ADOpy: A Python Package for Adaptive Design Optimization

Jaeyeong Yang[†], Mark A. Pitt[‡], Woo-Young Ahn[†*], and Jay I. Myung[‡*]

[†] Department of Psychology, Seoul National University, Seoul, Korea

[‡] Department of Psychology, Ohio State University, Columbus, Ohio

[*] Corresponding authors (wahn55@snu.ac.kr, myung.1@osu.edu)

Abstract

Experimental design is fundamental to research, but formal methods to identify good designs are lacking. Advances in Bayesian statistics and machine learning offer algorithm-based ways to identify good experimental designs. Adaptive design optimization (ADO; Cavagnaro, Myung, Pitt, & Kujala, 2010; Myung, Cavagnaro, & Pitt, 2013) is one such method. It works by maximizing the informativeness (and efficiency) of data collection, thereby improving inference. ADO is a general-purpose method for conducting adaptive experiments on the fly, and can lead to rapid accumulation of information about the phenomenon of interest with the fewest number of trials. The technical skills required to use ADO have been a barrier to its use. To increase its accessibility to a wide range of researchers, we introduce an open-source Python package, ADOpy, that implements ADO for optimizing experimental design. The package, available on GitHub, is written using high-level modular-based commands such that users do not have to understand the computational details of the ADO algorithm. In this paper, we first provide a tutorial introduction to ADOpy and ADO itself, and then illustrate its use in three walk-through examples: psychometric function estimation, delay discounting, and risky choice. Simulation data are also provided to demonstrate how ADO designs compare with other designs (random, staircase).

Keywords: computational modeling, optimal experimental design, Bayesian adaptive estimation, Python software package

1 Introduction

A main goal of psychological research is to gain knowledge about brain and behavior. Scientific discovery is guided in part by statistical inference, and the strength of any inference depends on the quality of the data collected. Because human data always contain various types of noise, researchers need to design experiments so that the signal of interest (experimental manipulations) is amplified while influences from uncontrolled variables (noise) are attenuated. The design space, the stimulus set that arises from decisions about the independent variable (number of variables, number of levels of each variable) is critically important for creating a high-signal experiment.

A similarly important consideration is the stimulus presentation schedule during the experiment. This issue is often guided by two competing goals: efficiency and precision. How much data must be collected to be confident that differences between conditions could be found? This question is similar to that asked when performing power analysis, but is focused on the performance of the participant during the experiment itself. Too few trials yield poor precision (low signal-to-noise ratio); there are simply not enough data to make any inference, for or against a prediction, with confidence. Although adding more trials can increase precision, too many trials will start to reduce it (increasing noise) through the inconsistencies in responding that come with participant fatigue and boredom. What then is the optimal number of trials that will provide the most precise performance estimates? A partial answer lies in recognizing that not all stimuli are equally informative. By optimizing stimulus selection in the design space, efficiency and precision can be balanced.

Methods of optimizing efficiency and precision have been developed for some experimental paradigms. The most widely used one is the staircase procedure for estimating a threshold (Cornsweet, 1962; Feeny et al., 1966; Rose et al., 1970), such as when measuring hearing or visual acuity. Stimuli differ along a one-dimensional continuum (intensity). The procedure operates by a simple heuristic rule, of which there are a handful of variants: The stimulus to present on one trial is determined by the response on the previous trial. Intensity is increased if the stimulus was not detected, decreased if it was. The experiment is stopped after a given number of reversals in direction has been observed. The staircase method is efficient because the general region of the threshold is identified after a relatively small number of trials, after which the remaining trials concentrate on obtaining a precise threshold estimate. Its ease of implementation and generally good results have made it a popular method across many fields in psychology.

