Calculating SSBM speedrun total time

Author: Mario 64 Master
Last updated: 2025-10-11

TL;DR summary

- SSBM uses special formulas when adding, subtracting, and converting times.
- Doing these calculations yourself using “normal” math will result in slightly incorrect totals.
- **Use the SSBM speedrun IGT calculator** to properly convert and add times.
    - Speedrun.com rules have changed to **require** this approach.

You only need to read the rest of the doc if you’re interested in more details!

Introduction

This doc describes how to correctly add times for SSBM speedruns, and discusses related Speedrun.com rules.

Background

In 2017, the SSBM full-game speedrun community unified on a total time calculation method described in this Speedrun.com thread. The approach involves manual subtraction and addition of times.

While it brought much-needed consistency at the time, this calculation method ends up having flaws which lead to incorrect time conversions, “impossible” times, and the possibility of incorrect ranking order. It also has a high chance of human error.

Over several years of developing the SSBM Stadium Score Database, I learned about how the game handles times, and implemented various spreadsheet conversion functions to reproduce them. I realized these functions could be easily reused for a helper spreadsheet that calculates totals for full-game speedruns, which could accompany Speedrun.com rule fixes.

How timers work

SSBM always displays times to two decimal places, but the game only updates visuals at (approximately) 60 frames per second. This means if you watch a running game timer frame-by-frame, only 60 of the 100 possible decimal values will be shown in a given second. Each second shows the same 60 decimals. We call the other 40 “impossible decimals”, as they can never appear. Here’s a representation of which decimals are possible (green) or impossible (red) on a **timer that counts up**:

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

And on a **timer that counts down**:

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

In reality, 1/60 of a second passes each frame. So an “exact” timer would increase or decrease by 1/60 = 0.01666… every second. You can think of the game as effectively rounding each of these “exact” values to two decimals, e.g. as shown below for timers that count up:

# frames since timer start	**“Exact” time**	**Actual time shown in game**
0	0.00	0.00
1	0.01666…	0.01
2	0.03333…	0.03
3	0.05	0.05
4	0.06666…	0.06
…	…	…

However, the exact rounding method used is **not** as simple as something like “round up”, “round down”, or “round to nearest”. Understanding exactly how it works is key for correct total calculations.

Conversion to frames

Under the hood, the game keeps track of a time as a **number of frames**. To add up times (such as for a BtT total), it just adds the number of frames normally. Only when an individual time or total time needs to be displayed on screen is the number of frames converted to a human-readable format.

This leads to a key observation: if we had conversion functions between number of frames and time displayed by the game, we could accurately add times like the game does, as follows:

1. Convert each time to frames.
2. Add up all the frames using normal addition.
3. Convert the frame sum back to a time.

This is generally the best approach, and is used e.g. by the SSBM speedrun IGT calculator.

Conversion formulas

Time to frames

To convert the decimal part of a time (represented as a two-digit number) to frames, the formulas are:

`frames = ceiling(decimal*59/99)` (for timers that count up)

`frames = floor(decimal*59/99)` (for timers that count down)

Examples:

- A count-up time of 0.46 represents exactly ceiling(46*59/99) = 28 frames since the clock began at 0.
- A count-down time of 0.46 represents exactly floor(46*59/99) = 27 frames until the clock reaches 0.

Frames to time

To convert a number of frames to the decimal part of a time (represented as a two-digit number), the formulas are:

`decimal = floor(frames*99/59)` (for timers that count up)

`decimal = ceiling(frames*99/59)` (for timers that count down)

Examples:

- A count-up time of 28 frames is represented by the decimal floor(28*99/59) = 46 (that is, the time 0.46).
- A count-down time of 27 frames is represented by the decimal ceiling(27*99/59) = 46 (that is, the time 0.46).

The frames-to-time conversion formulas can be visualized here.

Minutes and seconds

Time math works “normally” for minutes and seconds - special formulas are only needed for decimals.

For example, to add the count-up times 1:23.03 and 3:16.11, you can use special formulas above to add the decimals (0.03 + 0.11 = 0.15), then simply add the seconds (23 + 16 = 39), and add the minutes (1 + 3 = 4), giving a total of 4:39.15.

Note this is equivalent to converting each of the full times 2:23.03 and 3:16.11 to frames, adding them up, and converting the resulting frame count back to a time.

Carrying

Carrying also works normally. For example, to add 2:56.50 and 3:31.73 you could:

- Convert 0.50 to 30 frames, and 0.73 to 44 frames, and add them to get 74 frames
- Treat the 74 frames as 1 second plus 14 frames
- Convert 14 frames back to the decimal of **0.23** using the special formulas above
- Add the seconds, including the carried second: 56 + 31 + 1 = 88
- Treat the 88 seconds as 1 minute plus **28** seconds
- Add the minutes, including the carried minute: 2 + 3 + 1 = **6**

The final time in this case is **6:28.23**.

Manual calculation discrepancies

For modes with count-up timers (e.g. BtT), manual adding can be a source of error. For example, 1.01 + 1.01 = 2.02 with “normal” math, but is actually 2.03 using proper conversion formulas.

For modes with count-down timers (e.g. Classic), there are actually **two** potential sources for error: manual subtraction and manual addition. For example, for levels completed with timers of 3:59.98 and 3:59.94 (and timers starting at 4 minutes), “normal” math gives elapsed times of 0.02 and 0.06, for a total of 0.08 (5 frames), but proper conversion formulas give elapsed times of 0.03 and 0.06, for a total of 0.10 (6 frames).

To ensure accurate results, **never add or subtract the decimal part of times using normal math**!

Does it matter?

Yes! It is actually possible for **ranking to change** between two players when calculating times via regular math versus correct formulas.

For example, suppose player A has these times for 4 levels that all have a 4-minute timer:

3:59.98
3:59.98
3:59.98
3:59.98

And player B has these times:

3:59.99
3:59.99
3:59.99
3:59.94

If you use normal math, then player A has a total of 0.08 and player B has 0.09 - **player A wins**.

If you use the correct formulas, then player A has a total of 0.13 (8 frames) and player B has 0.11 (7 frames) - **player B wins**.

The more correct approach is that player B should win because their actual total play time was less (7 frames instead of 8).

This is one primary motivator for updating the Speedrun.com rules to require this corrected approach.

Typical discrepancy

In this section we explore: by how much will a total time typically change when switching from regular math to correct formulas?

Count-up timers

For count-up timers, the “worst case” scenario (largest possible discrepancy) would occur when all the times being added have been rounded maximally in the same direction.

One example is adding times that all end with a decimal of 0.01. As shown in the above table, 0.01 represents 1 frame, corresponding to an exact time of 0.01666… seconds. The difference between these numbers is that the exact time is 0.00666… seconds **higher**, which is the maximum possible discrepancy in this direction.

