Behavior Research Methods
https://doi.org/10.3758/s13428-021-01598-2

SweetPea: A standard language for factorial experimental design

Sebastian Musslick[1] · Annie Cherkaev[2] · Ben Draut[2] · Ahsan Sajjad Butt[2] · Pierce Darragh[2] · Vivek Srikumar[2] · Matthew Flatt[2] · Jonathan D. Cohen[1,3]

Accepted: 12 April 2021

Abstract

Experimental design is a key ingredient of reproducible empirical research. Yet, given the increasing complexity of experimental designs, researchers often struggle to implement ones that allow them to measure their variables of interest without confounds. SweetPea (https://sweetpea-org.github.io/) is an open-source declarative language in Python, in which researchers can describe their desired experiment as a set of factors and constraints. The language leverages advances in areas of computer science to sample experiment sequences in an unbiased way. In this article, we provide an overview of SweetPea's capabilities, and demonstrate its application to the design of psychological experiments. Finally, we discuss current limitations of SweetPea, as well as potential applications to other domains of empirical research, such as neuroscience and machine learning.

Keywords Factorial design · Randomization and sampling · Sequential constraints · Nuisance factor

Introduction

Reproducibility is an issue of increasing concern in scientific research. The problem has gained considerable attention in psychological research, where behavioral phenomena are often confounded by the presence of various moderator variables (Sherman & Pashler, 2019; Klein et al., 2014; Open Science Collaboration & et al., 2015; Stroebe & Strack, 2014; Dijksterhuis, Van Knippenberg, & Holland, 2014). More generally, the problem arises in any field where an experiment systematically manipulates a set of variables, controlling as best as possible for variables of non-interest, to evaluate the effects of experimental manipulation. This approach, a fundamental staple of the scientific process, applies to empirical studies in the natural sciences as well as the training and evaluation of artificial systems in engineering, such as machine learning algorithms (Langley, 1988; Zadrozny, 2004; Drummond, 2006; Bergstra & Bengio, 2012; Gardner, Neumann, Grus, & Lourie, 2018).

Experimental design

Methods of experimental design date back to the study of inheritance through the cultivation of pea plants[1] (Miller, 2011), and remains a gold standard in psychological research. In experimental settings, an ideal design samples variables of experimental interest in an unbiased way, while suitably controlling for variables of non-interest. While this is easy to state in principle, it is often difficult to achieve in practice. This can be for many reasons. One is that it may be impossible to sample all of the combinations of variables needed to achieve a fully balanced design. Random sampling is often used to address this. However, when data collection is expensive and therefore sample sizes are constrained, random sampling may be inadequate to ensure an adequately balanced or controlled design (Button et al., 2013; Kühberger, Fritz, Scherndl, 2014; Rossi, 1990; Wells & Windschitl, 1999). Randomization is also often used when the complexity of a design makes it difficult to determine how to satisfy constraints imposed by the design in a more

✉ Sebastian Musslick
musslick@princeton.edu

[1] Princeton Neuroscience Institute, Princeton University, Princeton, NJ 08544, USA

[2] School of Computing, University of Utah, Salt Lake City, UT 84112, USA

[3] Department of Psychology, Princeton University, Princeton, NJ 08544, USA

[1] This involved the crossing of pure-breeding lines of plants as experimental factors and assessing traits of the hybrid progeny (Druery & Bateson, 1901; Mendel, 1866).

Published online: 06 August 2021

systematic way (e.g., Mayr & Keele, 2000); however, naive randomization can render the design vulnerable to unintended and unrecognized biases. There is an increasing risk that this problem will be amplified by platforms that provide the infrastructure for automated data collection (De Leeuw, 2015; Gureckis et al., 2016; Hartshorne, de Leeuw, Goodman, Jennings, & O'Donnell, 2019); without proper experimental design, these approaches to large-scale data collection may encounter the same issues of reproducibility as traditional data collection, but in more arcane and difficult to detect ways. Finally, interest is increasing in optimal experimental design where, rather than simple counterbalancing, it is helpful to identify experimental conditions that are best for discriminating between models (Dale, 1999; Myung & Pitt, 2009)

These issues also arise in engineering. For example, in machine learning, the outcome of training is profoundly influenced by the statistics of the stimuli on which the model is trained; small sample sizes or violations of proper random sampling can limit performance or validity of statistical learning techniques, such as naive Bayes or decision tree learners (Zadrozny, 2004) or, conversely, it may be useful to identify designs that contrast conditions or segment the data in ways that optimize learning. Furthermore, the success of a machine learning system crucially depends on the choice of feature representations, models and their hyperparameters, which together call for searching through a potentially intractable search space (Chapelle, Vapnik, Bousquet, & Mukherjee, 2002; Bergstra & Bengio 2012).

Factorial structure vs. implementation constraints

The best approach to the concerns raised above is a well designed experiment. Factorial structure is the cornerstone of well-controlled experimental design involving repeated measures: The experimental variables manipulated in each repeated measure, or *trial*, are defined as *factors*, and the values of each variable to be sampled in the experiment are defined as *levels* of the corresponding factors. As an example, consider a Stroop experiment (Stroop, 1935) in which participants are asked to name the color in which a color word is displayed (e.g., say "green" to the written word RED displayed in green). The color and word are factors of the experiment. Instances of colors (e.g., green) and color words (e.g., "RED") can be considered levels of the factors color and word, respectively. The trials of the experiment are then constructed by crossing the factors—that is, selecting the levels from each factor to be used in each trial. For instance, a simple design for the Stroop experiment may seek to counterbalance just the combination of words and colors to be used (i.e., by insuring that every word appears in every color). In addition to providing a concise, formally rigorous description of the experimental design, expression in factorial form also aligns with standard approaches to statistical analysis; for example, each factor (e.g., color and word) can be represented as an effect in an analysis of variance (ANOVA) or, more generally, as a regressor in an analysis using the general linear model (GLM).

When an experimental design consists of a small number of independent factors, each with a modest number of levels, and all combinations of their levels can be subjected to measurement, then generating the trials for an experiment is straightforward: cross all of the factors, generate every possible permutation of the crossings (i.e., sequences of trials), and use each permutation in a different run of the experiment (this is often referred to as *full counterbalancing*). However, for many designs this can be difficult to achieve for one or more of several reasons. First, the full crossing of all factors may generate too many combinations of levels to sample in a single run of the experiment (e.g., for a single participant to complete). To illustrate this, consider a Stroop experiment with ten words, ten colors and two additional factors: a task factor indicating the task to be performed (i.e. whether to name the color or read the word), and a task transition factor (task transition) indicating whether the task on the current trial is repeated or switched, relative to the previous trial. A full crossing of all factors would require $10 \times 10 \times 2 \times 2 = 400$ trials to include all combinations of levels—an experiment sequence that may require too much time for a single participant to complete. Second, even if the full crossing can be sampled in a single run, there may be too many permutations (i.e. possible trial orders) to sample over runs of the experiment. To avoid a sampling bias, one could distribute all possible runs of the experiment over participants; but this may require too many participants. These challenges are often met by writing a program that sub-samples from the full crossing and permutation of sequences (i.e. trial orders). Thus, for the first problem, the program might select a subset of combinations of levels from the full crossing for use in each run (e.g., only certain combinations of the factors color, word, task or task transition in the Stroop experiment), either randomly or by insuring that all possible subsets are used across runs of the experiment (e.g., using a Latin square design). For the second problem, sequences can be drawn randomly from the full set of permutations (of either the full crossing, or the subset of it used for each run). In the Stroop experiment, this would amount to sampling, for each participant, from the set of trial sequences in which all factors (color, word, task transition) are fully crossed.

Sometimes, random sampling is sufficient to achieve an adequately balanced sample of the different implementations of the design, in which case measurements across runs are not biased in ways that confound interpretation. For more complicated designs, e.g., with sequential constraints, random sampling can be combined with rejection and/or repair algorithms to satisfy all design constraints. For instance, in a more complicated design of the Stroop experiment may demand that the color and word disagree (e.g., that the color and word used on a given trial be different than the ones used on the trial before and after it) or that the task to be performed (i.e., reading the word vs. naming the color) be repeated a maximum number of times in a row. One could generate a valid trial sequence, by randomly shuffling the counterbalanced set of factors in the experiment. If the generated sequence satisfies all required constraints (e.g., that colors and words in adjacent trials are different from one another) it is accepted and otherwise rejected. However, there are three problems that often arise with this approach: 1) Random sampling with rejection may be unlikely to yield experiment sequences that satisfy the set of desired constraints. This issue is particularly problematic in designs in which researchers attempt to account for sequential (across-trial) dependencies between experiment factors (e.g., that both the word and the color switch between all trials). 2) Even if a solution can be found, simple forms of randomization do not ensure that the samples are unbiased. This is of particular concern when the factors are not independent of one another (e.g., the factor task transition is dependent on the factor task), and especially when there are sequential dependencies (e.g., that a task can only be repeated for a certain number of trials). This can greatly complicate, and in some cases seriously constrain how subsets of factors can be crossed with one another, introducing biases in the sampling process that can lead to unintended (and often unrecognized) confounds in the design. 3) Even when randomization is executed in a way that ensures an unbiased design, it is often done using custom-written code that can be difficult to understand and therefore replicate, and is almost always specific to the particular experiment and thus not more generally useful.

Factor dependencies and sequential constraints have become increasingly common in cognitive psychological studies, as researchers seek to implement more sophisticated designs that more precisely address processes of interest (e.g., the mechanisms responsible for switching attention between tasks). Constructing algorithms that satisfy the constraints of such designs, while uniformly sampling (i.e., counterbalancing) all legal variants of the design, can quickly become a challenge even for a professional programmer. However, failure to do so may either not yield the desired design and/or introduce unintended biases. Indeed, as designs have become more complex, so too has the frequency of confounds identified in such designs (e.g., Allport & Wylie, 1999; Cooper & Marí-Beffa, 2008; Jou, 2014; Logan & Schneider, 2010; Zmigrod & Hommel, 2013). Furthermore, even when appropriate algorithms are used, the code that implements them is usually complex and design-specific, posing problems for interpretability and reproducibility (Miłkowski, Hensel, & Hohol, 2018, Peng, 2011; Sochat et al., 2016).