Formal approaches to achieving these same ends (good efficiency and precision) have also been

developed. They originated in the fields of optimal experimental design in statistics (Lindley, 1956; Atkinson and Donev, 1992) and active learning in machine learning (Cohn et al., 1994; Settles, 2009). In psychology, the application of these methods began in visual psychophysics (e.g., Kontsevich and Tyler, 1999), but has since expanded into other content areas (neuroscience, memory, decision making) and beyond. Common among them is the use of a Bayesian decision theoretic framework. The approach is intended to improve upon the staircase method by using not only the participant's responses to guide the choice of the stimulus on the next trial, but also a mathematical model that is assumed to describe the psychological process of interest (discussed more fully below). The model-based algorithm integrates information from both sources (model predictions and participants' responses) to present what it identifies as the stimulus that should be most informative on the next trial.

The method developed in our lab, adaptive design optimization (ADO), has shown to be efficient and precise. For example, in visual psychophysics, contrast sensitivity functions (thresholds) can be estimated so precisely in 50 trials that small changes in luminance (brightness) can be differentiated (Gu et al., 2016; Hou et al., 2016). In delayed discounting, precise estimation of the k parameter of the hyperbolic model (a measure of impulsivity) can be obtained in fewer than 20 trials, and the estimate is 3-5 times more precise than the staircase method (Ahn et al., 2019). Other applications of ADO can be found in several areas of psychology such as retention memory (Cavagnaro et al., 2010, 2011), risky choice decision (Cavagnaro et al., 2013a,b; Aranovich et al., 2017), and in neuroscience (Lewi et al., 2009; DiMattina and Zhang, 2008, 2011; Lorenz et al., 2016).

The technical expertise required to implement the ADO algorithm is nontrivial, posing a hurdle to its wider use. In this paper, we introduce an open-source Python package, dubbed ADOpy, that is intended to make the technology available to researchers who have limited background in Bayesian statistics or cognitive modeling (e.g., the hBayesDM package, Ahn et al., 2017). Only a working knowledge of Python programming is assumed.[1] For an in-depth, comprehensive treatment of Bayesian cognitive modeling, the reader is directed to the following excellent sources written for psychology researchers (Lee and Wagenmakers, 2014; Farrell and Lewandowsky, 2018; Vandekerckhove et al., 2018). ADO is implemented in three two-choice tasks: psychometric function estimation, the delay discounting task (Green and Myerson, 2004) and the choice under risk and ambiguity (CRA) task (Levy et al., 2010). ADOpy easily inmterfaces with Python code running one

[1] ADOpy is available for download at `https://github.com/adopy/adopy`.

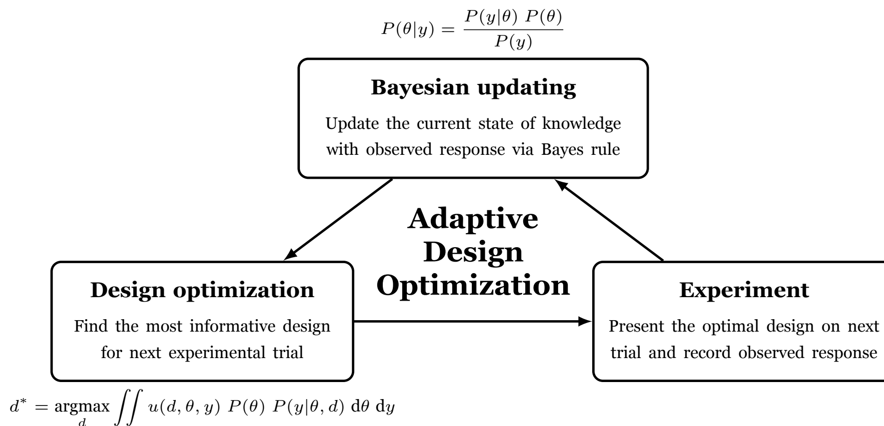

Figure 1: Schematic diagram illustrating the three iterative steps of adaptive design optimization (ADO).

