Monaco for Programmers

Introduction

Monaco is a program whose primary purpose is to answer probability related problems. It is a command line program – any alternative use is just to construct that command line.

For example, if asking "what is the distribution summing the values of three standard six-sided dice?", a run that answers that question is:

```
monaco -exact -histogram 3d6
```

where `monaco` is whatever you use to run the program. `3d6` is an *expression*, written in a language specific to Monaco, but with many parts borrowed from familiar sources. This expression is the main expression; other expressions are beyond the scope of this note. An expression can be evaluated to a value of its type – for the main expression that is always an integer. Most expressions are random, here `3d6` is the sum of three standard six-sided dice.

Before the expression there are *options*. Here, `-exact` is for an exact answer to the question and `-histogram` outputs a distribution of the possible results, which is:

```
 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
```

Other forms of output include forward and backward cumulative histograms using `-cumulative` and `-rcumulative` instead of or as well as `-histogram`.

The alternative to an exact result is an approximate result. For example, also introducing the additional option `-percent` – which can also be used in exact mode – using:

```
monaco -histogram -percent 3d6 10000000
```

the expression is evaluated ten million times, and produces the output:

```
 3 -   46364 ~  0.46364%  [0.459448%, 0.46787%]
 4 -  138809 ~  1.38809%  [1.38086%, 1.39536%]
 5 -  276908 ~  2.76908%  [2.75893%, 2.77927%]
 6 -  462777 ~  4.62777%  [4.61477%, 4.64081%]
 7 -  694454 ~  6.94454%  [6.9288%, 6.96031%]
```

```
 8 -  972865 ~  9.72865%  [9.7103%, 9.74703%]
 9 - 1157432 ~ 11.5743%   [11.5545%, 11.5942%]
10 - 1249310 ~ 12.4931%   [12.4726%, 12.5136%]
11 - 1250542 ~ 12.5054%   [12.4849%, 12.5259%]
12 - 1159223 ~ 11.5922%   [11.5724%, 11.6121%]
13 -  970829 ~  9.70829%  [9.68996%, 9.72666%]
14 -  695689 ~  6.95689%  [6.94114%, 6.97268%]
15 -  462863 ~  4.62863%  [4.61563%, 4.64167%]
16 -  276656 ~  2.76656%  [2.75641%, 2.77674%]
17 -  138966 ~  1.38966%  [1.38242%, 1.39693%]
18 -   46313 ~  0.46313%  [0.458941%, 0.467357%]
```

The figures in `[]` are a 95% confidence interval, suggesting how accurate each probability is.

For approximate results, that is always the output for that run, because the random number generator starts at a fixed point. For a different run each time add the option `-new`.

The Structure of an Expression

The expression is constructed from *terms*, which are anything that can be evaluated. Each term has a *type*: integer, list (of integers), real number or string, but only the first two types are considered here. Terms can be combined using *operators* and *functions*. Terms can be sequenced separated by semicolons, the value of such a sequence is the value of its last term.

Operators combine terms of the same type, producing a term of the same type. For example, the expression 3d6>=2d10 contains the terms 3d6 and 2d10 combined using the operator >=. Logical values are as the C programming language – zero is false, anything else is true, but a true result is always one. The operators are modelled after those in standard C/C++, with some changes (logical operators are not doubled) and some additions (including ~ @ # ?< ?> and ??).

The question to be answered here is how often 3d6 is greater than or equal to 2d10. We also replace `-histogram` by `-probability` and `-statistics` and the exact output is:

```
Number of results          = 21600
Number of false results    = 10800
Number of true results     = 10800
Probability                = 0.5 = 1/2
```

A function has a name followed by comma separated arguments in parentheses (). If the function has no arguments the parentheses can be omitted. Otherwise, each argument has a type. Argument types can be different from each other and from that of the function, i.e. the type of its value. When an expression is parsed it is always known which type of term is expected. There is an exception to this rule, the assignment of a variable, considered below.

For example, product(3d6) is a valid term, as product is a function with a list argument and an integer value. This 3d6 is not the sum of three dice but is a list of three dice values. Monaco can tell the difference between those because it always knows which type of term is expected.

Here we assume that we are interested in the average value of that product, and we use the option `-statistics` as well as `-exact` to produce the output:

```
Number of results          = 216
Mean                       = 42.875 = 343/8
```

```
Standard deviation          = 40.6262
Minimum result              = 1
Maximum result              = 216
```

Randomness can also be introduced by functions. For example, the function shuffle creates a random reordering of a list, thus shuffle{2,3,5,7} has 24 possible values. This term uses that a single braced list argument that could be used as ({...}) can be replaced by {...} for any ...

