Behav Res (2015) 47:1–12
DOI 10.3758/s13428-014-0458-y

jsPsych: A JavaScript library for creating behavioral experiments in a Web browser

Joshua R. de Leeuw

Published online: 28 March 2014

Abstract Online experiments are growing in popularity, and the increasing sophistication of Web technology has made it possible to run complex behavioral experiments online using only a Web browser. Unlike with offline laboratory experiments, however, few tools exist to aid in the development of browser-based experiments. This makes the process of creating an experiment slow and challenging, particularly for researchers who lack a Web development background. This article introduces jsPsych, a JavaScript library for the development of Web-based experiments. jsPsych formalizes a way of describing experiments that is much simpler than writing the entire experiment from scratch. jsPsych then executes these descriptions automatically, handling the flow from one task to another. The jsPsych library is open-source and designed to be expanded by the research community. The project is available online at www.jspsych.org.

Keywords Online experiments · JavaScript · Amazon Mechanical Turk

Putting an experiment online, particularly when it is coupled with a crowdsourcing marketplace like Amazon's Mechanical Turk, enables extremely rapid data collection at much lower cost than a comparable experiment in a laboratory environment. Although online experiments will not work for every kind of experiment, some studies produce similar results for laboratory and online experiments across a range of behavioral tests (Buhrmester, Kwang, & Gosling, 2011; Crump, McDonnell, & Gureckis, 2013; Goodman, Cryder, & Cheema, 2013; Paolacci, Chandler, & Ipeirotis, 2010; Zwaan & Pecher, 2012). Although some concerns remain about certain kinds of behavioral experiments, such as cognitive-learning and priming paradigms (Crump et al., 2013), the general consensus seems to be that online experiments produce results similar to those of laboratory experiments for many tasks. Given the cost and time savings of online experiments, plus such additional benefits as easy access to a different subject pool demographic than undergraduate college students (Ross, Irani, Silberman, Zaldivar, & Tomlinson, 2010), the popularity of online experiments seems likely to grow over the next several years.

In order to conduct an online experiment, researchers must design experiments that either can be downloaded to a subject's local computer or can be run in a Web browser. There are many options for running experiments in a Web browser: Platforms such as Qualtrics (www.qualtrics.com) and Amazon's Mechanical Turk (www.mturk.com) have simple templates that can be customized with a point-and-click-style interface. WEXTOR (Reips & Neuhaus, 2002) is an online graphical user interface designed for building Web-based survey-like experiments that offers more flexibility than does something like Mechanical Turk, but the content that can be included is limited by the built-in functionality of the system. Alternatively, experimenters can write their own customized code to achieve designs that may not be possible otherwise. With the increasing sophistication of Web technology, it is now possible to create many different kinds of computer-based behavioral experiments in a Web browser. In particular, the adoption of HTML5 by all of the major Web browsers enables programmatic control over both pixel- and vector-based graphics embedded in a website, allowing rich graphical interactions akin to playing a videogame. However, programming these experiments can be challenging and may require substantial experience in Web development, particularly for more complicated designs.

This article introduces an open-source JavaScript library called jsPsych, which accelerates the development process for browser-based experiments. The library contains code to perform a variety of tasks that are common across behavioral

J. R. de Leeuw (✉)
Department of Psychological & Brain Science, Cognitive Science Program, Indiana University, Bloomington, IN, USA
e-mail: jodeleeu@indiana.edu

experiments, such as displaying a stimulus, getting a response, and measuring response times. It also formalizes a structure for describing experiments, which enables experimenters to create experiment descriptions, and thus the corresponding experiment, by writing only a few lines of code. The library is designed so that experiments can be assembled from different components, called *plugins*. By providing plugins that can be used in different experimental contexts, jsPsych allows researchers to create custom experiments without generating all of the necessary code from scratch. Since the library is based on a modular architecture, researchers can share and utilize new plugins as they are developed, without any centralized modification to the library. The design of the library is covered in the next section of this article.

The main purpose of jsPsych is to control the content that the subject will interact with during the experiment. Note that jsPsych will not handle several other aspects of creating an online experiment: For example, although jsPsych will record all of the data generated by a subject, it will not store the data in a database or other permanent location on the Web server. The experimenter must decide how to handle and implement data storage. (The online documentation contains examples of how to do this, but there are many possible solutions, and jsPsych is agnostic as to which to use.) If jsPsych is used to create an experiment for Mechanical Turk or other crowdsourcing marketplace, numerous steps must be gone through in order to launch the experiment on these platforms (for an excellent primer, see Mason & Suri, 2012). One possibility is to combine jsPsych with something like psiTurk (McDonnell et al., 2012), a framework that handles all of the "back-end" aspects of running an online experiment, such as hosting the experiment on a server and interfacing with Mechanical Turk to get subjects to complete the experiment. jsPsych complements this library nicely. psiTurk can be used to serve a jsPsych-based experiment, simplifying both the front-end (what the subject sees; jsPsych's specialty) and back-end (server hosting and interfacing with Mechanical Turk, psiTurk's specialty) aspects of online experiments. Finally, it is worth noting that jsPsych can be used for offline experiments, as well. Websites can be developed and hosted locally on a laboratory computer, which subjects can then complete without ever connecting to the Internet. Since there are very few differences between developing a browser-based experiment for the lab and for an online environment, the same code can be used to run the experiment in both locations, if desired. This may be a particularly useful feature for checking the validity of online experiments against those conducted in the laboratory.

Overview of jsPsych

jsPsych is an open-source project hosted, at the time of writing this article, by GitHub. The URL for the project is www.jspsych.org. The project website contains all of the source code for the current and previous versions of jsPsych, as well as a wiki containing additional documentation. Extensions or modifications of jsPsych can be integrated into the project through GitHub, enabling the community of users to share code.

