Focuses: Implementation and usage of GeoGebra, JSXGraph, or other Programs; New users and authoring of questions.

Article number: 15

STACKing Further with STACK-JS

Sam Fearn[*]

Durham University

Abstract

This paper explores the use of STACK-JS, as a tool for enhancing STACK questions through the addition of custom JavaScript and the inclusion of existing JavaScript libraries. As an example of how STACK-JS may be used, we focus here on an example which creates a custom method of user interaction, designed to allow for a novel means of testing mathematical proof. In particular, this example was created to be used in a proof-heavy first-year undergraduate Analysis module at Durham University, emulating a type of problem often used in Computer Science courses known as Parsons Problems.

[*] Corresponding author: s.m.fearn@durham.ac.uk

1. Introduction

Assessment and feedback is a fundamental part of a degree in mathematics. Formative assessment may be defined as “encompassing all those activities undertaken by teachers, and/or by their students, which provide information to be used as feedback to modify the teaching and learning activities in which they are engaged” (Black and Wiliam, 1998). This style of assessment is also often referred to as ‘assessment for learning’, to emphasise the idea that the assessment is in support of learning, as opposed to ‘assessment of learning’ which is designed to certify that a learner has met a set of learning objectives. Such assessments give course leaders both a snapshot into the progress of the class as a whole, as well as a chance to identify particular students who may be struggling and in need of further support. They also allow students to gauge their own level of understanding, and the feedback obtained from an assessment should enable a student to further their learning. Indeed, studies have shown that “innovations that include strengthening the practice of formative assessment produce significant and often substantial learning gains,” but moreover that “improved formative assessment helps low achievers more than other students and so reduces the range of achievement while raising achievement overall” (Black and Wiliam, 2005).

However, in order for feedback to be effective, students must be given the opportunity to make use of it (Shute, 2007). Even when staff are able to return marked work quickly, by the time students receive their feedback, they have often moved on to a new topic and the feedback they receive may seem less pertinent. Automated assessment offers the opportunity for feedback to be generated at the moment of submission, giving students the chance to immediately close the feedback cycle, putting the received feedback into practice by re-attempting a given question (Sangwin, 2013). Moreover, automated assessments allow for questions to be generated using randomised inputs, so that when a student does re-attempt a question, they cannot simply input a correct solution they have been given in previous feedback. Not only does automated assessment therefore represent an opportunity for enhancing learning, it also clearly offers workload benefits to staff and their departments, saving the need for manually marking students' submissions.

Automated assessments have traditionally been best suited to types of questions that ask a student to do some routine calculation, as is often required in first courses in calculus or linear algebra (Sangwin, 2013). However, higher level mathematics also requires students to develop their proof comprehension, and this has traditionally been an area where it is difficult to take advantage of automated assessment. Although ideas for the automated assessment of mathematical proof have been discussed in the literature previously (Bickerton & Sangwin, 2022), this remains a crucial area for the development of automated assessment tools.

While exploring these ideas in the context of a proof-heavy first-year undergraduate Analysis module, I experimented with testing students’ proof-comprehension skills using a type of problem known in the Computer Science literature as Parsons Problems (Ericson et al., 2022). Specifically, rather than asking a student to write their own proof of a given result, they were instead asked to arrange pre-defined statement blocks in order to construct a valid proof. The Department of Mathematical Sciences at Durham University uses STACK (Sangwin, 2013) for automated assessment, and at that time STACK did not natively provide a means of constructing such a question. A crude approximation was possible within our environment using an alternative tool (another Moodle question type), but was limited by requiring a single fixed answer without distractors. Since many valid mathematical proofs allow for the interchange of particular logical blocks, such as when proving the equality of sets by showing each is a subset of the other, this was not sufficient for the required use case. However, STACK does allow a means of augmenting the inbuilt capabilities through the addition of custom JavaScript, enabling advanced Parsons problems to be created. In this paper, I will discuss

how STACK-JS can be used to extend STACK (Harjula, 2023), enabling advanced and custom forms of user interaction among many other possibilities. The previously mentioned Parsons problems will serve as a contextualising example throughout.