Lists

Lists are of integers. All lists have a fixed length; for example, the list 3d6 has length 3. Three other examples of lists with length 3 are {d6,2d6+1,3d6+3}, 4[3], which is equivalent to {4,4,4}, and sequence3, which is equivalent to {0,1,2}. List lengths can be omitted when they can be deduced; for example, sequence3+4 can be used instead of sequence3+4[3].

Variables and Assignment

There are ten integer variables, called r0 to r9, and ten list variables, called v0 to v9, that can be used in an expression.

We assign to a variable with, for example, v0:=3d6, or a modifying assignment, for example v0+=3d6. List variables have a constant length, so, for example, once the length of v0 is set it cannot be changed. Either of those assignments sets the length of v0 to 3 if not already set.

Assignments are terms; the value of an assignment term is the new value of the assigned to variable. Assignments can be used in a semicolon separated term sequence, even if a different term type is expected. However, the last term in such a sequence should have the expected type. For example, v0:=3d6;sum(v0*v0) is a valid integer term. (Like other list operators, * works element by element, and sum is similar to product.)

There are also some derived variables. For example, e01 is element r1 of the list v0. It can be used to extract information from v0, or to set individual elements of v0 as r1 is varied.

Before each evaluation of the expression, integer variables are all set to zero and list variables are all set to a list with the appropriate number of all zero elements.

Constants

There are also ten integer constants, c0 to c9, and ten list constants, u0 to u9. These can only be set once, at the start of the expression and followed by one or more semicolon separated terms. Also, the constants s0 to s9 are the lengths of the variable lists v0 to v9; these can be set to define the list length. The lengths of u0 to u9 are k0 to k9, but these cannot be set. In addition, for example, the variable i01 is element r1 of u0; this cannot be modified.

Branching and Loops

Branches and loops are implemented within the expression, not outside it.

Branching can be by using operators, as in C/C++. The ternary ?: operator evaluates either its second or third operand according to the value of its first operand (true or false). Also, the logical and & and inclusive or | operators only evaluate their second argument if needed.

One form of loop uses the @ (accumulation) operator. This evaluates its first operand, and then evaluates its second operand that many times, summing them. For example, 3@d6 is the same as 3d6. However, for d6@d6, the number of dice rolled is variable. That means that term cannot be used in exact mode. (However, that case can be analysed exactly because the randomness is bounded. The best way uses the *randomness pool*, but is beyond the scope of this note.)

Other loops use functions. There are many loop functions, both looping in different ways and providing cases such as summing or taking the maximum over a loop to simplify the expression. However, this note just presents two loops to show the basic principles.

The first example loop rolls a d6 until a 6 is rolled, accumulating the results without the final 6, using the term `while(r0:=d6;r0!=6,r1+=r0)`. The value of this term is the last value of its second argument. This is the required sum if that is the complete expression, using that `r1` is initialised to zero. This example of this loop cannot be used in exact mode.

The second example loop is through the list `v0`, with length `s0`. We assume there is also a constant list `u0` with at least the same length. We replace each element of `v0` by its square or cube if the corresponding element of `u0` is true or false. As `s0` and `u0` are not initialised here, this is only a term, not a complete expression: `rloop1(s0,e01*=(i01?e01:e01*e01))`, which loops `r1` from 0, inclusive, to `s0`, exclusive. The term's value is again the final value of the second argument, but that is not very useful here (nor is the whole term).

Faster Exact Results

Rolling nine standard dice the number of different values rolled is `count_diff(9d6)`. On my computer (as for all times here) that takes just under 2 seconds. We can greatly speed that up using `count_diff(sorted9d6)`. The list term `sorted9d6` assumes that which die is which has no effect, and so generates just one representative of each significantly different set of dice, the one where they are sorted in non-descending order. It handles that not all sets of dice are equally likely. The time is reduced to about a millisecond. Increasing 9 dice to 12 dice the time is reduced from about 6 minutes to 2.5 milliseconds, which is by five orders of magnitude.

There are more terms like sorted9d6 for other cases, including drawing cards as well as rolling dice, and for other forms of symmetry than the "all dice are interchangeable" used here. They rely on the user to identify which term is appropriate, but the gains can make that worthwhile.

References

Monaco's runs can include many more instances of things mentioned in this note, including options, named entities, random terms, operations and functions. For an introduction to some of them there are three tutorial documents, but for a full reference to all features the extensive full documentation is required; see its Section 1.2.2 for a starting point. The program itself also provides its own documentation; as a starting point for that use `monaco -help`.

Contact Information

Monaco was created by Christopher Dearlove, christopher.dearlove@gmail.com. Further information is available at `http://www.mnemosyne.uk/monaco`.