A number of software solutions have been proposed to address the issues raised above. Some of them are embedded in general-purpose packages for running psychological and/or neuroscientific experiments, such as PsychoPy (Peirce, 2007) or E-Prime (Schneider, Eschman, & Zuccolotto, 2002). These enable basic counterbalancing of experimental factors but do not allow the user to impose complex constraints on the generated sequence. More sophisticated solutions, tailored to the problem of generating pseudorandom sequences, rely on iterative rejection sampling[2] (Mathôt, 2016; van Casteren & Davis, 2006) or genetic algorithms (Klein et al., 2014; Wager & Nichols, 2003). However, these methods do not permit to sample experiment sequences uniformly from the space of all possible solutions. Moreover, rejection sampling and genetic algorithms are not guaranteed to identify a valid experiment sequence, even if it exists. Finally, all existing software solutions to experiment design are limited in that they tie the specification of an experiment design to one particular way of sampling that design. A novel approach is needed to mitigate these limitations.

A new language for experimental design

To address the concerns raised above, we have developed the first generation of a software tool that is designed to narrow the gap between the specification of an experimental design and its implementation in trial-generating programs. Our approach is to create a declarative language implemented in **Python** for describing experimental designs in factorial form, and then express the design in a format that can be used by standard computational sampling algorithms to ensure—as best as possible—that the trial sequences generated are as unbiased as possible. That is, instead of separately describing the intended design and then writing down a set of `for` loops and calls to `random` that are intended to implement that design, an experimenter can describe the design precisely in a language that is tailored to the problem of experimental design. The generation of trials from that description can then be fully automated, insuring that it is done in a consistent and well-defined way, and with the hope of providing increasingly strong guarantees about the statistical properties of generated trials as

[2] Iterative rejection sampling involves the following steps: (1) Append the experiment sequence with a randomly selected trial, (2a) reject the sampled trial if the appended experiment sequence fails to satisfy all specified constraints or (2b) if the experiment sequence remains valid, randomly sample the next trial and repeat steps (1) and (2) until the experiment sequence is complete. Rejection sampling methods may invoke additional repair algorithms to ensure that the generated sequence remains valid.

the tool is improved by incorporating new advances in sampling and sample analysis algorithms, and/or computational power increases.

The language we describe, **SweetPea**, builds on the foundation of factor-based design. In addition to independent factors that can be crossed in various ways, the language adds a notion of *derived factors*, which allow an experimenter to give names to properties of interest (e.g., whether two properties of a stimulus are "congruent" or "incongruent"). Finally, the experimenter can impose explicit *constraints* on the factors, derived factors, and even the relationship of different factors within a window of adjacent trials in the experiment. This combination of *factors* and *constraints* directly addresses the growing complexity of experimental designs in empirical sciences and machine learning. Generating a run of the experiment corresponds to sampling from the solution space of those constraints, ideally with a guarantee of uniformity that avoids accidental correlations in generated trials.

The implementation of SweetPea mirrors the design goal of separating the "what" of experimental design from the "how" of generating trials. Our language front-end converts an experimental design into a constraint-solving problem, where various constructs imply constraints on trial content and order. The constraint problem then can be delegated to a combination of custom and existing solvers and samplers that act as the back-end of the implementation. This structure substantially decouples the problem of creating an expressive experiment-design language from the problem of implementing it efficiently enough, so researchers can explore both directions concurrently.

The SweetPea language is an open-source project[3] and, as such, work in progress. The development of SweetPea focused on two fronts. First, with respect to expressiveness, while the language has proven rich enough for several designs of immediate interest (e.g., for which generating trials by one-off programs has been challenging; see Section *Walkthrough Examples*), it is still limited in a number of ways. For example, it currently does not support the weighted sampling of factor levels and weighted crossings thereof. Our hope is that continued development (on our part, and through community support), will refine and expand the language and its semantics to cover a greater range of experimental designs. Second, with respect to sampling guarantees, we have not yet found a practical solution to the problem of guaranteeing a uniform sample of the space of all possible constraint solutions (valid trial sequences); that general problem is an area of active research in computer science, and we report on our experience to date using currently available solutions. However, by standardizing the expression of experimental designs, and its interface with sampling methods, our tool ensures that as advances are made in computational algorithms for sampling and sample analysis, these can be easily introduced into SweetPea, thus allowing experimental design methods to readily leverage these advances.

Using SweetPea

SweetPea is a "language" in the sense that it defines a vocabulary and set of abstractions for describing experiments, but it is implemented as a library in Python. An experiment description in SweetPea is a Python value that is constructed with lists, objects, and primitive data types (e.g., strings or integers) that represent factor names and level values. Passing an experiment description to SweetPea's `synthesize_trials` function generates a list of runs. Each run in that result is reported as a dictionary mapping factor names to level values, with one level value for each trial in the run.

The interpretation of factor and level strings (e.g., interpreting a `"color"` factor's `"red"` level to mean that a stimulus should be displayed in the color red) is outside the scope of SweetPea, but embedding SweetPea in Python simplifies the bridge between trial generation and presentation, since experiment-presentation systems, such as PsychoPy (Peirce, 2007, 2009) or Expyriment (Krause & Lindemann, 2014), are also available as Python libraries. Similarly, SweetPea can interface with data processing pipelines used to train artificial systems in Python, such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016), as well as simulation environments targeted specifically at cognitive and neuroscientific models such as PsyNeuLink (psyneulink.org).

Defining factors

Figure 1 shows the complete grammar of SweetPea experiment descriptions in terms of Python values. For example, the production `[`*block*`, ...]` means a list of *block*s values, and the production `factor(`*factor_name*`, [`*level_value*`, ...])` means an object produced by passing a *factor_name* and a list of *level_value*s to the `factor` function.

An experiment consists of one or more blocks, where each block is a sequence of trials, and each trial combines one level from each of the block's factors. A factor is either *independent* or *derived*, depending on whether its levels are primitive data types (e.g., string or integer) or created by the `derived_level` function. For example, consider the Stroop experiment mentioned above, in which participants are asked to name the color in which a color word is

[3] We invite contributions to SweetPea's open-source repository (https://github.com/sweetpea-org/).

```
experiment     =  block | [block, ...]

block          =  fully_cross_block(design, crossing, constraints)
design         =  [factor, ...]
crossing       =  [factor, ...]
constraints    =  [constraint, ...]

factor         =  factor(factor_name, [level_value, ...])
                | factor(factor_name, [derived_level, ...])

derived_level  =  derived_level(level_value, derivation)
derivation     =  within_trial(predicate, [factor, ...])
                | transition(predicate, [factor, ...])
                | window(stride, width, function, [factor, ...])
predicate      =  a function from [level_value, ...] to boolean

constraint     =  exclude(factor_name, level_value)
                | at_most_k_in_a_row(k, levels)
                | exactly_k_in_a_row(k, levels)
                | no_more_than_k_in_a_row(k, levels)
                | minimum_trials(trial_number)
levels         =  (factor_name, level_value)
                | factor

factor_name    =  a string
level_value    =  a primitive data type (string, integer, float, boolean)
k, stride, width = a positive integer
```

Fig. 1 SweetPea grammar

displayed. Each trial of the Stroop experiment consists of the independent factors `color` and `word` that can be defined as

```
color_list = ["red", "green", "blue"]
color = factor("color", color_list)
word  = factor("word",  color_list)
```

Note that `color` here is a Python variable, while `"color"` is a factor name. A SweetPea experiment description has factor names, not variables, but Python variables are useful to give names to parts of SweetPea experiments.

In a simple variant of the Stroop experiment, it may be sufficient to fully cross the independent factors `color` and `word`, and then randomize the order in which the trials are drawn from the crossing to produce a block of trials. In other cases, the experimenter may want to constrain the occurrences of “congruent” trials (i.e., in which the color and word match) versus “incongruent” trials (in which they are different). This notion of congruence can be made explicit by deriving it in terms of the `color` and `word` factors:

```
congruent   = derived_level("con",
   within_trial(operator.eq, [color, word])
   )
incongruent = derived_level("inc",
   within_trial(operator.ne, [color, word])
   )
congruence  = factor("congruence",
    [congruent, incongruent])
```

A derived level such as `congruent` or `incongruent` applies a predicate to levels drawn from other factors. The levels might be candidates within a trial or across trials; the `within_trial` function applies the predicate to candidate levels within a single trial, which is what is needed for defining congruence in the example above. The predicate given to `within_trial` must take the same number of arguments as factors in a list given to `within_trial`. In the example, the Python predicate `operator.eq` expects two arguments and returns `True` when they are the same, while `operator.ne` returns `True` when two arguments are different, so those predicates make sense given levels from `color` and `word` as specified by `[color, word]`. Thus, the uses of `within_trial` for `congruent` and `incongruent` select complementary portions of the `color`–`word` crossing. The `derived_level` function assigns a name to each of those spaces, and the resulting levels are combined with `factor` to define a `congruence` factor. The `factor` function requires that each derived level identifies a distinct portion of a crossing and that every part of the crossing is covered by one of the levels.

If `color`, `word`, and `congruence` are declared to be the factors of a block, then each trial in that block will have a level for each of those three factors. For example, two possible trials are

```
{ 'color': 'red', 'word': 'green', '
   congruence': 'inc' }
{ 'color': 'red', 'word': 'red',  'congruence
   ': 'con' }
```

SweetPea’s `print_experiments` function can be used to print these Python dictionaries more compactly:

```
color red | word blue | congruence inc
color red | word red  | congruence con
```

The following is *not* a possible trial, because the derived factor’s level is not consistent with the other factors’ levels:

```
color red | word blue | congruence con
```

While the `congruence` level could be inferred from the `color` and `word` levels, a benefit of explicitly defining `congruence` is that a block can then be specified to be the crossing of, say, `color` and `congruence`, which ensures that the block contains every color once with each congruence. A possible block produced by synthesize_trials in that case is

```
color red   | word green | congruence inc
color red   | word red   | congruence con
color blue  | word blue  | congruence con
color green | word green | congruence con
color green | word red   | congruence inc
color blue  | word red   | congruence inc
```

Note that this sequence does not include every possible color–word combination, as would be produced by instead crossing color and word. Crossing word and congruence would also work, and would ensure that each word appears twice within the block. Crossing all three factors would *fail* and cause synthesize_trials to report an error, because some combinations of levels violate the constraint inherent in the definition of congruence, such as

```
color red   | word blue | congruence con
```

Crossing and constraints

In addition to the constraints that are inherent to a derived factor, an experimental design can also impose other constraints on generated trials. Those may rule out particular combinations of levels (i.e., portions of a crossing), or they may constrain the order of combinations. Typically and most usefully, some aspects of the trials and ordering can remain unconstrained. For example, in the cases above the order of trials within a crossing is unconstrained, so that synthesize_trials can randomly select among all possible orderings. Similarly, crossing color and congruence allows the word level to vary randomly among the two possibilities for each incongruent trial.