85 of these tasks, requiring only a few definitions and one function call. Most model parameters have default values, but a simulation mode is provided for users to assess the consequences of changing parameter values. As we discuss below, this is a useful step that we encourage researchers to use to ensure the algorithm is optimized for the given test situation.

The algorithm underlying ADO is illustrated in Figure 1. It consists of three steps that are 90 executed on each trial of an experiment: (1) design optimization; (2) experimentation; and (3) Bayesian updating. In the first step, we identify the optimal design (e.g., stimulus) of all possible designs, the choice of which is intended to provide the most information about the quantity to be inferred (e.g., model parameters). In Step 2, an experiment is carried out with the chosen experimental design. In Step 3, the participant's response is used to update the belief about the 95 informativeness of all designs. This revised (updated) knowledge is used to repeat the ADO cycle on the next trial of the experiment.

The following section provides a short technical introduction to the ADO algorithm. Subsequent sections introduce the package and demonstrate how to use ADOpy for optimizing experimental design with walk-through examples from three domains: psychometric function estimation, delay 100 discounting, and risky choice. Readers who prefer to concentrate on practical applications of the algorithm rather than its technicalities should skip Section 2 and jump directly to Section 3.

2 Adaptive Design Optimization (ADO)

ADO follows in the tradition of optimal experimental design in statistics (Lindley, 1956; Atkinson and Donev, 1992) and active learning in machine learning (Cohn et al., 1994; Settles, 2009).
105 ADO is a model-based approach to optimization in the sense that it requires a quantitative (statistical, cognitive) model that predicts experimental outcomes based on the model's parameters and design variables (e.g., experimentally controllable independent variables). Statistically speaking, a model should be defined as a *likelihood function*, a parametric family of probability distributions indexed by its parameters, denoted by $p(y|\theta, d)$, where y represents a vector of experimental
110 outcomes, θ is the parameter vector, and finally, d is the vector of design variables.

ADO is formulated in a Bayesian framework of optimal experimental design (Chaloner and Verdinelli, 1995; Müller, 1999; Müller et al., 2004; Amzal et al., 2006). On each ADO trial, we seek to identify the optimal design d^* that maximizes some real-valued function $U(d)$ that represents the utility or usefulness of design d. Formally, the "global" utility function $U(d)$ (Chaloner and
115 Verdinelli, 1995) is defined as:

$$U(d) = \iint u(d, \theta, y)\, p(y|\theta, d)\, p(\theta)\, dy\, d\theta, \tag{1}$$

where $p(\theta)$ is the prior distribution. In the above equation, $u(d, \theta, y)$, called the "local" utility function, measures the utility of an experiment carried out with design d when the model outputs an outcome y given the parameter value θ. Note that the global utility $U(d)$, which is a function of
120 design d, represents the mean of the local utility $u(d, \theta, y)$ calculated across all possible outcomes and parameter values, weighted by the likelihood function $p(y|\theta, d)$ and the prior $p(\theta)$.

As typically done in ADO, the ADOpy package adopts an information theoretic framework in which the optimal design is defined as the one that is maximally informative about the unknown quantity of interest, i.e., the values of the parameter θ in our case. Specifically, by using Shannon's
125 entropy, a particular local utility function is defined as $u(d, \theta, y) = \log \frac{p(\theta|y,d)}{p(\theta)}$. The global utility function in Equation (1) becomes the mutual information between the outcome random variable $Y(d)$ and the parameter random variable Θ conditional on design d (Cover and Thomas, 1991):

$$U(d) = H(Y(d)) - H(Y(d)|\Theta), \tag{2}$$

where $H(Y(d))$ is the marginal entropy (i.e., overall uncertainty) of the outcome event and $H(Y(d)|\Theta)$
130 is the conditional entropy of the outcome event *given* the knowledge of the parameter θ.[2] According-

[2]See Step 1 in Figure 2 for specific equations defining the entropy measures in Equation (2)

Grid-based ADO Algorithm

Step 0. Pre-computation and initial prior

1) Precompute the likelihood $p(y|\theta, d)$ for all discretized values of y, θ, and d.
2) Precompute the entropy $H(Y(d)|\theta) = -\sum_y p(y|\theta, d) \ln p(y|\theta, d)$ for all discretized values of d and θ.
3) Let $t = 0$, and initialize the prior $p_t(\theta)$ for all discretized values of θ.