Another example is adding times that all end with a decimal of 0.99. The corresponding exact time is 0.98333… seconds. In this case the exact time is 0.00666… seconds **lower**, which is the maximum possible discrepancy in this direction.

In other words, the discrepancy when switching to correct formulas will always be **at most** 0.00666… seconds per time being summed, potentially in either direction. For example, when adding 25 BtT times together, the total can change up to 0.16666… seconds in either direction when switching to correct formulas.

However, it would be *extremely* unlikely for all times being added to have maximal rounding in the same direction. What’s the more likely, “average case” scenario?

Assuming the decimal part of times is randomly and evenly distributed across all possible decimals (not quite true, but close enough), it turns out that on average, the total will increase by 0.0015 seconds per time. For example, when adding 25 BtT times together, the total will increase by about 0.04 seconds on average (although again, the discrepancy could be anywhere from -0.16666… seconds to +0.16666… seconds).

Countdown timers

Unlike for count-up timers, discrepancy for countdown timers can only skew in one direction.

The worst-case scenario here is that every time is too low by 0.01333… seconds. For example, a time of 4:59.98 (with a timer starting at 5 minutes) actually represents exactly 0.03333… seconds, not 0.02 seconds as simple math would suggest.

The best case scenario is no discrepancy at all, which can only occur if all times end in 0.00.

So when e.g. adding 11 times for a Classic run, switching to correct formulas will *increase* the time by anywhere from 0.00 to 0.14666… seconds.

The average case expected increase is 0.008166… seconds per time, e.g. about 0.09 seconds for an 11-time Classic run.

Summary

When switching to correct formulas, the possible changes are as follows:

Mode	**Typical timer count**	**Lowest possible discrepancy**	**Highest possible discrepancy**	**Average discrepancy**
Classic	11 countdown timers	0.00 seconds	+0.14666… seconds	+0.09 seconds
Adventure	21 countdown timers	0.00 seconds	+0.28 seconds	+0.17 seconds
All Target Tests	25 count-up timers	-0.16666… seconds	+0.16666… seconds	+0.04 seconds
All Events	18 countdown timers, 33 count-up timers	-0.22 seconds	+0.46 seconds	+0.20 seconds

For full probability distributions, see Discrepancy distributions.

Important caveats:

1. Typical timer count can vary (e.g. Adventure having 20 to 22 stages depending on whether Giant Kirby and/or Giga Bowser is fought)

2. The above table **does not account for retries**. Retries increase the magnitude of lowest / highest / typical discrepancies. Since there is no limit on retries, there is actually **no true bound on the lowest or highest possible discrepancy**.

Human error

The rest of this doc describes discrepancies that arise when players perfectly follow a (flawed) method for calculating times. But after recalculating many existing runs, the SSBM Speedrun.com mods found that a substantial percentage had mistakes in mistiming, sometimes causing times to be off by a lot (multiple seconds or tens of seconds).

For “All Events” (the hardest to get right, because it has the most times to add), about **80% of submissions had a math error** leading to a substantially wrong submission time!

Using the SSBM speedrun IGT calculator significantly reduces the chance for human error, because it only requires copying times rather than calculating with them, and it outputs a summary that can be easily verified by a moderator.

So although this was not an original motivation of the rule change and associated retimings, this project has also had a **significant positive effect** on the accuracy of the leaderboards.

Unfixable totals

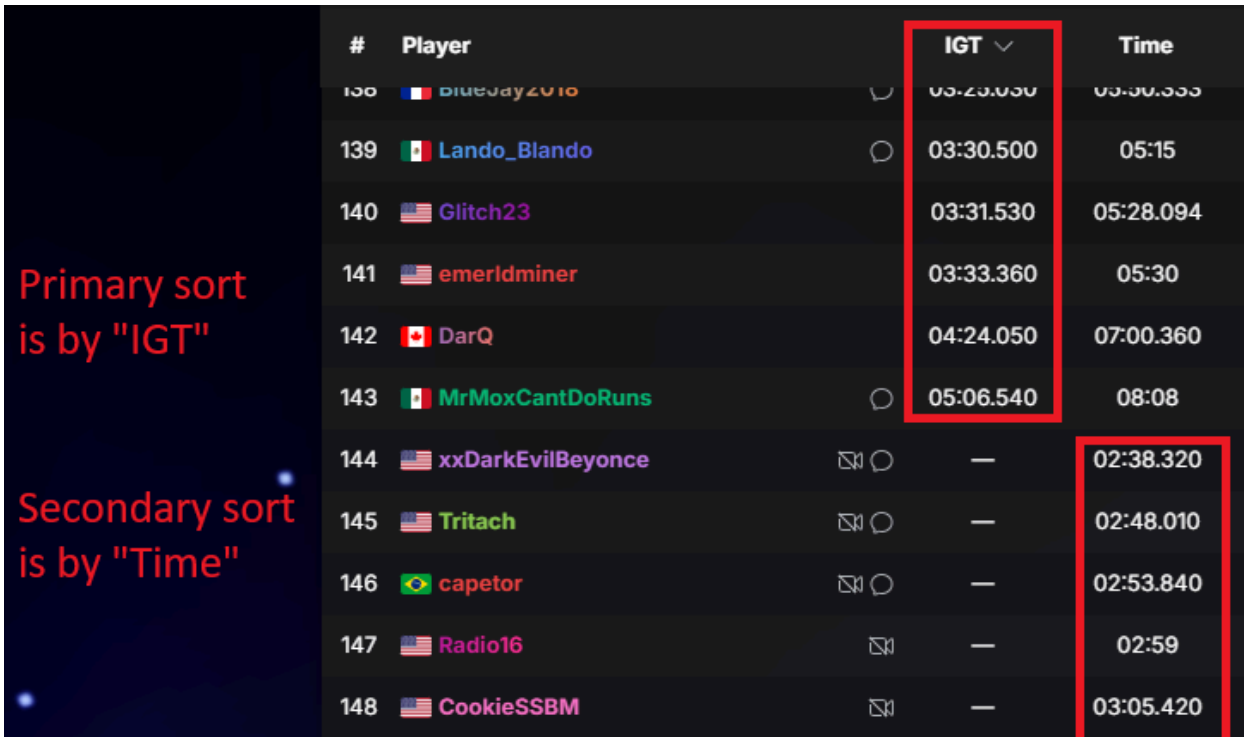
Some existing totals may be impossible to fix, due to lack of information. For example, if a Speedrun.com submission has no video (or a blurry video), and insufficient time info recorded in the Description field, it will be impossible to calculate its true total. In such cases, we have to decide if / how to adjust the time.

