variable loop*, the number of times it is used is not so known.

Constant Loop

One use of a constant loop is to do something for each element of a list. As an example, we assume function `max` does not exist and implement it. For simplicity, unlike `max`, we assume all elements of *list* are greater than zero, so can use the zero initial value of `r0`, the maximum.

The loop used here is `rloop1(size(`*list*`),r2:=get(r1,`*list*`);r2>r0&(r0:=r2))`.

The function `rloop1` (one of ten, `rloop0` to `rloop9`) loops `r1`, the `1` matching `rloop1`, as 0, 1, … `rloop1` has two comma-separated arguments: `size(`*list*`)`, the number of elements in *list*, is how many times we loop, and `r2:=get(r1,`*list*`);r2>r0&(r0:=r2)`, is what we do that many times. The latter demonstrates that we can sequence terms with a `;` anywhere.

Now suppose that, for example, `size(`*list*`)` is 3. What now happens is the sequence:

```
r1:=0
r2:=get(r1,list);r2>r0&(r0:=r2)
r1:=1
```

```
r2:=get(r1,list);r2>r0&(r0:=r2)
r1:=2
r2:=get(r1,list);r2>r0&(r0:=r2)
r1:=3
```

`r1` is the number of each element of *list*, so `r2:=get(r1,list)` is equal to each element of *list*. We then check if `r2` is greater than the maximum so far `r0` using `r2>r0` and only update `r0` if this is true. We need parentheses because all assignments have precedence below `&`.

We finish the expression with `;r0` because `r0` is the required answer. Sequencing rules mean that what is before a `;` must be the same *type* (number or list) or must be an assignment. `rloop1(m,n)` is a number, the last result of *n*, but followed by `;r0` that number is not used.

To only calculate *list* once we need to put it in a variable. We use `v0`, thus giving us:

`v0:=`*list*`;rloop1(size(v0),r2:=get(r1,v0);r2>r0&(r0:=r2));r0`

We can simplify that. There is a notation `s0` for `size(v0)`, and similarly for `v1` etc. There is a notation `e01` for `get(r1,v0)` and similarly for the other 99 possible cases. That gives us:

`v0:=`*list*`;rloop1(s0,r2:=e01;r2>r0&(r0:=r2));r0`

and, even better, we can now get rid of `r2` as:

`v0:=`*list*`;rloop1(s0,e01>r0&(r0:=e01));r0`

Yet better still, but not needed in this tutorial, is that we can assign to `e01`, i.e. use `e01:=`… for some … and thus change one element of `v0`. We can even use assignments like `e01+=`…

In our expression, let *list* be `sorted3d6`. The distribution, the second output from `-all`, is:

```
1 -  1 ~ 0.00462963 = 1/216
2 -  7 ~ 0.0324074  = 7/216
3 - 19 ~ 0.087963   = 19/216
4 - 37 ~ 0.171296   = 37/216
5 - 61 ~ 0.282407   = 61/216
6 - 91 ~ 0.421296   = 91/216
```

This the same output as using the expression `max(sorted3d6)` or `last(sorted3d6)`.

Variable Loop

The most basic variable loop function is `while`. `while(m,n)` is equivalent to:

(a) Set *k* to 0. (*k* is a temporary variable that is just used to explain this loop.)

(b) Evaluate *m*; if it is false then finish with result *k*, otherwise continue.

(c) Evaluate *n* and set *k* to its value.

(d) Go back to (b).

We now use this function, together with some other new features, to solve the next problem.

Seventh Problem: A Race Game

To demonstrate a variable loop and another new feature, we use a simple race game. Players 0 and 1 race down a track with a finish line after 10 spaces. Every turn each rolls a die with faces 2, 3 and 4, i.e. a `d3+1`, and moves that far, at the same time. The game ends when

either or both of them have crossed the line. A player's score is how many spaces travelled minus how many spaces the other player travelled. Both count spaces beyond 10 if reached.

We let player 0 have moved `r0` spaces, and player 1 have moved `r1` spaces. Here is an expression that looks like it should work to give player 0's score:

```
while(r0<10&r1<10,r0+=d3+1;r1+=d3+1);r0-r1
```

But using `-exact` that expression does not work, with the error message:

```
Error: Function while cannot be used that way in exact mode.
```

This is because exact results depend on there being known amounts of randomness, which is not so here. One solution is switching to approximate results. But in this case there is an exact solution possible because there is a maximum amount of randomness. So we act as if we pre-roll all the dice we might possibly want and put those in a *pool*. We then take all of our dice from the pool. The program efficiently handles any remaining unused dice in the pool.