Step 1. Design Optimization

1) Compute the marginal likelihood $p(y|d) = \sum_\theta p(y|\theta, d)\, p_t(\theta)$ for all discretized values of y and d.
2) Compute the conditional entropy $H(Y(d)|\Theta) = \sum_\theta p_t(\theta)\, H(Y(d)|\theta)$ for all discretized values of d.
3) Compute the marginal entropy $H(Y(d)) = -\sum_y p(y|d) \ln p(y|d)$ for all discretized values of d.
4) Identify the optimal design d^* that maximizes the mutual information $I(Y(d); \Theta) = H(Y(d)) - H(Y(d)|\Theta)$.

Step 2. Experimentation

-) Run the experiment with the design d^* and observe an outcome $y_{obs}(t)$.

Step 3. Bayesian Updating

1) Compute the posterior $p(\theta|y_{obs}(t), d^*) = \frac{p(y_{obs}(t)|\theta, d^*)\, p_t(\theta)}{p(y_{obs}(t)|d^*)}$ via Bayes rule for all discretized values of θ.
2) Set $p_{t+1}(\theta) = p(\theta|y_{obs}(t), d^*)$ and $t = t + 1$, and go to **Step 1** above.

Figure 2: Three-steps of a grid-based ADO algorithm with an initial step for pre-computation.

ingly, the optimal design d^* that maximizes the mutual information in Equation (2) is the one that maximally reduces the uncertainty about the parameters of interest.

Once the optimal design d^* is identified, we then conduct an experiment on the current trial with the optimal design and observe an experimental outcome y_{obs}. The prior distribution $p(\theta)$ is updated
135 via Bayes rule with this new observation to obtain the posterior distribution $p(\theta|y_{obs})$, which in turn becomes the *new* prior on the next trial, i.e., by replacing $p(\theta)$ with $p(\theta|y_{obs})$ in Equation (1). This "trilogy scheme" of design optimization, experimentation, and Bayesian updating, depicted in Figure 1, is applied successively on each ADO trial until the end of the experiment.

Finding the optimal design d^* that maximizes $U(d)$ in Equation (1) is computationally non-
140 trivial as it involves solving a high dimensional maximization and integration problem. As such, obtaining an analytic form solution for the problem is generally not possible; instead, approximate solutions must be sought numerically. For this purpose, the ADOpy package implements a grid-based algorithm for both the design optimization and Bayesian updating steps in Figure 1. Implementation of the algorithm requires the discretization of both the continuous parameter and
145 design spaces. That is, each element of the parameter vector θ and the design vector d is represented as a one-dimensional discretized line with a finite number of grid points. Further, the local utility function $u(d, \theta, y)$, the likelihood function $p(y|\theta, d)$, and the prior $p(\theta)$ are all represented numerically as vectors defined on the grid points.

Putting the preceding specifications together, Figure 2 lays out in greater detail the grid-based ADO algorithm implemented in the ADOpy package, which is adapted from Bayesian adaptive estimation algorithms in psychophysics (Kontsevich and Tyler, 1999; Kujala and Lukka, 2006; Lesmes et al., 2006). The algorithm requires creating and storing in memory a pre-computed look-up table of various functions over all possible (discretized) outcomes, parameter values, prior to executing the three steps of the ADO trilogy scheme in Figure 1. For a more thorough description of the algorithm, the reader is directed to other sources (e.g., Cavagnaro et al., 2010; Myung et al., 2013).

3 ADOpy

In this section, we provide a step-by-step guide on how to use the ADOpy package to compute optimal designs adaptively with walk-through examples. It is assumed that readers are familiar with Python programming and have written experiment scripts using Python or some other language.

