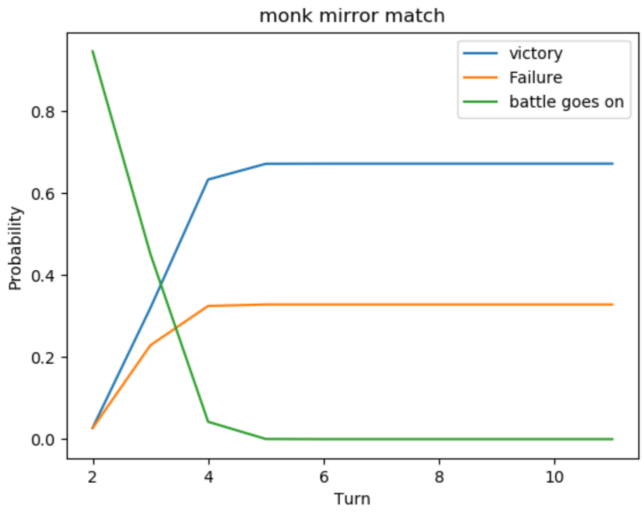
How the probability code works

Frederik M. Jørgensen,

February 21, 2019



mirror match.png

1 Purpose of the code

The code is written in Python 3, and uses the libraries NumPy, Math, and MatPlotLib.
The main purpose of the code is, simply put, to calculate the probability of victory versus failure in a duel in Dungeons and Dragons as a function of turns. Such a calculation can be seen in the above figure, which shows a calculation of a character versus itself, where the ”victory” graph refers to the victory chance of the one who goes first, and the ”failure” refers to the victory chance of the one who goes second.
In order to understand the boundaries of the calculated event, we will start by stating some general rules of a duel:

- The game is turn based, so somebody goes first. This is the one whose chance of victory we calculate.
- A turn consists of a sequence of attacks from each participant.
- Each participant does its sequence of attacks by rolling some dice on its part of the turn. The values of the dice is related to the damage dealt, resulting in a probability distribution for the damage the attacks deal.
- Each participant has some amount of health points (HP).
- One side achieves victory when the other side has taken an amount of damage that is greater than or equal to its HP.
- When one side has won, the duel ends. If one wins, the other cannot win.

With these rules in mind, we can begin to understand why probabilistic calculations are necessary, as the many combinations of dice rolls lead to many outcomes. Sure, one can look at averages, but the average does not give one a win percentage. For a given damage distribution per turn for each character, as well as their HP, one can then calculate these probabilities.

2 Calculating the three states

The duel allows for three states that are always present in each turn: There is some probability W that the one who goes first has won, there is some probability L that the other person has won, and there is some probability C that the game is still ongoing.
There is a symmetry between W and L, in that they only differ by having different damage distributions and HP, and that one goes first. Therefore, the way to calculate W and L are very similar.
To calculate W, let us pretend that you and I duel, and let's say you go first. The chance that you have won after round 1 is equal to the probability that your character knocked my character out in round 1. What is the chance that you have won after round 2?
The chance that you have won in round 2 is equal to the chance that you win in round 1 plus the chance that you win in round 2, but what is the chance that you win in round 2? If you answered "it's the chance that I knock you out with my second attack," then you are mistaken.
There's an important rule about probabilities that states: **If two things, A and B have probabilities a and b of happening, the chance that A and B both happen is the product of a and b.**
If you have some W_n probability of having won after turn n, and some C_{n-1} probability of the game still going on right before your n'th attack, and the probability w_n of knocking your opponent out in your n'th attack, then a relation can be written:

$$W_n = W_{n-1} + C_{n-1}w_n$$

In other words, we multiply an attack with the probability of getting to make that attack.
What of the chance that the game still goes on, then? Well, if we for simplicity assume that you are not taking damage while attacking from some magical effect, then the likelihood that the game is still ongoing after the n'th attack (let's call that c_n) is

$$c_n = C_{n-1}(1 - w_n)$$

We can write this better by utilising vectors and matrices:

$$\begin{pmatrix} W_n \\ c_n \\ L_n \end{pmatrix} = \begin{pmatrix} 1 & w_n & 0 \\ 0 & (1-w_n) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} W_{n-1} \\ C_{n-1} \\ L_{n-1} \end{pmatrix}$$