2. Using STACK-JS

STACK-JS is a tool for including custom JavaScript (and existing JavaScript libraries) within a STACK question, whilst providing security by separating the executing code from the host Virtual Learning Environment (VLE). In principle this VLE could be Moodle or ILIAS, though the rest of this paper assumes STACK is being used via Moodle. This separation is achieved through the creation of an iframe in which the JavaScript runs, with limited interaction possible between the JavaScript and the VLE session. Crucially however, STACK input fields can be updated by the JavaScript, meaning that user interactions within the iframe can result in inputs that are then evaluated by STACK.

At a high level, which we then discuss in more detail through the example of creating a Parsons problem below, the key steps for integrating new JavaScript with STACK using STACK-JS are:

1. Use `[[iframe]] and [[script]]` blocks to create the iframe, and import any required JavaScript libraries;
2. Create a (hidden) STACK input for any state data you want to communicate between STACK and the JavaScript;
3. Use the STACK helper function `stack_js.request_access_to_input(ans1,true)` to get access to the STACK input `ans1` in the JavaScript;
4. Optionally, update the STACK input after changes in the iframe.

These steps may look familiar to readers who are already familiar with writing STACK questions that utilise the graphing tool JSXGraph (Sangwin, 2018). JSXGraph is itself a JavaScript graphing library which is officially supported for use in STACK. Using JSXGraph in STACK is therefore simplified when compared to using arbitrary JavaScript, through the existence of a number of helper functions. However, the underlying structure of a STACK question using JavaScript is very similar in either case.

3. Worked example: creating a simple Parsons problem

In order to create a simple Parsons problem in STACK, we need to create a list of statements that the students should sort, as well as provide a means of interacting with the list of statements. We first consider the case where there are no distractors in the list of statements, and therefore the student simply needs to sort a given list into a correct order. A simple example question may then be as shown below in Figure 1, where the statements to be sorted are simply the numbers one to five.

Arrange the steps of working shown to answer the following question.

Question is missing tests or variants.

Each item will be marked as correct if it is preceded and followed by the correct items. It will be marked partially correct if it is *either* preceded *or* followed by the correct item, and it will be marked as incorrect if it is neither preceded nor followed by the correct item.

Put the following statements in ascending order

1
2
5
3
4

Check

Figure 1: A simple Parsons problem using STACK-JS and SortableJS

Initialising the iframe and loading libraries

There are many JavaScript libraries one could use to create a sortable list of items, but for the purposes of this example we will use the Sortable library (Mills, 2024). We therefore create an iframe within the question text of our question, and load the STACK-JS and Sortable libraries:

```
<p>Put the following statements in ascending order</p>
[[iframe]]
[[script type="module"]]
import {stack_js} from '[[cors src="stackjsiframe.js"/]]';
import '[[cors src="Sortable.js"/]]';
```

Here, the `[[iframe]]` block is used to create an iframe within the generated HTML of the question, and the `[[script]]` block is used to define a script element within the header of this iframe. The script element is given the attribute `type="module"`, so that other JavaScript libraries can be imported in, but this element can also be the one in which we include any additional custom JavaScript we need. The `[[cors]]` blocks provide a simple way to reference the paths to our JavaScript libraries, which are stored locally on our Moodle server. The file location for such libraries is then specified relative to the `moodle/question/type/stack/corsscripts` directory. Note that later in the question text we will need to close the script and iframe blocks.

Input, State and STACK-map

All STACK questions should have at least one input, though since students are to interact with our Parsons problem by dragging to reorder the list of statements, we may not want this input to be visible. Even when not visible, this input will be used to store the state of a student's response to our problem. We can create such a hidden input in our question text as follows:

```
style="display:none">[[input:statestringinput]][[validation:statestr
inginput]]
```

The "`display:none`" style command is used to visually hide the input box from the student. There are of course serious accessibility issues with creating a question that only allows interaction through using a mouse (or touch input) to re-order list items, and so this approach may not be the most suitable. Since we use this Parsons problem merely as an example of how to use STACK-JS, we leave further discussion of this point for other work.

