Behavior Research Methods (2023) 55:1863–1873
https://doi.org/10.3758/s13428-022-01899-0

Simulating behavior to help researchers build experiments

Joshua R. de Leeuw[1] · Rebecca A. Gilbert[2] · Nikolay Petrov[3] · Björn Luchterhandt[4]

Accepted: 1 June 2022 / Published online: 29 June 2022

Abstract
Testing that an experiment works as intended is critical for identifying design problems and catching technical errors that could invalidate the results. Testing is also time-consuming because of the need to manually run the experiment. This makes testing the experiment costly for researchers, and therefore testing is less comprehensive than in other kinds of software development where tools to automate and speed up the testing process are widely used. In this paper, we describe an approach that substantially reduces the time required to test behavioral experiments: automated simulation of participant behavior. We describe how software that is used to build experiments can use information contained in the experiment's code to automatically generate plausible participant behavior. We demonstrate this through an implementation using jsPsych. We then describe four potential scenarios where automated simulation of participant behavior can improve the way researchers build experiments. Each scenario includes a demo and accompanying code. The full set of examples can be found at https://jspsych.github.io/simulation-examples/.

Keywords Methodology · Behavioral experiments · Automation · Simulation · jsPsych

In 1970, Philip Corsi and Brenda Milner arranged nine one-inch, black cubes on a board roughly the size of a standard piece of paper to create what would become one of the most widely used tests of nonverbal working memory (Milner, 1971; Corsi, 1972; Berch et al., 1998; Richardson, 2007). The experimenter used a six-inch wooden stick to tap on a sequence of blocks and the participant attempted to repeat the sequence while the experimenter noted their response. Today this task is routinely administered using a computer. Squares on a screen take the place of the physical cubes, the experimenter's stick has been replaced by a programmed sequence of cues, and the responses of the participant are automatically tracked with greater precision (Brunetti et al., 2014; Gibeau, 2021; Arce & McMullen, 2021).

✉ Joshua R. de Leeuw
jdeleeuw@vassar.edu

[1] Department of Cognitive Science, Vassar College, Poughkeepsie, NY, USA

[2] MRC Cognition and Brain Sciences Unit, University of Cambridge, Cambridge, UK

[3] Cardiff University Brain Research Imaging Centre, University of Cardiff, Cardiff, UK

[4] Department of Computer Science, Paderborn University, Paderborn, Germany

The transition to computerized experiments is, of course, a widespread development in the relatively recent history of psychological and behavioral research. Computer-based experiments are so widespread that it can be easy to forget that many of the digital experiments we run today were created with a very different set of tools. However, tools have changed, and as a result, so have our experiments. Unlike the original block-tapping task using physical objects, experiments today are created using code.

This gradual, but now nearly total, shift to experiments-as-code has changed the kinds of research that are possible. Methodological innovations began almost immediately upon the introduction of computers as a research tool, initially with an emphasis on automating tasks, creating new kinds of stimuli, and collecting more detailed data than was previously possible (Aaronson et al., 1976). In recent years, innovations such as machine-assisted/algorithmic experimental design (Suchow & Griffiths, 2016; Ouyang et al., 2016), software-based eye tracking (Papoutsaki et al., 2016; Yang & Krajbich, 2021), and real-time networking between large numbers of participants (Almaatouq et al., 2021; Balietti, 2017) have all continued the tradition of enabling new kinds of experimentation through building new software. Computer-based data collection has also, especially in recent years, radically changed who can participate in research and

the speed at which data collection is done. The widespread use of online experiments has removed the need for geographic proximity to a research lab and changed the model of research from a synchronous interaction between experimenter and participant to an asynchronous event where a virtually unlimited number of participants can complete an experiment in parallel with no experimenter supervision (Hartshorne et al., 2019).

Alongside the widespread adoption of software-based experiments has been the development of tools that make it easier to create experiments. Many experiments have a common set of needs, such as precise control over visual displays, presenting sequences of events, recording key presses and mouse clicks, and various kinds of randomization. The development of programming libraries and frameworks that are targeted to experiment-specific problems (e.g., Brainard, 1997; Forster & Forster, 2003; Peirce, 2007; Stoet, 2010; Mathôt et al., 2012; de Leeuw, 2015; Zehr & Schwarz, 2018; Henninger et al., 2021) has made it easier for researchers to create software-based experiments. Furthermore, once an experiment is created as software, it can usually be shared and modified without too much friction, making it easier to replicate and build upon existing experiments (e.g., Sochat et al., 2016; Gureckis et al., 2016).

In addition to all of these tools tailored towards experiment development, there is an even longer list of innovations that have changed how people write code more generally. Integrated development environments provide features such as code completion and error detection, allowing developers to write code both faster and more accurately. Version control software such as Git tracks and manages all changes to the code, enabling collaborative development at large scales. Collaboration platforms like GitHub and GitLab enrich version control with features for communication, organization, and continuous integration services, which allow developers to automatically build, test, and deploy code whenever a change is made. All of these innovations are now essential parts of any professional software engineer's toolkit.

Since modern experiments are often code, we think there is room for researchers to continue to innovate by borrowing (more) ideas from software engineering and adapting them for use in experiment development. Of course, many researchers already use a wide range of software engineering tools as part of their experiment building process. We think that innovations in this space are likely to come from taking concepts from software engineering and applying them in new ways that are targeted at experiment development. For example, the Experiment Factory (Sochat et al., 2016) took the concept of containers—a bundle of software that is portable across different computing environments—and developed tooling for reproducible experiments.

Simulating user behavior is one technique used in software engineering that has potential to change how we develop experiments. Software development typically involves the development of smoke tests—bare-minimum tests of functionality, named for the puff of smoke that a miswired circuit can generate—and integration tests. These tests are crucial for detecting major errors and verifying that components interact correctly with each other. These test procedures often require user interaction to determine how the software responds to various situations. Automating these tests by simulating user behavior can make testing faster and easier to reproduce. Once tests are automated, they can be run with substantially less effort than manual tests, typically resulting in better test coverage and earlier bug detection, thereby improving the developer experience and helping developers write more reliable code.