In the basic workflow of jsPsych, the experimenter generates a description of the experiment, making use of various jsPsych plugins, and then the core jsPsych library is used to execute the experiment in conjunction with the appropriate plugins. The core library provides a set of functions that are necessary across all experiments, but plugins are where the bulk of the action is. Each jsPsych plugin defines a task that a subject might engage in during an experiment. Examples include: reading instructions and pressing a key to continue, categorizing a stimulus and receiving feedback, responding as quickly as possible to a stimulus, and answering a survey question. At the time of writing, 14 plugins were well-tested and well-documented (see Table 1), and an additional 13 plugins either were in earlier stages of development or were not well-documented. The completed plugins can be found in the *plugins* folder (available at the jsPsych project website), and the plugins that are in development can be found in the *dev* folder, which is a subfolder of the plugins folder.

In describing the process of generating an experiment with jsPsych, I will use two terms: *trials* and *blocks*. *Trials* are the basic unit of jsPsych. Each plugin defines the process of executing a trial. For example, the jspsych-categorize plugin defines a trial consisting of presenting a stimulus to the subject, getting a response, and presenting feedback. All of this functionality is embedded within the plugin, and the experimenter only has to define a small set of parameters to utilize it. To use the jspsych-categorize plugin, the experimenter would define the stimulus, the acceptable responses, and the text used to give feedback. Other plugins have different parameters that are specific to their functionality.

A *block* is a collection of trials that are all defined by the same plugin, and thus all have the same overall structure. A block of categorization trials might involve the successive categorization of a few dozen stimuli, for example. Creating an experiment with jsPsych involves defining a series of blocks and then telling jsPsych to execute the experiment that has been defined. As is shown in the tutorial section of this article, this process is relatively simple, and certainly easier than writing the full program that would provide the equivalent functionality from scratch. Once a description of the experiment is created, the core library handles the execution of the experiment, keeping track of where the user is in the experiment and delegating control to the various plugins when appropriate.

Two key goals in the design of jsPsych were *modularity* and *expandability*. Both are achieved by using the core + plugin architecture. jsPsych is highly modular; only the core

Table 1 List of jsPsych plugins

Plugin File	Description of Trial
jspsych-animation.js	Displays a sequence of images at a specified frame rate. Sequence can be looped for a specified number of times. Timing of subject keyboard responses is recorded.
jspsych-call-function.js	Calls a specified function. This allows the experimenter to insert arbitrary function calls throughout the experiment.
jspsych-categorize.js	Displays an image or HTML object. Subject gives a keyboard response. Feedback is provided about the correctness of the subject's response.
jspsych-categorize-animation.js	Displays a sequence of images at a specified frame rate. Subject gives a keyboard response. Feedback is provided about the correctness of the subject's response.
jspsych-free-sort.js	Displays a collection of images on the screen that the subject can interact with by clicking and dragging. All of the moves that the subject performs are recorded.
jspsych-html.js	Displays an external HTML document (often a consent form). Either a keyboard response or a button press can be used to continue to the next trial. Allows the experimenter to check if conditions are met (such as indicating informed consent) before continuing.
jspsych-palmer.js	Displays a programmatically generated stimulus designed to mimic stimuli used by Palmer (1977) and Goldstone and colleagues (Goldstone, Rogosky, Pevtzow, & Blair, 2005). Gives the option to allow the subject to manipulate the stimulus (for example, to test the subject's ability to recreate a stimulus that they previously saw). Can optionally display corrective feedback if the stimulus is editable.
jspsych-same–different.js	Displays two stimuli sequentially. Stimuli can be images or HTML objects. Subject responds using the keyboard, and indicates whether the stimuli were the same or different. Same does not necessarily mean identical; a category judgment could be made, for example.
jspsych-similarity.js	Displays two stimuli sequentially. Stimuli can be images or HTML objects. Subject uses a draggable slider that is shown on screen to give a response. Anchor labels for the slider can be specified.
jspsych-single-stim.js	Displays an image or HTML object. Subject gives a response using the keyboard.
jspsych-survey-likery.js	Displays a set of questions with Likert scale responses. Subject uses a draggable slider to respond to the questions.
jspsych-survey-text.js	Displays a set of questions with free response text fields. Subject types in answers.
jspsych-text.js	Displays HTML formatted text. Variables can be inserted at the moment the text displays (useful for giving overall accuracy feedback, for example). Subject presses a specified key to continue.
jspsych-xab.js	Displays either an image or HTML object stimulus (X). After a short gap, displays two additional stimuli (A and B). Subject selects which of the two stimuli matches X using the keyboard.

The columns display a list of jsPsych plugins and a brief description of the trial structure that each defines. More information, including a working example for each plugin, is available at the project wiki (https://github.com/jodeleeuw/jsPsych/wiki/List-of-Plugins).

library and plugins that are actually used by an experiment are required to be included in the experiment Web page. Plugins are designed so that they can interact with any other subset of plugins. Creating a new plugin allows jsPsych to be expanded by experienced programmers. Plugins have a very flexible structure, making it possible to create a plugin to define most kinds of trials that an experiment requires. The hope is that as more experiments are created using jsPsych, the library of available plugins becomes large enough that most experiments can be developed by assembling preexisting plugins and writing only a few dozen lines of code (this is already possible with the currently available plugins for many common types of experiments that psychologists conduct). For a detailed description of how to create a plugin, see this wiki page at the jsPsych project website: https://github.com/jodeleeuw/jsPsych/wiki/Create-a-Plugin. The use of preexisting plugins is covered in the tutorial part of the article.

Features of the core library

Although most of the code used to run an experiment will exist in the plugins, the core library is the glue that holds everything together. Besides handling the execution of the experiment, the core library has a few other functions that range from critical (e.g., data storage) to optional.