As implied by the given name of the input, we will store the student's answer as a string, so the input should be given type string. Since the input field itself is hidden, we should use the input options to not require or display validation, and the extra option `hideanswer` may be specified to ensure that information about the "teacher's answer" is not displayed at any point to students. Note that we have not yet given the student a way to submit their answer through this input; this will be discussed in the next section.

The statements that our students will sort are the key question variables, and so within the question variables field we initialise a list of statements. Since these statements may be long strings, potentially involving mathematics displayed using MathJax, it will be much simpler to refer to these strings wherever possible using unique short keys. A data structure consisting of keys and associated values (our statements to be sorted) can be represented in our JavaScript as a JavaScript Object. Although such a structure is not native to Maxima, STACK offers helper functions for dealing with data in this form as a so-called STACK-map. This is a nested list structure in Maxima, with the first element of the parent list being the string "`stack_map`", and key:value pairs as sublists for our keys and statements. In our example, we might therefore initialise our question variable as follows:

```
correctlist:["stack_map",["f","\\(1\\)"],["y","\\(2\\)"],["j","\\(3\
\)"],["q","\\(4\\)"],["v","\\(5\\)"]]
```

Here we have simply chosen random keys, though of course in practice one could easily write a function to create this structure in Maxima simply from a list of the strings we want to sort. We also note that as well as both our keys and statements being stored as strings in Maxima, the statements make use of inline LaTeX maths. The double backslash is required here, as the backslash character is a so-called special character, and hence requires 'escaping' with the additional backslash to ensure it is parsed properly once passed to the JavaScript. In this simple example, this STACK-map stores the statements in the unique correct order; we discuss alternative correct orders in the context of a more complicated problem later in the section 4 of this paper.

Our statements are now stored in key:value pairs in Maxima as a STACK-map, and will be handled as an object in the JavaScript. We can easily convert the data between these two forms by utilising JSON, more specifically JSON strings. It is the JSON string representation which we will store in the `statestringinput` STACK input. STACK includes a helper function for creating a JSON string from a STACK-map, which we can use as `stackjson_stringify(correctlist)`. We can similarly convert back from a JSON string `stateString` to STACK-maps using `stackjson_parse(stateString)`.

Connect the Maxima and JavaScript

We now have a Maxima STACK-map variable representing the statements we want our student to sort, a hidden input which should be a string representing the student's answer, and an iframe with a script element in which we have loaded a JavaScript library and can also write additional JavaScript. We now need to connect these separate pieces together, displaying the

question statements, providing a means for students to sort the list items in the question, and have this update the hidden string input so we can then check the student's answer.

We first tell our JavaScript about the STACK input field we have created:

```
var stateStorepromiseinput =
stack_js.request_access_to_input("statestringinput", false);
```

This uses a JavaScript Promise to asynchronously return the id of a hidden HTML input which is created inside the iframe, and whose contents are synchronised (on change events, unless the additional boolean option to `stack_js.request_access_to_input` is set to `true`, in which case also on input events) with our hidden STACK input. This means that if our JavaScript sets the contents of the hidden HTML input in the iframe to a value representing the student's answer, our STACK input will be updated to contain the same string, which can then be submitted and checked (as discussed in the next section).

