Monaco

version 2.17

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

Christopher Dearlove
christopher.dearlove@gmail.com

© Christopher Dearlove, 2008-2023

What these slides include

Eight sections:

- What the program is for.
- The first approach: estimation.
- The second approach: exact.
- How to use the program.
- Example output.
- Faster exact results.
- So why not always exact?
- Final comments.

© Christopher Dearlove, 2008-2023

What the program is for

© Christopher Dearlove, 2008-2023

An example problem

When playing games, sometimes you want to know how likely something is.

For example, in the games *Yspahan* and *Corinth* you usually roll nine standard dice. (You can also buy more, later we will consider twelve dice.)

One thing useful to know is the probability of rolling all six values. (Later we add other cases.)

© Christopher Dearlove, 2008-2023

Possible ways to get an answer

You could work that out theoretically, but:

- it is a little tricky, and tedious, and
- there are much harder interesting problems.

You could write a program to solve the problem.

- That would take some time to do, and you would need a new program for each problem.

Or there are two other approaches, either or both of which this program can do for you.

© Christopher Dearlove, 2008-2023

Exact and estimated answers

These two approaches provide *exact* answers and *estimated* (approximate) answers.

Exact answers are obviously better.

However, the first approach described here is that providing estimated answers, because the explanation is easier that way.

Why have estimated answers at all? Because not all problems can be solved quickly and exactly.

© Christopher Dearlove, 2008-2023

The first approach: estimation

© Christopher Dearlove, 2008-2023

The first approach: estimation

Here is the first approach:

- Roll the dice and count the number of values.
- Do it again. And again. And again.
- Do it many times, and produce suitable statistics from the results.

Using this program you can do that – or its electronic equivalent – many millions of times to get a good estimate of the required answer.

© Christopher Dearlove, 2008-2023

A good enough answer

I just rolled the dice a million times.

- Exactly how I will describe later in these slides.

The answer, for all six values, was about 18.9%.

Do you want it more accurately than that?

I just rolled the dice a hundred million times.

- That took me about twenty seconds to run.

The answer was about 18.90%.

That should be good enough for most purposes.

© Christopher Dearlove, 2008-2023

How good is the answer?

In the last slide the probability was said to have improved from about 18.9% to about 18.90%, or as probabilities from about 0.189 to 0.1890.

How do we know that?

Each of those estimates came with a *confidence interval*, the first with [0.188208, 0.189742].

This means that the exact answer is probably in that interval. 0.188 and 0.190 are outside that interval, so it is reasonable to say “about 0.189”.

© Christopher Dearlove, 2008-2023

The second approach: exact

© Christopher Dearlove, 2008-2023

The second approach: exact

Here is the second approach:

- Try all the possible rolls of the dice.
- There are 6^9 possibilities, or about ten million.
- That gives us our exact answer (which is 245/1296 or about 0.189043 or 18.9043%).

Obviously this way is better.

Except …

© Christopher Dearlove, 2008-2023

Exact is not always possible

Some problems cannot be solved exactly, others are too big; there are examples later.

For example, suppose, as sometimes you can choose in the game, you instead roll twelve dice.

The number of possibilities is over two billion.

That takes me about ten minutes.

Approximate looks more attractive now.

But you can make this much faster, see later.

© Christopher Dearlove, 2008-2023

How to use the program

© Christopher Dearlove, 2008-2023

How to use the program

Regardless of which approach you take, you:

- Express the problem in a way the program can handle.
- Run the program, specifying what you want to know.
- Get the statistics output from the program.

The only difference is telling the program which approach to use, and how often in the first one.

© Christopher Dearlove, 2008-2023

Specifying the problem

We start with standard gaming notation.

- The result of a six sided die is denoted `d6`.

We want nine of these, which we will put in what is called a *list*.

- One way to write the list we want is `{d6,d6,d6,d6,d6,d6,d6,d6,d6}`.

But it is too easy to get the number of dice wrong when writing or reading such a list.

© Christopher Dearlove, 2008-2023

Specifying the problem (cont.)

An alternative notation allows us to write `9d6` for the same list.

Then we want to know how many different values there are in that list.

Another piece of notation allows us to write `count_diff(9d6)` for that.