Note that switching to proper formulas almost always makes a total time “worse” (slower) - this is true about 99% of the time for BtT, and more than 99.9999% of the time for other modes (data). So it would be unfair to diligent players (those who have provided clear video proof and/or individual level times in the Description) if we “fixed” their times, but left others alone, leading to potential ranking swaps.

Potential solutions:

1. Leave the unfixable times alone. This would give an unfair advantage to players with missing information, but it would be simple to understand and implement, and in practice it would be unlikely to have any significant impact on ranking.
    a. Update: After manually retiming many runs, we’ve realized that a substantial percentage were originally mistimed, sometimes by a lot (off by seconds or tens of seconds). So it’s not true in practice that leaving times unfixed is unlikely to affect their ranking. See Human error.

2. We could entirely remove the “IGT” for totals that can’t be properly recalculated. This would also be easy enough to understand and implement, and would guarantee that these submissions don’t get an advantage, but would often drop their rankings considerably - submissions without an IGT get sorted by default to the bottom of the ranking list, as shown below. Also, this could potentially cause some former WRs to disappear from the “History” tab of Speedrun.com (depending on whether there are any older runs with IGT specified).



3. We could penalize the score by a “worst case scenario” amount. Some examples:
    a. The Summary table above shows that fixing an All Events total would add at most 0.46 seconds to the time. So we could just add 0.46 to any All Events time that can’t be correctly calculated. This would aim to ensure that players without sufficient proof are not gaining an advantage, but also not being too harsh on them. Unfortunately, this does not *guarantee* that the full advantage is mitigated, because the “worst case scenario” does not account for retries. A run with many retries could plausibly need to be penalized by more than the above table shows. To address this, we could pick any arbitrary penalty amount that we feel is fair to add for each game mode (never *guaranteeing* that it’s enough, but almost certainly being enough without penalties needing to go above 1 second or so).
    b. Blurry videos where the decimal cannot be viewed can have timers rounded to the next second. For example, this time would be treated as 4:47.00 because the 4:47 is clear in the video, but the decimal is not.

In general, the above options present a tradeoff between the types of inaccuracies the leaderboard has. Specifically, option #2 leads to **missing** scores, while options #1 and #3 lead to **incorrect** scores.

Decision

After much consideration, the SSBM Speedrun.com mods have **chosen and implemented option #2**. Any runs that could not be retimed have had their IGT removed but their Time left alone, with the Description updated to explain why (example).

In a few cases this was not possible due to the run only having an IGT but not a Time; in these cases the runs unfortunately needed to be removed from the leaderboards (example).

Speedrun.com rule change

As of **October 12, 2025**, all SSBM Speedrun.com submissions for affected game modes are required to use totals as output by the SSBM speedrun IGT calculator.

Affected game modes are **Classic**, **Adventure**, **All Target Tests**, and **All Events**. (Other modes are not affected since they don't require the player to add up their own times.) This applies to both SSBM and SSBM Category Extensions leaderboards.

All existing submissions for these modes have been retroactively updated to use the correct calculation methodology. In some cases, total times were unfixable and needed special handling.

A new rule requires all submissions to copy/paste the output of the calculator (a full list of accurate splits) into the Description of their submission. This will allow times to be easily tracked and retained going forward - if a video disappears at some point, the Description will still give all information needed to justify the total time.

All Events retry/reset timing

The All Events mode poses an extra challenge for accurate timing, specifically for retries (Start+Z) and resets (L+R+A+Start).

In many game modes, the time to be recorded for a level is always shown on the screen for long enough to easily read it. For example, when you pause and retry a BtT level, the game screen (including the timer) freezes for about half a second as the level is reloaded. Classic, Adventure, and All Target Tests have no problems in this regard.

But in event matches, the **timer disappears** when the game is paused. Per All Events category rules, the time to be recorded for a retry or reset is the time shown on the last frame before the

pause. Unfortunately, this means we sometimes need to determine a time that is displayed on screen for **only one frame**. Any video recording that is not a perfectly smooth 60 fps will have a chance of dropping this key frame, risking accidentally marking the time as faster than it was.

For example, suppose an event is paused and the actual last time shown on the count-up timer was 1.23. A 30 fps recording may miss that frame, only showing the 1.21 frame followed by the first pause frame. However, it's also possible that a 30 fps recording shows the 1.23 frame followed by the second pause frame. How can we tell what happened, as accurately as possible?

The answer is to use our understanding of what happens when the game pauses:

1. Everything in the game scene freezes (characters, visual effects, background animations, etc.), matching the last frame before pause.
2. The game camera starts to pan and zoom, beginning on the first frame of pause.

By carefully inspecting the video around the moment of pause, we can use various tricks to often determine with high confidence exactly when the pause occurred. For example, if everything in the game screen looks identical except for camera pan and zoom between the last frame shown before pause and the first frame of pause, the video is probably showing the "true" frame whose time we need to record. If there's at least one clear difference in the scene, then **at least** one non-pause video frame must have been dropped.

Determining *how many* frames were dropped is sometimes possible, but often tricker. For example, the below diagram shows 6 consecutive frames of Samus's aerial backwards rejump animation. Since she spins very quickly, each position is easily distinguishable. Now suppose an event match video shows Samus in position B on the last frame before pause, then in position E on the first frame of pause. This means there must have been two non-pause dropped frames in between (and two frames need to be added to the time shown on the frame before pause). In other cases this determination may be very difficult or impossible, especially if very little has moved between the frames being inspected.