Once our iframe has created its hidden input, the JavaScript promise resolves to the id of this element. We can then load some state into this input and display the current state.

```
stateStorepromiseinput.then((stateid) => {
    let stateStore = document.getElementById(stateid);
    var state;
    // Load existing state, or initialise from a default
```

Here, `stateStore` is the hidden input element in the iframe. When a student first attempts our problem, we simply want to load a default state by randomising the order of our statements. However, if a student returns to view the question, after checking their answer for example, we will want to use the state which has been stored in our STACK input, `statestringinput`. As discussed in the previous section, `statestringinput` will be a JSON string, and so when we load this into our JavaScript, we will want to parse this JSON string into a JavaScript object. The default state will similarly need to parse a JSON string representation of our initial list of statements `correctlist`, before randomising the order of the key:value pairs (which for brevity we omit the details of below).

```
// If we already have a stored state in the statestringinput input,
then we use this state
if ( stateStore.value && stateStore.value != ''){
    state = JSON.parse(stateStore.value);
}
// otherwise our state is loaded from the correct list given as a
Maxima variable
else {
    var stateCorrect = JSON.parse({#
stackjson_stringify(correctlist) #});
    // state = shuffle(stateCorrect)
}
```

Creating a Sortable list

The preceding steps are quite generic, and will cover many possible use cases for STACK-JS. What follows in this section is specific to our example of a Parsons problem, and is included as an example of how one might use the state data within the JavaScript.

With the problem data now available in the JavaScript, we can create the visual display of our statements as an HTML list, and use the Sortable library to both make this a manipulatable element and to update our stored state whenever the student re-orders the statements. We should firstly create an empty HTML list within our iframe, which we can then populate using our JavaScript. We create the empty list after the closing of the script block, but before the closing of the iframe block as follows:

```
<div class="container"><div class="row">
<ul class="list-group col" id="correctListHTML"></ul>
</div></div>
```

Once we've loaded our state, we can then populate this empty list from inside our JavaScript:

```
let correctListHTML = document.getElementById("correctListHTML");
for (const key in state) {
    let li = document.createElement("li");
    li.innerText = state[key];
    li.setAttribute("data-id",key);
    li.className = "list-group-item";
    correctList.appendChild(li);
};
```

Here, we set the key as the value of the `data-id` attribute, as the Sortable library provides a method for returning an array of the `data-id` attributes when the list is sorted by the student. The HTML class is specified to allow for styling using Bootstrap, though we do not discuss this point further.

Finally, we use the Sortable library to add drag-and-drop interactivity to our HTML list:

```
var sortableMainList = Sortable.create(correctListHTML, {
    onSort: (evt) => {
        updateState(sortableMainList);
    },
});
```

This adds an event handler to the list, which calls the following `updateState` function every time the list is sorted:

```
function updateState(sortedCorrect) {
    stateStorepromiseinput.then((stateid) => {
        const newState = {};
        sortedCorrect.toArray().forEach((mykey) => {
            if (state[mykey]) {newState[mykey] = state[mykey]};
        });

        let stateStore = document.getElementById(stateid);
        stateStore.value = JSON.stringify(newState);
        stateStore.dispatchEvent(new Event('change'));
        state = newState;
```

```
    });
}
```

This creates an array of the `data-id` attributes of the list items, representing the order the student has sorted the original keys into, and then creates a `newState` object consisting of the keys and corresponding statements in this new order. This is then converted to a JSON string using `JSON.stringify(newState)`, which is in turn stored in the hidden input in the iframe. Since this iframe input has been configured to be synchronised with our hidden STACK input, we therefore have a JSON string representing the sorted list in our STACK input.

Marking the attempt

In this simple Parsons problem, the correctness of a student's answer may be determined by comparing the student's ordered list of keys from the `statestringinput` STACK input with the ordered list of keys specified in the initial question variables within the `correctlist` variable. Since the marking of the answer is done within STACK, using variables configured in the feedback variables field of a potential response tree (PRT), we should convert the JSON string from `statestringinput` back to a STACK-map for analysis with Maxima. As mentioned above in the section on inputs and STACK-maps, we can create a Maxima variable `stateString` using the helper function `stateString:stackjson_parse(statestringinput)`. Since we only need to compare the keys from the STACK-maps, we can extract just these keys into two respective lists as follows:

```
correctkeys:stackmap_keys(correctlist);
sakeys:stackmap_keys(stateString);
```