The general constructor to describe a block of trials is fully_cross_block, which expects three arguments:

- *design*: A list of all factors in the experiment. Every trial in the block will have a single level value for each of these factors.
- *crossing*: A subset of the factors in *design* that are to be fully crossed in each run of the block. This crossing imposes two constraints on the result: the minimum number of trials per run is the product of the factor sizes, and each of those runs must have the same number of instances for every distinct combination of levels among the factors.
- *constraints*: Additional constraints on either (a) the minimum number of trials, (b) the order of the *crossing* combinations, or (c) the factor levels in *design* that are not in *crossing*.

As an example constraint, suppose that we want to fully cross color and word in our Stroop experiment, but we do not want two consecutive congruent trials. The at_most_k_in_a_row function lets us express this constraint:

```
one_con_at_a_time = at_most_k_in_a_row(1, (
    congruence, congruent))
```

The at_most_k_in_a_row, exactly_k_in_a_row, and no_more_than_k_in_a_row functions take a count for "k" and one or more levels. A single level is specified as a tuple of a factor_name and level_value, while a factor argument indicates all of the levels of the factor, effectively replicating the constraint for each level.

Assembling the factors and constraints into a block completes the design of a single-block experiment:

```
design       = [color, word, congruence]
crossing     = [color, word]
constraints  = [one_con_at_a_time]

block   = fully_cross_block(design, crossing,
    constraints)
```

A potential run of this experiment is

```
color blue  | word green | congruence inc
color green | word blue  | congruence inc
color blue  | word blue  | congruence con
color blue  | word red   | congruence inc
color green | word red   | congruence inc
color red   | word red   | congruence con
color red   | word green | congruence inc
color green | word green | congruence con
color red   | word blue  | congruence inc
```

Note that trying to constrain the experiment to at most one incongruent trial in a row would fail, because there are more incongruent pairs than congruent pairs in a color–word crossing, so adjacent incongruent trials are inevitable. SweetPea's synthesize_trials function reports an error when constraints have no solution, but it cannot always provide a simple explanation for why an experiment is over-constrained.

Transitions and windows

The one_con_at_a_time constraint sets up a kind of transition constraint, in that a congruent trial must transition to an incongruent trial. SweetPea also supports a more direct and general way of controlling transitions. For example, suppose that each trial of our experiment should be related to the previous trial by having either the same color or the same text, but not both. By using transition instead of within_trial, we can create a derived factor that categorizes a trial as either changing exactly one independent factor or both, relative to the previous trial:

```
def one_diff(colors, words):
    if (colors[0] == colors[1]):
        return words[0] != words[1]
    else:
        return words[0] == words[1]

def both_diff(colors, words):
    return not one_diff(colors, words)

one = derived_level("one", transition(one_diff
    , [color, word]))
both = derived_level("both", transition(
    both_diff, [color, word]))
changed = factor("changed", [one, both])
```

Like within_trial, the predicate provided to transition must accept one argument for each factor listed to transition. But while within_trial gets a level value for each argument, transition gets a list of two level values for each argument. The first element in the list is the previous trial's value, and the second element in the list is the current trial's value. The one_diff predicate here determines whether exactly one of the lists has the same values for its two elements.

If we just add changed to our experiment, then the result shows whether one of color and word change or both:

```
color blue   | word green | congruence inc
| changed
color green | word blue   | congruence inc
 | changed both
color blue   | word blue   | congruence con
 | changed one
color blue   | word red | congruence inc
 | changed one
color green | word red   | congruence inc
 | changed one
color red   | word red   | congruence con
 | changed one
color red   | word green | congruence inc
 | changed one
color green | word green | congruence con
 | changed one
color red   | word blue   | congruence inc
 | changed both
```

Notice that there is no changed value for the first trial, since a trial needs a preceding trial to determine its transition type. To rule out trials that change both color and word relative to the previous one, we can add an exclude constraint:

```
constraints  = [one_con_at_a_time,
               exclude(changed, both)]
```

As it happens, the one_con_at_a_time constraint is now redundant, since a congruent trial must be followed by one that changes one of color or word. Such redundancies are not always obvious, however, and including them in the design causes no problems. With this new constraint, synthesize_trials cannot generate the previous run, and it might instead generate this one:

```
color blue | word green | congruence inc
| changed
color red | word green | congruence inc
| changed one
color red | word red | congruence con
| changed one
color red | word blue | congruence inc
| changed one
color green | word blue | congruence inc
| changed one
color blue | word blue | congruence con
| changed one
color blue | word red | congruence inc
| changed one
color green | word red | congruence inc
| changed one
color green | word green | congruence con
| changed one
```

Derived factors defined by transition can also be included in an experiment's crossing. For example, if we define the experiment's crossing to be color and changed and omit further constraints, then a possible experiment is

```
| color blue | word green | congruence
 inc | changed
color red   | text red   | congruence con
| changed
color green | text green | congruence con
 | changed both
color blue | text green | congruence inc
 | changed one
color blue | text green | congruence inc |
 changed both
color red   | text blue | congruence inc |
 changed both
color green | text blue | congruence inc |
 changed one
color red   | text blue | congruence inc |
 changed one
```

In this experiment, each color is included once for each changed, which is why the experiment includes six trials to cover all combinations, plus one more at the start to enable the the changed transition. Not every color and word combination appears, since word has been removed from the crossing. If we cross all three of color, word, and changed, the resulting experiment would have $3 \times 3 \times 2 + 1 = 19$ trials.

The within_trial and transition functions for derived levels are both shorthands for a general window constructor. The general window form supports relationships among any fixed number of trials, and it allows the trials to be separated by a given stride within the run instead of requiring them to be adjacent. We illustrate the use of the window function in the walkthrough example *Designing a 2-Back Task*.

Exporting experiment sequences

The process of designing an experiment is usually followed by stimulus presentation and data collection. Unlike other software packages, SweetPea does not provide functionality for executing experiments. However, as illustrated in Section *Walkthrough Examples*, SweetPea allows the user to export experiment sequences obtained from synthesize_trials into a comma-separated values (CSV) file, using

```
experiment_to_csv(experiments, file_prefix="
    path/experiment")
```

The function experiment_to_csv accepts two arguments, the second of which is optional. The first argument specifies the set of experiments to be exported, i.e. the output of synthesize_trials. The second argument file_prefix specifies both the path and the prefix of each experiment file. Executing this function will produce as many CSV files as dictionaries are contained in the list experiments. In this example, each CSV file will be located in the relative path "path/experiment_X.csv" where X corresponds to the index of each experiment sequence in experiments.

The generated CSV files can be imported by the majority of software packages used for stimulus presentation and data collection, such as Psychtoolbox (Kleiner, Brainard, & Pelli, 2007), PsychoPy (Peirce, 2009), OpenSesame (Mathôt, Schreij, & Theeuwes, 2012) or E-Prime (Schneider, Eschman, & Zuccolotto, 2002). Thus, SweetPea can be easily combined with other platforms for executing experiments. For Python-based environments, SweetPea can also be directly integrated into the programmatic workflow for executing experiments (e.g., in PsychoPy (Peirce, 2007) or in OpenSesame (Mathôt, Schreij, & Theeuwes, 2012)), as well as for running machine learning models and/or machine learning algorithms (e.g., in TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019) or PsyNeuLink (psyneulink.org).

Solving experimental designs

The simple Stroop experiment introduced introduced above has only 9 trials in a full crossing of color and word. Those 9 trials can be ordered in $(3 \times 3)! = 362,880$ ways. For that scale, a practical implementation could simply enumerate all permutations, filter ones that do not conform to a constraint, and then randomly select from the rest. Raising the number of color names from 3 to 5, however, creates 25 elements in the crossing, which means 1.5×10^{25} permutations; it's still a tiny experiment, but generate-all-and-filter is not remotely practical. For sufficiently constrained experiments, even generate-one-and-test is not practical, because the generated experiment is unlikely to satisfy the constraints.

The SweetPea implementation can avoid generate-and-test and instead use an external program to find a trial sequence that crosses all specified factors and that satisfies all indicated constraints (Cherkaev, 2019). The external program is a *SAT solver*, which can find a solution to a general logical formula (or report that no such solution exists). In principle, finding a solution to an arbitrary logical formula is intractably difficult. In practice, modern SAT solvers have heuristics that efficiently find solutions. SweetPea compiles the constraints that define an experiment into a disjunctive normal form—a logical expression that a SAT solver can recognize—and then it translates the SAT solver's answer back to experiment terms.

Using a SAT solver is good for finding *one* valid run of an experiment. A second run can be generated by further constraining the solution so that it's different from the first solution. While this strategy can be used to generate multiple experiments, it unfortunately fails to solve one problem raised above, which is to sample the space of solutions uniformly; new solutions generated by simply ruling out previous solutions might explore only a small part of the solution space.

A *SAT sampler* can drive a SAT solver to explore the full space of solutions. SweetPea uses Unigen2 (Chakraborty, Meel, & Vardi, 2014) to implement uniform sampling of small experimental designs.[4] Our attempt to use SAT samplers for medium-to-large designs has not so far been effective, however. The problem is that current SAT samplers rely on a formula having a certain structure, and the way that SweetPea currently encodes experiments does not fit that structure. Indeed, the mismatch appears to be fundamentally related to the problem of generating permutations of possible trials, which suggests that no encoding will work.

SAT solving and sampling can work for some experimental designs, generate-and-test can work for some designs, and strategies that involve sampling from numbers with a bijection between numbers and solutions can work in other designs. By framing the problem in terms of a declarative specification of an experiment, SweetPea separates the

[4] In future releases of SweetPea, we will leverage Unigen4 (https://github.com/meelgroup/unigen).

problem of describing an experiment and generating the experiment—much the same way that SAT solvers separate the problem of specifying formulas and solving those formulas.

Much of our ongoing work is about finding new implementation strategies to “solve” experiments and to automatically determine which strategies will work for a given experiment description. For now, SweetPea offers a sampling strategy that can handle experiments without constraints by just picking a random permutation among the crossing. SweetPea also currently offers a strategy for finding trial sequences for small-to-medium designs, which can be found using a SAT solver without a sampler, but it does not provide a guarantee of uniform sampling. New samplers will likely involve a hybrid of these approaches.