Let us interpret this for a second. This operation that you do on the three states moves probability from "the game is still ongoing" to "You have won," but this is only half a turn, and now it's my turn. By symmetry, I must make a similar operation of the state, but in stead of moving probability from the middle element of the vector to the top element of the vector, I move it to the lowest. Therefore, if I have probability k_n of knocking you out with my n'th attack, the entire turn will look like this:

$$\begin{pmatrix} W_n \\ C_n \\ L_n \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & (1-k_n) & 0 \\ 0 & k_n & 1 \end{pmatrix} \begin{pmatrix} 1 & w_n & 0 \\ 0 & (1-w_n) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} W_{n-1} \\ C_{n-1} \\ L_{n-1} \end{pmatrix}$$

or in more compact notation:

$$\vec{v}_n = M_n Y_n \vec{v}_{n-1}$$

There is here an unspoken rule that is used:

$$\vec{v}_0 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

because the probability of anybody having won at the beginning of the game is always 0.

2.1 Steady states

In thermodynamics and statistical physics, the things of greatest interest is "steady states," which are states that do not change, and are therefore there after an infinitely long time. We're trying to find the "end result" of the battle, and therefore we are looking for some steady state, because if the probability of winning changed significantly, we wouldn't be done with our calculation.
In terms of this, we're looking for a state where

$$\vec{v}_N = M_N Y_N \vec{v}_N$$

In other words, an eigenvector with eigenvalue 1.
We can, however, see that any vector where $C_N = 0$ is an eigenvector with eigenvalue 1 to any of the two matrices.
Fortunately, we notice that, due to the way probabilities have to work, $0 \leq w_n \leq 1$ and $0 \leq k_n \leq 1$ $\forall n$, which means that:

$$C_n = (1 - k_n)(1 - w_n)C_{n-1} \leq C_{n-1}$$

Furthermore, as the game progresses, the likelihood of somebody knocking somebody out increases, so therefore the system converges towards a steady state.

2.2 Summary and epilogue of three states

The entire thing is carried out by the code, and we rewrite our formula to

$$\vec{v}_n = \left(\prod_{i=1}^{n} M_i Y_i \right) \vec{v}_0$$

so that we always have a vector with the current chance of victory, chance that the game is still ongoing, and chance of loss after each operation. Practically, the calculations are continued for some amount of rounds until the center element of the vector becomes small, as any change in the victory and loss states is then proportional to a small number.
The exact values of w_n and k_n directly depend on the distributions that for each character's attacks, and therefore figuring out how to make a distribution is going to be most of the remainder of this document.

3 Making a distribution

A typical attack consists of rolling a 20-sided die (abbreviated 1d20 - one-d-twenty), adding your "attack score" (abbreviated atk), and then comparing it to the armor class (AC) of the one whom you attack. This leads to three options:

- A miss if $1d20 + atk < AC$ or if $1d20 = 1$
- A critical hit (abbreviated crit) if $1d20 = 20$
- A hit otherwise