3.1 Overview

ADOpy is designed in a modular fashion to ensure functional flexibility and code readability. At the core of the package are three classes: `Task`, `Model`, and `Engine` classes. The `Task` class is used to define design variables of a task. The `Model` class is used to define model parameters and the likelihood function (e.g., Myung, 2003; Farrell and Lewandowsky, 2018). The `Engine` class is used for implementing design optimization and Bayesian updating.

The general workflow using these classes is illustrated in Figure 3. After loading the three classes, users should initialize each object, with the engine requiring the most parameters. The for-loop is an experiment itself divided into three parts: 1) obtain the design (stimulus) for the next trials and present the stimulus to the participant; 2) obtain a response from the participant, which would come from a keyboard or mouse, as defined by the experimenter; 3) update the ADO engine using the participant response together with the design.

ADOpy implements a *grid-search algorithm* in which the design space and parameter space are discretized as sets of grid points. How to set grid points and the range of each grid dimension is described in detail in Section 3.5.

Owing to the modular structure of ADOpy, users do not have to concern themselves with how the `Engine` works, other than defining the `Task` and the `Model` classes. Consequently, ADOpy dra-

matically reduces the amount of coding, and the likelihood of coding errors, in the implementation of ADO.

180 ## 3.2 Prerequisites

Before installing ADOpy, users should install Python (version 3.5 or higher). Using the Anaconda distribution (`https://www.anaconda.com`) is greatly recommended, since it provides a rich environment for scientific data analysis.

ADOpy depends on several core packages for scientific computing: NumPy, SciPy, and Pandas. 185 Since ADOpy uses high dimensional matrices to compute optimal designs, it is strongly recommended to install linear algebra libraries (e.g., Intel Math Kernel Library, LAPACK, BLAS) to make the operations fast. If the Anaconda distribution is used, the Intel Math Kernel Library will be used as the default.

3.3 Installation

190 The ADOpy package is available from the Python Package Index (PyPI) and GitHub. The easiest way to install ADOpy is from PyPI using `pip` as follows:

```
pip install adopy
```

To install the developmental version, users can install it from GitHub. However, it can be unstable, so use it with caution.

Step 0. Initialization

1) Define a task using `adopy.Task`.
2) Define a model using `adopy.Model`.
3) Define grids for design variables and model parameters.
4) Initialize an engine using `adopy.Engine`.

Step 1. Design optimization

- Compute an optimal design.

Step 2. Experiment

- Conduct an experiment using the design.

Step 3. Bayesian optimization

- Update the engine based on the observation.

```
# Load ADOpy
from adopy import Task, Model, Engine

# Step 0. Initialization
task = Task(designs, responses)
model = Model(params, function)
grid_design = {...}; grid_param = {...}
engine = Engine(task, model, grid_design, grid_param)

for trial in trials:
    # Step 1. Design optimization
    design = engine.get_design()
    # Step 2. Experiment
    response = ...  # Get a response from users' own codes
    # Step 3. Bayesian updating
    engine.update(design, response)
```

Figure 3: Procedures of using ADOpy and its general workflow. Each function call above is described in greater detail in Section 3.5. Note that ADOpy itself does not provide actual code to conduct an experiment; the user must program his/her experiment.

195

```
git clone https://github.com/adopy/adopy.git
cd adopy
git checkout develop
pip install .
```

To check that ADOpy was installed successfully, run the following code at the Python prompt.
200 As of now, the latest version is `0.3.0`.

```
import adopy
adopy.__version__  # '0.3.0'
```

3.4 Module structure

Inside the ADOpy package, the two most important modules are `adopy.base` and `adopy.tasks`.
205 The module adopy.base contains three basic classes: `Task`, `Model`, and `Engine` (see more details in Section 3.5). Using these classes, users can apply the ADO procedure into their tasks and models. For convenience, users can load these classes directly from `adopy` itself as follows:

```
# Load three classes from ADOpy
from adopy import Task, Model, Engine
```

210 The other module, `adopy.tasks`, contains three pre-implemented tasks and models (see Section 4 and Table 1). The three tasks are psychometric function estimation (`adopy.tasks.psi`), the delay discounting task (`adopy.tasks.ddt`), and the choice under risk and ambiguity task (`adopy.tasks.cra`).