Data storage Whenever a plugin stores data, it should call the writeData() method defined in the core library. This method does two important things. First, it stores the data in memory so that it can be retrieved later by various jsPsych functions. Second, it invokes a function call to indicate that new data has been recorded (see the next section on event related function calls). At any point, the method jsPsych.data() can be called to recover all of the data stored so far. The data is organized as a multilevel array. The top level corresponds to each block of the experiment. The second level corresponds to individual trials. Each second level element is a JavaScript object defined by the plugin that stored the data. Plugins can store arbitrarily defined objects, allowing any kind of data to be stored. The object returned by calling jsPsych.data() can be represented as a JSON (JavaScript object notation) string, enabling relatively simple transfer between jsPsych and other programming languages, such as R, for further data analysis.

Data from jsPsych can also be retrieved as comma-separated value (CSV) strings, suitable for importing into spreadsheet or statistical analysis packages. The method jsPsych.dataAsCSV() will return a string containing all of the data in CSV format. Another method, which has limited support (see the Current Limitations section), is to write a CSV file to the hard drive of the computer running jsPsych, by calling the jsPsych.saveCSVdata() method (the tutorial section of this article will give an example of using both of these methods). However, this is not the recommended way to save data. Rather, it is meant to be used as a quick way to check the data during the development process. Ultimately, data should be saved in a database or as a file on the server hosting the experiment. This ensures that the experimenter can access the data, and that they are securely stored. The online documentation gives some pointers on how to accomplish this.

Event-related function calls Certain events in the course of an experiment are designed to trigger calls to functions. These functions can be specified by the experimenter, meaning that arbitrary code can execute when these events happen. The four events that trigger a function call are (1) the moment before a trial begins, (2) the moment after a trial ends, (3) the end of the experiment, and (4) the moment that data are recorded. No built-in functionality is associated with these events, and the events can be used (or not used) however the experimenter sees fit. As an example, the event that occurs when data is recorded could be used to permanently store the new data in a database, enabling data to be progressively saved while the experiment is being completed.

Progress tracking The core library keeps track of how many trials are completed and how many are left in the experiment. Calling jsPsych.progress() returns a JavaScript object with information about the total number of blocks in the experiment, the total number of trials in the experiment, the current trial number (relative both to the whole experiment and to the current block), and what the current block number is. This can be used in a variety of ways. One common use case is to combine the progress information with the event related function call that occurs when a trial ends. A function can be specified that checks the current progress and updates a progress bar on the screen, so that the subject is aware of their approximate position in the experiment.

Preloading images For many experiments, it is important to preload image files. This ensures that images will display in the browser as soon as jsPsych attempts to display them, rather than having to wait for them to be downloaded from the server hosting the experiment. To facilitate preloading of images, jsPsych has a built in preloading function: jsPsych.preloadImages(). This function has optional callback function parameters that can be triggered as images load and when all images finish loading. These can be used to display a loading progress screen, and to wait until all images are loaded before beginning the experiment.

Interfacing with amazon mechanical turk Since Mechanical Turk is a common way to recruit subjects for online experiments, jsPsych provides some basic functionality to interface with Mechanical Turk. The jsPsych.turkInfo() method returns a JavaScript object containing the Mechanical Turk worker ID, HIT ID, and assignment ID. It also determines whether the page was loaded from Mechanical Turk, which can be useful to prevent unauthorized users from completing the experiment.

Keeping track of experiment start time Finally, the core library records the time that the experiment began. This can be useful when determining how long a subject took to get through an experiment. This is particularly useful in an online context, where subjects could get up and leave their computer in the middle of the experiment and return later to complete it.

Current limitations

jsPsych was initially created as a way of rapidly building relatively simple experiments. Although the complexity of possible experiments is now quite high, there are still some limitations. Because the library is built around the idea of creating an experiment description before executing the experiment, it does very well with static experiments, but has

few features for dynamically altering the course of the experiment as the subject is engaged. The software has no core library features to handle conditional or looping changes in the experiment structure. One solution, though perhaps not an ideal one, is to create a plugin that implements conditional changes for the specific task. A plugin in the /dev folder, called jspsych-adaptive-category-train, implements a specific category-training protocol in which subjects must reach a certain threshold of performance on a category learning task. It does this by treating the entire training protocol as a single trial, and implementing the conditional logic within the trial structure. Although this works, it is not a particularly elegant or general solution. At the moment, jsPsych is best used in cases in which the structure of the experiment is completely known before the subject begins. There are plans to implement a general set of conditional structures in the core library for the future.

Browser compatibility is always a major concern for any Web development project. The core library and most plugins will work in all of the latest versions of the major browsers (Chrome, Firefox, Safari, and Internet Explorer). jsPsych has been tested most thoroughly in Chrome; other browsers may have compatibility issues that haven't been discovered yet. One feature that is only available in Chrome and Firefox is the ability to write the data to a local text file. If you are planning to use this feature in a laboratory environment, Chrome is the best option.

One specific kind of browser compatibility issue worth noting is the use of mobile devices and tablets with Web browsers. The core jsPsych library works with these devices, but specific plugins may need to be used for certain kinds of devices. For example, devices that use touchscreens in lieu of a mouse require different plugins when touch-based interaction is required, and some devices may lack a dedicated keyboard, making certain kinds of user interaction different, such as responding to a stimulus by pressing a key. All of the supported plugins at the time of writing this article were designed for traditional mouse and keyboard interaction, but jsPsych has been used with the iPad in laboratory environments to collect touch-response data. The plugin used for this experiment is jspsych-storybook, located in the /dev folder. It may be a useful example for developing this kind of tool.

If one is concerned about browser compatibility issues for a particular experiment, then the best course of action is to use JavaScript's feature detection capabilities to determine whether the browser is capable of running the experiment. This can be done before the experiment starts, with an error message being displayed if the browser is not compatible. A summary of feature detection is available at http://learn.jquery.com/code-organization/feature-browser-detection/. Currently, automatic feature detection is not integrated into jsPsych, but it is something that will likely be added in the future.