If it is a miss, then the attack deals 0 damage. If it is a hit, then one rolls some damage dice, sums the dice, and adds a number called the "flat damage" (also called flat) to the result. If it is a critical hit, then one rolls twice as many damage dice and adds the flat damage.
These are the rules of an attack. Next is to describe distributions in terms of something that we can calculate.
This is done my making a vector $\vec{P}$, whose i'th element, p_i is the probability of dealing i damage.
An ordinary die of N sides is a flat distribution that has entries: $p_i = \frac{1}{N}$.
Thus, if one rolls a six sided die and adds 1 number of flat damage, the resulting distribution would have elements:

$$\begin{aligned}
&\vec{P} = \begin{pmatrix} 0 & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \end{pmatrix} \\
&q_{i+1} = p_i \\
&\vec{Q} = \begin{pmatrix} 0 & 0 & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \end{pmatrix}
\end{aligned}$$

or alternately

$$q_i = p_{i-1}.$$

This obviously makes for a problem if the vectors are of finite elements, as can be seen from the relations, as $\vec{P}$ and $\vec{Q}$ have different amounts of elements, while having a simple relation to each other, but in the mathematics, we can think of the vectors as being with infinitely many elements and indexes going into the negatives, and simply infer that $p_i = 0$ if $i < 0$, for all of our damage distributions. It is done in such a manner that a six sided die would have infinitely many zeroes in each end, and in the 1st to 6th index it would have 1/6. As such, we will only show from element 0 to the element after which we have an infinite amount of zeroes.
In that case, adding a number, F, is equivalent to shifting a vector F indexes to the right, so that is the first function to define - the **shift** function:

$$\begin{aligned} if\ q_{i+F} &= p_i \\ then\ \vec{Q} &= \mathbf{shift}(\vec{P}, F) \end{aligned}$$

This is the first thing we do, because we use this to add dice together.
In the code, we simply insert an array of zeroes of appropriate length in the beginning of $\vec{Q}$, and copy $\vec{P}$ and insert that after the zeroes.

3.1 Adding dice - the matfold operator

When you add two dice together, and you want the probability of each outcome, then you must remember the important rule of probabilities:
If two things, A and B have probabilities a and b of happening, the chance that A and B both happen is the product of a and b.
So if we roll two six sided dice (2d6), labelled $\vec{P}$ and $\vec{Q}$, and add them together, then we calculate the probability of each possible outcome, and add the probabilities of identical outcomes together. We write the resulting vector, $\vec{Z}$ as $\vec{Z} = \mathbf{matfold}(\vec{P}, \vec{Q})$, where the **matfold** operator is some sort of special addition that happens when you sum the eyes on two dice.
The chance that the first one rolls 1 and the second rolls 1 is $p_1 q_1 = \frac{1}{6}\frac{1}{6}$, the chance that the first one rolls 2 and the second one rolls 1 is $p_2 q_1 = \frac{1}{6}\frac{1}{6}$, the chance that the first rolls 1 and the second rolls 2 is $p_1 q_2 = \frac{1}{6}\frac{1}{6}$ and so on.

$$\begin{aligned} z_0 &= 0 \\ z_1 &= 0 \\ z_2 &= \frac{1}{6}\frac{1}{6} = \frac{1}{36} \\ z_3 = \frac{1}{6}\frac{1}{6} &+ \frac{1}{6}\frac{1}{6} = \frac{2}{36} \end{aligned}$$

And so on, but if we want to add more complicated distributions together like we add the eyes on the dice (which is necessary in the long run), then we must think harder and realise the underlying mechanics of this strange kind of addition.
We understand that we want to add our probabilities so that $p_i q_j$ contributes to z_{i+j}. In other words: As long as i and j sum to x, it contributes to z_x. This can be formulated as such:

$$\begin{aligned} i + j = x &\Rightarrow j = x - i \\ z_x &= \sum_i p_i q_{x-i} \end{aligned}$$

Here we might see that the **shift** operator is more important than previously seen, because if we write $\vec{T_i} = shift(\vec{Q}, i)$, then $t_x = q_{x-i}$, and therefore, if we think hard about it, we will realise that:

$$\begin{aligned} \vec{Z} &= \sum_i p_i \mathbf{shift}(\vec{Q}, i) \\ &= \mathbf{matfold}(\vec{P}, \vec{Q}) \end{aligned}$$

This is the definition of the **matfold** operation. We note that we could just as well have written $i = x - j$, leading to $\vec{Z} = \sum_j q_j \mathbf{shift}(\vec{P}, j) = \mathbf{matfold}(\vec{Q}, \vec{P})$, so rolling dice 2 after dice 1 is the same as rolling dice 1 after dice 2 in the dice addition process.