3.5 Basic usage

ADOpy requires four steps of its procedure to be utilized in an experiment as shown in Figure 3.
215 Among the four, the most important and complex is the *Initialization* step, in which ADOpy objects to be used in the subsequent steps are defined. The *Initialization* step itself comprises four sub-steps: defining a task, defining a model, defining grids, and initializing an ADO engine. In this section, we explain the coding involved in each of these sub-steps using the delay discounting task as an example.

220 **Defining a task** The `Task` class is for defining the experimental task. Using the `Task` class, a task object is initialized by specifying three types of information: the name of the task (`name`), the design variables (`designs`), and the response variable (`responses`).

Delay discounting (DD; the task is depicted in Figure 4), refers to the well-established finding that animals, including humans, tend to discount the value of a delayed reward such that the discount progressively increases as a function of the receipt delay (e.g., Green and Myerson, 2004; Vincent, 2016). The delay discounting task has been widely used to assess individual differences in temporal impulsivity and is a strong candidate endophenotype for addiction (Green and Myerson, 2004; Bickel, 2015). In a typical DD task, a participant is asked to indicate his/her preference between two options, a smaller-sooner (SS) option or stimulus (e.g., 8 dollars now) and a larger-later (LL) option (e.g., 50 dollars in a month). Let us use a formal expression (R_{SS}, t_{SS}) to denote the SS option where R_{SS} represents the reward amount, and t_{SS} represents the receipt delay. Similarly, (R_{LL}, t_{LL}) denotes the LL option. By definition, the following constraints are imposed on the reward amounts and the delay times: $R_{SS} < R_{LL}$ and $t_{SS} < t_{LL}$ for a given pair of options. The choice response is recorded as either $y = 1$ (LL option) or $y = 0$ (SS option).

The DD task therefore has four design variables, i.e., $d = (t_{SS}, t_{LL}, R_{SS}, R_{LL})$, with a binary response on each trial (i.e., 0 or 1). As such, we define a `Task` object for the DD task as follows:

```
from adopy import Task

task = Task(name='Delay discounting task',
            designs=['t_ss', 't_ll', 'r_ss', 'r_ll'],
            responses=[0, 1])
```

where the four symbols (`t_ss`, `t_ll`, `r_ss`, `r_ll`) denote short notations for the respective design variables $(t_{SS}, t_{LL}, R_{SS}, R_{LL})$.

With the task object defined, the information passed into the object can be accessed by

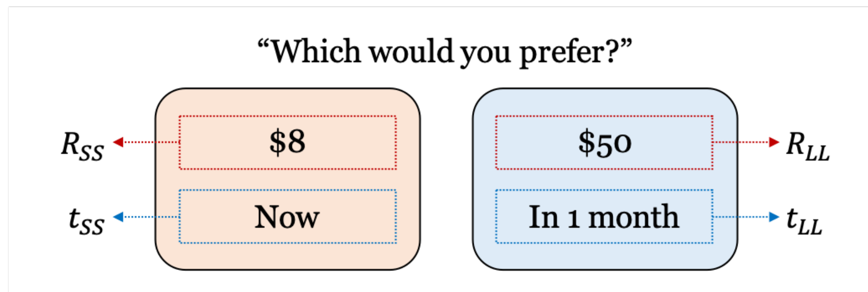

Figure 4: Illustrated scheme of the delay discounting (DD) task. On each trial, a participant is asked to choose between two options, a smaller-sooner (SS) option on the left and a larger-later (LL) option on the right. The dotted lines and arrows indicate the design variables of the task to be optimized.