In this game the maximum number of turns possible is 5, so the maximum number of dice needed is 5 each, a total of 10 d3s. We put those 10 dice in the pool with `pool_nset10of3`. When using the pool we replace the use of `d3` by `pool_d(3)` and our expression becomes:

```
pool_nset10of3;while(r0<10&r1<10,r0+=pool_d(3)+1;r1+=pool_d(3)+1);r0-r1
```

We can run this expression with the options `-exact -all` to produce the first output:

```
Number of evaluations      = 3241
Number of results          = 59049
Mean                       = 0
Standard deviation         = 2.12924
Minimum result             = -6
Maximum result             = 6
```

and the second, basic distribution, output:

```
-6 -   81 ~ 0.00137174 = 1/729
-5 -  549 ~ 0.00929736 = 61/6561
-4 - 2070 ~ 0.0350556  = 230/6561
-3 - 4729 ~ 0.080086   = 4729/59049
-2 - 7972 ~ 0.135007   = 7972/59049
-1 - 9644 ~ 0.163322   = 9644/59049
 0 - 8959 ~ 0.151721   = 8959/59049
 1 - 9644 ~ 0.163322   = 9644/59049
 2 - 7972 ~ 0.135007   = 7972/59049
 3 - 4729 ~ 0.080086   = 4729/59049
 4 - 2070 ~ 0.0350556  = 230/6561
 5 -  549 ~ 0.00929736 = 61/6561
 6 -   81 ~ 0.00137174 = 1/729
```

We can use mixed dice in a problem, in any order, by using more than one `pool_nset` term.

Eighth Problem: Conditional Probabilities

Sometimes we might be interested in a problem such as what is the distribution of our D&D characteristics (roll four standard dice and sum the three largest) but where we reroll if none of our dice is a 6. To do this exactly we only roll the dice once, but we discard any rolls without a 6. In this way we are using *conditional probabilities* (these are conditional on at least one 6).

We can do this with `v0:=tail3(sorted4d6);conditional(last(v0)==6,sum(v0))`. Here we only need to check the last die in `v0`, because it is sorted, and that being a 6 is our condition. If that condition is true then we use `sum(v0)` as our result, otherwise we have no result. To save space, here we just show the `-statistics` output, but you would usually produce the `-all` or `-table` output. The 56 tells us that is how many of the 126 different dice patterns had a 6 in them, and the 671 tells us how many of the possible 1296 results (a number that is not reported here) had a 6 in them. The mean has gone up to 13.9314 (from 12.2446).

```
Number of evaluations      = 126
No. of weight > 0 evals    = 56
Number of results          = 671
Mean                       = 13.9314 = 9348/671
Standard deviation         = 2.12811
Minimum result             = 8
Maximum result             = 18
```

Approximate Results

We now consider approximate results. These are produced by what is called *Monte Carlo simulation* (hence the program name). We evaluate the expression using actual random numbers many times, in examples here ten million times. The program *estimates* probabilities, average values and distributions by counting results and performing the required arithmetic.

An important question is how good are these estimates? They obviously get better as the number of results increases. But, for probabilities and distributions, they also get worse if the event they are looking at gets less likely. To use the program properly we should have a realistic assessment as to how accurate our estimates are.

Here I suggest that we do this by quoting results to a reasonable number of significant figures. So if we say that the average is 10.5, then ideally we would believe that all three digits are correct. We cannot actually do this, there is always some uncertainty left. So instead we say that the average is “about 10.5”, meaning that we think it is probably 10.5 to three figures. However, it might be 10.4 or 10.6, or even further from 10.5, but that gets increasingly unlikely as we get further from 10.5. We do not do that with full rigour, but we do it reasonably well.

First, we need to know how to produce approximate results. We start with a known case, the sum of three standard dice. I recommend that you use:

```
monaco -new -all sum(sorted3d6) 10000000
```

Note that `-new` has replaced `-exact` and we have a new parameter after the expression, how many times we are going to evaluate the expression to get the requested `-all` statistics.

However, that is not what is used in this tutorial, which instead is:

```
monaco -all sum(sorted3d6) 10000000
```

Why? Because what `-new` does is to make it such that if you repeat a run, you get new random numbers and new estimates. But here we want examples that you can repeat and compare.

Also, if we produce many approximate results then we should replace `sorted3d6` with `3d6`. This creates an unsorted list; for approximate results this saves an unnecessary sorting step.