The ability to automate user interactions has the potential to change how researchers build experiments in a number of ways. The dominant use case in software engineering—automated testing—is of course applicable to experiments as well, but we think the use cases go beyond this to things like rapidly generating simulated data sets for building analysis pipelines, creating visualizations of participant behavior during the experiment, and simply speeding up the development process by allowing the researcher to simulate some chunks of the experiment while carefully interacting with other parts. Later in this paper, we describe four scenarios in which simulations can help researchers build experiments. These scenarios apply regardless of the tool used to build the experiment. Simulations can be used with pretty much any software-based experiment. There are general-purpose software tools that automate the kinds of inputs that a user would provide for any platform that experiments are commonly deployed on (e.g., AutoIt for Windows; PyAutoGUI for any platform running Python; Selenium for anything that runs in a web browser).

The downside of simulation is that it is often cumbersome to set up. Writing code to automate a behavior may take a lot more time than just manually performing the behavior many times. In order for a simulation to be an efficient use of a researcher's time, the simulation must be useful enough to make up for the time spent implementing. Where exactly this cost-benefit analysis tilts in favor of simulation depends on several factors, including how the simulation is used, how difficult it is to set up, how complicated the experiment is, and how comprehensively the researcher wants to test it.

With most experiment software, the cost of implementing a simulation would be relatively high. Even borrowing general-purpose behavior simulation tools from software engineering is unlikely to make simulation an attractive option for researchers given the need to learn how to use the tools and then customize the behavior of the simulation for the particular experiment. However, if simulation is built into tools for building experiments, then the implementation cost shifts from individual researchers to the developers of the tools. This would make simulations a practical option for

researchers, while also allowing the code for implementing simulation to benefit from shared development.

Considering the needs that researchers typically have when building experiments, we think that experiment-building software can maximize the value of simulation by supporting four key features.

1. *Simulation tools should allow researchers to run a simulation with as little configuration as possible.* One way to minimize the cost of using simulation is to make it require minimal additional configuration to set up. We describe one approach for generating sensible default simulation behavior with no researcher configuration in the Implementation section below.
2. *Simulation tools should allow researchers to fully customize the behavior of the simulation*. In many situations researchers will need to customize the behavior of the simulation to achieve their goals. For example, a researcher may want to test that when an instructions comprehension quiz is answered incorrectly the experiment repeats the instruction. A tool that allows the researcher to specify the exact responses generated by the simulation is important for this kind of testing.
3. *Simulation tools should allow researchers to pick and choose which portions of an experiment are simulated.* This allows the researcher to simulate a portion of the experiment while interacting with other parts. For example, a researcher may want to skip through the instructions and comprehension check while developing the portions of the experiment that come later. We describe more examples of this in use case #3 later in the paper.
4. *Simulation tools should be able to run in real time and run faster than real time.* Depending on the goals of the simulation, sometimes a researcher may want to run the simulation as fast as possible, allowing for rapid testing of the experiment logic or quickly generating data files. Other times, the researcher may want to run the simulation in real time in order to observe the behavior of the experiment directly.

Two experiment-building tools (other than the jsPsych implementation we will describe in the remainder of this paper) currently offer support for some of these simulation features. Both are closed-source, commercial platforms, so implementation details are not available, but their features are worth describing. Qualtrics (Qualtrics, Provo, UT) is able to generate test responses[1]. This feature allows a researcher to generate random data for a survey with no configuration required. The simulation runs faster than real time and cannot be controlled by the researcher. The second example is Inquisit's test monkey (Millisecond, Seattle, WA). This is a more customizable tool that allows the researcher to specify a list of valid responses to a trial, from which the simulation will pick one at random[2]. For experiments that already have a list of valid responses for each trial the simulation requires very little configuration to run. The simulation behavior is customizable by changing the list of valid responses, and the response latency and accuracy of the simulation can be set by the researcher. However, the test monkey only runs a real-time simulation, so simulating the behavior of a long experiment can take a while. The simulation can be sped up by controlling the latency of the generated responses, but it isn't possible to rapidly simulate how the experiment responds to realistic response times. To the best of our knowledge, neither the Qualtrics nor Inquisit simulation features allow the researcher to simulate portions of the experiment while manually completing others.

In the remainder of the paper, we describe how we implemented a simulation tool for behavioral experiments created with jsPsych. This tool supports all four of the features that we identified above. For most experiments, the researcher can simulate the entire experiment by changing a single line of code. If the situation warrants it, the researcher also has the option to simulate portions of the experiment while manually completing others and/or customize the simulation by adjusting its parameters. The simulation tool is capable of running in two distinct modes; one allows for real-time observation of the simulation and the other allows for nearly instantaneous runs that generate data as if the experiment were run in real time. After describing the tool, we explore four use cases for jsPsych's simulation modes and features.

Implementation

The problem of simulating plausible behavior The word processor that we are using to write this paper has at least 60 clickable options on the screen and responds to many different keyboard shortcuts. There are a lot of plausible behaviors that a user could perform. To robustly test an application, a software engineer will want to test as many of the behaviors that users might perform as possible. Failing to test a critical situation might result in a catastrophic bug or security vulnerability[3]. Because of the need to test a wide range of possible behaviors across a wide range of possible applications,

[1] Documentation of Qualtrics' test response feature: https://www.qualtrics.com/support/survey-platform/survey-module/survey-tools/generating-test-responses/

[2] Documentation of Inquisit's test monkey feature: https://www.millisecond.com/support/docs/current/html/howto/howtotestexperiment.htm

[3] One approach that software engineers use is "fuzzing" or "fuzz tests." This involves generating many different possible inputs randomly in an effort to find edge cases that cause a bug.

a general-purpose tool for simulating user behavior typically requires substantial configuration to tailor the simulation to the particular application.