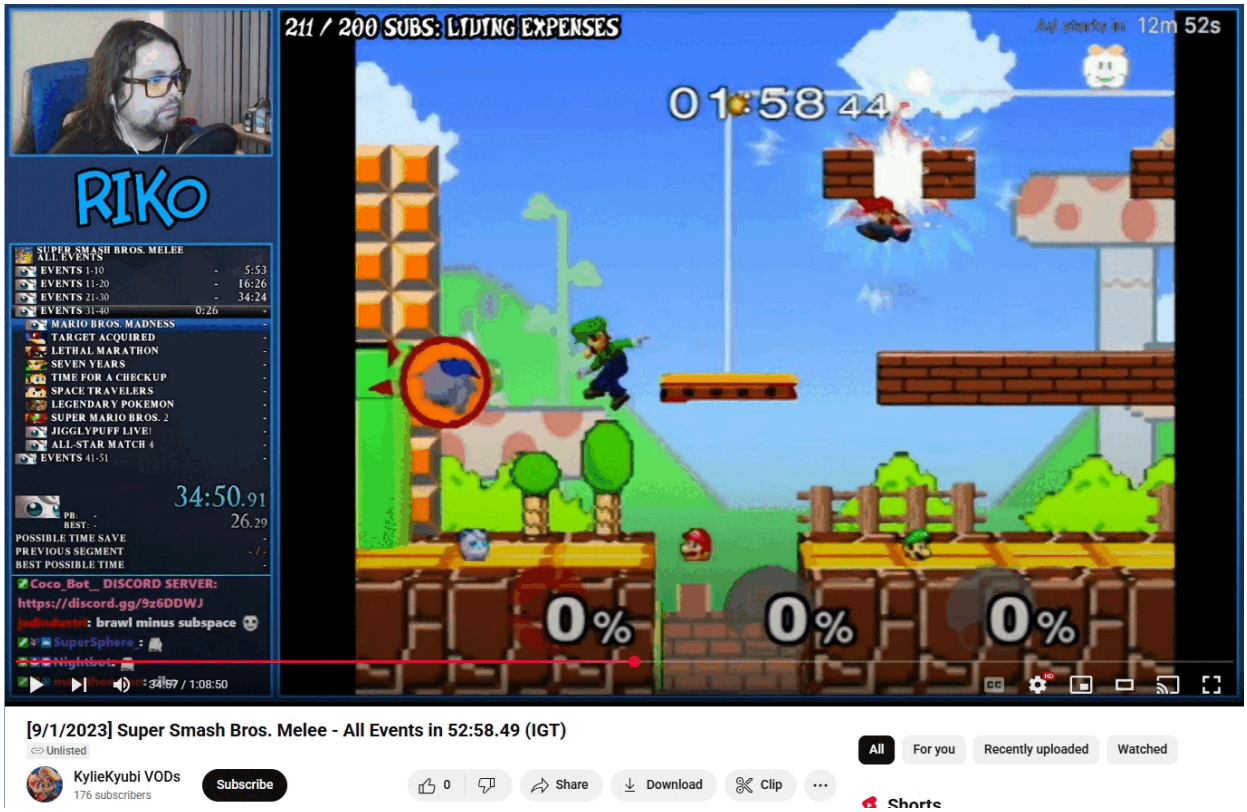
All Events timing examples

Each example below shows 3 frames from an All Events run - the two frames before pause, and the first frame of pause. Comparing frames 1 and 2 gives a sense of what's actively moving, and then comparing frames 2 and 3 for differences can determine whether any frames are missing.

Example 1

We can be fairly certain that no frame is missing here, as the only difference between frames 2 and 3 is a slight pan of the camera. Note for example that the red and white flash of the block breaking above Mario's head looks exactly the same in frames 2 and 3 - these flashes tend to change significantly on each frame while the game is not paused.

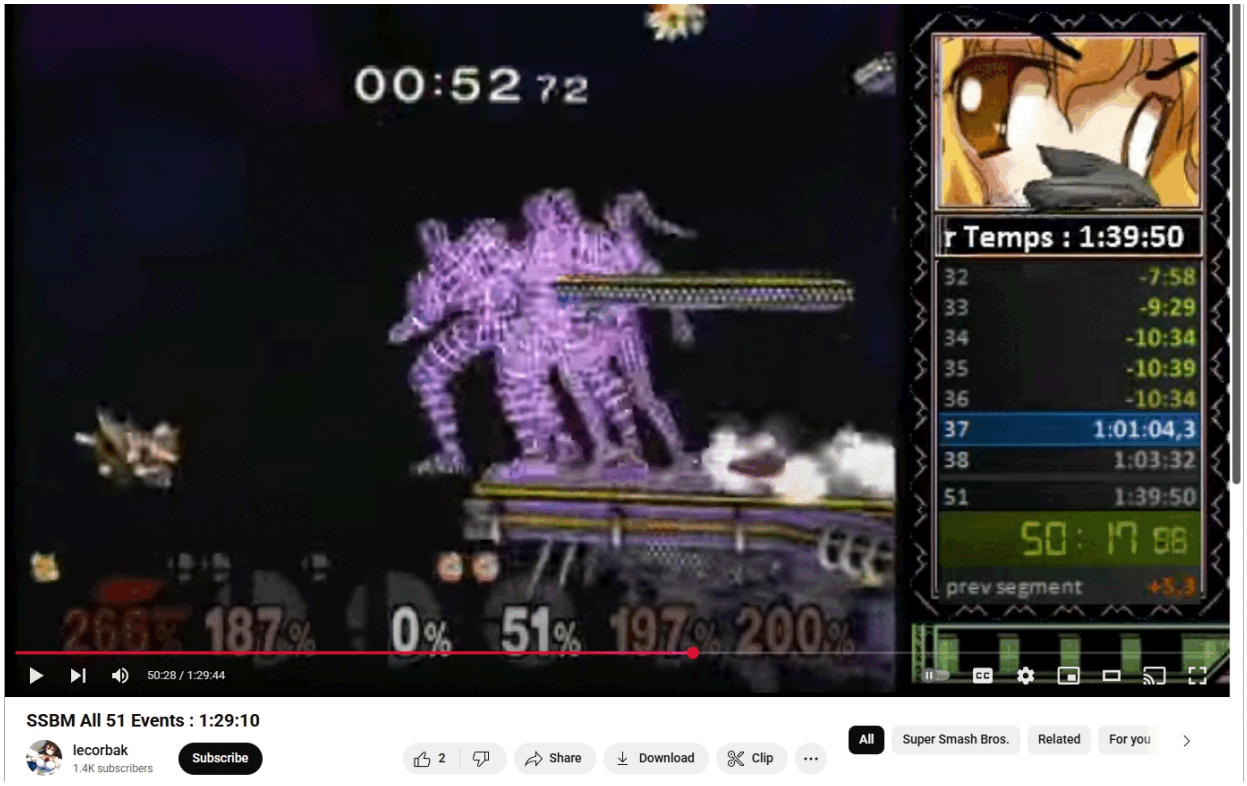
https://youtu.be/QKLebdmcB4A?t=2096



Example 2

Inconclusive. The first 2 frames show that Fox is falling and the front male wireframe is bending down, but it's hard to tell if Fox has fallen farther or the wireframe has bent more on frame 3. One frame should be added here since this video is at 30 fps (note the time jumps from 52.72 to 52.75, so the recorded time would be 52.77).