245 `task.name`, `task.designs`, and `task.responses`, respectively:

```
task.name  # 'Delay discounting task'
task.designs  # ['t_ss', 't_ll', 'r_ss', 'r_ll']
task.responses  # [0, 1]
```

Defining a model Before making a model object, users should define a function that describes how to compute the likelihood function given design variables and model parameters. For example, the hyperbolic model for the delay discounting task is defined with the following set of equations:

$$\begin{aligned} D(t) &= \frac{1}{1+kt} \\ V_{LL} &= R_{LL} \cdot D(t_{LL}) \\ V_{SS} &= R_{SS} \cdot D(t_{SS}) \\ P(LL\, over\, SS) &= \frac{1}{1+\exp[-\tau(V_{LL}-V_{SS})]} \end{aligned} \tag{3}$$

where $P(\text{LL over SS})$ denotes the probability of choosing the LL option over the SS option, and V_{LL}
250 and V_{SS} denote subjective value estimates for the LL and SS options respectively. There are two model parameters: k represents the discounting rate and τ the inverse temperature that measures the consistency or stability in choice responses. For further details about the above model, the reader is referred to Section 4.2.

Based on the above model, a likelihood function that computes the response probability can be
255 defined as

```
import numpy as np


def compute_likelihood(t_ss, t_ll, r_ss, r_ll, k, tau):
    v_ss = r_ss * np.divide(1, 1 + t_ss * k)
    v_ll = r_ll * np.divide(1, 1 + t_ll * k)
    p_obs = np.divide(1, np.exp(-tau * (v_ll - v_ss)))
    return p_obs
```

The argument names for design variables in the above function definition *must* be the same as those used in the task definition (i.e., `t_ss`, `r_ss`, `t_ll`, `r_ll`). We also recommend using NumPy
265 functions for the definition, given that it can vectorize basic mathematical operations.

Using the likelihood function (`compute_likelihood`), ADOpy handles a mathematical model by the `Model` class. The `Model` class requires four arguments: the name of the model (`name`), a task object related to the model (`task`), labels of model parameters (`params`), and the likelihood function of the model (`func`). In terms of these arguments, a model object is defined as below:

```
from adopy import Model


model = Model(name='Hyperbolic model',
              task=task,
              params=['k', 'tau'],
              func=compute_likelihood)
```

As in the task object, the information passed into the model object can be accessed by `model.name`, `model.task`, and `model.params`:

```
model.name   # 'Hyperbolic model'
model.task   # Task('Delay discounting task', ...)
model.params   # ['k', 'tau']
```

Further, users can run the likelihood function passed into the model object by `model.compute()`, which uses the same arguments that are used for the `compute_likelihood()` function, as follows:

```
model.compute(t_ss, t_ll, r_ss, r_ll, k, tau)
```

Defining grids As mentioned earlier, ADOpy implements a grid-based algorithm that requires the discretization of both parameter and design spaces. As such, before running ADO using model and task objects, users must specify the grid resolution to be used for the design optimization and Bayesian updating steps in Figure 1. This amounts to defining the number and spacing of grid points on each dimension of the design and parameter variables. The grid passed to the ADO engine determines (1) the range of values in design variables that the ADO engine can suggest and (2) the range of the model parameters over which the computations will be carried out.

It is important to note that the number of grid points affects the efficiency and reliability of parameter estimation. The more sparse the grid, the more efficient but less precise parameter estimation will be; the denser the grid, the more precise but less efficient parameter estimation will be. Specifically, sparse grids can lead to poorly estimated model parameters whereas dense grids can require large amounts of memory and long computing times. Thus, before conducting

an ADO-based experiment with participants, it is worth identifying the optimal grid resolution for each parameter/design variable. A simulation mode provided with ADOpy can help facilitate this process.