Walkthrough examples

This section illustrates the application of SweetPea to three different experimental design problems. The first design problem (Stroop task) illustrates the use of basic and derived factors, the second design problem (task switching) introduces factors that describe transitions between two trials, and the third design problem (2-Back task) requires solving sequential dependencies between more than two trials. We chose to focus on experiments that are commonly used to study cognitive control (i.e., the ability to flexibly pursue goal-directed behavior in the face of distraction). However these experiments share similarities with experiments in other domains of psychology, as well as other empirical disciplines, such as neuroscience and machine learning. Before walking through each example, we describe the installation of SweetPea. We then focus on each experiment, by first describing the experimental design problem, then showing the code for expressing and solving the problem in SweetPea, and finally walking the reader line-by-line through the example code. In addition, each example includes a solution for the design problem that was identified by SweetPea. Due to their medium complexity, the experiments described in this section are not amenable to uniform sampling, so we sampled experiment solutions non-uniformly from the space of possible solutions. The reader may find examples that implement uniform sampling, as well as other examples—both proof of concept and for realistic experimental designs—online (https://sweetpea-org.github.io/). The materials for all walkthrough examples described in this section are available at https://osf.io/b4nsy/.

Installation of SweetPea

The current version of SweetPea requires Python 3.7 or later and can be installed in a local Python environment using

```
pip install sweetpea
```

Designing a Stroop Task

In previous sections we described the Stroop task (Stroop, 1935) in which participants are typically asked to name the color in which a color word is displayed (e.g., say “green” to the word RED displayed in green). Here, we consider the Stroop color naming experiment with the two regular factors described above: the color factor representing the color in which the stimulus is displayed, and having four levels: red, green, blue, brown; and the word factor representing the word itself, also having four levels corresponding to each of the colors (Fig. 2a). On a given trial, one level from each factor is used to generate the stimulus, and the participant is required to respond to the color factor, indicating the color in which the word was displayed. For instance, they may be required to press the left arrow key if the word was displayed in red, the right arrow key if it was in green, the up arrow key if it was in blue, the down arrow if it was in brown. This results in a response factor with four levels: left, right, up and down.

In the general case, an experimenter may want to ensure that each color is paired with each word. This could be achieved by crossing all colors with all words. The full crossing of colors and words includes conditions in which the color and the word are the same (congruent; e.g., the word RED displayed in red), as well as conditions in which they are different (incongruent; e.g., the written word GREEN displayed in red). Together, the two conditions define the congruency of a trial. In this particular experiment, we wish to pair each color with each word, subject to the constraint that only incongruent trials are included. Finally, we wish to generate a minimum of 20 experiment trials. For the ease of the reader, we interleave each chunk of code with an explanation of what it does for the first Walkthrough Example. The full code implementing the specified design is shown in Listing 1.

We begin with importing the SweetPea modules:

```
from sweetpea.primitives import factor, derived_level, within_trial
from sweetpea.constraints import minimum_trials, exclude
from sweetpea import fully_cross_block, synthesize_trials_non_uniform, \
    print_experiments, tabulate_experiments, experiment_to_csv
```

Modules in line 1 are needed to specify regular and derived factors. Modules in line 2 are required to implement design constraints, such as specification of the minimum number of trials and the exclusion of factor levels. The imported modules in lines 3 and 4 serve to generate, sample, print, tabulate and export the final experiment.

We continue with defining the two basic factors `color` and `word`,

```
6  # color and word factors
7
8  color      = factor("color",  ["red", "green", "blue", "brown"])
9  word       = factor("word", ["red", "green", "blue", "brown"])
```

as well as the response factor:

```
11 # response factor
12
13 def is_response_left(color):
14     return color == "red"
15 def is_response_right(color):
16     return color == "green"
17 def is_response_up(color):
18     return color == "blue"
19 def is_response_down(color):
20     return color == "brown"
21
22 response = factor("response", [
23     derived_level("left", within_trial(is_response_left, [color])),
24     derived_level("right", within_trial(is_response_right, [color])),
25     derived_level("up", within_trial(is_response_up, [color])),
26     derived_level("down", within_trial(is_response_down, [color]))
27 ])
```

Levels of the `response` factor are dependent on the factor `color`. Each of its derived levels is defined by a predicate that takes as input the `color` factor describing the current stimulus (lines 13–20). For instance, the following function is used to define a leftward response:

```
def is_response_left(color):
    return color == "red"
```

The function returns true if the factor `color` on a given trial evaluates to the level `"red"`. The function is used to specify the derived level `"left"` in the definition of the factor `response` (lines 22–27):

```
derived_level("left", within_trial
    (is_response_left, [color])),
```

The first argument specifies the level name `"left"`. The second argument is the function `within_trial` described in Section *Defining Factors*. It passes the `color` factor of the current trial to the `is_response_left` predicate, to determine whether the factor `response` evaluates to the level `"left"`. The `congruency` factor is defined in a similar fashion:

```
29 # congruency factor
30
31 def is_congruent(color, word):
32     return color == word
33
34 def is_incongruent(color, word):
35     return not is_congruent(color, word)
36
37 congruent = derived_level("congruent", within_trial(is_congruent, [color, word]))
38 incongruent = derived_level("incongruent", within_trial(is_incongruent, [color, word]))
39
40 congruency = factor("congruency", [
41     congruent,
42     incongruent
43 ])
```

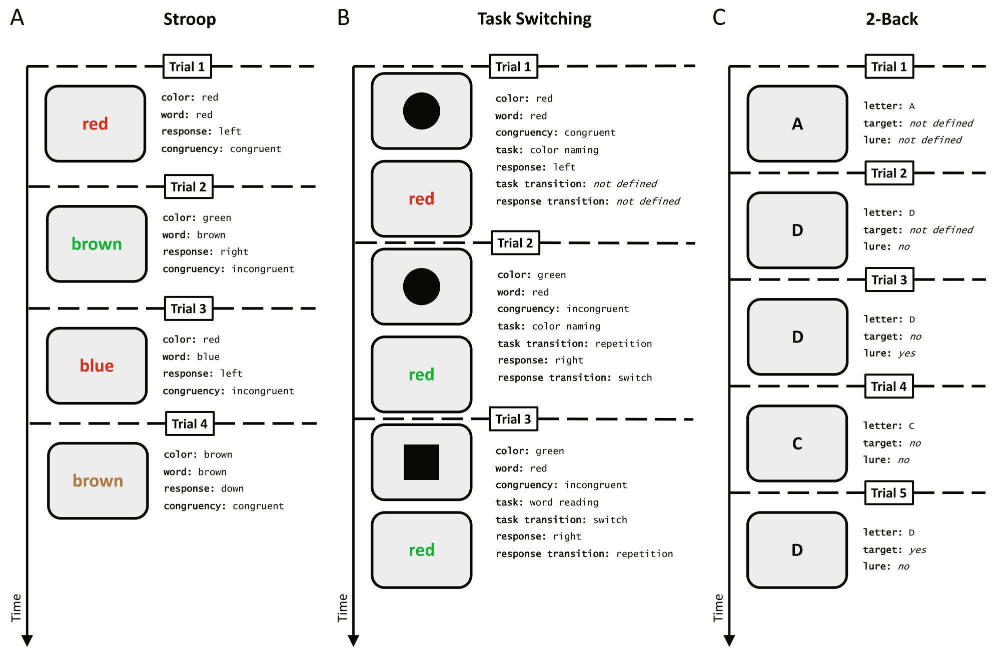

Fig. 2 Experimental designs used in examples. Each experiment can be described as a sequence of trials. In each of the three examples, a trial consists of one or two displays of a stimulus that appear in sequence. The figure shows experiment factors (in bold) with their respective levels (regular font) describing each trial. The reader may refer to the text for a detailed description of each experiment. **a** Stroop color naming experiment (Stroop, 1935). Participants are asked to indicate the color in which a color word is displayed, in this case with a button press. Relevant experiment factors include the color of the word, the word itself, the correct response button associated with the color, as well as stimulus congruency. A stimulus is considered congruent if the color and word are the same, and incongruent otherwise. **b** Cued task switching experiment (Meiran, 1996; Sudevan & Taylor, 1987). The same stimuli are used as in the Stroop experiment, however in this case participants are instructed to press a button corresponding either to the color of the stimulus (color naming) or to the word (word reading). Each trial begins with the display of a shape cue that signals which task to perform. A circle indicates that the color naming task should be performed for the ensuing stimulus, and a square indicates that the word reading task should be performed on that trial. Each trial (except the first) can be characterized as a transition between tasks relative to the previous trial: Participants may either repeat the same task or switch to a different task. The same applies to transitions between responses from one trial to the next. **c** 2-Back task (Cohen et al., 1997). Participants are presented with a sequence of letters, and instructed to press a button if the letter on the current trial matches the letter presented two trials back. The letter on the current trial is designated as a "target" if it matches in this way, and is considered to be a "lure" if it (a) is not a target and (b) matches the letter on the previous trial

Both levels of the congruency factor are dependent on the color and word factors of the current trial. Thus, the predicates specifying these levels take the color and the word factors as arguments (lines 31–35). For instance, a trial is considered congruent if the color and word of the current trial match:

```
def is_congruent(color, word):
    return color == word
```

Note that the definition of the corresponding derived level must include within_trial, with both color and word as arguments (lines 37–38):

```
derived_level("congruent", within_trial
    (is_congruent, [color, word]))
```

The experiment code specifies two constraints:

```
45 # constraints
46
47 trial_constraint = minimum_trials(20)
48 exclusion_constraint = exclude(congruency, congruent)
49
50 constraints = [trial_constraint, exclusion_constraint]
```

First, it specifies the minimum number of trials (line 47). The minimum_trials constraint ensures that the experiment sequence includes at least 20 trials. Note that a full crossing of all valid trials (without 4 possible congruent trials) requires a multiple of $4 \times 4 - 4 = 12$ trials. Thus, minimum_trials will not satisfy a full crossing unless

Listing 1 Example of a Stroop task

```python
from sweetpea.primitives import factor, derived_level, within_trial
from sweetpea.constraints import minimum_trials, exclude
from sweetpea import fully_cross_block, synthesize_trials_non_uniform, \
    print_experiments, tabulate_experiments, experiment_to_csv

# color and word factors

color      = factor("color",  ["red", "green", "blue", "brown"])
word       = factor("word", ["red", "green", "blue", "brown"])

# response factor

def is_response_left(color):
    return color == "red"
def is_response_right(color):
    return color == "green"
def is_response_up(color):
    return color == "blue"
def is_response_down(color):
    return color == "brown"

response = factor("response", [
    derived_level("left", within_trial(is_response_left, [color])),
    derived_level("right", within_trial(is_response_right, [color])),
    derived_level("up", within_trial(is_response_up, [color])),
    derived_level("down", within_trial(is_response_down, [color]))
])

# congruency factor

def is_congruent(color, word):
    return color == word

def is_incongruent(color, word):
    return not is_congruent(color, word)

congruent = derived_level("congruent", within_trial(is_congruent, [color, word]))
incongruent = derived_level("incongruent", within_trial(is_incongruent, [color, word]))

congruency = factor("congruency", [
    congruent,
    incongruent
])

# constraints

trial_constraint = minimum_trials(20)
exclusion_constraint = exclude(congruency, congruent)

constraints = [trial_constraint, exclusion_constraint]

# experiment

design      = [color, word, response, congruency]
crossing    = [color, word]
```