Experiments, however, are usually far more constrained than most software applications. For example, consider the computerized Corsi block task. The set of *plausible* behaviors that a reasonable participant might engage in is limited to clicking on one or more of the squares on the screen at a rate of about one square every second. The set of *possible* behaviors is larger, including things like not clicking any squares for several minutes while taking an unscheduled break from the experiment, but given the limited set of options available compared to most software applications, there is still a fairly narrow range of behaviors that could happen.

With a narrow space of plausible behaviors, it is possible for the developers of the simulation tool to build in information about these task-specific behaviors. Thus, the simulation tool requires less application-specific configuration and becomes easier to use. For example, in a task where the participant needs to choose between two alternative choices using a keyboard press, the simulation tool can simply sample randomly from the space of plausible behaviors and provide a reasonable approximation of what a participant might do in an experiment. The researcher building the experiment might not even need to configure the simulation tool at all.

For a developer of the experiment simulation tool, the problem is to figure out what the space of plausible behaviors is at any given moment based on the code created by the researcher. If this space can be determined from the code needed to run the experiment, then the software can randomly pick plausible behaviors to generate a basic simulation. Customization of the simulation by the researcher can further refine the actions taken from that point. In the next section, we look at how one experiment framework works as a case study in how to determine the space of plausible behaviors.

Background on jsPsych experiments We implemented a simulation mode in version 7.1 of jsPsych. In order to understand how the simulation mode works, it is necessary to have a basic understanding of how experiments are created in jsPsych. jsPsych is a JavaScript framework for creating behavioral experiments that run in a web browser (de Leeuw, 2015). Experiments in jsPsych are constructed with jsPsych plugins. Plugins are the primary unit of abstraction in jsPsych; each plugin defines a self-contained trial or event and the data that should be recorded. For example, a plugin might display an image on the screen along with a set of buttons and record which button is clicked and the response time. Plugins vary substantially in terms of their generalizability. A plugin to show some text and wait for a keyboard response could be used for a huge range of tasks, while another plugin might provide a template for a very specific task, like moving-window self-paced reading. This flexibility of abstraction is a key feature of jsPsych, and part of what makes it a particularly powerful way of declaratively creating experiments. jsPsych v7.2 includes 50 plugins, and many more plugins have been developed by the research community (e.g., Callaway et al., 2017; Rajananda et al., 2018; Kuroki, 2021; Galang et al., 2021; Strittmatter et al., 2021; Cousineau, 2021; Gibeau, 2021; Donhauser & Klein, 2021; Kinley, 2022).

To create a jsPsych experiment, a researcher must specify a timeline. The timeline controls the order of events in the experiment, where each event is the execution of a plugin with a particular set of parameters. Timelines can also contain control logic, such as loops and conditional branches, as well as events that trigger the execution of code at different moments during the experiment, such as every time new data are recorded.

For the purpose of understanding how the simulation mode is built, the key thing to understand is that in a jsPsych experiment each moment is controlled by a single plugin. This means, as we will show in a moment, that we can simplify the problem of finding the space of plausible behavior to a problem that is specific to each individual plugin, regardless of the context that the plugin is used in. And while the terminology of "timelines" and "plugins" is specific to the jsPsych library, many other experiment-building tools have their own versions of these same design concepts, and these tools could take a similar approach to implementing simulations.

Building a simulation mode Our primary design goal for jsPsych's simulation mode was to allow a researcher to simulate an experiment without making any modifications to the code (other than a single line specifying that the experiment should be simulated). We reasoned that this would maximize the number of different ways that a simulation mode could be used. If simulating an experiment *requires* the user to make changes to the code or add additional parameters, such as specifying what an expected set of actions are for each trial, then the cost of simulating the experiment is higher and there will be fewer scenarios where the benefits justify the cost. There is also the risk that changes to the code to support simulation will accidentally change the behavior of the experiment, breaking the validity of the simulation.

Researchers create jsPsych experiments by assembling a timeline that contains information about each trial/event in the experiment. Because the timeline contains information about what plugin is being used for a trial, we have a pretty good idea about what plausible user behavior is during that piece of the experiment. This is true even for plugins that are extremely general in their use cases. For example, one commonly used plugin is the html-keyboard-response plugin. This plugin displays arbitrary HTML content on the screen, which means the screen could contain pretty much anything,

and waits for the participant to press a key on the keyboard. The plugin has a few adjustable parameters, including the allowable set of keys and the duration of the trial. This is enough information to generate plausible responses. We can randomly sample from the set of allowable keys and randomly sample a response time from a realistically skewed distribution, like an exponentially modified Gaussian distribution. If the response time is longer than the duration of the trial, we can simulate a nonresponse.

We can also simulate more complex behavior, like filling out a questionnaire. With free-form and open-ended response formats, the space of possible behaviors starts to get larger, but it is still constrained enough that we can simulate plausible behavior. For instance, text box responses can be filled in with random words. We are also able to use some information to further constrain the behavior; for example, if a text box is small, we can assume that shorter answers are more likely, and if a response is required, then we can ensure a response is given.

Regardless of complexity, in all cases the simulation logic can be part of the code for the plugin. A plugin developer can determine the space of plausible behaviors for that plugin and provide a set of default simulation behaviors that cover that space. When a jsPsych experiment is run in simulation mode, the jsPsych engine can then check if a plugin supports simulation and then hand over execution to the plugin.

The default simulated behavior will vary in realism depending on the plugin. For instance, if the trial is a multiple-choice survey, then the default simulation behavior will be to pick out choices at random. These responses may realistically represent some (rather low effort) participants. If the trial, however, contains a text box to answer an open-ended question, then the default behavior is to simulate a random string of words, which is unlikely to match any real participant. However, for many purposes, realistic responses are not needed, and the default simulation behavior should suffice. When realism is needed, we also provide a way for users to write code to change the default behavior to something more realistic for their own experiments.