There are many ways one could produce a score, and indeed feedback, given these two lists. One possible algorithm, which we present here without the explicit Maxima code for brevity, is:

- Let n be the number of items the student is sorting, so in our simple example n = 5.
- For the first item in the student's list, give a score of $\frac{1}{2n}$ if the key for this item is also the first key in the teacher's list, otherwise set the score to 0.
- For the first item in the student's list, add a score of $\frac{1}{2n}$ if the second key is the second key in the teacher's list.
- For items $k \in \{2, \ldots, n-1\}$ in the student's list: check whether key $k-1$ in the student's list is the same as key $k-1$ in the teacher's list, if so add a score of $\frac{1}{2n}$. Similarly, add a score of $\frac{1}{2n}$ if key $k+1$ matches in both lists.
- For the final item in the student's list, add a score of $\frac{1}{2n}$ if the preceding key matches in both lists. Similarly, add a score of $\frac{1}{2n}$ if this is also the final key in the teacher's list.

This algorithm gives partial credit for items being locally correct when compared to their neighbours, even if a student has the wrong absolute order (due to an incorrect first item for example). If a student submitted the answer as shown above in Figure 1 for example, they would score 0.5/1.0, since the numbers 1 and 2 are in the absolute correct position (scoring a total of 0.3), and the numbers 3 and 4 are adjacent items in the correct order (scoring an additional 0.2) despite being in the wrong absolute position. We briefly return to the case of their being multiple correct answers in the following section.

4. A more complicated Parsons problem

While the example discussed in detail in the previous section demonstrates how to use STACK-JS for a simple Parsons problem, we can make some additions which greatly improve how useful this would be for assessing a mathematical proof. Here we present only the broad steps required for two important additions we might make.

The first limitation of our simple example is that it doesn't allow for distractors — items which should be separated out of the correct list. As an example, a distractor could be used to test a student's understanding of logical qualifiers in a proof, with statements that differ only by whether "there exists an element of the set *X*", or "for all elements of the set *X*". In order to allow for this, we can add a second list to our Parsons problems, with one list for indicating the correct statements in the correct order, and the second for indicating the statements which should not be included in the proof (irrespective of order). We start with all statements displayed in the 'incorrect' list and ask the student to filter the valid proof steps into the 'correct' list, in the correct order. Although we would still only need to mark the 'correct' list, we would want to preserve the order of the items in both lists visually when a student checks their answer. We can achieve this by extending our state variable such that the corresponding JavaScript object contains both a `correctlist` and an `incorrectlist`, both of which are themselves JavaScript objects of key:value pairs similar to our simple example.

Secondly, our simple example supports only a unique correct answer. A proof involves showing two sets are equal, by showing each set contains the other as a subset, has the obvious freedom to demonstrate the subset inclusions in either order. Such a Parsons problem would therefore not have a unique solution. We can support such cases by allowing the question author to specify a list of alternative correct orders for the keys. When marking the student's answer, we can then apply a modified version of the algorithm presented previously. In particular, for each key in the student's answer we can consider the following keys in all alternative answers, and give a mark if the following key in the students answer matches any of the valid following keys. We then do similarly for the preceding keys.

An example question demonstrating these features is shown below in Figure 2. In this image, the student has already constructed a correct answer to the question. If the student were to move the fifth and final statement from their 'correct' left-hand list into the second position in this list (with statements two to four simply moving down in fixed order), they would still score full marks for this question, as this represents an alternative correct order as specified by the question author.

Let $f : X \to Y$ be a function, and assume that $A, B \subset X$, and $C, D \subset Y$. Arrange the following steps into order to construct a proof of the statement

Question is missing tests or variants.

$$f^{-1}(Y \setminus C) = X \setminus f^{-1}(C).$$

Arrange the following items into the correct order in the left-hand list. Any items that you don't want as part of your answer should be placed into the right-hand list.