This also exemplifies the important rules of probabilities:

$\vec{Z}$ is the sum over i of $\vec{Q}$ being shifted with i multiplied with the probability p_i that it is shifted with i, because the distribution $\vec{Q}$ is *only* shifted with i when the distribution $\vec{P}$ results in index i, which it does with probability p_i, so it is a case that the two events necessarily happen simultaneously, and our rule holds with distributions as well as the indices in the distributions.

3.2 Hit or miss - the activate function

Now that we know the general gist of how to add dice together, and how to add a number to them, it's time to understand how to take into account that you can miss and critically hit.
This is all taken care of by the **activate** function.
It takes in the following inputs:

- The AC of the enemy
- The atk bonus of your particular attack
- The dice distribution you make on an ordinary attack, $\vec{D}$.
- The flat damage, F.

It uses the atk bonus and AC to calculate the probability that you hit (H), the probability that you miss (M), and has a stationary 1/20 chance to critically hit (C=1/20).
It then makes a miss function, $\vec{Z}$, that has the element $z_0 = 1$ and the rest of the elements are 0.
It expands the length of the distributions $\vec{D}$ and $\vec{Z}$ to match the amount of elements that you have when you critically hit, as this must necessarily have more elements (although in the mathematics we think of them as being of equal dimensions).
Then it, once again, uses the *very important rule of probabilities:*
If two things, A and B have probabilities a and b of happening, the chance that A and B both happen is the product of a and b, which leads to the following resulting distribution:

$$\begin{aligned}\vec{P} &= M\vec{Z} + H\textbf{shift}(\vec{D}, F) + C\textbf{shift}(\textbf{matfold}(\vec{D}, \vec{D}), F)\\ &= \textbf{activate}(\vec{D}, F, AC, atk)\end{aligned}$$

It reads as:
"There is M chance to have the $\vec{Z}$ distribution, H chance to have the $\textbf{shift}(\vec{D}, F)$ distribution, and C chance to have the $\textbf{shift}(\textbf{matfold}(\vec{D}, \vec{D}), F)$ distribution. The total distribution is the sum of all the possible distributions, weighed with their probability of occurring."
Recall, that the critical hit means that one rolls twice as many damage dice as on an ordinary hit, resulting in the distribution $\textbf{shift}(\textbf{matfold}(\vec{D}, \vec{D}), F)$. The equations for M and H are simple. Recall that one only have to match the AC with ones attack bonus plus the d20, a one always misses, a 20 always critically hits, and the problem reduces itself to just counting the number of outcomes, and dividing with 20.
If $1 \leq AC - atk - 1 \leq 19$, then: $M = \frac{1}{20}(AC - atk - 1)$, $H = 1 - M - \frac{1}{20}$.
If $AC - atk - 1 < 1$, then $M = \frac{1}{20}$, $H = \frac{18}{20}$.
If $AC - atk - 1 > 19$, then $M = \frac{19}{20}$, $H = 0$.
C is always $\frac{1}{20}$ when rolling only 1d20.

3.3 Distribution for a single and multiple attacks

When we put this together, we understand that if one person makes an attack against 18 AC, with an attack bonus of 5, and which deals 2d6+5 on a hit, we'll have to start out by adding together the die. This is, in the code, reduced to a function called **Ndice**(N,d), so that the distribution of 2d6 is:

$$\begin{aligned}\vec{D} &= \textbf{Ndice}(2, 6)\\ &= (0, 0, \frac{1}{36}, \frac{2}{36}, \frac{3}{36}, \frac{4}{36}, \frac{5}{36}, \frac{6}{36}, \frac{5}{36}, \frac{4}{36}, \frac{3}{36}, \frac{2}{36}, \frac{1}{36})\end{aligned}$$