To allow finer control over the simulation, we allow the programmer to specify the particular data that should be generated for a trial. This simulation-specific parameter can be specified directly in the code for a trial, or the user can provide a list of trials and their associated data. The latter option allows for users to create different files that could be imported, each with different simulation options. We cover the ways that this could be used in the next section.

Finally, we implemented two different simulation modes: visual and data-only. In both modes, jsPsych runs through the timeline using the same control logic as if the experiment was running normally. When a plugin is reached, simulated data is generated based on the parameters for that trial. In data-only mode, the trial ends immediately and the simulated data are stored as if generated by a participant. This means that an experiment will complete nearly instantly. In visual mode, the normal trial code is executed, but the simulated data are used to generate the actions that the human participant would normally perform. For example, if the simulated data for a trial indicate that the spacebar is pressed after 1500 ms, then the simulation mode will create a keydown event at 1500 ms with the key being the spacebar. In this mode, the researcher can observe the experiment running in the browser in real time, without needing to interact with it.

Documentation for jsPsych's simulation modes, which goes into detail about the different modes and explains the options for customizing the simulation, is available at https://www.jspsych.org/7.2/overview/simulation/.

Illustration of use cases

In this section we describe four scenarios in which we that think simulation mode can be useful for researchers. For each scenario, we describe the problem that researchers face and discuss how simulation can help solve the problem. We also provide an example of implementing a simulation in jsPsych for each scenario. The full set of examples can be found at https://github.com/jspsych/simulation-examples. Each example includes a live demonstration that can be run in a web browser as well as the code.

Use case 1: Automated testing

Code: https://github.com/jspsych/simulation-examples/tree/main/use-case-1-testing
Live example (demo task): https://jspsych.github.io/simulation-examples/use-case-1-testing/demo-task/
Live example (basic testing): https://jspsych.github.io/simulation-examples/use-case-1-testing/basic/
Live example (testing with Jasmine): https://jspsych.github.io/simulation-examples/use-case-1-testing/with-jasmine/

Like all software, computer-based experiments will occasionally contain errors. Errors in experiment code are costly. A worst-case scenario might be that the error is never detected, and data are collected and published without anyone being aware that the data are invalid because of a critical error. We have no way of knowing how many papers this scenario describes given that the premise is that these are undetected errors, but there are plenty of reported cases where a programming error detected after publication led to retraction of the paper (e.g., Dolk et al., 2019; Grave et al., 2021) or major corrections that change the interpretation of the original finding (e.g., Strand, 2020). Errors are also often detected during the development of the experiment, usually through time-consuming testing, or during the execution

of the experiment, following some amount of data collection. These errors might be less consequential because they are caught before publication, but they are costly in terms of the time that is required to check for, identify, and fix them, and potentially in terms of wasted participant time and reimbursements.

Part of what makes errors time-consuming to catch is that systematic testing of experiments is difficult. To ensure that the behavior of the experiment is correct, we may (i) test how the experiment behaves with different kinds of input, (ii) run it through experiments a handful of times once it is complete, (iii) pilot the experiment on a small number of subjects, and/or (iv) analyze data from these pilot runs. All of these approaches have one thing in common, which is that they generally require a human in the loop to be the source of the behavior that is being tested. This makes testing time-consuming. Furthermore, human involvement makes this approach more error-prone; for instance, researchers may comprehensively test their experiment, but then forget to repeat all tests following a code change.

As with any programming, time spent fixing errors is to be expected. One study of software engineers livestreaming their work on open-source projects found that, on average, nearly half the development time was spent fixing bugs (Alaboudi & LaToza, 2021). Because errors are an unavoidable problem of software development, a variety of tools exist to help programmers find and fix bugs. These tools help reduce the cost of errors by making them both easier to find and faster to fix. One of the most widely used tools is automated testing. Automated testing tools require the programmer to craft a series of test cases that specify what the programmer expects to happen in a given situation, and then the tool can run these test cases automatically and notify the programmer which tests fail. The programmer can use the pattern of test results to identify whether and where bugs exist. This allows for systematic and repeatable testing of the code.

For many kinds of tests, the user's behavior must be simulated. This allows the programmer to specify tests—for example, *If the user clicks this button, then this image should disappear*. General-purpose user simulation tools could be used to automate the testing of experiment code, and certainly some researchers with the necessary programming skills have done so (e.g., Sochat et al., 2016; Pronk et al., 2021). But, as we discussed earlier, customizing such tests is time-consuming and researchers may reasonably decide that the time investment is not worth it in most cases.

What a built-in simulation mode offers is the option to use simulations for the automated testing of some aspects of the experiment, such as its logic and flow. Automated testing may seem like an unnecessary step, and this may be true for simple experiments. But modern behavioral experiments are often more complicated than they first seem, with multiple events and parameters that can change depending on the participant's behavior during the experiment. Consider, for instance, an experiment where the participant must complete a set of practice trials, and only move on to the main task if they've reached a given performance threshold. In this case the researcher should test that the experiment continues when the performance threshold is reached, otherwise the experiment should stop. But what if the researcher wants to allow the participant multiple attempts to reach the performance threshold? Now they must also test that the experiment (i) loops back to the instructions and repeats the practice trials after a failed attempt, (ii) responds correctly to the participant's performance after each attempt, and (iii) ends after a certain number of failed attempts. To test this, the experimenter would need to run through the practice trials multiple times while paying close attention to their response accuracy within and across the trial sets.