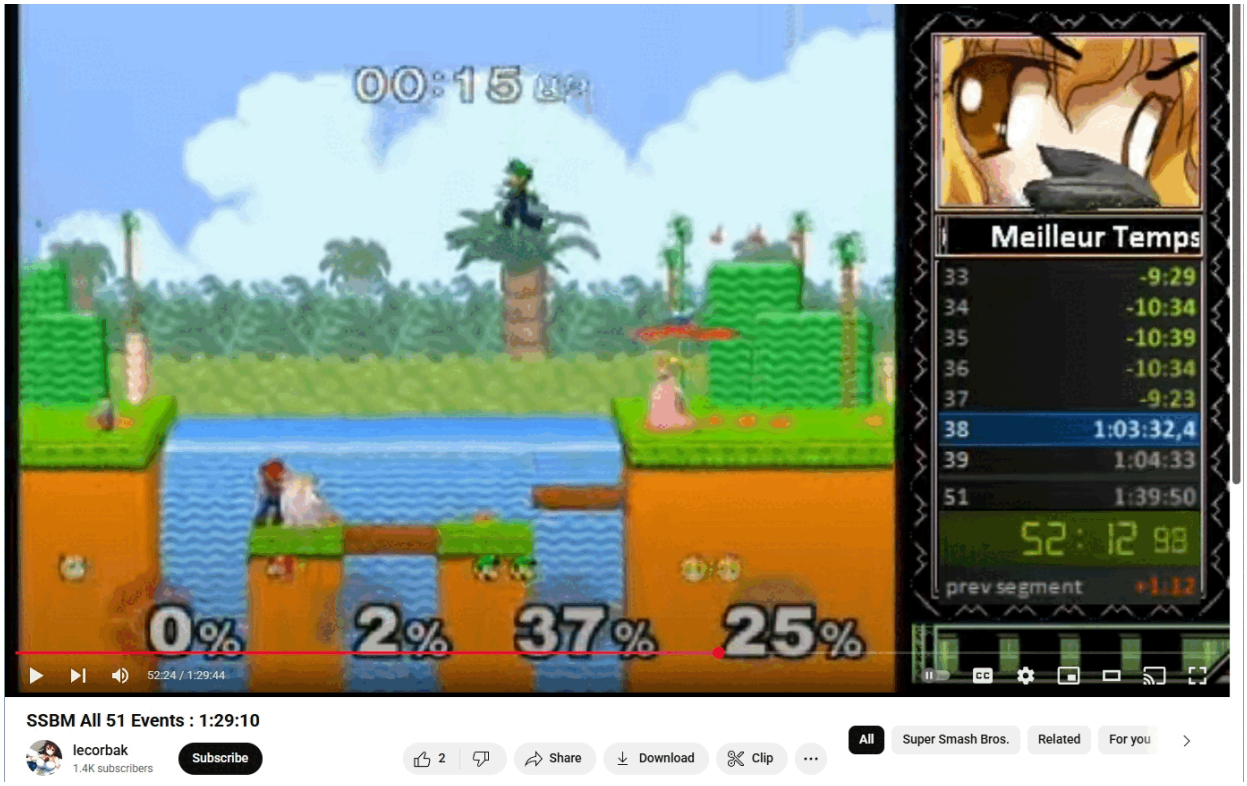
https://youtu.be/JVPD6uq_DTg?&t=3026



Example 3

Inconclusive. Mario and Jigglypuff are both very still, so it's too hard to tell if anything has changed. The video is 30 fps, so 1 frame should be added (giving 15.87 instead of 15.85).

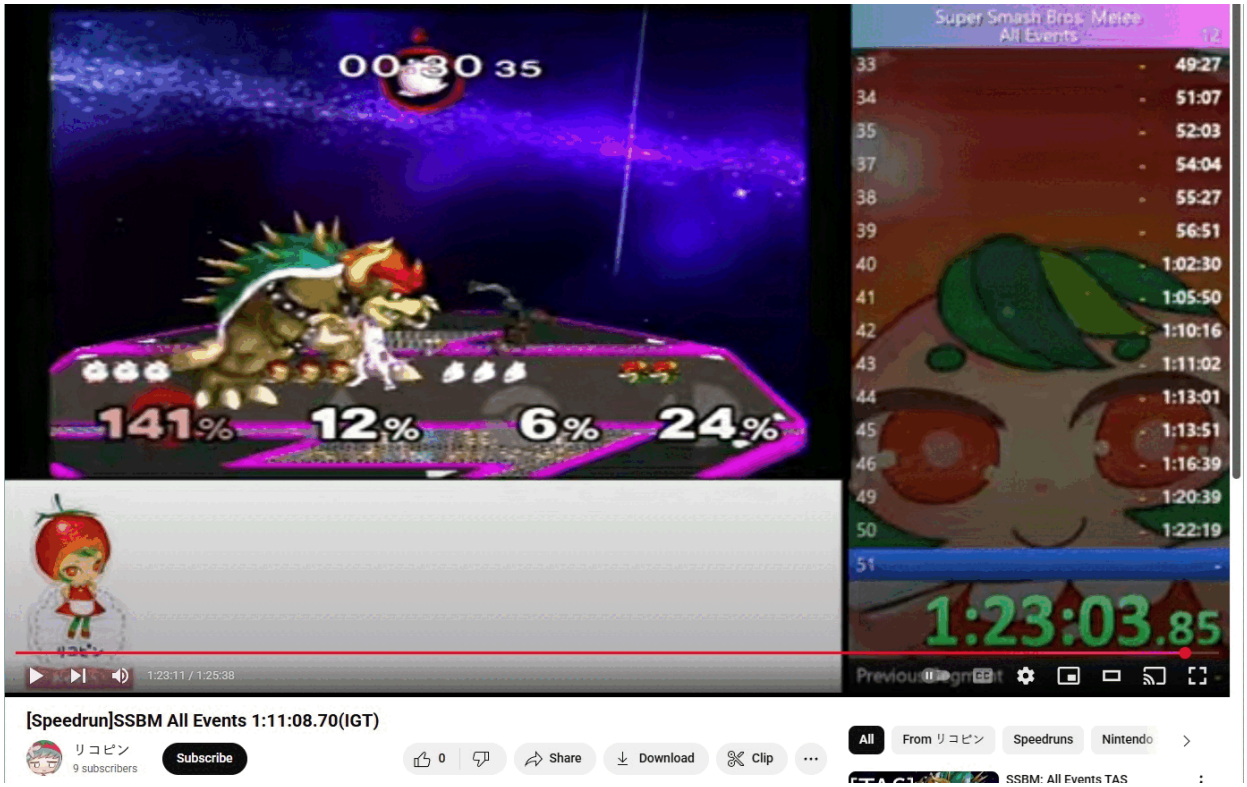
https://youtu.be/JVPD6uq_DTg?t=3142



Example 4

We can be fairly certain that no frame is missing here. The first two frames show some obvious movement (e.g. Giga Bowser blinking, Mewtwo turning, Ganondorf falling), while the only difference between frames 2 and 3 is a slight pan of the camera.