Which makes the distribution for a six sided die, and then adds **matfold**s it with itself to get the resulting distribution.
The final distribution for this attack is then popped into the **activate** function,

$$\vec{P} = \textbf{activate}(\vec{D}, 5, 18, 7)$$

which takes care of the rest.
When dealing with multiple attacks, and one is interested in the distribution for how much damage a character has taken, we can realise the following:
The mathematics of adding die is the mathematics for adding outcomes of discrete probability distributions, and is therefore not restricted to ordinary dice.
Alternately, we can think of the distribution for a single attack as a strange die, in fact, because it is made up of dice and by multiplying it with rational numbers, there is a typically large number that one can multiply the distribution with to make every element in it an integer, at which point it would look like a die with many sides being the same, so it must necessarily follow the same mathematics, when adding damage from damage distributions together, as for adding the eyes of several dice.
In total, the distribution for two identical attacks is:

$$\vec{P}_2 = \textbf{matfold}(\vec{P}, \vec{P})$$

For three identical attacks:

$$\vec{P}_3 = \textbf{matfold}(\vec{P}_2, \vec{P})$$

and so on.
The probability distribution for multiple attacks is the probability distribution for how much damage a character has taken after those attacks.

4 Distributions for multiple turns

Now that we have a probability distribution for the damage, we can easily figure out the three states.
When you have made your first attack with the distribution $\vec{P}$, the probability that you *don't* knock out your opponent is trivial if you know the opponent's HP:

$$S_1 = \sum_{i=0}^{HP-1} p_i$$
$$w_1 = 1 - S_1$$

So your first turn's operation on the three states would look like this:

$$\begin{pmatrix} 1 & w_1 & 0 \\ 0 & (1-w_1) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1-S_1 & 0 \\ 0 & S_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A problem arises when you want to calculate your 2nd turn and beyond. To understand this issue, we must ask: what is the damage distribution, $\vec{Q}$, that the opponent has taken after your 2nd turn? If we say $\vec{Q} = \textbf{matfold}(\vec{P}, \vec{P})$, we can run into a problem. This problem is that **the opponent only takes damage from your 2nd turn if it survived the 1st turn,** meaning that it necessarily survived the 1st attack, and therefore did not take damage higher than its HP.
We resolve this by writing our attack $\vec{P}$ as the sum of two un-normalised distributions: $\vec{S}$, which contains the distribution that is survivable by your opponent, and $\vec{D}$, which contains the distribution that knocks your opponent out. In essence:

$$\vec{P} = \vec{S} + \vec{D}$$
$$s_i = p_i \ for\ i < HP$$
$$s_i = 0 \ for\ i \geq HP$$
$$d_i = 0 \ for\ i < HP$$
$$d_i = p_i \ for\ i \geq HP$$

If the opponent made it to your 2nd turn, it must necessarily have taken the damage distribution

$$\hat{S} = \frac{\vec{S}}{\sum_{i=0}^{\infty} s_i}$$

in the first round.
Therefore:

$$\vec{Q} = \mathbf{matfold}(\hat{S}, \vec{P})$$

In the round after, it must only have taken the damage it can survive from $\vec{Q}$ and so on and so on. Therefore, this is taken care of by the function $\mathbf{partdist}(\vec{P}, HP) = \hat{S}$.
Strictly speaking, we should do similarly with the individual attacks, but as it will be shown below, it ends up not being necessary.