The use of performance thresholds to determine the experiment flow is just one common example of how experiments can become surprisingly complex. There are, of course, many other experiment procedures and task paradigms in which the precise series of events that should occur depends on the responses made during the experiment. And, in contrast to the performance threshold example, in some cases it can be difficult for the researcher to intentionally trigger a specific condition and/or easily see whether the experiment has produced the intended behavior. Adaptive tasks use the participant's response timing and/or accuracy to continuously adjust the task difficulty, often in subtle ways and following complex rules. For instance, the go-no-go/stop-signal task requires that the participant responds as quickly as possible to every "go" stimulus unless a stop signal is shown, and the timing of the stop signal is continuously adjusted throughout the experiment in order to achieve a target response accuracy. A researcher testing their code for this task will likely be unable to intentionally produce the full range of response patterns that they might want to test. In this case, the researcher would benefit from being able to programmatically generate the patterns of responses that should be tested, and examining the resulting data files to determine whether the appropriate shifts were made to the stimulus timing. Indeed, the researcher may want to publish their experiment simulations and resulting data files as a form of validation alongside their experiment.

In our example code for this use case, we have created a very simple demo task (see the "demo task" link at the top of this section), along with two tests that we can immediately run using jsPsych's simulation mode (see the "simulation with tests" link). In the demo task, numbers are presented one at a time, and for each number, the participant must identify if it is odd or even as quickly as possible by pressing the "A" key for odd and the "L" key for even. The experiment begins with an instruction check, in which the participant sees two example numbers and must indicate,

by clicking either the "A" or "L" button on the screen, the correct keyboard response for each one. If the participant responds incorrectly to either of these trials, they should see a message saying that they must repeat the instructions, which triggers the repetition of the instructions page and check trials. Once the participant has responded correctly to both instruction check trials, they should move on to the main task. The main task randomly presents single-digit numbers, with the constraints that (i) there should be the same number of even and odd trials, and (ii) there should be the same number of small (≤ 4) and large (≥ 6) trials.

The experiment code also contains a custom function that runs at the end of the experiment which checks the data to determine whether two test scenarios have passed or failed. The first test is: was there an equal number of all trial types (even/odd x small/large) during the main task? The second test is: if there was one or more incorrect responses during the instructions check, then did instructions repeat (or, if all responses were correct during the instructions check, then did the instructions not repeat)? For the purposes of the demonstration, we have written the data checks into the experiment code, but these data checks could be moved into any other program (e.g., R) and run on the data separately.

For researchers who are new to the idea of automated testing, we offer a few suggestions about what to focus on when adding tests to an experiment. (1) Test portions of the experiment that change in response to user input. This might include testing the behavior of the experiment when a response is too slow, no response is detected, or when performance is below threshold during a set of practice trials. Include both expected and unexpected input as test cases to check that the experiment behaves properly under atypical conditions. (2) Test any constraints on randomization. Randomized code is difficult to test because errors may occur only some of the time. Adding test cases to catch errors in randomization (such as too many trials of one type or incorrect spacing between conditions) and running these tests many times can help catch any such errors before data collection begins. (3) Test factors that are key to the experimental manipulation. In experiments, some errors are simply more costly than others. Use tests to verify that condition assignment, stimulus selection, and other factors that could potentially invalidate the results of the experiment are working as expected.

Use case 2: Generating simulated data sets for analysis scripts

Code: https://github.com/jspsych/simulation-examples/tree/main/use-case-2-generate-dataset
Live example: https://jspsych.github.io/simulation-examples/use-case-2-generate-dataset/

Creating an analysis pipeline prior to collecting data is a useful practice when building an experiment. It is becoming increasingly common as more studies are preregistered and helps avoid making analysis decisions based on desired outcomes. It also helps avoid integration errors between the experiment and analysis. For example, in our experience an easy mistake to make is to forget to include a piece of key information in the data generated by the experiment, such as a label indicating which condition a trial belongs to. This kind of error might not be caught by the kinds of automated testing we described in the previous scenario, since it is difficult to detect until the analysis is begun.

Generating an analysis pipeline in advance requires knowing the structure of the input to that pipeline. A typical solution is to generate sample data sets by completing the experiment several times. This can be done by the researcher(s) or with a small number of pilot participants. In either case, the process can be time-consuming because of the human in the loop. It is especially costly when errors are discovered, because the experiment must be run again (and again) as changes are made and tested.

With simulation, the experiment does not need to be executed in real time and sample data sets can be generated rapidly. Not only does this make developing an analysis pipeline less time-consuming, but it also changes the scope of what kinds of test analyses are reasonable to conduct. For example, the analysis pipeline can be developed with a data set that is the same size and structure as the target sample without requiring the researchers to invest substantial resources in pilot data or manually completing the experiment. Furthermore, the analysis can be simulated with different kinds of results by changing the parameters of the simulation. This could be used to validate parts of the analysis that are contingent on participant behavior (exclusion of outliers, for example), testing model assumptions, and generating visualizations that work under different outcomes.

Our example for this scenario highlights the ability to customize simulation parameters to generate different outcomes. We created a simple Stroop task with 48 trials, half of which are congruent and half incongruent. The experiment is simulated in data-only mode, and a JSON file is automatically downloaded when the simulation is complete. We used the `simulation_options` parameter to create different response time distributions for the congruent and incongruent trials, allowing for control over the expected effect size. We also included two parameters to control the probability of a correct response for each kind of trial. Finally, we included a probability that a subject is inattentive and responding as fast as possible. This would allow the researcher to test that their analysis script can detect and exclude those subjects. All of the parameters of the responses are in a configuration script, and the researcher could create multiple configuration scripts to simulate the experiment under different conditions.

To generate a data set, the researcher could simply refresh the web browser several times. Each refresh will generate a unique data set for one subject. It is also possible to script it, running the jsPsych experiment in a loop, for example, but for a small or medium-sized data set it is probably faster to just refresh the page since the simulation happens nearly instantly.

The customizable parameters of the simulation also allow the researcher to explore intuitions about the design of the experiment and the data that are generated. In our Stroop example, for instance, the researcher could vary the number of trials to explore how it affects the reliability of the aggregate measures. This could be useful for exploring different parameters for a power analysis. This is, of course, possible to do in other ways, such as simulating the data directly in a programming language, but once the researcher has the experiment built it may be less work to simulate the experiment directly rather than building a separate simulation of the data.