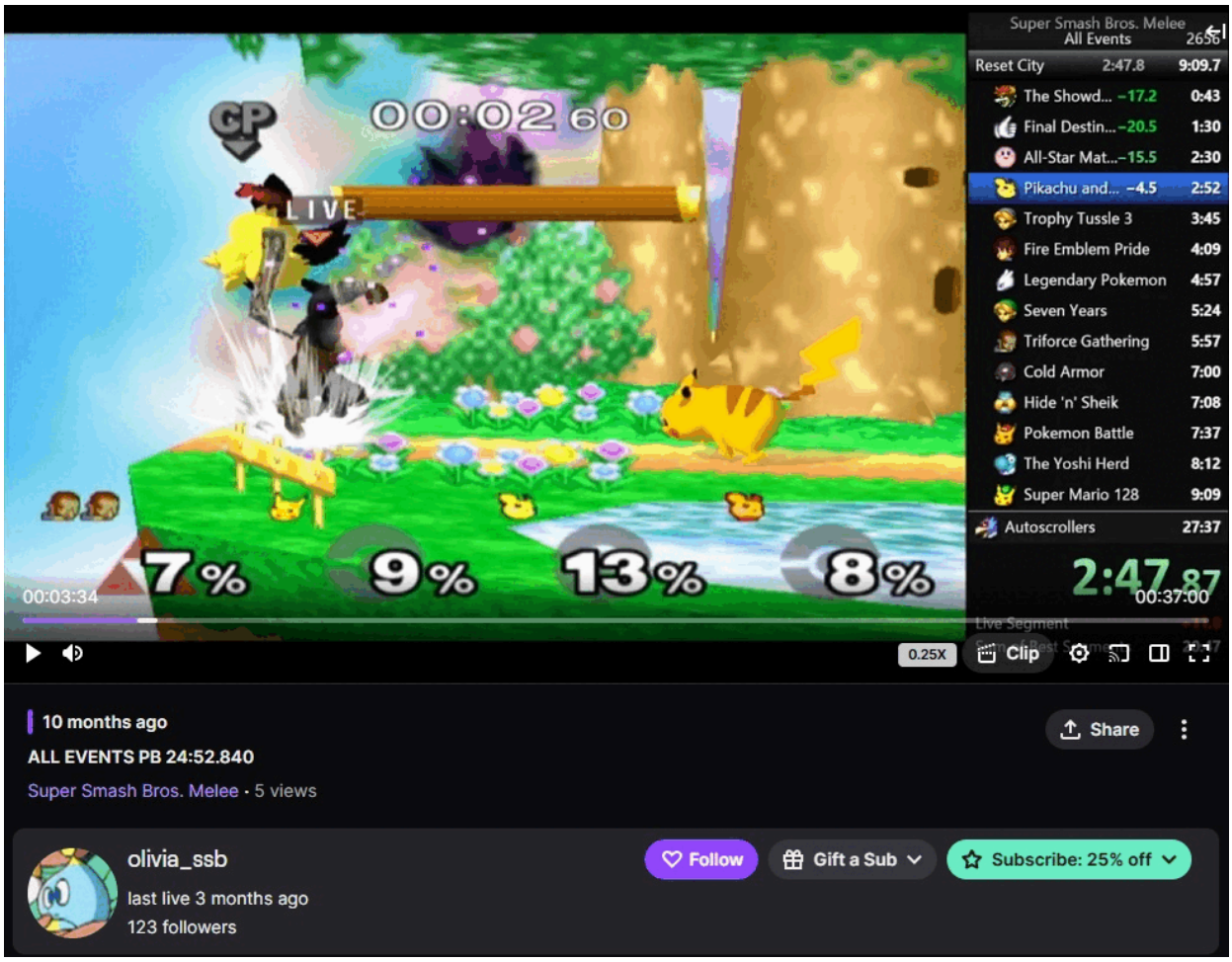
https://youtu.be/Ul40vAYG8vw?t=4989



Example 5

At least one frame has been dropped here. Between frames 2 and 3, the bright purple flash has started to fade, the Pichus are tilted in a slightly different orientation, and the white landing flash on the ground around Ganon's feet does not spike out as far. The time should be recorded as 2.65.

https://www.twitch.tv/videos/2307033670?t=0h3m32s



Example 6

This video has screen tearing, such that two consecutive frames are combined into one. The frame showing 0.20 has a tear, with the top half of the screen not yet being paused, and the bottom half showing a significant zoom (larger Jigglypuff). Hence the 0.20 must have been the last frame before pause, with no extra penalty needed.

https://www.twitch.tv/videos/2307033670?t=0h13m25s

All Events SRC retry/reset timing rules

In light of the above complications, the Speedrun.com rule for timing retries (Start+Z) and resets (L+R+A+Start) for SSBM All Events is:

For each retry or reset, penalize the last shown time by as few frames as is required to have high confidence that the time is not being recorded as better than it actually is.

Ultimately, this requires some judgment. To apply this rule as accurately and consistently as possible, use this approach:

First note the video's overall framerate. One technique is to frame advance through the video and count the number of different frames shown within a one-second segment. Common examples of framerates include 60 fps (no skips), 30 fps (skip every other frame), 25 fps (skip frames in a 1 2 1 1 2 pattern), and so on. Also try spot-checking to see how consistent the framerate is - there can sometimes be momentary lag spikes that disrupt the regular pattern. Make sure you're familiar with the impossible frames.

Then for each retry or reset, carefully observe the frames just before and just after the pause. Try to determine if the position of anything in the scene has changed at all (any character moved, visual effect changed, etc.) between the last frame before pause, and the first frame of pause. Be wary of the fact that the camera starting to zoom can change the relative position / angle of things, so you have to look for specific changes that would be more than just due to that. See the above examples.

The penalty is then as follows:

1. If you're confident that nothing has moved between those 2 key frames, don't apply any frame penalty.
2. If you're confident in how many non-pause frames have been dropped, apply a penalty of that many frames.
3. Otherwise, penalize the time by however many frames could have possibly been missing in between the two frames shown in the video based on the video frame rate. For example:
    a. A consistently 60 fps video would have no penalty.
    b. A consistently 30 fps video would have a 1 frame penalty.
    c. A consistently 25 fps video would have either 1 frame or 2 frames of penalty applied depending on where in the 1 2 1 1 2 cycle the video currently is.
    d. A pause around a laggy moment of a 30 fps video may have 2, 3, or more frames of penalty applied depending on how many frames are typically skipped in nearby lag spikes.
    e. Special case: Event 33 (Lethal Marathon) runs at double speed. This typically means the worst-case scenario is twice as many frames needing to be penalized (e.g. on a consistently 25 fps video, potentially 4 frames penalized instead of 2).