4.1 The effects of omitting partdist

To see what would happen, should we not include the partdist function, we can look at how the matfold operator works. The two most important features of it in this case are:

- It is distributive: $\mathbf{matfold}(\vec{x} + \vec{y}, \vec{z}) = \mathbf{matfold}(\vec{x}, \vec{z}) + \mathbf{matfold}(\vec{y}, \vec{z})$
- It does the following when multiplied by a scalar: $\mathbf{matfold}(A * \vec{x}, \vec{z}) = \mathbf{matfold}(\vec{x}, A * \vec{z}) = A * \mathbf{matfold}(\vec{x}, \vec{z})$
- It is symmetric when its inputs are exchanged: $\mathbf{matfold}(\vec{x}, \vec{z}) = \mathbf{matfold}(\vec{z}, \vec{x})$
- It preserves normalisation: If $\vec{Y} = \mathbf{matfold}(\vec{x}, \vec{z})$, $\sum_{i=0}^{\infty} y_i = (\sum_{i=0}^{\infty} x_i) * (\sum_{j=0}^{\infty} z_j)$

Let us use the same definitions of $\vec{S}$ and $\vec{D}$ as above:

$$\begin{aligned}
\vec{P} &= \vec{S} + \vec{D} \\
\sum_{i=0}^{\infty} s_i &= S \\
\vec{S} &= S\hat{S} \\
\vec{Q} &= \mathbf{matfold}(\vec{P}, \vec{P}) \\
&= \mathbf{matfold}(\vec{S} + \vec{D}, \vec{S} + \vec{D}) \\
&= 2 * \mathbf{matfold}(\vec{S}, \vec{D}) + \mathbf{matfold}(\vec{S}, \vec{S}) + \mathbf{matfold}(\vec{D}, \vec{D}) \\
&= 2S * \mathbf{matfold}(\hat{S}, \vec{D}) + S * \mathbf{matfold}(\hat{S}, \vec{S}) + \mathbf{matfold}(\vec{D}, \vec{D})
\end{aligned}$$

When we sum over the relevant indices to see whether or not the opponent survives, the first one does not contribute and the last one *definitely* does not contribute to the sum.
The only term that *does* contribute is $S * \mathbf{matfold}(\hat{S}, \vec{S})$.
If we included the **partdist** function, the contributing term would be $\vec{Z} = \mathbf{matfold}(\hat{S}, \vec{S})$ - they differ with a factor of S.
In sequential attacks on the same turn, however, $\vec{Z}$ would have to be multiplied with the chance that it happens, which is S, so this would be extra coding for nothing, except maybe some processing time.
In between your turns, however, it looks very different, as your opponent also has a turn.
The end result is that if we do not use this **partdist** function in between your turns, the matrix operation on the three states underestimates how survivable each turn after the first is, effectively overestimating the damage in consecutive turns.

In a simulated match, the victory chance differed with about 3 % after this was realised.

5 Summary - how it's done

The procedure goes something like this:

- Make the dice with **Ndice**(N,d).
- If different types of dice, add them together with **matfold**$(\vec{x}, \vec{y})$.
- Make the distribution for an attack with **activate**$(\vec{D}, F, AC, atk)$.
- If multiple attacks per turn are made, **matfold** them to add the damage and get the distribution, $\vec{P}$, for a turn.
- Do the same for the opponent.
- Calculate the chance to survive a turn for you and your opponent.
- Calculate the matrices and make the product for the first turn.
- Calculate the distribution for survival by $\hat{S} =$ **partdist**$(\vec{P}, HP)$ for yours and the opponent's distribution.
- Calculate the next turn as $\vec{Q} =$ **matfold**$(\hat{S}, \vec{P})$.
- Calculate survival chances and do the matrix product again.
- Calculate the distribution for survival as $\hat{S} =$ **partdist**$(\vec{Q}, HP)$.
- Repeat the previous 3 steps until the middle element is sufficiently small.

With this, we finally have a fully functioning code.
If one would wish to make more intricate strategies than using the same attack all the time, one would simply have to adapt the damage distributions to something like this: $\vec{P} = \sum_i z_i \vec{Y}_i$, where $\vec{Y}_i$ is the damage distribution for some attack that is made when some condition is fulfilled, and z_i is the chance that this exact condition is fulfilled. It is difficult to code, but it can be done.