This notation expresses what we want to try many times, and we call it an *expression*.

© Christopher Dearlove, 2008-2023

Running the program

We run the program by typing a command line.

- I use an alias `monaco` to run the program using macOS (Unix). Here we use `monaco` to mean that or whatever you have to type.

We then need to type in, on the same line:

- Some *options*, to be described, and
- the expression, and, using the first approach,
- the number of times you will try the problem.

© Christopher Dearlove, 2008-2023

Expression, options, number

We have an example expression.

- I need to put it in quotes `'…'` or "…". You may need to do that, and/or some other things.

If not exact we need the number, e.g. `1000000`.

Which leaves the options.

- Include the option `-exact` for exact answers.
- Here I just describe some options that allow you to produce particular forms of output.

© Christopher Dearlove, 2008-2023

Output options

For the average (mean) add `-statistics`.

To know how often each result happened, add `-histogram`. That reports how often the result is equal to each number that occurs.

To report how often the result is each number or less, or the result is each number or more, add `-cumulative` or `-rcumulative`.

For output as percentages, add `-percent`.

© Christopher Dearlove, 2008-2023

Putting that all together

If I just want e.g. the `-rcumulative` statistics, on my computer I can type, all on one line (split here for space) one of the two commands:

```
monaco -rcumulative -percent
'count_diff(9d6)' 1000000
```

```
monaco -exact -rcumulative -percent
'count_diff(9d6)'
```

You may need to use something else as the program name, and may not need the quotes.

© Christopher Dearlove, 2008-2023

Example output

© Christopher Dearlove, 2008-2023

Estimated statistics

Estimated statistics using `-rcumulative`:

```
>= 2 - 1000000 ~ 100%       [99.9995%, 100%]
>= 3 -  999289 ~ 99.9289%   [99.9235%, 99.9339%]
>= 4 -  963006 ~ 96.3006%   [96.2634%, 96.3374%]
>= 5 -  685268 ~ 68.5268%   [68.4357%, 68.6178%]
>= 6 -  188974 ~ 18.8974%   [18.8208%, 18.9742%]
```

This reports estimated probabilities; the true probabilities are probably in the `[…]` confidence intervals.

© Christopher Dearlove, 2008-2023

Exact statistics

Exact statistics using `-rcumulative`:

```
1 -       6 ~ 5.95374e-05% = 1/1679616
2 -    7650 ~ 0.0759102%   = 425/559872
3 -  363000 ~ 3.60201%     = 15125/419904
4 - 2797200 ~ 27.7563%     = 6475/23328
5 - 5004720 ~ 49.6614%     = 11585/23328
6 - 1905120 ~ 18.9043%     = 245/1296
```

You can compare the estimated statistics with these. (There were too few estimated results to see any 1s.)

© Christopher Dearlove, 2008-2023

Reporting the average (mean)

Exact:

```
Number of results           = 10077696
Mean                        = 4.83716 = 8124571/1679616
Standard deviation          = 0.768773
Minimum result              = 1
Maximum result              = 6
```

Estimated:

```
Number of results           = 1000000
Mean                        = 4.83654
Standard deviation          = 0.769106
Standard deviation of mean  = 0.000769106
95% confidence interval     = [4.83503, 4.83804]
Minimum result              = 2
Maximum result              = 6
```

© Christopher Dearlove, 2008-2023

Another way to present statistics

Exact statistics using the option `-table`:

N	P(N)	P(<=N)	P(>=N)
1	5.95374e-07	5.95374e-07	1
2	0.000759102	0.000759697	0.999999
3	0.0360201	0.0367798	0.99924
4	0.277563	0.314343	0.96322
5	0.496614	0.810957	0.685657
6	0.189043	1	0.189043

This can replace all histogram options if decimal values are all that is wanted. Note that there are no confidence intervals, so only use when exact.

© Christopher Dearlove, 2008-2023

And with 12 dice?

We can use `count_diff(12d6)` as the expression.

The probability of getting all 6 values goes up to about 43.8%, and of getting 5 or 6 values goes up to about 89.4%.

An exact answer takes longer than most of us have patience for. But there is a better way.