SSBM Speedrun.com mods can be expected to apply a **reasonable best effort** to time accurately. This would entail a close frame comparison, but not necessarily additional outside research (such as looking up move animations, making a TAS recreation, etc.). If a player objects to a time (e.g. a moderator penalized a retry by 2 frames but the player believes it should only have been penalized by 1), the time can be re-reviewed on a case-by-case basis. Ultimately the moderator team has the final say.

Appendix

True game framerate

Technically, it's not accurate to say that SSBM runs at 60 fps and therefore each frame is displayed for exactly 1/60 of a second. The game (NTSC or PAL in "60 Hz mode") actually outputs frames at closer to 59.94 fps, and a side effect of this is that about 1 out of every 1000 frames is "skipped". The detailed description of how this all works is not covered in this doc,

because it doesn't affect how we convert or sum times, and a 60 fps approximation is close enough for what we're discussing here.

Delayed timer start

In most cases, players can actually move before the timer starts counting, as soon as "Go!" appears.

At the start of a BtT level for example, there are 39 actionable frames where "Go!" is shown and the timer is 0.00, then one frame where "Go!" has disappeared but the timer still shows 0.00, then finally the timer shows 0.01 on the next frame.

Countdown timers work similarly, with e.g. the timer showing 2:00.00 for 40 actionable frames before finally ticking down to 1:59.99.

In other words, times calculated via the approach described in this doc are *technically* 40 frames (2/3 of a second) lower than the true actionable play time. In theory, we could add in this extra time to get the most accurate measure of actionable playtime across all levels. However, the convention is instead to ignore the extra frames and just calculate based on the displayed timer. Reasons include:

- This matches how the game handles summing of times, in modes like BtT.
- It's simpler to go by just the displayed timer.

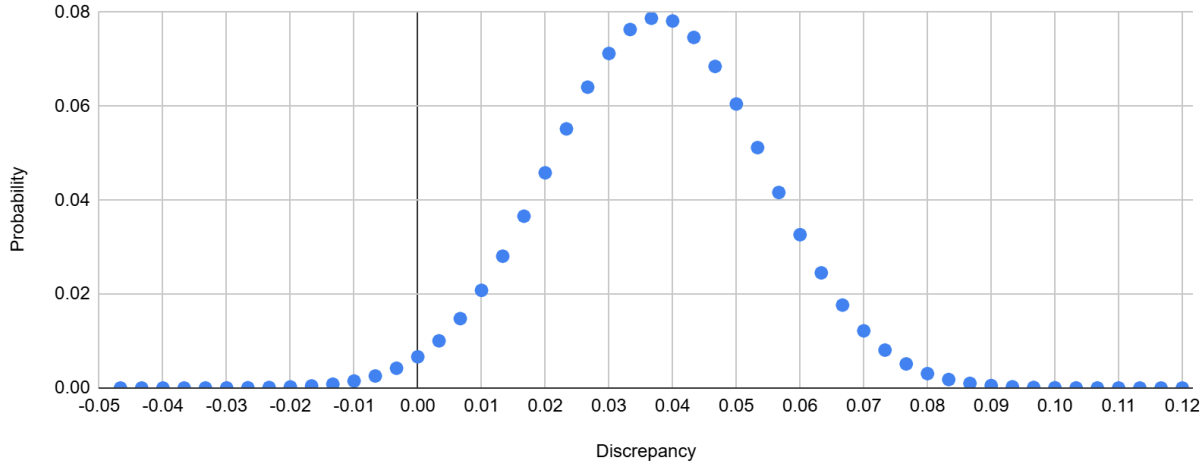
One implication of ignoring these extra frames is that each retry is 40 frames less costly than it would be if every actionable frame were counted. It's even quite possible that a run with more retries ranks better on the leaderboard despite having more total actionable frames of playtime. However, this is an existing convention that this doc will not try to change, as this would affect strategy for some levels (how "worth it" it is to retry versus trying to finish the level).

A notable exception to the above is the Master Hand fight, where the timer starts at 4:59.99 rather than a round 5 minutes, there is no "Go!", and the first actionable frame is the one where the timer ticks down to 4:59.98. Here we just calculate the time as we do with all other levels, using a starting time of 4:59.99.