Finally, measuring response times (RTs) with JavaScript has inherent limitations that researchers should be aware of. When responses like pressing a key are generated by the subject, they are treated as events by JavaScript and added to an event-processing queue. The time it takes to processes the event will depend on a number of factors, including which browser the subject is using, how fast their computer is, and the number of other tasks that the computer is executing. Researchers should treat JavaScript-based RT measures with some caution, though certain factors will mitigate these problems: within-subjects RT differences will be more reliable than between-subjects RT differences since within-subjects designs eliminate variability caused by computer hardware and software, and long RTs will be less sensitive to these problems than short RTs because the relative magnitude of the noise caused by the event-driven processing will be less.

Creating an experiment with jsPsych: Eriksen flanker task

The rest of this article is a tutorial, covering how to set up a simple experiment using the jsPsych library. I have chosen to use a variation of the Eriksen flanker task (Eriksen & Eriksen, 1974), closely inspired by an article by Kopp and colleagues (Kopp, Mattler, & Rist, 1994). In the Eriksen flanker task, the subject responds to a visual stimulus (e.g., a left-pointing arrow) by pressing a specified key (e.g., the left arrow key). In some trials, incongruent stimuli (e.g., a right-pointing arrow) are presented in the surrounding visual field, flanking the target, whereas in other trials, congruent (or neutral) items are presented. It has been consistently found that incongruent flanking items slow down RTs relative to congruent flankers (Fox, 1995). This experiment is simple enough to serve as an introductory tutorial, and hopefully it will clearly illustrate how to use the jsPsych library to rapidly build experiments. You can try the complete experiment yourself by going to www.jspsych.org/examples/tutorial/. You can also download the completed tutorial, including all of the source code and image files, at www.jspsych.org/examples/tutorial/download/.

Some background information that might be necessary to fully understand how the code works is outside the scope of this article. In particular, I will not be covering HTML, CSS, or JavaScript fundamentals. A wealth of freely available online resources can be used to learn the basics of these tools. I will only be covering how to integrate jsPsych with these tools.

Step 1 Installing jsPsych

For most applications, the Web page containing the experiment will be hosted on a Web server. However, it is just as easy to develop the experiment on a personal computer. Fortunately, in both cases the installation process for jsPsych is

the same. In fact, installation is a bit of a misnomer here, since all that needs to happen are the following steps:

1. Go to https://github.com/jodeleeuw/jsPsych/releases or www.jspsych.org and download the latest version of jsPsych. At the time of writing, the latest was version 2.0. You may want to start with that version (available at the first link) to guarantee that the instructions below will work.
2. Put the downloaded files in an easy-to-reference location. For example, if you are developing the experiment in a folder on your personal computer, you might want to create a subfolder called "scripts" that will hold the JavaScript files your experiment needs to work. When jsPsych is loaded on the experiment Web page, the location of the files will be referenced.

If you want to follow along and recreate the experiment as I am describing it, you should create a folder on your personal computer to hold all of the files that you will need (I will call this folder the *project* folder). Within the project folder, create a subfolder called *scripts*. Place the jsPsych library in the scripts folder; you should see jspsych.js in the scripts folder, and a plugins folder that contains all of the plugins also in the scripts folder.

Step 2 Creating stimuli

For most experiments, you will need to create the stimuli outside of jsPsych. For this experiment, I have created four images (Fig. 1). In this version of the Eriksen flanker task, subjects must press the arrow key that corresponds to the direction that the middle triangle is pointing. Two stimuli have congruent flankers, and two have incongruent flankers.

A note for advanced users: One exciting thing about HTML5 is the possibility of programmatically generating stimuli using <svg> or <canvas> elements. Many jsPsych plugins support the display of HTML content, and <svg> or <canvas> elements could be used in these cases. Libraries including D3 (www.d3js.org) or Raphael (www.raphaeljs.com/) can be used to generate interactive stimuli. The *jspsych-palmer* library uses Raphael to create an interactive stimulus. This is outside the scope of the tutorial example, but I mention it here to point out the possibility.

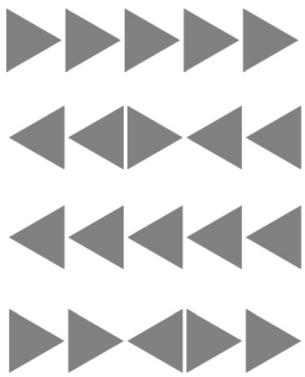

Fig. 1 The four images that serve as stimuli for the demo experiment. The file names, in order from top to bottom, are *congruent-right.gif*, *incongruent-right.gif*, *congruent-left.gif*, and *incongruent-left.gif*

You can download the images in Fig. 1 at www.jspsych.org/examples/tutorial/download.html. Create a folder called *img* in the *project* folder, and place the four image files inside the *img* folder.

Step 3 Setting up an HMTL page for jsPsych

jsPsych works by dynamically manipulating the content of a Web page. To start using jsPsych, only a minimal framework of a Web page is needed. The minimal page to run jsPsych will look something like this:

```
<!doctype html>
<html>
<head>
  <title>My experiment</title>
  <script src="scripts/jquery-1.11.0.min.js">
  </script>
  <script src="scripts/jspsych.js">
  </script>
</head>
<body>
</body>
<script>
  // code to initialize jsPsych will go here.
</script>
</html>
```

This is a very bare-bones HTML document. The only lines of substance are the two <script> tags contained within the <head> section of the document. The first links to the jQuery library (http://jquery.com/), which is a dependency of the jsPsych library. Without this link, jsPsych will not work. You can download the jQuery library at http://code.jquery.com/jquery-1.11.0.min.js. Place the downloaded file in the scripts folder. The second links to the *jspsych.js* file, which is the core library file for jsPsych. If you are following along, copy the HTML code into a blank document using a basic text editor (such as Notepad on Windows or TextMate on OSX) and save as *index.html* in the project folder. You can then make sure the file is correctly formatted as an HTML document by opening it in a Web browser. If you see a blank window with the title "My experiment" then everything is working.