Listing 1 (continued)

```
56 block        = fully_cross_block(design, crossing, constraints,
57                                 require_complete_crossing=False)
58
59 experiments  = synthesize_trials_non_uniform(block, 1)
60
61 print_experiments(block, experiments)
62
63 tabulate_experiments(experiments, crossing)
64
65 experiment_to_csv(experiments, file_prefix="experiment")
```

the specified minimum number of trials is a multiple of 12. To mitigate this issue, SweetPea successively samples trials without replacement from counterbalanced blocks. In this example, the first 12 trials are sampled from a block of 12 counterbalanced trials, and the remaining 8 trials are sampled without replacement from another counterbalanced block.[5] The second constraint defines an exclusion criterion according to which the level `congruent` of the factor `congruency` is excluded from the experiment (line 48). All factors to be included in the design are listed in line 50. We continue with specifying the full experiment:

```
52 # experiment
53
54 design       = [color, word, response, congruency]
55 crossing     = [color, word]
56 block        = fully_cross_block(design, crossing, constraints,
57                                 require_complete_crossing=False)
```

The entire experimental design is defined by the factors `color`, `word`, `response` and `congruency` (line 54). The crossing between all colors and all words is specified in line 55. The design, crossing and constraints are used to define a fully crossed experiment block (subject to said constraints; lines 56–57). However, a complete crossing between all colors and words is not possible because we want to exclude all congruent trials for which the color and the word match. Thus, `fully_cross_block` would return no solution to the experimental design unless we allow the crossing to be incomplete, by setting `require_complete_crossing` to `False`. We can now generate, print, tabulate and save a desired sequence of trials:

```
59 experiments  = synthesize_trials_non_uniform(block, 1)
60
61 print_experiments(block, experiments)
62
63 tabulate_experiments(experiments, crossing)
64
65 experiment_to_csv(experiments, file_prefix="experiment")
```

Line 59 specifies how the experiment should be generated:

```
experiments  = synthesize_trials_non_uniform
   (block, 1)
```

In this case, we sample the experiment block only once, thus the argument `1`. The function `synthesize_trials_non_uniform` solves for experiment sequences without guaranteeing that they are sampled uniformly from the space of all possible solutions (see Section *Solving Experimental Designs*). We print the experiment in line 61, yielding the following output (only first six lines are shown; see Table 1 for the full output):

```
1 trial sequences found.
Experiment 0:
color brown | word red   | response down |
congruency incongruent color green | word
red   | response right | congruency
incongruent color brown | word blue |
response down | congruency incongruent
color red   | word blue | response left |
congruency incongruent
```

To check the frequency of each factor combination, we tabulate the generated experiment sequence for the factors specified in the crossing (line 63). The generated table lists the frequency and proportion of each factor level

[5] The current version of SweetPea only supports this counterbalancing scheme for experimental designs without transition factors and transition constraints. The counterbalancing described in the text is done independently for each experimental design. Future versions of SweetPea will coordinate this counterbalancing scheme across multiple experiment sequences, e.g., to allow for the counterbalancing of remaining trials across participants (Latin square design).

Table 1 Example solution to a Stroop color naming design

Trial	color	word	response	congruency
1	brown	red	down	incongruent
2	green	red	right	incongruent
3	brown	blue	down	incongruent
4	red	blue	left	incongruent
5	brown	green	down	incongruent
6	blue	red	up	incongruent
7	red	green	left	incongruent
8	green	brown	right	incongruent
9	red	brown	left	incongruent
10	green	blue	right	incongruent
11	blue	green	up	incongruent
12	blue	brown	up	incongruent
13	blue	brown	up	incongruent
14	green	red	right	incongruent
15	brown	blue	down	incongruent
16	red	brown	left	incongruent
17	brown	green	down	incongruent
18	blue	red	up	incongruent
19	red	green	left	incongruent
20	green	brown	right	incongruent

combination (only first six lines of the output are shown; see Table 2 for the full output):

```
Experiment 0:
color red   | word red   | frequency 0 |
 proportion 0.0 %
color red   | word green | frequency 2 |
 proportion 10.0%
color red   | word blue  | frequency 1 |
 proportion 5.0%
color red   | word brown | frequency 2 |
 proportion 10.0%
color green | word red   | frequency 2 |
 proportion 10.0%
```

Finally, we export the generated experiment sequence to a CSV file named “experiment_0.csv”[6] into the local folder (line 65).

Designing a task switching experiment

In many experiments, the sequence in which trials are presented is an important part of the design. For example,

Table 2 Frequencies and proportions of factor level combinations in an example solution to the Stroop experiment (cf. Table 1)

color	word	frequency	proportion
red	red	0	0.0%
red	green	2	10.0%
red	blue	1	5.0%
red	brown	2	10.0%
green	red	2	10.0%
green	green	0	0.0%
green	blue	1	5.0%
green	brown	2	10.0%
blue	red	2	10.0%
blue	green	1	5.0%
blue	blue	0	0.0%
blue	brown	2	10.0%
brown	red	1	5.0%
brown	green	2	10.0%
brown	blue	2	10.0%
brown	brown	0	0.0%

in task switching paradigms—used to study the flexibility with which people can adapt their behavior—the transition between trial types is an important factor. One example of this is the cued task switching paradigm, in which participants receive a task cue on every trial that instructs them which of two (or more) tasks they should perform on the current trial (Fig. 2b). The cue may instruct them either to repeat the task from the previous trial (task repetition) or to switch to a different task (task switch). A common measure in such designs is the cost associated with task switches—that is, reaction time or accuracy on switch relative to repetition trials. Thus, the design must consider task transition as a factor.

Here, we consider a task switching paradigm that builds on the Stroop experiment described in the previous example. In this experiment, participants are presented with a sequence of Stroop stimuli (i.e., color words displayed in a particular color). For simplicity, we consider only two colors (red and green), resulting in two levels for the color factor and two levels for the word factor. The task factor indicates which of the two tasks the participant is instructed to perform on a given trial, with two levels: color naming (respond to the color in which the word is displayed) and word reading (respond to the word). Thus, the correct response on every trial depends on both the stimulus and the relevant task. As in the previous example, we assume that the participant responds by pressing one of two buttons, but the stimulus-response mapping is the opposite for the two tasks: for color naming, the left button should be pressed for the color red and the right button for the color green; conversely, for word reading, the right button should be

[6] Both the `print_experiments` and the `experiment_to_csv` functions require a list of experiments. Here, we want to generate just one experiment, so both functions will output one experiment sequence.

pressed for the word “red” and the left button for the word “green”. Finally, there is a task transition factor, with two levels: repeat (if the instructed task for the current trial is the same as the last trial) and switch (if it is different).

In addition to the factors described above, it may also be important to include a response transition factor, that determines whether the response required on the current trial (i.e., whether the left or right button is the correct response) is the same or different as the one required on the previous trial. For example, this response transition factor has been shown to interact with performance costs associated with task switches (Kiesel et al., 2010; Rogers & Monsell, 1995). Thus, to control for this, it may be important to ensure that the same number of response transitions occurs in the task switch and task repetition conditions. This can be done by specifying a full crossing of the task transition, response transition, color, word and task factors.

Fully crossing the factors described above will ensure a balanced sampling of all combinations of their levels, including types of task and response transitions, over the course of the experiment. However, when selecting trials to execute, it does not necessarily preclude the possibility of undesired local sequences, or “runs”—that is, sequences of trials in which some level of a factor (e.g., whether the current trial is a task repetition or response repetition) remains the same, or alternates in some seemingly predictable but undesired way—for several trials in a row. This could invite misleading expectations from the participant, or otherwise impact performance in undesired ways. While these could also be specified as higher level factors, it can be more convenient to control for this by imposing constraints on the number times a particular level of a given factor, or a particular crossing of two or more factors is allowed to occur in sequence. As an example, here we limit task transitions to four of the same type, and the number of consecutive response repetitions or response switches to four. The code shown in Listing 2 implements this design.

Lines 1–4 import the SweetPea modules needed to specify the experiment as described for the previous example. We also include transition (line 1) to define factor levels based on transitions between trials.

Lines 8–10 define the three regular factors, color, word and task. Lines 14–25 define the derived factor congruency, as in the previous example (cf. Listing 1, lines 31–43). However, for conciseness, here the derived_level function is directly passed to factor.

Lines 29–41 define the derived factor response. The levels of this factor are defined by the two functions is_response_left and is_response_right. Note that each of the functions depends on the three factors color, word and task. For instance, the is_response_left function

```
def is_response_left(color, word, task):
    return (task == "color naming" and color
    == "red") or \
           (task == "word reading" and word ==
     "green")
```

implements the rule that the left response button should be pressed if the task is "color naming" and the color is "red", or if the task is "word reading" and the word is "green".

The task transition factor is defined in lines 45-56. The levels of this factor are dependent on the factor task in the previous trial and the current trial. The predicate is_task_repetition expresses this between-trial dependency to define the task repetition level:

```
def is_task_repetition(task):
    return task[0] == task[1]
```

In this example, the factor task is passed as an ordered list with two elements. The first and second elements of the list encode the task of the previous trial (task[0]) and the current trial (task[1]), respectively. If the two are the same, the current trial is considered a task repetition, and a task switch otherwise. Note that the derived levels of task_transition are now defined using the transition function. For instance,

```
derived_level("repetition",
                transition(is_task_repetition,
      [task])),
```

defines the level "repetition", and uses the transition function to pass the factor task as a list (encoding the task on the current and previous trial) to the predicate is_task_repetition. Lines 60–71 define response_transition in an analogous manner.

Lines 75-77 define constraints for the experiment. The function no_more_than_k_in_a_row in line 76 implements the sequential constraint that each level of the factor task_transition cannot occur more than four times in a row; the same is declared for the factor response_transition in line 77.

Analogous to Listing 1, the design, the crossing, as well as the constraints are integrated into an experiment (lines 81-84) that is first sampled non-uniformly (line 86), and then printed (line 88).

The output shown in Table 3 illustrates one solution to the specified design. Note that the factors task_transition and response_transition are not defined for the first trial of the experiment, simply because there exists no preceding trial with the factors task and response, respectively. Full counterbalancing of five factors, each of which has two levels, requires at least $2^5 = 32$ trials.