© Christopher Dearlove, 2008-2023

Faster exact statistics

© Christopher Dearlove, 2008-2023

Sorted dice

We got exact statistics by trying all possible rolls.

But we do not care about the order of the rolls.

So we could just consider each possible set of dice values once, in sorted order is convenient.

Not all sets are equally likely. But the program handles that for you if you use the expression `count_diff(sorted9d6)`.

© Christopher Dearlove, 2008-2023

Example output

That is a lot faster, it takes milliseconds rather than minutes. To understand why, the `-statistics` output is now:

```
Number of evaluations       = 2002
Number of results           = 10077696
Mean                        = 4.83716 = 8124571/1679616
Standard deviation          = 0.768773
Minimum result              = 1
Maximum result              = 6
```

The time taken depends on the `2002`, the number of sets of values, not on the `10077696`.

© Christopher Dearlove, 2008-2023

Other faster results

Sorting dice is not the only way to get faster results.

In the *Yspahan*/*Corinth* example we also do not care which number is which and can use, with 12 dice, `pattern_list12rand6`.

Other *symmetries* that can be used include if a die has repeated faces, and we do not care which face was rolled, only what was on it.

© Christopher Dearlove, 2008-2023

So why not always exact?

© Christopher Dearlove, 2008-2023

Not all problems work exactly

Problems with an indefinite length do not work. For example, if I roll a d6 until I get a 6, summing all values before that, what is my average result?

Some problems are too big even with a weighted approach. An example is next. (This problem can be solved exactly with bigger integers.)

This next example also shows a more difficult expression. They can get much more difficult.

© Christopher Dearlove, 2008-2023

Dealing a standard deck of cards

Dealing the deck to 4 players what proportion of time does at least someone get a 7+ card suit?

14.6%, higher than most people guess. People do not shuffle well, but the computer does.

An expression that gives the longest suit length:

```
u0:=sequence52/13;count_mode(shuffle(u0)+u0*4)
```

I am not expecting that expression to be obvious, but nor am I going to explain it here.

© Christopher Dearlove, 2008-2023

The corresponding output

Using `-rcumulative` over 100,000,000 deals:

```
>=  4 - 100000000 ~ 1           [1, 1]
>=  5 -  97073396 ~ 0.970734    [0.970701, 0.970767]
>=  6 -  56849959 ~ 0.5685      [0.568403, 0.568597]
>=  7 -  14634185 ~ 0.146342    [0.146273, 0.146411]
>=  8 -   1986128 ~ 0.0198613   [0.019834, 0.0198886]
>=  9 -    154871 ~ 0.00154871  [0.00154102, 0.00155644]
>= 10 -      6830 ~ 6.83e-05    [6.66992e-05, 6.99392e-05]
>= 11 -       144 ~ 1.44e-06    [1.22245e-06, 1.69597e-06]
>= 12 -         1 ~ 1e-08       [0, 6.27034e-08]
```

Of course a 13 card suit is possible, but none happened in those hundred million deals.

© Christopher Dearlove, 2008-2023

Final comments

© Christopher Dearlove, 2008-2023

What else?

A lot more, including:

- More output options – including forms of user-defined output for when you need it.
- Options for various non-output purposes.
- A complete expression “language”.

The expression language includes both features directly applicable to dice and card problems, and general features to solve new problems.

© Christopher Dearlove, 2008-2023

What use is the program?

Some examples of problems I have analysed:

- Determining some probabilities in dice games such as *Yahtzee* and *To Court the King*.
- How likely you are to fail playing *Can't Stop*.
- Finding the suit distribution of *bridge* hands.
- Finding hand distributions in *poker* and *cribbage*, and comparing *Texas hold'em* hands.

All games are referenced in the documentation.

© Christopher Dearlove, 2008-2023

Where next?

There are three documents about the program:

- The main document, with explanatory material and extensive reference material.
- A much shorter getting started tutorial.
- A “teaser” file with results but no explanations just to show what the program can do.

If you are interested, please get in touch.

© Christopher Dearlove, 2008-2023

And finally

Why *Monaco*?

The way of estimating result statistics by running a simulation of a problem many times is known as *Monte Carlo simulation*.

And Monte Carlo is in Monaco.

© Christopher Dearlove, 2008-2023