Although this is a good template to start from, it will not actually do anything at this point. Two more aspects are needed: loading the relevant jsPsych plugins, and a few lines of JavaScript code that will describe the experiment to be run.

Step 4 Loading jsPsych plugins

As was described earlier in this article, jsPsych is modular; different plugins are needed to perform different kinds of trials. The first step in setting up an experiment is determining which plugins are needed to run the various kinds of tasks that

the experiment requires. For this demo experiment, two different kinds of tasks are created:

1. Subjects will read instructions.
2. Subjects will see an image on the screen and press a key in response.

These are two very simple tasks that many different experiments would use, and existing jsPsych plugins will accomplish both. The *jspsych-text* plugin displays a block of text to the user (it can be used to display any HTML content), which can be used to show instructions. The *jspsych-single-stim* plugin displays a stimulus (either a single image file or an HTML element) on the screen and records a keyboard response from the subject. This plugin can be used to show the flanker task stimuli and to measure the RT.

To load a plugin, insert the appropriate <script> tag *after* the <script> tag that loads the core jspsych.js file. Plugins can be loaded in any order, but they must come after the jspsych.js file is loaded. Here is the <head> section of the index.html file after the appropriate <script> tags have been inserted:

```
<head>
  <title>My experiment</title>
  <script src="scripts/jquery-1.11.0.min.js">
  </script>
  <script src="scripts/jspsych.js">
  </script>
  <script src="scripts/jspsych-text.js">
  </script>
  <script src="scripts/jspsych-single-stim.js">
  </script>
</head>
```

Step 5 Creating the experiment description

In order for jsPsych to run the experiment, we have to describe the structure of the experiment to jsPsych. This can be done by specifying a set of *blocks*, in which each block is a set of trials using a single plugin. For example, if the subject is to see 20 flanker stimuli in a row, we would create a block of 20 trials using the jspsych-single-stim plugin.

To specify a block, we create a JavaScript object (JavaScript objects can be created using { } brackets) that tells jsPsych which plugin to use and what parameters to use for that plugin. For example:

```
var experiment_block = {
  type: "text",
  parameter1: value,
  parameter2: value,
  // and so on ...
}
```

The first part of the object is a parameter called *type*. This parameter must always be set. It tells jsPsych which plugin to use for the block. In this case, the block will use the jspsych-text plugin. After the declaration of the *type* parameter, other parameters can be specified. Each plugin has a different set of parameters. This is a big part of what makes jsPsych a flexible library; as the need arises, different plugins can be created with different parameters. To figure out what parameters a plugin uses, you can look it up at the jsPsych wiki (https://github.com/jodeleeuw/jsPsych/wiki/List-of-Plugins), or you can view the source code for a plugin (https://github.com/jodeleeuw/jsPsych/tree/master/plugins). The source code for all of the nondevelopmental plugins contains a header section that describes the parameter options in detail.

The first block that we need in our demo experiment is the instruction block. We are using the jspsych-text plugin for this. The only parameter (other than the type parameter) that *must* be set for this plugin is called *text*, and unsurprisingly, this parameter specifies what text is displayed to the subject. (A few optional parameters could also be set, such as what key the subject can press to advance to the next part of the experiment, but jsPsych has default values that can be used, and you only need to set a value if you want to override the default value.)

One potentially confusing aspect of defining a value for this parameter is that the value for the text parameter is going to be an array. This is because we can specify multiple *trials* in a single block, and jsPsych will automatically take each element of the array and construct a new trial. To illustrate this, I will break the instructions into two parts. The following is JavaScript code added inside the <script> tag near the bottom of the document described in Step 3.

```
// Experiment Instructions
var welcome_message = "<div id='instructions'><p>Welcome to the " +
  "experiment. Press enter to begin.</p></div>";

var instructions = "<div id='instructions '><p>You will see a " +
  "series of images that look similar to this:</p><p>" +
  "<img src='img/incongruent_right.gif'></p><p>Press the arrow " +
  "key that corresponds to the direction that the middle arrow " +
  "is pointing. For example, in this case you would press the " +
  "right arrow key.</p><p>Press enter to start.</p>";

var instruction_block = {
  type: "text",
  text: [welcome_message, instructions]
};
```

The code above defines two variables that contain strings of HTML code. Because the strings are too long to fit on a single line, they are broken up into multiple lines, and then concatenated together with the + operator. These two strings are the two parts of the instructions that will show for the subject. The third variable is a block that we will use as part of the experiment description for jsPsych. It specifies the type of plugin and the key parameter for that plugin. The text plugin will automatically show each different string defined in the *text* parameter in sequence, advancing to the next string when the enter key is pressed by the subject.

Now we can look at how to send this information to jsPsych.