Listing 2 Example of a task switching experiment

```
1  from sweetpea.primitives import factor, derived_level, within_trial, transition
2  from sweetpea.constraints import no_more_than_k_in_a_row
3  from sweetpea import fully_cross_block, synthesize_trials_non_uniform, \
4      print_experiments, tabulate_experiments
5
6  # color, word and task factors
7
8  color      = factor("color",  ["red", "green"])
9  word       = factor("word", ["red", "green"])
10 task       = factor("task", ["color naming", "word reading"])
11
12 # congruency factor
13
14 def is_congruent(color, word):
15     return color == word
16
17 def is_incongruent(color, word):
18     return not is_congruent(color, word)
19
20 congruency = factor("congruency", [
21     derived_level("congruent",
22                  within_trial(is_congruent, [color, word])),
23     derived_level("incongruent",
24                  within_trial(is_incongruent, [color, word]))
25 ])
26
27 # response factor
28
29 def is_response_left(color, word, task):
30     return (task == "color naming" and color == "red") or \
31            (task == "word reading" and word == "green")
32 def is_response_right(color, word, task):
33     return (task == "color naming" and color == "green") or \
34            (task == "word reading" and word == "red")
35
36 response = factor("response", [
37     derived_level("left",
38                  within_trial(is_response_left, [color, word, task])),
39     derived_level("right",
40                  within_trial(is_response_right, [color, word, task]))
41 ])
42
43 # task transition factor
44
```

SweetPea adds an additional filler trial at the beginning of the experiment sequence to accommodate the circumstance that transition factors cannot be defined for the first trial of an experiment. We can check the generated sequence by breaking down the frequencies of every factor level combination in the crossing (lines 90–91; Table 4). Note that we instruct the `tabulate_experiments` function to only consider all trials from the second trial (indexed as `1`) to the last trial (denoted as `33`) since we can ignore the first filler trial.

Designing a 2-Back Task

Some experiments may involve factors that are dependent on more than two consecutive trials. The function `window` allows the user to define such factors in SweetPea. Here, we illustrate its functionality in the design of an N-Back task—a psychological task commonly used to assess working memory performance, e.g., how well participants can update and maintain task-relevant information over time. In the N-Back task considered here, participants are presented with

Listing 2 (continued)

```python
def is_task_repetition(task):
    return task[0] == task[1]

def is_task_switch(task):
    return not is_task_repetition(task)

task_transition = factor("task_transition", [
    derived_level("repetition",
                 transition(is_task_repetition, [task])),
    derived_level("switch",
                 transition(is_task_switch, [task]))
])

# response transition factor

def is_response_repeat(response):
    return response[0] == response[1]

def is_response_switch(response):
    return not is_response_repeat(response)

response_transition = factor("response_transition", [
    derived_level("repetition",
                 transition(is_response_repeat, [response])),
    derived_level("switch",
                 transition(is_response_switch, [response]))
])

# constraints

constraints = [
    no_more_than_k_in_a_row(4, task_transition),
    no_more_than_k_in_a_row(4, response_transition)]

# experiment

design       = [color, word, task, congruency, response,
                task_transition, response_transition]
crossing     = [color, word, task, task_transition, response_transition]
block        = fully_cross_block(design, crossing, constraints)

experiments  = synthesize_trials_non_uniform(block, 1)

print_experiments(block, experiments)

tabulate_experiments(experiments, crossing,
                     trials=list(range(1, 33)))
```

a sequence of letters, one letter per trial (Cohen et al., 1997). Participants are instructed to press a button if the letter on the current trial matches the letter some number N trials back. For simplicity, we consider a 2-Back task in which participants should press a button if the current letter was presented $N = 2$ trials back (Fig. 2c). The letter factor describes which of the following letters is presented on the current trial: “A”, “B”, “C”, “D”, “E”, “F”. The target factor determines whether the letter on the current trial matches the letter two trials back, in which case the participant has to press the button. Finally, we consider a trial to be a lure if (a) it isn’t a target trial but (b) the letter on the current trial matches the letter one trial back. Lure trails are important because they can help determine

Table 3 Example solution to a task switching design

Trial	color	word	task	congruency	response	task_transition	response_transition
1	red	green	color naming	incongruent	left		
2	red	red	word reading	congruent	right	switch	switch
3	green	green	color naming	congruent	right	switch	repetition
4	green	red	color naming	incongruent	right	repetition	repetition
5	green	green	word reading	congruent	left	switch	switch
6	green	green	word reading	congruent	left	repetition	repetition
7	red	green	color naming	incongruent	left	switch	repetition
8	red	red	color naming	congruent	left	repetition	repetition
9	green	red	color naming	incongruent	right	repetition	switch
10	red	red	word reading	congruent	right	switch	repetition
11	red	green	word reading	incongruent	left	repetition	switch
12	green	red	word reading	incongruent	right	repetition	switch
13	green	green	word reading	congruent	left	repetition	switch
14	red	red	color naming	congruent	left	switch	repetition
15	red	green	word reading	incongruent	left	switch	repetition
16	green	red	color naming	incongruent	right	switch	switch
17	red	red	color naming	congruent	left	repetition	switch
18	green	red	word reading	incongruent	right	switch	switch
19	green	red	color naming	incongruent	right	switch	repetition
20	green	red	word reading	incongruent	right	switch	repetition
21	green	red	word reading	incongruent	right	repetition	repetition
22	red	red	word reading	congruent	right	repetition	repetition
23	red	green	color naming	incongruent	left	switch	switch
24	red	green	color naming	incongruent	left	repetition	repetition
25	green	green	word reading	congruent	left	switch	repetition
26	red	green	word reading	incongruent	left	repetition	repetition
27	red	red	word reading	congruent	right	repetition	switch
28	red	red	color naming	congruent	left	switch	switch
29	green	green	color naming	congruent	right	repetition	switch
30	green	green	color naming	congruent	right	repetition	repetition
31	red	green	word reading	incongruent	left	switch	switch
32	green	green	color naming	congruent	right	switch	switch
33	red	green	color naming	incongruent	left	repetition	switch

whether, when participants make mistakes, it is because they are having trouble remembering the letters themselves (e.g., a problem with maintenance) or the order in which they were presented (e.g., a problem with updating). It is also possible that participants may be better at responding to some letters than others. Measures of working memory performance could be biased if those letters occur more often in the experiment. This bias can be avoided by ensuring that each letter is a target and a non-target for equal numbers of trials. Previous studies address this issue by sampling letters randomly. However, as discussed above, random sampling of individual trials is only reliable if the sample size is large. Experiments with a small number of trials may risk confounding the target factor with the letter factor. SweetPea can be used to address this problem directly, by balancing the letter factor with the derived target factor. Listing 3 shows SweetPea code that generates such an experiment sequence.

Lines 1–3 import relevant SweetPea modules, as described in the previous two examples. Note, however, that we also include `window` (line 1) to derive factor levels from sequences of more than two trials.

Lines 7–9 define the regular factor `letter` with its six levels.

The `target` factor is defined in lines 13-21. The predicates implementing each level of the factor `target`

Table 4 Frequencies and proportions of factor level combinations in an example solution to the task switching design (cf. Table 3)

`color`	`word`	`task`	`task_transition`	`response_transition`	`frequency`	`proportion`
red	red	color naming	repetition	repetition	1	3.125%
red	red	color naming	repetition	switch	1	3.125%
red	red	color naming	switch	repetition	1	3.125%
red	red	color naming	switch	switch	1	3.125%
red	red	word reading	repetition	repetition	1	3.125%
red	red	word reading	repetition	switch	1	3.125%
red	red	word reading	switch	repetition	1	3.125%
red	red	word reading	switch	switch	1	3.125%
red	green	color naming	repetition	repetition	1	3.125%
red	green	color naming	repetition	switch	1	3.125%
red	green	color naming	switch	repetition	1	3.125%
red	green	color naming	switch	switch	1	3.125%
red	green	word reading	repetition	repetition	1	3.125%
red	green	word reading	repetition	switch	1	3.125%
red	green	word reading	switch	repetition	1	3.125%
red	green	word reading	switch	switch	1	3.125%
green	red	color naming	repetition	repetition	1	3.125%
green	red	color naming	repetition	switch	1	3.125%
green	red	color naming	switch	repetition	1	3.125%
green	red	color naming	switch	switch	1	3.125%
green	red	word reading	repetition	repetition	1	3.125%
green	red	word reading	repetition	switch	1	3.125%
green	red	word reading	switch	repetition	1	3.125%
green	red	word reading	switch	switch	1	3.125%
green	green	color naming	repetition	repetition	1	3.125%
green	green	color naming	repetition	switch	1	3.125%
green	green	color naming	switch	repetition	1	3.125%
green	green	color naming	switch	switch	1	3.125%
green	green	word reading	repetition	repetition	1	3.125%
green	green	word reading	repetition	switch	1	3.125%
green	green	word reading	switch	repetition	1	3.125%
green	green	word reading	switch	switch	1	3.125%

receive a list as input, similar to levels for transition factors described in the previous example. For instance,

```
def is_target(letter):
    return letter[0] == letter[2]
```

expects a list labeled `letter`. Each element of the list refers to a different trial within a specified window of trials relative to the current trial. In the function `is_target`, the argument coding for the factor `letter` is treated as a window of size 3. The last element (`letter[2]`) refers to the letter on the current trial and the first element (`letter[0]`) refers to the letter two trials back. A trial is considered a target if the letter on the current trial matches the letter two trials back. The window is specified in the declaration of the derived level `"yes"` for the `target` factor (line 19):

```
derived_level("yes", window(is_target,
    [letter], 3, 1)),
```

`window` is given the predicate `is_target` that returns true if the sequence satisfies requirements for this level. It passes the list of factors `[letter]` to the function `is_target`. The last two arguments of `window` define the window size—the number of past trials to consider, including the current trial—as well as the stride. The stride determines the number of trials to skip between the trials that are considered when selecting the new, derived level. For instance, if we were to determine the presence of a target

Listing 3 Example of a 2-Back experiment

```
 1 from sweetpea.primitives import factor, derived_level, window
 2 from sweetpea import fully_cross_block, synthesize_trials_non_uniform, \
 3     print_experiments, tabulate_experiments
 4 
 5 # letter factor
 6 
 7 all_letters = ["A", "B", "C", "D", "E", "F"]
 8 
 9 letter = factor("letter",  all_letters)
10 
11 # target factor
12 
13 def is_target(letter):
14     return letter[0] == letter[2]
15 def is_no_target(letter):
16     return not is_target(letter)
17 
18 target = factor("target", [
19     derived_level("yes", window(is_target, [letter], 3, 1)),
20     derived_level("no",  window(is_no_target, [letter], 3, 1))
21 ])
22 
23 # lure factor
24 
25 def is_lure(letter, target):
26     return target[1] == "no" and letter[0] == letter[1]
27 def is_no_lure(letter, target):
28     return not is_lure(letter, target)
29 
30 lure = factor("lure", [
31     derived_level("yes", window(is_lure, [letter, target], 2, 1)),
32     derived_level("no",  window(is_no_lure, [letter, target], 2, 1))
33 ])
34 
35 # experiment
36 
37 design       = [letter, target, lure]
38 crossing     = [letter, target]
39 block        = fully_cross_block(design, crossing, [])
40 
41 experiments  = synthesize_trials_non_uniform(block, 1)
42 
43 print_experiments(block, experiments)
44 
45 tabulate_experiments(experiments, crossing,
46                      trials=list(range(2, 14)))
```

on every other trial (instead of every trial), we may specify a stride of 2 (instead of a stride of 1).