The first output, using `sorted3d6` or `3d6`, including reporting the number of results used, is:

```
Number of results          = 10000000
Mean                       = 10.4989
Standard deviation         = 2.95791
```

```
Standard deviation of mean = 0.000935373
95% confidence interval    = [10.497, 10.5007]
Minimum result             = 3
Maximum result             = 18
```

Instead of an exact mean of 10.5, we now have an estimate of the mean that apparently is 10.4989. However, that is not how we should quote it. For how to quote it, we look at the two new lines of output, and the line that we have ignored up to now, the standard deviation.

The standard deviation measures the width of a distribution. We can say some things (more if we knew the distribution) about how many results are within different numbers of standard deviations from the mean. The standard deviation is an estimate; the more results we have, the closer it will get to its true value, which from our earlier exact results is about 2.95804.

The mean is also an estimate, and also has a standard deviation. But as the number of results gets larger, the mean varies less, getting closer to its true value. This is shown in the new line of output for the standard deviation of the mean. That goes down as the square root of the number of results: to make it ten times smaller you need a hundred times as many results.

We do not, in general, know enough about the distribution of the results to use the standard deviation very usefully. (Sometimes we might, so it is there for those cases.) But for large numbers of results the mean tends towards a known distribution, one known as a *normal* distribution. Knowing that, and that ten million is more than large enough, we can do more.

In particular, the program can estimate a *confidence interval* that the true mean should lie in about 95% of the time, and that confidence interval is shown. Why 95% rather than, say 99%? There is no particularly good reason, it is just the most commonly used value.

However, it is easy to misinterpret the confidence interval. Although (about, because this is an estimate) 95% of the time the mean will be in the confidence interval, this is not the same as saying that there is a 95% chance that the mean is in any given confidence interval that the program produces, but explaining why that is so is beyond the scope of this tutorial.

Instead, less precisely, we use the interval as a guide to quote the mean to an appropriate precision. From those results we can quote the mean as “about 10.50”. Why? Because the next values to that precision, 10.49 and 10.51, are both (well) outside that confidence interval.

What about the more precise (although, as we happen to know, less accurate) next estimate of “about 10.499”? The next values to that precision are 10.498 and 10.500, which are both inside the confidence interval. So quoting the mean as about 10.499 would be over-precise.

This is not a rigorous process, sometimes we have to make a judgement call when numbers are just inside or outside the interval. But it is better than just quoting over-precise figures. And to recognise this we always quote estimated figures with that qualifying “about”.

The mean is not the only output with a confidence interval. The second output from `-all` is:

```
 3 -   46637 ~ 0.0046637  [0.00462166, 0.00470612]
 4 -  138764 ~ 0.0138764  [0.0138041, 0.0139491]
 5 -  277086 ~ 0.0277086  [0.0276071, 0.0278105]
 6 -  463492 ~ 0.0463492  [0.0462191, 0.0464797]
 7 -  695550 ~ 0.069555   [0.0693975, 0.0697128]
 8 -  972369 ~ 0.0972369  [0.0970534, 0.0974207]
 9 - 1156903 ~ 0.11569    [0.115492, 0.115889]
10 - 1251252 ~ 0.125125   [0.12492, 0.12533]
11 - 1249253 ~ 0.124925   [0.124721, 0.12513]
12 - 1158734 ~ 0.115873   [0.115675, 0.116072]
13 -  970634 ~ 0.0970634  [0.0968801, 0.097247]
14 -  694195 ~ 0.0694195  [0.0692621, 0.0695772]
15 -  462408 ~ 0.0462408  [0.0461108, 0.0463711]
```

```
16 -  277349 ~ 0.0277349  [0.0276333, 0.0278369]
17 -  138981 ~ 0.0138981  [0.0138257, 0.0139708]
18 -   46393 ~ 0.0046393  [0.00459737, 0.00468161]
```

We should now quote the probability of a 3 as “about 0.0047” and of an 18 as “about 0.0046”, as in the values 0.0046 and 0.0048 in the first case and 0.0045 and 0.0047 in the second case are outside the reported intervals. However, we definitely need the “about” in each case, as by symmetry those two probabilities must be equal, hence one of them must be slightly inaccurate. (We know from the exact result of about 0.00462963 – which is in both confidence intervals – that the true answer to the appropriate precision is about 0.0046.)

Note that `-table` output has no confidence intervals, so we prefer to use `-all` in this case.