In the most basic case, you will only need to call a single method for jsPsych to run the experiment. This is the init() method. The init method has a single parameter, which is an object that specifies different options. The only option that you *must* specify is the "experiment_structure" option. This is

how you tell jsPsych the structure of your experiment. The value for the experiment_structure option is an array of blocks. The blocks will be executed in the order that they appear in the array. For now, we have only a single block using the jspsych-text plugin. This is what the jsPsych.init() call looks like for the experiment so far:

```
jsPsych.init({
  experiment_structure: [instruction_block]
});
```

Although the version of the init method above will work, it is often very useful to specify which HTML element will display the jsPsych content. This can be done by using the *display_element* option in the init method. If this option is omitted, then jsPsych will automatically display the content in the <body> element (and will create this element if it does not exist). Specifying what element jsPsych appears in provides control over how the content is displayed. CSS rules can be specified to alter the appearance of the jsPsych content (see Step 10). There is also the possibility of mixing jsPsych content with other content, or displaying elements on the screen like a progress bar that persists from trial to trial. Here we will add a element and display the jsPsych content in the . First, add the following HTML in the <body> section of the HTML document:

```
<body>
  <div id="jspsych_target"></div>
</body>
```

Then change the init method to specify the display_element option:

```
jsPsych.init({
  display_element: $("#jspsych_target"),
  experiment_structure: [instruction_block]
});
```

The experiment will run with the code that we have now created. Place the jsPsych.init method call after the JavaScript code that defines the instruction_block variable. Only the instruction block will be displayed so far (which will show the two different messages we defined in that block), but this will confirm that jsPsych is working. The complete code up to this point can be found at www.jspsych.org/examples/tutorial/part1.html. You can use the View Source option in your browser to see the code for the live version.

Step 6 Finishing the experiment description

At this point, all of the basic functionality of jsPsych has been covered. The following material fills out the rest of the experiment description using the same mechanics described above: define an experiment block using JavaScript, and add it to the *experiment_structure* parameter in the jsPsych.init method.

To display the stimuli for the flanker task, we use the jspsych-single-stim plugin. This plugin shows a stimulus on the screen and records the key that the subject presses in response. It also records the RT. To use it, we need to specify what stimuli to show. We can specify the stimuli as relative paths to image files. As we did with the jspsych-text plugin, we can specify multiple trials within a single block statement. We do that with the single-stim plugin by giving an array of stimuli to show. The plugin will automatically create a trial for each element in the array.

We only have four different stimuli for this experiment, but we want to show them multiple times. We do that by simply repeating elements in the array. The single-stim plugin will not do randomization or multiple presentations for us; we must build the order outside of jsPsych, and then tell jsPsych what to do. There are many reasonable ways to do this. Here is one example:

First, define two variables above the experiment instructions we created earlier. One variable is the number of trials shown to the subject, and the second is an array containing the paths of the four images:

```
var n_trials = 20;
var stimuli = [
  "img/congruent_left.gif",
  "img/congruent_right.gif",
  "img/incongruent_left.gif",
  "img/incongruent_right.gif"
];
```

Below that, create an array to hold a random order of these stimuli.

```
var stimuli_random_order = [];
```

Finally, just below the above code, add a random selection from the *stimuli* array to the *stimuli_random_order* array *n_trials* times, using a for loop.

```
for (var i = 0; i < n_trials; i++) {
    var random_choice = Math.floor(Math.random() * stimuli.length);
    stimuli_random_order.push(stimuli[random_choice]);
}
```

Once that code executes, *stimuli_random_order* will have 20 randomly selected stimuli to present. From an experimental design perspective, this process is crude. Better versions might ensure that all four stimuli are shown the same number of times. This illustrates an important design feature of jsPsych: It does not lock the experimenter into a particular method for randomization or ordering; the experimenter can choose whatever method is appropriate for the experimental design. However, this flexibility comes at the cost of increasing the amount of code that the experimenter must write.

Now we have everything we need to define the single-stim block. The only parameter that we must specify that is not already defined is called *choices*. *Choices* is an array of integers. The integers correspond to JavaScript key codes, which represent keys on the keyboard (a useful tool for finding key codes is available at www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes). The single-stim plugin will only continue if the user selects a key that is in the *choices* array. This means that the subject can be forced to make a binary (or any *n*-alternative) choice. For this

experiment, we want the binary choice to be the left arrow key (key code 37) or the right arrow key (key code 39). Add this line after the line where we defined the instruction_block:

```
var test_block = {
  type: "single-stim",
  stimuli: stimuli_random_order,
  choices: [37, 39]
}

jsPsych.init({
  display_element: $("#jspsych_target"),
  experiment_structure: [
    instruction_block,
    test_block
  ]
});
```

Next, amend the jsPsych.init() function to include the new block:

If you are following along, go ahead and run the experiment. After the instructions, you should see 20 trials with different flanker stimuli.

The final block to add is a simple debriefing. Add the variable containing the debriefing string and the variable to define the block, and then modify the init method to include the new debriefing block:

```
var debrief = "<div id='instructions'><p>Thank you for " +
  "participating! Press enter to see the data.</p></div>";

var debrief_block = {
  type: "text",
  text: [debrief]
};

jsPsych.init({
  display_element: $('#jspsych_target'),
  experiment_structure: [
    instruction_block,
    test_block,
    debrief_block]
});
```

Step 7 Looking at the data using an event-related function call

The debrief instructions indicated that pressing Enter would reveal the data, but so far the only thing that happens is the experiment ends and a blank screen is shown. We can use the event that is triggered by the end of the experiment to display the raw data to the user. Obviously this is not something that would happen in a normal experimental context. It is included here to illustrate event-related function calls and to show what the raw data recorded by jsPsych look like.

To add an event-related function call, we need to modify the jsPsych.init() method to tell jsPsych what function to use for the event. The event that we are interested in is *on_finish*, which is triggered at the very end of the experiment. By default, jsPsych will pass the complete data object as the first parameter to whatever function is defined. Here we are going to define the function within the jsPsych.init() method call:

```
jsPsych.init({
  display_element: $('#jspsych_target'),
  experiment_structure: [
    instruction_block,
    test_block,
    debrief_block],
  on_finish: function(data) {
    $('#jspsych_target').append($('<pre>', {
      html: JSON.stringify(data, undefined, 2)
    }));
  }
});
```

The code inside the function is adding a <pre> element to the HTML document and inserting the string representation of the data inside the new <pre> element. It is using jQuery's selectors and append method to do this. You can learn more about this at the jQuery documentation (http://api.jquery.com/). You can also read about the JSON.stringify method here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify.