Lines 25–33 define the `lure` factor in an analogous manner, but using a window size of 2. Note that a trial is considered a lure if it is not a target and if the letter on the previous trial is the same as the letter on the current trial. The `is_lure` predicate, used to determine whether a trial is a lure, takes two arguments, `letter` and `target`,

```
def is_lure(letter, target):
    return target[1] == "no" and letter[0] ==
    letter[1]
```

with both factors passed as a window of size two. Thus, target[1] and letter[1] refer to the target and letter factors, respectively, on the current trial and letter[0] refers to the letter factor on the previous trial. The window size of two trials for the predicate is_lure is specified in line 31:

```
derived_level("yes", window(is_lure, [letter,
    target], 2, 1))
```

Finally, line 37 lists the factors to consider in the design and line 38 defines the crossing between letter and target. The design and crossing are embedded without constraints in an experiment block (line 39). Line 41 generates an experiment sequence from the block with non-uniform sampling. Line 43 displays the output. Table 5 shows an example output. Note that the target is not defined for the first two trials and the lure is not defined for the first trial due to a window size of two and one, respectively.

Finally, we validate the generated sequence by breaking down the frequencies of every combination of factor levels in the crossing (lines 45–46; Table 6). Here, we instruct the tabulate_experiments function to only consider trials from the third trial (indexed as 2) to the last trial (indexed as 14) since we can ignore the first two filler trials.

Discussion

The SweetPea language is intended to provide a format for describing experimental designs in a declarative form that is: (a) as concise and natural as possible; (b) sufficiently precise as to allow the application of standard computational algorithms for sampling and sample analysis; and (c) sufficiently general as to be useful for the widest possible range of experimental applications. Fully achieving all of these goals is of course a considerable challenge. Here, we present a first implementation, that we hope takes a meaningful step in this direction, by providing at least one concrete formulation of how to approach these challenges, that is already useful for some applications, and that provides a foundation and benchmark for future development.

As noted above, there are a number of existing software packages that support experimental design. Table 7 compares SweetPea to some of these packages, including E-Prime (Schneider et al., 2002), PsychoPy (Peirce, 2009), OpenSesame (Mathôt et al., 2012; Mathôt, 2016), Mix (van Casteren & Davis, 2006) and GAMixit (Ihrke & Behrendt, 2011). SweetPea differs from these packages in several ways. First, SweetPea is a declarative language for factorial design, allowing the user to specify the logic of the entire experimental design (such as the set of factors to be counterbalanced or the constraints to be imposed on the experimental sequence) without having to specify its implementation. Other packages, such as Mix (van Casteren & Davis, 2006), require the user to implement parts of the factorial design by themselves (e.g., the user has to provide Mix with a full list of possible factor-level combinations to be included in the design) and are bound to a single sampling method (e.g., rejection sampling with a specific repair algorithm). Second, SweetPea introduces a novel approach to solving for experimental sequences, by formulating experimental design as a Boolean satisfiability problem. This makes the space of solutions amenable to uniform sampling, e.g., based on Unigen. The uniform sampling of experiment sequences cannot be guaranteed if experimental designs are realized via rejection sampling with subsequent repair (Mathôt, 2016; van Casteren & Davis, 2006) or via genetic algorithms (Ihrke & Behrendt, 2011;

Table 5 Example solution to a 2-Back task design

Trial	letter	target	lure
1	A		
2	B		no
3	B	no	yes
4	D	no	no
5	B	yes	no
6	D	yes	no
7	F	no	no
8	C	no	no
9	F	yes	no
10	C	yes	no
11	A	no	no
12	E	no	no
13	A	yes	no
14	E	yes	no

Table 6 Frequencies and proportions of factor level combinations in an example solution to the 2-Back task design (cf. Table 5)

letter	target	frequency	proportion
A	yes	1	8.333%
A	no	1	8.333%
B	yes	1	8.333%
B	no	1	8.333%
C	yes	1	8.333%
C	no	1	8.333%
D	yes	1	8.333%
D	no	1	8.333%
E	yes	1	8.333%
E	no	1	8.333%
F	yes	1	8.333%
F	no	1	8.333%

Table 7 Comparison of software solutions for experimental design

	Packages for experimental design & execution			Packages tailored to experimental design		
	E-Prime	PsychoPy	OpenSesame	Mix	GAMixit	SweetPea
Implementation						
Open Source	×	✓	✓	×	✓	✓
Language Interface	×	Python	Python	×	×	Python
GUI	✓	✓	✓	✓	✓	×
Counterbalancing						
Regular Factors	✓	✓	✓	×	×	✓
Derived Factors	×	×	×	×	×	✓
Constraints						
Algorithm	N/A	N/A	Rejection Sampling with Repair	Rejection Sampling with Repair	Genetic Algorithm	Solver-aided Sampling
Global	×	×	✓	✓	✓	✓
Transition	×	×	✓	✓	✓	✓
Window	×	×	×	✓	×	✓
Uniform Sampling	×	×	×	×	×	*
Exp. Execution	✓	✓	✓	×	×	×

Implementations of software solutions vary based on whether they are open sourced, whether they are designed to interface with a programming language and whether they provide a GUI. The counterbalancing scheme may include regular factors and/or derived factors. Software packages may rely on specific algorithms to sample experiment sequences with respect to constraints. Such constraints include global constraints (e.g. minimum number of trials or maximum occurance of a given factor level), constraints on transitions (e.g. maximum number of factor level repetitions) or constraints on windows of trials (e.g. maximal occurance of certain patterns). Moreover, constrained experiment sequences may be sampled uniformly from the space of all possible solutions or not. Note that some of the listed software solutions are embedded in larger packages for stimulus presentation and data collection

✓ Functionality Supported × Functionality Not Supported * Functionality Supported in Future Release

Wager & Nichols, 2003), especially if complex sequential constraints need to be taken into account. Another distinct feature of SweetPea is that the user can derive novel factors from regular factors—based on custom rules—and can include these into the counterbalancing scheme. However, unlike some experimental design packages (e.g., GAMixit, OpenSesame or E-Prime), SweetPea does currently not come with a graphical user interface (GUI) and does not provide functionality for executing experiments. Instead, the user may leverage SweetPea to generate experimental sequences as part of a Python-based programming environment for experiment execution (e.g., using PsychoPy or OpenSesame) or import generated experiment sequences as CSV files into their preferred package for stimulus presentation, e.g., E-Prime (Schneider et al., 2002) or Psychtoolbox (Kleiner et al., 2007).

The experimental design challenges addressed by SweetPea are faced by researchers in other domains, such as neuroscience and machine learning. The measurement of neural correlates is often confounded by the order in which stimuli appear, especially if measurement techniques have a low temporal resolution (Aarabi, Osharina, & Wallois, 2017, Gorgolewski, Storkey, Bastin, Whittle, & Pernet, 2013). SweetPea addresses this problem, by allowing the user to counterbalance transition factors, and to impose sequential constraints on the generated sequence of trials. Another issue concerns proper sampling of training data for machine learning systems. Improper sampling of stimuli can bias statistical learning systems to misrepresent the space of all stimuli, leading to poor generalization performance (Zadrozny, 2004). Machine learning researchers can leverage SweetPea to specify the space of stimuli in terms of factors and levels, and use it to generate counterbalanced training data.

Despite the broad scope of experiments that can already be expressed in SweetPea, there are several ways in which the language can be improved. By making the code available in an open-source repository (https://github.com/sweetpea-org), we hope that the community can help contribute to that effort. Toward that end, below we discuss potentially valuable directions for further development.

Continuous factors SweetPea frames experimental design as a Boolean satisfiability problem, and thus requires that factors are composed of discrete levels. However, some experiments may require continuous factors, such as intertrial intervals or monetary rewards offered based on participant performance. At present, SweetPea can represent continuous factors by binning them into a limited set of discrete levels. However, a more efficient strategy would be to allow the user to sample these factors from pre-specified distributions (e.g., sampling the factor "intertrial interval" from an exponentially modified Gaussian distribution).

Uniform sampling of complex experimental designs The current version of SweetPea can provide a SAT-sampler (Unigen; Chakraborty et al., 2014) with a general logical formula describing the experiment suitable for use by automated sampling methods. Ideally, these would sample uniformly from the space of all possible solutions to the general logical formula provided by SweetPea. However, the current generation of SAT-samplers are not efficient enough to handle the scale of experimental designs that can be formulated using SweetPea. The uniform sampling of solutions to general logical formulas is still an area of active research in computer science. SweetPea stands to leverage progress made in this direction, by describing experimental designs in a form that can be used by SAT-samplers. In the future, this may make it possible to design larger experiments with the guarantee that all constraints are satisfied and solutions are uniformly sampled.

More constraints Constraints provide a convenient way of expressing desired conditions for experimental sequences. The user can currently specify a number of constraints in SweetPea, e.g., to exclude trials with certain factor levels, or to enforce a maximum number of factor level repetitions. In future releases, we seek to expand the functionality of SweetPea to include additional constraints, such as the requirement that numeric factor levels of adjacent trials must lie within a certain distance (e.g., that the monetary reward provided in a trial can only be a certain amount larger or smaller than the monetary reward on the previous trial), as implemented by Mix (van Casteren & Davis, 2006). However, it should be noted that many complex constraints are already accommodated in the SweetPea language. For instance, the numeric constraint may be enforced by constructing a derived factor, indicating whether the numeric distance in factor levels between two adjacent trials is smaller than a certain amount, and by excluding levels of this factor that violate this condition. Future releases of SweetPea will allow the user to express these requirements more naturally as single constraints.