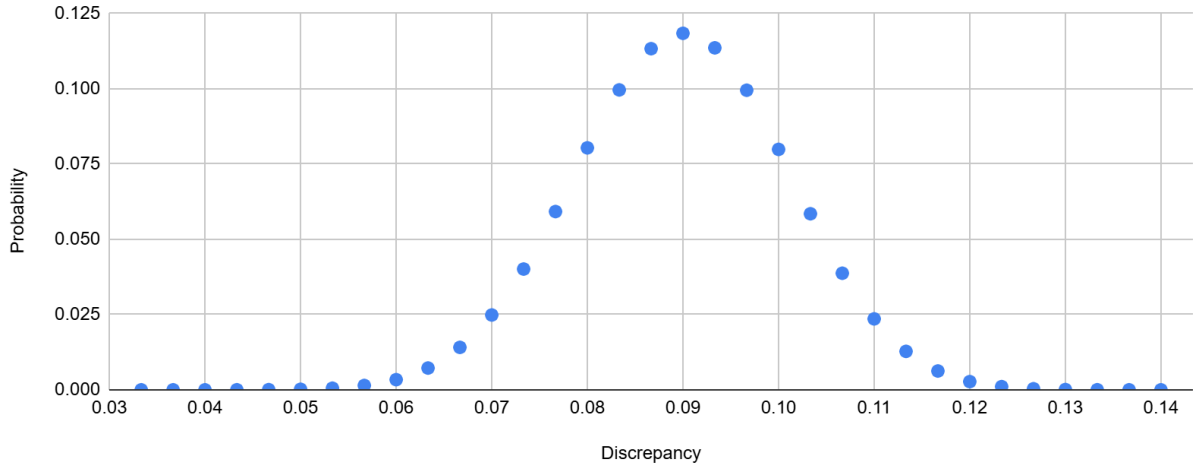
Discrepancy distributions

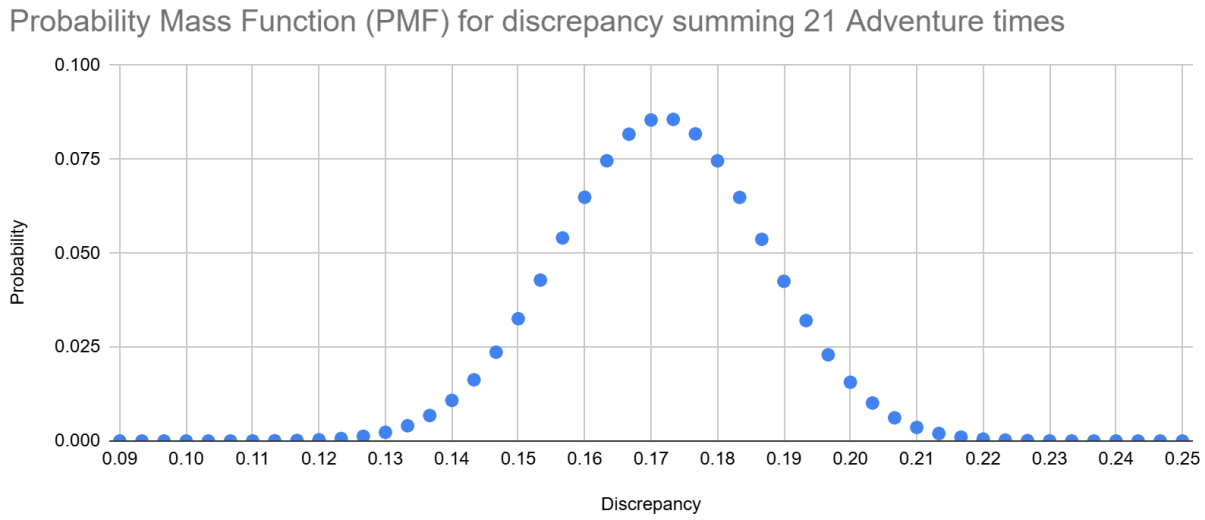
Data generated by the Sum of rolls simulation code:

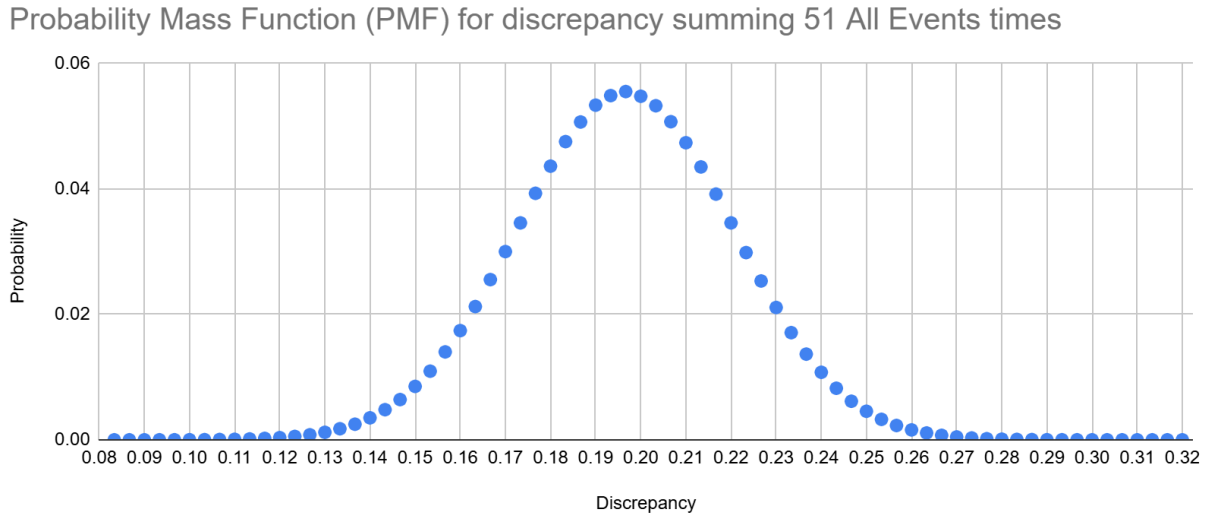
Probability Mass Function (PMF) for discrepancy summing 25 BtT times



Probability Mass Function (PMF) for discrepancy summing 11 Classic times







Sum of rolls simulation

Monte Carlo simulation of expected discrepancy when summing times. Used to generate [Discrepancy distributions]. Java code generated by Gemini with a few subsequent manual adjustments:

```
import java.util.Random;
import java.util.TreeMap;
import java.util.Map;

public class SumOfRollsSimulation {

    public static void main(String[] args) {
```

```
        // All Target Tests
        // int numCountUpRolls = 25;
        // int numCountdownRolls = 0;
        // Classic
        // int numCountUpRolls = 0;
        // int numCountdownRolls = 11;
        // Adventure
        // int numCountUpRolls = 0;
        // int numCountdownRolls = 21;
        // All Events
        int numCountUpRolls = 33;
        int numCountdownRolls = 18;

        int numSimulations = 10000000;
        int decimalPlaces = 6;

        Random random = new Random();
        Map<Double, Integer> sumCounts = new TreeMap<>();

        double[] countUpOutcomes = {-0.00666666666, -0.00333333333,
0.00333333333, 0.00666666666, 0.0};
        double[] countUpProbabilities = {1.0/6.0, 19.0/60.0, 1.0/6.0,
1.0/60.0, 1.0/3.0};
        double[] countdownOutcomes = {-0.01333333333, -0.01, -0.00333333333,
-0.00666666666, 0.0};
        double[] countdownProbabilities = {1.0/6.0, 19.0/60.0, 1.0/6.0,
1.0/3.0, 1.0/60.0};

        for (int i = 0; i < numSimulations; i++) {
            double currentSum = 0;
            for (int j = 0; j < numCountdownRolls; j++) {
                currentSum += getRandomOutcome(countdownOutcomes,
countdownProbabilities, random);
            }
            for (int j = 0; j < numCountUpRolls; j++) {
                currentSum += getRandomOutcome(countUpOutcomes,
countUpProbabilities, random);
            }
            double roundedSum = roundToDecimalPlaces(currentSum,
decimalPlaces);
            sumCounts.put(roundedSum, sumCounts.getOrDefault(roundedSum, 0) +
1);
        }
```

```
        System.out.println("Discrepancy,Probability");

        for (Map.Entry<Double, Integer> entry : sumCounts.entrySet()) {
            double sum = entry.getKey();
            int count = entry.getValue();
            double probability = (double) count / numSimulations;
            System.out.println(String.format("%.6f", sum) + "," +
String.format("%.8f", probability));
        }
    }

    private static double getRandomOutcome(double[] outcomes, double[]
probabilities, Random random) {
        double randomNumber = random.nextDouble();
        double cumulativeProbability = 0;
        for (int i = 0; i < outcomes.length; i++) {
            cumulativeProbability += probabilities[i];
            if (randomNumber < cumulativeProbability) {
                return outcomes[i];
            }
        }
        return 0.0;
    }

    private static double roundToDecimalPlaces(double value, int places) {
        double scale = Math.pow(10, places);
        return Math.round(value * scale) / scale;
    }
}
```