Use case 3: Speeding up development work

Code: https://github.com/jspsych/simulation-examples/tree/main/use-case-3-partial-simulation
Live example: https://jspsych.github.io/simulation-examples/use-case-3-partial-simulation/

When building an experiment, researchers will often need to test some piece of it. For example, imagine having already set up a series of trials to explain how the task works, and then working on developing the task itself. This is a phase of development where lots of iteration is typically required. Code will be changed, and the task will be tested—over and over and over again.

During this phase it is especially annoying and time-consuming to have to move through parts of the experiment that are not relevant to the part under active development. Often those parts of the code can simply be temporarily removed from the experiment to make it faster to iterate, but sometimes it is not so simple. For example, consider a situation where the researcher is working on a screen that will show the participant some feedback about their performance over the past twenty trials. This screen needs to get the data from the last twenty trials in order to function properly. The typical solution to this problem would probably be to just try and get through the 20 trials as quickly as possible and try to minimize the number of iterations needed to develop the feedback screen. This is still time-consuming, especially if there are parameters that force a delay, like a minimum trial duration. In that case the code could be edited to shorten or remove the delay, but this creates new problems where the code needs to be reverted back to the correct state before the experiment is deployed.

Simulation offers an alternative. If the researcher can simulate a portion of the experiment, then the parts of the experiment that are not under active development can be simulated while the rest of the experiment runs normally. When jsPsych is run in simulation mode, the researcher can specify that some trials should *not* be simulated. Those trials instead happen as if the experiment were running normally. Using this feature, the developer could simulate the entire experiment except for the piece that is actively under development. This allows for faster iterative development. While there are other ways of achieving a similar outcome, such as commenting out portions of the code that can be skipped, simulation has the advantages that (i) the code does not need to be altered and (ii) any data generated by the simulated trials will exist as if those trials had run for real, allowing the developer to test portions of the experiment that are contingent on actions taken earlier.

For the example of this scenario, we imagined a visual search task in which the researcher is adding a feedback screen. The researcher has already constructed the main task. Participants must identify whether a red X is present in a set of 84 trials. The trials vary in terms of the set size (4, 6, or 8 total objects), whether a red X is present, and whether the search is a feature search (all other objects are black Xs, so there is only one feature to attend to) or conjunction search (the other objects are a mix of black Xs and red Os, so shape and color must both be attended to). In all trials there is a 1-second presentation of the fixation point before displaying the search array. Completing all 84 trials takes 3–5 minutes.

The goal of the feedback screen is to show participants their average response time for different kinds of trials in the experiment. This would be useful information to show if the experiment was being used in a classroom to demonstrate the difference between feature search and conjunction search, or in training studies in which participants receive feedback about their performance after each session. To show this feedback, the researcher needs to get the data from the 84 trials, partition it into the correct groupings, and calculate summary scores.

Simulating all of the visual search trials will speed up development work, and save the researcher from the boredom of repetitively testing the experiment. The researcher may also want to do something more sophisticated, like show a graph based on these data points. We don't include that in the example in order to keep things as simple as possible, but any increase in complexity will benefit from simulation because of the speed-up it enables in development.

To simulate all of the trials before the feedback screen, we ran the jsPsych experiment in data-only simulation mode and used the `simulation_options` parameter of the feedback trial to turn off its simulation mode by setting `simulate: false`. Opening the experiment in a web browser will load the feedback screen almost immediately. Refreshing the page will show that new data are generated each time the experiment is loaded. Here the researcher

could also specify simulated data directly to see if the feedback screen is working properly, though that would make this scenario a little more like the automated testing cases. Our main goal here was to show that, even without careful specifying of test cases, simulation mode is useful for speeding up development of portions of the experiment that are dependent on other portions.

Use case 4: Generating animations of participant behavior

Code: https://github.com/jspsych/simulation-examples/tree/main/use-case-4-replay
Live example: https://jspsych.github.io/simulation-examples/use-case-4-replay/

At some point a researcher may want to show other researchers what an experiment looks like. For some experiments, an image or short sequence of images might be able to convey the critical information. In other cases, a movie might be more effective, particularly with animated stimuli or experiments where the specific timing of events is meaningful. Even without these kinds of features, a movie might convey what it is like to complete the experiment more effectively than images.

It is of course already possible to record movies of experiments. Screen recording software is easily accessible[4]. Simulating an experiment doesn't make it easier to create a movie. But it does make some options for visualization easier than they were.

One useful tool that simulation mode offers is to quickly create an accelerated playthrough of the experiment. For example, using the visual simulation mode in jsPsych and using the `simulation_options` to set all response times to relatively short values results in playback that is analogous to a time-lapse movie. This can be useful for quickly demonstrating the entirety of the experiment to other researchers.

A second, more unique possibility is the ability to "playback" participant behavior. Using the data collected from a participant as the input to the simulation results in a visual playback of what the participant did in the experiment. This could be used for different purposes. It could be used to communicate better, by showing what typical behavior looked like in different conditions or the variability of behavior. It could also be used in the development process, allowing researchers to rewatch pilot participants complete the experiment to better understand how participants are actually interacting with the tasks. Finally, it could be used to build better intuitions about expected behavior in the experiment. For example, sometimes researchers use too-fast response times as a decision criterion for excluding participants. It can be hard to decide what is a reasonable amount of time to have plausibly engaged with the task. Simulating trials from pilot participants who are completing the experiment quickly can give the researcher a sense for what a reasonable cutoff could be.

For our example of this scenario, we wanted to illustrate how to replay the data from a participant. We created a short serial response time task that allows you to complete the experiment once as the participant and then watch a playback of your behavior in visual simulation mode. We implemented this by controlling the parameters of the simulation on a trial-by-trial basis. The data from the actual run of the experiment are used to generate the `simulation_options` for the simulated run. Each trial is uniquely identified by its trial index, and we set the `simulation_options` separately for each trial index. This can all be scripted rather efficiently because jsPsych has a variable that denotes the current trial's index at any point.