Debugging experimental designs In some cases, a researcher may over-constrain an experimental design, such that there exists no possible experiment that satisfies all constraints. In these cases, SweetPea can inform the user that the experiment is overspecified. However, at present, SweetPea can only offer limited insight into which constraints are in conflict with one another. Therefore, a useful target for ongoing development of SweetPea would be tools that aid researchers in debugging their experimental design. This could involve iterative procedures that achieve partial satisfiability by dropping design constraints with low priority.

Automatic design specification Another useful avenue for development would be to automatically derive the minimal set of counterbalancing conditions that satisfies the statistical analysis specified by the researcher. A major goal of experimental design is to warrant proper statistical inference from the data generated by the experiment. As a consequence, counterbalancing schemes and sequential constraints are often determined by the statistical analysis that a researcher attempts to perform. For instance, a researcher who seeks to contrast two levels of an experiment factor with a Student's t-test, would want to counterbalance the two levels with respect to all levels of nuisance factors. This can be achieved by distributing the levels of nuisance factors uniformly across the levels of factors of interest. SweetPea could assist in generating such a counterbalancing scheme, but currently the specification itself would have to be performed by the researcher. More generally, a particularly promising direction for future work would the integration of a high-level interface for specifying an experimental design based on a desired statistical analysis (e.g., contrasts defined by a Student's t-test, ANOVA or linear mixed model).

We plan to continue work along these lines, and hope others will join us in this effort. Doing so offers the promise of providing a rigorous and standardized method for experimental design that would help avert many of the problems that have been identified in past and current experimental research.

Funding and Acknowledgements This project was made possible through the support of a grant from the John Templeton Foundation, as well as the National Science Foundation (NSF-1813123). The opinions expressed in this publication are those of the authors and do not necessarily reflect the views of the John Templeton Foundation. We thank Markus Spitzer for providing helpful feedback on the manuscript.

Availability of Data and Materials Data sharing is not applicable to this article as no datasets were generated or analysed during the current study. The materials for all walkthrough examples are available at https://osf.io/b4nsy/.

Code Availability We invite contributions to SweetPea's open-source repository at https://github.com/sweetpea-org/.

Declarations

Conflicts of Interest/Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this article.

Ethics Approval, Consent to Participate and Consent for Publication Ethics approval, consent to participate and consent for publication are not applicable to this article as no datasets were generated or analysed during the current study.

References

Aarabi, A., Osharina, V., & Wallois, F. (2017). Effect of confounding variables on hemodynamic response function estimation using averaging and deconvolution analysis: An event-related nirs study. *Neuroimage*, *155*, 25–49.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ..., Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th usenix conference on operating systems design and implementation* (pp. 265–283). USA: USENIX Association.

Allport, A., & Wylie, G. (1999). Task-switching: Positive and negative priming of task-set. In G. W. Humphreys, J. Duncan, & A. Treisman (Eds.) *Attention, space, and action: Studies in cognitive neuroscience*, (pp. 273–296): Oxford University Press.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, *13*(1), 365–376.

Button, K. S., Ioannidis, J. P., Mokrysz, C., Nosek, B. A., Flint, J., Robinson, E. S., & Munafò, M. R. (2013). Power failure: Why small sample size undermines the reliability of neuroscience. *Nature Reviews Neuroscience*, *14*(5), 365–376.

Chakraborty, S., Meel, K. S., & Vardi, M. Y. (2014). Balancing scalability and uniformity in sat witness generator. In *2014 51st acm/edac/ieee design automation conference (dac)* (pp. 1–6).

Chapelle, O., Vapnik, V., Bousquet, O., & Mukherjee, S. (2002). Choosing multiple parameters for support vector machines. *Machine Learning*, *46*(1–3), 131–159.

Cherkaev, A. (2019). SweetPea: A language for experimental design (Unpublished master's thesis). The University of Utah Salt Lake City.

Cohen, J. D., Perlstein, W. M., Braver, T. S., Nystrom, L. E., Noll, D. C., Jonides, J., & Smith, E. E. (1997). Temporal dynamics of brain activation during a working memory task. *Nature*, *386*(6625), 604–608.

Cooper, S., & Marí-Beffa, P. (2008). The role of response repetition in task switching. *Journal of Experimental Psychology: Human Perception and Performance*, *34*(5), 1198.

Dale, A. M. (1999). Optimal experimental design for event-related fmri. *Human Brain Mapping*, *8*(2–3), 109–114.

De Leeuw, J. R. (2015). jspsych: A javascript library for creating behavioral experiments in a web browser. *Behavior Research Methods*, *47*(1), 1–12.

Dijksterhuis, A., Van Knippenberg, A., & Holland, R. W. (2014). Evaluating behavior priming research: Three observations and a recommendation. *Social Cognition*, *32*(Supplement), 196–208.

Druery, C., & Bateson, W. (1901). Experiments in plant hybridization. *Journal of the Royal Horticultural Society*, *26*, 1–32.

Drummond, C. (2006). Machine learning as an experimental science (revisited). In *Aaai workshop on evaluation methods for machine learning* (pp. 1–5).

Gardner, M., Neumann, M., Grus, J., & Lourie, N. (2018). Writing Code for NLP Research. In *Proceedings of the 2018 conference on empirical methods in natural language processing: Tutorial abstracts*. Melbourne: Association for Computational Linguistics.

Gorgolewski, K. J., Storkey, A. J., Bastin, M. E., Whittle, I., & Pernet, C. (2013). Single subject fmri test-retest reliability metrics and confounding factors. *Neuroimage*, *69*, 231–243.

Gureckis, T. M., Martin, J., McDonnell, J., Rich, A. S., Markant, D., Coenen, A., ..., Chan, P. (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.

Ihrke, M., & Behrendt, J. (2011). Automatic generation of randomized trial sequences for priming experiments. *Frontiers in Psychology*, *2*, 225.

Jou, J. (2014). Task-switching cost and repetition priming: Two overlooked confounds in the fixed-set procedure of the sternberg paradigm and how they affect memory set-size effects. *Quarterly Journal of Experimental Psychology*, *67*(10), 1871–1894.

Kiesel, A., Steinhauser, M., Wendt, M., Falkenstein, M., Jost, K., Philipp, A. M., & Koch, I. (2010). Control and interference in task switching-a review. *Psychological Bulletin*, *136*(5), 849.

Klein, R. A., Ratliff, K. A., Vianello, M., Adams, J.r. R. B., Bahník, Š., Bernstein, M. J., ..., et al. (2014). Investigating variation in replicability. *Social Psychology*, *45*, 142–152.

Kleiner, M., Brainard, D., & Pelli, D. (2007). What's new in psychtoolbox-3?

Krause, F., & Lindemann, O. (2014). Expyriment: A python library for cognitive and neuroscientific experiments. *Behavior Research Methods*, *46*(2), 416–428.

Kühberger, A., Fritz, A., & Scherndl, T. (2014). Publication bias in psychology: A diagnosis based on the correlation between effect size and sample size. *PloS One*, *9*(9), e105825.

Langley, P. (1988). Machine learning as an experimental science. *Machine Learning*, *3*(1), 5–8.

Logan, G. D., & Schneider, D. W. (2010). Distinguishing reconfiguration and compound-cue retrieval in task switching. *Psychologica Belgica*, *50*(3), 413–433.

Mathôt, S. (2016). A package for pseudorandomization of datamatrix objects. https://github.com/open-cogsci/python-pseudorandom. GitHub.

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.

Mayr, U., & Keele, S. W. (2000). Changing internal constraints on action: The role of backward inhibition. *Journal of Experimental Psychology: General*, *129*(1), 4–26.

Meiran, N. (1996). Reconfiguration of processing mode prior to task performance. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *22*(6), 1423–1442.

Mendel, G. (1866). Versuche uber pflanzen-hybriden. *Verhandlungen des naturforschenden Vereins in Brunn fur*, *4*, 3–47.

Miłkowski, M., Hensel, W. M., & Hohol, M. (2018). Replicability or reproducibility? On the replication crisis in computational neuroscience and sharing only relevant detail. *Journal of Computational Neuroscience*, *45*(3), 163–172.

Miller, G. (2011). The mating mind: How sexual choice shaped the evolution of human nature. Anchor.

Myung, J. I., & Pitt, M. A. (2009). Optimal experimental design for model discrimination. *Psychological review*, *116*(3), 499.

Open Science Collaboration, et al. (2015). Estimating the reproducibility of psychological science. *Science*, *349*(6251), 1–10.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., …, et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems* (pp. 8026–8037).

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

Peirce, J. W. (2009). Generating stimuli for neuroscience using psychopy. *Frontiers in Neuroinformatics*, *2*, 10.

Peng, R. D. (2011). Reproducible research in computational science. *Science*, *334*(6060), 1226–1227.

Rogers, R. D., & Monsell, S. (1995). Costs of a predictible switch between simple cognitive tasks. *Journal of Experimental Psychology: General*, *124*(2), 207–231.

Rossi, J. S. (1990). Statistical power of psychological research: What have we gained in 20 years? *Journal of Consulting and Clinical Psychology*, *58*(5), 646.

Schneider, W., Eschman, A., & Zuccolotto, A. (2002). E-prime: User's guide. reference guide getting started guide. Psychology Software Tools, Incorporated.

Sherman, R., & Pashler, H. (2019). Powerful moderator variables in behavioral science? Don't bet on them (version 3). PsyArXiv preprint: https://psyarxiv.com/c65wm/

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. https://doi.org/10.3389/fpsyg.2016.00610

Stroebe, W., & Strack, F. (2014). The alleged crisis and the illusion of exact replication. *Perspectives on Psychological Science*, *9*(1), 59–71.

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, *18*(6), 643.

Sudevan, P., & Taylor, D. A. (1987). The cuing and priming of cognitive operations. *Journal of Experimental Psychology: Human Perception and Performance*, *13*(1), 89.

van Casteren, M., & Davis, M. H. (2006). Mix, a program for pseudorandomization. *Behavior Research Methods*, *38*(4), 584–589.

Wager, T. D., & Nichols, T. E. (2003). Optimization of experimental design in fmri: A general framework using a genetic algorithm. *Neuroimage*, *18*(2), 293–309.

Wells, G. L., & Windschitl, P. D. (1999). Stimulus sampling and social psychological experimentation. *Personality and Social Psychology Bulletin*, *25*(9), 1115–1125.

Zadrozny, B. (2004). Learning and evaluating classifiers under sample selection bias. In *Proceedings of the twenty-first international conference on machine learning* (pp. 114–121).

Zmigrod, S., & Hommel, B. (2013). Feature integration across multimodal perception and action: A review. *Multisensory Research*, *26*(1-2), 143–157.