Running the experiment at this point should display the data at the very end. It should look something like this:


```
[
  [
    {
      "trial_type": "text",
      "trial_index": 0,
      "rt": 1751
    },
    {
      "trial_type": "text",
      "trial_index": 1,
      "rt": 12159
    }
  ],
  [
    {
      "trial_type": "single-stim",
      "trial_index": 0,
      "rt": 11336,
      "stimulus": "img/congruent_left.gif",
      "key_press": 37
    },

    … trials 1-18  …

    {
      "trial_type": "single-stim",
      "trial_index": 19,
      "rt": 624,
      "stimulus": "img/congruent_left.gif",
      "key_press": 37
    }
  ],
  [
    {
      "trial_type": "text",
      "trial_index": 0,
      "rt": 1724
    }
  ]
]
```

Notice that the data are organized by blocks, and then by trial within each block. Numerous variables are recorded for each trial, including what key the user pressed, the RT, what stimulus was shown, and what trial number it was within the block.

Although this hierarchically arranged data object may be useful for certain kinds of programming tasks, other formats may be more useful for data analysis. jsPsych currently offers one alternative format, a CSV string, suitable for importing into various spreadsheet and statistical analysis packages. To see the CSV representation of the data, modify the on_finish function definition to use the jsPsych.dataAsCSV() method:

```
jsPsych.init({
  display_element: $('#jspsych_target'),
  experiment_structure: [
    instruction_block,
    test_block,
    debrief_block],
  on_finish: function(data) {
    $('#jspsych_target').append($('<pre>', {
      html: jsPsych.dataAsCSV()
    });
  }
});
```

Now when the experiment finishes, the data should display in CSV format, with the first line representing the names of the columns, and each subsequent line representing a single trial.

Finally, it may be useful to have a downloaded version of the data. As was discussed in the Overview section, the following technique is not meant to be a solution for permanent data storage. This technique will enable the subject's computer to download data, and for online experiments this clearly will not work (but it may be reasonable to use this technique in a laboratory setting). Finally, the following technique only works with the latest versions of Chrome and Firefox at the time of writing, although it is likely that other major browsers will eventually support this as well. To download a CSV file containing the data, simple call the jsPsych.saveCSVdata() method, with a file name as the parameter:

```
jsPsych.init({
  display_element: $('#jspsych_target'),
  experiment_structure: [
    instruction_block,
    test_block,
    debrief_block],
  on_finish: function(data) {
    $('#jspsych_target').append($('<pre>', {
      html: jsPsych.dataAsCSV()
    });
    jsPsych.saveCSVData("demo_experiment_data.csv");
  }
});
```

After modifying the code and running the experiment, you should see the data on screen and also be prompted to download a CSV file containing the data, if you are using an up-to-date version of Chrome or Firefox.

Step 8 Using the optional data object

Each jsPsych plugin defines in advance what data it will store. The single-stim plugin, for example, records information about what stimulus was shown, what key the subject pressed, and the RT. However, in some cases it may be desirable to add additional information about the trial that is not defined by the plugin. In our experiment, we have two kinds of stimuli: incongruent and congruent. We may want to record which type of stimulus was shown on a trial, so that it is easy to divide the data for analysis. (Note that we could do this in principle by looking at which stimulus was shown on a trial, but this might not be possible in every context, and it might save time to record this information in the data and not recreate it during analysis.)

All plugins support a feature called the optional data object. There is a parameter called *data* for every plugin, which accepts an array of objects. The array needs to be the same length as the number of trials defined for that block. Each element of the array can contain arbitrary data that are appended to the data for the trial. Here is an example.

We are going to add a value to each trial in the testing block that indicates whether the stimulus is incongruent or congruent. To do this, we will modify our original code that generated the random order of stimuli. We defined an array that had the four different stimuli. Now define a corresponding array that indicates the type of each stimulus:

```
var n_trials = 20;
var stimuli = [
  "img/congruent_left.gif",
  "img/congruent_right.gif",
  "img/incongruent_left.gif",
  "img/incongruent_right.gif"
];
var stimuli_types = [
  "congruent",
  "congruent",
  "incongruent",
  "incongruent"
];
```

Note that the elements of the two arrays correspond to each other: The first stimulus in the stimulus array is congruent; the third is incongruent; and so forth.

We can modify the code that generated the random order to also generate an array containing a data object for each trial. The array of data objects will be the same length as the array of stimuli, and the elements of the arrays will correspond to each other:

```
var stimuli_random_order = [];
var opt_data = [];

for (var i = 0; i < n_trials; i++) {
  var random_choice = Math.floor(Math.random() * stimuli.length);
  stimuli_random_order.push(stimuli[random_choice]);

  opt_data.push({
    stimulus_type: stimuli_types[random_choice]
  });
}
```

We added a new item called *stimulus_type*, and the value of the item is either "congruent" or "incongruent," depending on which stimulus is shown for that trial. The final step is to modify the definition of the test block to include the data parameter:

```
var test_block = {
  type: "single-stim",
  stimuli: stimuli_random_order,
  choices: [37, 39],
  data: opt_data
}
```

With this modification, each trial will be tagged with an additional item that indicates what kind of stimulus was presented. This information will be visible when the data are displayed at the end of the experiment.

Step 9 Modifying default parameters

Plugins often have several parameters that have reasonable default values. The default value can be overridden in cases in which the experimenter wants to specify a different value. One such parameter for the jspsych-text plugin is called *timing_post_trial*. This defines how long a blank screen is shown after the text is removed from the screen. For this experiment, we might want a slightly longer delay between the time that the instructions disappear and the presentation of the first stimulus, to give the subject time to position his or her fingers over the response keys. We can modify the *timing_post_trial* parameter for the first block of the text plugin like this:

```
var instruction_block = {
  type: "text",
  text: [welcome_message, instructions],
  timing_post_trial: 2500
};
```

This change will cause a 2,500-ms delay between the time that the instructions disappear and the start of the testing block.