Let $x \in f^{-1}(Y \setminus C)$.	Then $f(x) \in Y \cap C$.
Then $f(x) \in Y \setminus C$.	Let $x \in f^{-1}(Y) \setminus C$.
This implies $f(x) \notin C$.	This implies $x \notin C$.
Hence we have $x \notin f^{-1}(C)$.	
Since $X = f^{-1}(Y)$, we have $x \in f^{-1}(Y)$.	

Check

Figure 2: A more complex Parsons problem using STACK-JS and SortableJS, demonstrating the possibility for distractors in the problem.

5. Summary

In this paper we have presented the key steps for integrating custom JavaScript into STACK questions using STACK-JS. This enables possibilities such as new input methods for students to use, or advanced visualisation methods, among others. Here, we presented an example of the former use case, creating drag-and-drop Parsons problems (Ericson et al., 2022) suitable for testing students' proof-comprehension skills. Examples of the latter use case might include using the D3 library (Bostock, 2024) for data visualisation, or VisualPDE (Walker et al., 2023) for the visualisation and exploration of 1d and 2d PDEs and their solutions.

Following conversations with the core developers, STACK now natively supports drag-and-drop Parsons problems (Sangwin, 2023), building on the ideas presented in this paper. We would therefore not recommend that question authors use the specific implementation of Parsons problems presented here, in favour of using the native implementation. However, we hope this example nevertheless serves as an effective demonstration of how to use STACK-JS to add advanced features to STACK questions.

Bibliography

Bickerton, R. and Sangwin, C. (2022). *Practical on- line assessment of mathematical proof.* International Journal of Mathematical Education in Science and Technology, 53(10), pp. 2637–2660. https://doi.org/10.1080/0020739X.2021.1896813

Black, P. and Wiliam, D. (1998). *Assessment and classroom learning*. Assessment in Education: principles, policy & practice, 5(1), pp. 7–74. https://doi.org/10.1080/0969595980050102

Black, P., & Wiliam, D. (2010). *Inside the Black Box: Raising Standards through Classroom Assessment*. Phi Delta Kappan, 92(1), pp. 81-90. https://doi.org/10.1177/003172171009200119

Bostock, M. (2024). D3. GitHub. https://github.com/d3/d3

Ericson, B. J., Denny, P., Prather, J., Duran, R., Hellas, A., Leinonen, J., Miller, C. S., Morrison, B. B., Pearce, J. L., and Rodger, S. H. (2022). *Parsons problems and beyond: Systematic literature review and empirical study designs*. Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education, pp. 191–234. https://doi.org/10.1145/3571785.3574127

Harjula, M. (2023). *STACK-JS*. STACK Docs. https://docs.stack-assessment.org/en/Developer/STACK-JS/

Mills, O. (2024). Sortable. GitHub. https://github.com/SortableJS/Sortable

Sangwin, C. (2013). *Computer aided assessment of mathematics*. OUP Oxford.

Sangwin, C. (2018). JSXGraph. STACK Docs. https://docs.stack-assessment.org/en/Authoring/JSXGraph/

Sangwin, C. (2023). *Support for proof in STACK*. STACK Docs. https://docs.stack-assessment.org/en/Proof/

Shute, V. (2007). The future of assessment: Shaping teaching and learning. *Tensions, trends, tools, and technologies: Time for an educational sea change*, pp. 139–187.

Walker, B.J., Townsend, A.K., Chudasama, A.K., and Krause, A. L. (2023). *VisualPDE: Rapid Interactive Simulations of Partial Differential Equations*. Bulletin of Mathematical Biology 85(113). https://doi.org/10.1007/s11538-023-01218-4

Information on the author

Sam Fearn is an Associate Professor (Education) in the department of Mathematical Sciences at Durham University. Before starting this role in 2020, he worked as a Teaching Fellow in the same department, following the completion of his Ph.D. in 2018.