A few caveats about this scenario: In jsPsych, the accuracy of the visual simulation will depend on a few things. First, plugins may not record enough detail about a participant's behavior to recreate the exact sequence of events. For example, the various plugins for collecting form responses (filling in text boxes, selecting options in a list, etc.) usually do not record the exact timing of each entry, let alone the exact timing of each keystroke in a text entry field, opting instead to just record the total time to complete a page. In this case, jsPsych will be able to simulate the total length of time on the page, but the timing of individual responses will not match the participant's behavior. In theory, this problem could be solved by recording more detailed data about the participant's behavior. It is possible that a future version of jsPsych may support the option to turn on more detailed logging, but at the moment, only the actions that are critical to data collection are recorded. Second, because of the way that the visual simulation mode works, the timing of events may differ by a few milliseconds from the participant's actual behavior. The reasons for this are explained in the documentation[5].

The other critical factor for replaying user behavior is that any randomness in the experiment must be reproducible. For example, if the experiment has a randomly ordered set of trials, then the same order must be generated during the simulation. A common approach to solving this problem is to use a seed for a pseudorandom number generator. Many programming languages support seeding the random number generator by default, but JavaScript is an exception.

[4] Our favorite option at the time of writing this article is https://gifcap.dev. It is free, open source, and runs entirely in a browser so no additional software is needed. It can record screens or windows and outputs GIF images that are easy to share online, archive in experiment repositories, or embed in a slideshow.

[5] https://www.jspsych.org/7.2/overview/simulation/#visual-mode

jsPsych, as of version 7.2, implements a seedable random number generator. The seed can be stored in the data and used to reproduce the exact pattern of random choices in the experiment.

Conclusion

The ability to simulate user behavior can help researchers build more robust experiments. Potential use cases include automated testing of experiment code, generating simulated data files for testing analysis pipelines, automating repetitive tasks during development, and visualizing participant behavior. The scenarios we described can apply to experiments developed using any programming language or experiment software, but require expertise with general-purpose user interface (UI) testing tools to set up. Abstractions in experiment building software that constrain the possible space of behaviors allow for the automated generation of plausible behaviors which, though not realistic, are still useful in many scenarios. As jsPsych's plugin-based architecture affords this kind of abstraction, we implemented visual and data-only simulation modes in jsPsych. Adding simulation features into experiment building software shifts the implementation costs from the researcher to the software developer, and can therefore significantly tilt the cost-benefit tradeoff of using simulation and change what is feasible for researchers to do when building experiments.

In addition to the use cases that we've described above, the existence of a built-in way to simulate behavior may encourage innovation in how we build and validate experiments. Developers can build additional tools that make use of simulation to provide new functionality for researchers. For example, software engineers often use code coverage tools to report the percentage of code that is run during testing. With automated simulation, it is possible to build an "experiment coverage" tool that could report the percentage of an experiment that is run during testing. This could be especially useful for experiments that involve a lot of branching and randomization. Such a tool would give researchers the ability to report on the comprehensiveness of testing that was done.

Another example is dealing with the ever-present problem of browser and device variation. If you have run an online experiment, you are likely familiar with the problem that an experiment may look as expected on one machine but appear or behave differently on another due to changes in browser, operating system, or monitor size. It can be difficult to test an experiment comprehensively across the range of device and browser combinations that online participants may use to complete it. This problem is exacerbated by the fact that web browsers undergo frequent updates and changes that could suddenly break an experiment, which means that experiments often benefit from repeated browser compatibility checks. The visual simulation mode could be used in conjunction with tools that automatically run code across different browsers and operating systems (e.g., BrowserStack) to verify that experiments look and behave correctly, regardless of the device and browser used.

In sum, the ability to simulate user behavior is an important tool in software engineering, but one that has been difficult to use in the context of building experiments. We hope that making this kind of tool more accessible to researchers will help researchers build more robust experiments and encourage the development of additional infrastructure to make these features more accessible in the future.

Acknowledgements Development of jsPsych's simulation mode was funded by an award from the Mozilla Foundation.

Author contributions All of the authors contributed to the development of jsPsych's simulation mode, writing the manuscript, and editing the manuscript. JRD developed the examples for this paper and BL helped edit the examples.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

Aaronson, D., Grupsmith, E., & Aaronson, M. (1976). The impact of computers on cognitive psychology. *Behavior Research Methods & Instrumentation, 8*(2), 129–138.

Alaboudi, A., & LaToza, T. D. (2021). An exploratory study of debugging episodes. arXiv preprint arXiv:2105.02162.

Almaatouq, A., Becker, J., Houghton, J. P., Paton, N., Watts, D. J., & Whiting, M. E. (2021). Empirica: A virtual lab for high-throughput macro-level experiments. *Behavior Research Methods*, 1–14.

Arce, T., & McMullen, K. (2021). The Corsi block-tapping test: Evaluating methodological practices with an eye towards modern digital frameworks. *Computers in Human Behavior Reports*, 100099.

Balietti, S. (2017). nodeGame: Real-time, synchronous, online experiments in the browser. *Behavior research methods, 49*(5), 1696–1715.

Berch, D. B., Krikorian, R., & Huha, E. M. (1998). The Corsi block-tapping task: Methodological and theoretical considerations. *Brain and Cognition, 38*(3), 317–338.

Brainard, D. H. (1997). The psychophysics toolbox. *Spatial Vision, 10*(4), 433–436.

Brunetti, R., Del Gatto, C., & Delogu, F. (2014). eCorsi: Implementation and testing of the Corsi block-tapping task for digital tablets. *Frontiers in Psychology, 5*, 939.