Step 10 Changing the visual appearance of the experiment

Most jsPsych plugins do not modify the visual style of elements on the Web page. Instead, they simply add elements (like text or images) and leave the visual style of the elements up to the experimenter. CSS can be used to modify the visual appearance of jsPsych elements. Covering CSS in any depth is outside the scope of this tutorial. Instead, I have created a simple style sheet (a file with a .css extension) that can be applied to the example. It changes the font, font size, alignment of elements, and spacing of elements:

```
/* experiment.css */

body {
  background: #fff;
  margin: 0;
  padding: 0;
  font-size: 18px;
  font-family: 'Palatino Linotype', FreeSerif, serif;
}

#jspsych_target {
  width: 1000px;
  margin-left: auto;
  margin-right: auto;
  margin-top: 50px;
  text-align: center;
}

#jspsych_target pre {
  text-align: left;
}

#instructions {
  width: 500px;
  margin-left: auto;
  margin-right: auto;
  text-align: left;
}
```

To include this in the experiment, copy the CSS code into a file named *experiment.css* (contained in the same directory as the index.html file for the experiment) and insert this <link> tag in the <head> section of the HTML document:

```
<link href="experiment.css" rel="stylesheet"></link>
```

This is the end of the tutorial. The code should now match exactly what is available at www.jspsych.org/examples/tutorial/. You can use the View Source option in your browser to compare your code with the online version.

Learning more

This article has given a broad overview of jsPsych, discussing the motivation and design principles behind the library. The tutorial covered the basic steps involved in creating an experiment with jsPsych. Several advanced topics were not covered, but these are covered in detail within the jsPsych project wiki (https://github.com/jodeleeuw/jsPsych/wiki). In particular, the wiki provides information on how to create new plugins for jsPsych, in the event that an experiment cannot be assembled using the plugins that already exist. The wiki also has more documentation for the many plugins that were not discussed in this tutorial and for features of the library that were not covered in depth. Finally, the wiki has information about topics that are not specifically related to the jsPsych library, but are important for online experiments, such as linking jsPsych to permanent data

storage in a database and authenticating Mechanical Turk users for jsPsych experiments.

Author note This material is based on work that was supported by a National Science Foundation Graduate Research Fellowship under Grant No. DGE-1342962. The author thanks Rob Goldstone, Nicholas de Leeuw, Rick Hullinger, and Peter Todd for feedback and suggestions on an earlier draft of this article, as well as the numerous people who have used jsPsych throughout the development process and have provided valuable feedback.

References

Buhrmester, M., Kwang, T., & Gosling, S. D. (2011). Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *Perspectives on Psychological Science, 6,* 3–5. doi:10.1177/1745691610393980

Crump, M. J. C., McDonnell, J. V., & Gureckis, T. M. (2013). Evaluating Amazon's Mechanical Turk as a tool for experimental behavioral research. *PloS ONE, 8,* e51382. doi:10.1371/journal.pone.0057410

Eriksen, B., & Eriksen, C. (1974). Effects of noise letters upon the identification of a target letter in a nonsearch task. *Perception & Psychophysics, 16,* 143–149. doi:10.3758/BF03203267

Fox, E. (1995). Negative priming from ignored distractors in visual selection: A review. *Psychonomic Bulletin & Review, 2,* 129–139. doi:10.3758/BF03210958

Goldstone, R., Rogosky, B., Pevtzow, R., & Blair, M. (2005). Perceptual and semantic reorganization during category learning. In H. Cohen & C. Lefebvre (Eds.), *Handbook of categorization in cognitive science* (pp. 651–678). Amsterdam: Elsevier.

Goodman, J. K., Cryder, C. E., & Cheema, A. (2013). Data collection in a flat world: The strengths and weaknesses of Mechanical Turk samples. *Journal of Behavioral Decision Making, 26,* 213–224. doi:10.1002/bdm.1753

Kopp, B., Mattler, U., & Rist, F. (1994). Selective attention and response competition in schizophrenic patients. *Psychiatry Research, 53,* 129–139.

Mason, W., & Suri, S. (2012). Conducting behavioral research on Amazon's Mechanical Turk. *Behavior Research Methods, 44,* 1–23. doi:10.3758/s13428-011-0124-6

McDonnell, J. V., Martin, J. B., Markant, D. B., Coenen, A., Rich, A. S., & Gureckis, T. M. (2012). psiTurk (Version 1.02) [Software]. New York, NY: New York University. Available from https://github.com/NYUCCL/psiTurk

Palmer, S. E. (1977). Hierarchical structure in perceptual representation. *Cognitive Psychology, 9,* 441–474. doi:10.1016/0010-0285(77)90016-0

Paolacci, G., Chandler, J., & Ipeirotis, P. G. (2010). Running experiments on Amazon Mechanical Turk. *Judment and Decision Making, 5,* 411–419.

Reips, U.-D., & Neuhaus, C. (2002). WEXTOR: A Web-based tool for generating and visualizing experimental designs and procedures. *Behavior Research Methods, Instruments, & Computers, 34,* 234–240. doi:10.3758/BF03195449

Ross, J., Irani, L., Silberman, M. S., Zaldivar, A., & Tomlinson, B. (2010). Who are the turkers? Worker demographics in Amazon Mechanical Turk. In *CHI '10: CHI Conference on Human Factors in Computing Systems* (pp. 2863–2872). New York: ACM.

Zwaan, R. A., & Pecher, D. (2012). Revisiting mental simulation in language comprehension: six replication attempts. *PloS ONE, 7,* e51382. doi:10.1371/journal.pone.0051382