Approximate Indefinite Loop

We previously noted that the following expression looked like it ought to work, but did not:

```
while(r0<10&r1<10,r0+=d3+1;r1+=d3+1);r0-r1
```

Now, for approximate results, it does. The second output from `-all` for ten million results is as follows. Comparing it with the exact results, one of the 13 exact probabilities is slightly outside the estimated confidence interval and one is on the edge. This is not surprising.

```
-6 -   13746 ~ 0.0013746  [0.00135183, 0.00139776]
-5 -   93475 ~ 0.0093475  [0.00928805, 0.00940733]
-4 -  350436 ~ 0.0350436  [0.0349298, 0.0351578]
-3 -  801800 ~ 0.08018    [0.0800118, 0.0803485]
-2 - 1349852 ~ 0.134985   [0.134774, 0.135197]
-1 - 1633447 ~ 0.163345   [0.163116, 0.163574]
 0 - 1516919 ~ 0.151692   [0.15147, 0.151914]
 1 - 1634008 ~ 0.163401   [0.163172, 0.16363]
 2 - 1349791 ~ 0.134979   [0.134767, 0.135191]
 3 -  799178 ~ 0.0799178  [0.0797499, 0.080086]
 4 -  350986 ~ 0.0350986  [0.0349847, 0.0352128]
 5 -   92642 ~ 0.0092642  [0.00920501, 0.00932377]
 6 -   13720 ~ 0.001372   [0.00134925, 0.00139513]
```

Approximation Required

We cannot produce exact results when the number of random numbers needed is unbounded. Sometimes we can get good estimates using exact results by capping the number of random numbers. However, an example where that fails is to roll and sum a standard die until we roll a 6, when we stop, not including the 6 in the sum. That sum’s mean can be shown to be 15.

A suitable expression is `while(r0:=d6;r0<6,r1+=r0)`, with `-statistics` output:

```
Number of results          = 10000000
Mean                       = 15.0031
Standard deviation         = 16.741
Standard deviation of mean = 0.00529396
95% confidence interval    = [14.9927, 15.0135]
Minimum result             = 0
Maximum result             = 266
```

If instead we use exact results capping the number of rolls at 12, the expression can become:

```
pool_nset12of6;while(r2+=1;(r2<12)&(r0:=pool_d(6);r0<6),r1+=r0)
```

The `-statistics` output for this example – which is not fast, even with the cap – is:

```
Number of evaluations      = 61035156
Number of results          = 2176782336
Mean                       = 12.9812 = 1569844655/120932352
Standard deviation         = 11.631
Minimum result             = 0
Maximum result             = 55
```

This is the exact result for the capped case but is a bad estimate for the uncapped case, so in this case we need to approximate. (To tell that, adjust the cap and see how the mean varies.)

We also need to approximate when the problem is too big. This applies to some examples using bridge hands. But for a simple example here, we roll a large number of standard dice and simply count 6s. Rolling 24 dice can use `count_eq(sorted24d6,6)`, with exact output:

```
Number of evaluations      = 118755
Number of results          = 4738381338321616896
Mean                       = 4
Standard deviation         = 1.82574
Minimum result             = 0
Maximum result             = 24
```

However, the usual version of the program cannot handle 25 dice and reports a failure. But an approximate run is possible and produces the output:

```
Number of results          = 10000000
Mean                       = 4.16679
Standard deviation         = 1.86334
Standard deviation of mean = 0.000589238
95% confidence interval    = [4.16563, 4.16794]
Minimum result             = 0
Maximum result             = 16
```

The reported maximum result 16 is the most 6s that were seen. Results up to 25 are possible, but need more results to become likely (a result >16 has probability about 1.6×10^{-8}). The mean estimate of about 4.167 (with some doubt about the 7) is close to the true figure of 4⅙ or about 4.16667. Using approximate results we can roll many more dice than the 25 in this example.

Conclusions

The aim of this tutorial is to provide a maximum “bang for the buck” in describing only some features of the program, but chosen to be enough to enable you to answer many questions.

This tutorial has emphasised using the program to give exact answers. It has presented more examples using dice than using cards, largely because dice are easier to get started with.

If you want to know more, there is a much longer main document that describes the program in full detail (some of which I have simplified here). There is also a “teaser” document that includes various real problems that can be solved using the program. Many use new features of the program, but some of them could probably be solved using just what is in this tutorial.

If you have any feedback about this tutorial or any other aspect of the program, please let me know. If you have a problem, let me have a complete description of it and I will try to help.