Callaway, F., Lieder, F., Krueger, P., & Griffiths, T. (2017, July 25). Mouselab-MDP: A new paradigm for tracing how people plan. https://doi.org/10.31219/osf.io/7wcya

Corsi, P. (1972). *Memory and the medial temporal region of the brain* [Doctoral dissertation, McGill University]. https://escholarship.mcgill.ca/concern/theses/05741s554

Cousineau, D. (2021, August 10). Born-Open Data for jsPsych. https://doi.org/10.31234/osf.io/rkhng

de Leeuw, J. R. (2015). jsPsych: A JavaScript library for creating behavioral experiments in a web browser. *Behavior Research Methods, 47*(1), 1–12. https://doi.org/10.3758/s13428-014-0458-y

Dolk, T., Freigang, C., Bogon, J., & Dreisbach, G. (2019). Retraction notice to "Auditory (dis-)fluency triggers sequential processing adjustments" [ACTPSY 191 (2018) 69–75]. *Acta Psychologica, 198*, 102886. https://doi.org/10.1016/j.actpsy.2019.102886

Donhauser, P., & Klein, D. (2021, October 16). Audio-Tokens: A toolbox for rating, sorting and comparing audio samples in the browser. https://doi.org/10.31234/osf.io/3j58q

Forster, K. I., & Forster, J. C. (2003). DMDX: A windows display program with millisecond accuracy. *Behavior Research Methods, Instruments, & Computers, 35*(1), 116–124.

Galang, C. M., Malik, R., Kinley, I., & Obhi, S. S. (2021). Studying sense of agency online: Can intentional binding be observed in uncontrolled online settings? *Consciousness and Cognition, 95*, 103217.

Gibeau, R. (2021). The corsi blocks task: Variations and coding with jsPsych. *The Quantitative Methods for Psychology, 17*(3), 299–311. https://doi.org/10.20982/tqmp.17.3.p299

Grave, J., Soares, S. C., Morais, S., Rodrigues, P., & Madeira, N. (2021). Retraction notice to "The effects of perceptual load in processing emotional facial expression in psychotic disorders" [Psychiatry Research Volume 250C April 2017, pages 121 - 128]. *Psychiatry Research, 303*, 114077. https://doi.org/10.1016/j.psychres.2021.114077

Gureckis, T. M., Martin, J., McDonnell, J., Rich, A. S., Markant, D., Coenen, A., et al. (2016). psiTurk: An open-source framework for conducting replicable behavioral experiments online. *Behavior Research Methods, 48*(3), 829–842.

Hartshorne, J. K., de Leeuw, J. R., Goodman, N. D., Jennings, M., & O'Donnell, T. J. (2019). A thousand studies for the price of one: Accelerating psychological science with Pushkin. *Behavior Research Methods, 51*(4), 1782–1803.

Henninger, F., Shevchenko, Y., Mertens, U. K., Kieslich, P. J., & Hilbig, B. E. (2021). lab.js: A free, open, online study builder. *Behavior Research Methods*. https://doi.org/10.3758/s13428-019-01283-5

Kinley, I. (2022, March 7). A jsPsych plugin for visual analogue scales. https://doi.org/10.31234/osf.io/avj92

Kuroki, D. (2021). A new jsPsych plugin for psychophysics, providing accurate display duration and stimulus onset asynchrony. *Behavior Research Methods, 53*(1), 301–310.

Mathôt, S., Schreij, D., & Theeuwes, J. (2012). OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behavior Research Methods, 44*(2), 314–324.

Milner, B. (1971). Interhemispheric differences in the localization of psychological processes in man. *British Medical Bulletin, 27*(3), 272–277.

Ouyang, L., Tessler, M. H., Ly, D., & Goodman, N. (2016). Practical optimal experiment design with probabilistic programs. *arXiv preprint arXiv:1608.05046*.

Papoutsaki, A., Sangkloy, P., Laskey, J., Daskalova, N., Huang, J., & Hays, J. (2016). Webgazer: Scalable webcam eye tracking using user interactions. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence-IJCAI 2016*.

Peirce, J. W. (2007). PsychoPy—psychophysics software in Python. *Journal of Neuroscience Methods, 162*(1-2), 8–13.

Pronk, T., Hirst, R., Wiers, R., & Murre, J. M. J. (2021, December 7). Can we measure individual differences in cognitive measures reliably via smartphones? A comparison of the flanker effect across device types and samples. https://doi.org/10.31234/osf.io/2kdca

Rajananda, S., Lau, H., & Odegaard, B. (2018). A random-dot kinematogram for web-based vision research. *Journal of Open Research Software, 6*(1).

Richardson, J. T. (2007). Measures of short-term memory: A historical review. *Cortex, 43*(5), 635–650.

Sochat, V. V., Eisenberg, I. W., Enkavi, A. Z., Li, J., Bissett, P. G., & Poldrack, R. A. (2016). The experiment factory: Standardizing behavioral experiments. *Frontiers in Psychology, 7*, 610.

Stoet, G. (2010). PsyToolkit: A software package for programming psychological experiments using Linux. *Behavior Research Methods, 42*(4), 1096–1104.

Strand, J. (2020, March 24). Scientists Make Mistakes. *I Made a Big One. [Blog Post]*. Retrieved from https://elemental.medium.com/when-science-needs-self-correcting-a130eacb4235

Strittmatter, Y., Spitzer, M., & Kiesel, A. (2021, July 12). A random-object-kinematogram plugin for web-based research: implementing oriented objects enables varying coherence levels and stimulus congruency levels. https://doi.org/10.31234/osf.io/hmq4u

Suchow, J., & Griffiths, T. E. (2016, December). Rethinking experiment design as algorithm design. In *30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain*.

Yang, X., & Krajbich, I. (2021). Webcam-based online eye-tracking for behavioral research. *Judgment and Decision Making, 16*(6), 1485–1505.

Zehr, J., & Schwarz, F. (2018). PennController for Internet Based Experiments (IBEX). https://doi.org/10.17605/OSF.IO/MD832

Open practices statement All the code is available at https://github.com/jspsych/simulation-examples.