A grid object for ADOpy can be defined as a Python dictionary object by using the name of a variable as its key and a list of the grid points as its values. If a design variable or model parameter needs to be fixed to a single value, users would simply assign a single grid point for the variable. Also, to restrict the values of a variable, users can manually make a matrix in which each column vector indicates possible values for the variable, then pass it as a value with a key of the column labels. Example codes below illustrate various ways of defining the grids for two arbitrary design variables (`x1` and `x2`):

```
# A grid object for two design variables, x1 and x2.
grid_design = {'x1': [1, 2, 3], 'x2': [1, 2, 3]}

# One variable can be fixed to a single value as below:
grid_design = {'x1': [1, 2, 3], 'x2': [0]}

# Constrain the grid by using a joint matrix.
x_joint = []
for x1 in [1, 2, 3]:
    for x2 in [1, 2, 3]:
        if x1 <= x2:  # Use design pairs in which x1 <= x2
            x_joint.append([x1, x2])
# [[1, 1], [1, 2], [1, 3], [2, 2], [2, 3], [3, 3]]
grid_design = {('x1', 'x2'): x_joint}
```

In much the same way, users can also define a grid for model parameters. For example, a grid for the two parameters of the delay discounting model in Equation (3), `k` and `tau`, can be defined as:

```
grid_param = {
    'k': np.logspace(-5, 0, 20),  # 20 points within [10^-5, 10^0] in a log scale
    'tau': np.linspace(0, 5, 20)
}
```

The reader is directed to Appendix A for more examples for defining grids for the delay discounting task.

Initializing an ADO engine With the defined `Model` and `Task` classes and grids for design and
330 parameter variables, users are now ready to load an `Engine` for ADO computation. It requires four arguments: (1) the task object (`task`); (2) the model object (`model`); (3) a grid for design variables (`grid_design`); and (4) a grid for model parameters (`grid_param`):

```
from adopy import Engine

engine = Engine(model=model, task=task,
                grid_design=grid_design, grid_param=grid_param)
```

When initializing an instance of `Engine`, it pre-computes likelihood and mutual information for given sets of designs and parameters. This step may take a while, with linearly increasing computing time in proportion to the number and resolution of the grids.

340 Once the engine object is in place, users can access its task objects: the exhaustive list of task objects is (`engine.task`), its model object (`engine.model`), the number of possible pairs on design variables (`engine.num_design`), the number of possible pairs on model parameters (`engine.num_param`), the grid matrix of design variables (`engine.grid_design`), the grid matrix of model parameters (`engine.grid_param`), the prior distribution on the grid matrix of model parameters (`engine.prior`),
345 the posterior distribution on the grid matrix of model parameters (`engine.post`), the posterior mean (`engine.post_mean`), the covariance matrix of the posterior (`engine.post_cov`), and the standard deviations of the posterior (`engine.post_sd`).

Two functions are available in ADOpy for the engine object: `engine.get_design()` and `engine.update()`. The `engine.get_design()` provides a set of designs on each trial of the experiment given a specified
350 design type. With an argument of `design_type`, users can indicate the type of design to use. There are two possible values: `'optimal'` and `'random'`, where `'random'` refers to a randomly chosen design. The output of this function call is a dictionary that contains key-value pairs for each design variable and its optimal or random value.

```
design = engine.get_design('optimal')  # Provides the optimal design
design = engine.get_design('random')  # Provides a random design
```

If no argument is given for `design_type`, the optimal design is returned by default:

```
design = engine.get_design()
```

The other important use of the engine object is `engine.update()`. Here, ADOpy first performs the Bayesian updating step described in Figures 1 and 2 based on a participant's response given
360 the design, and then computes a new optimal design for the next trial using the updated posterior distributions of model parameters. It takes two arguments: the design used on the given trial (`design`), and the corresponding response on that trial (`response`). For example, from the observation that a participant selects the SS option (`response = 0`) or the LL option (`response = 1`) on the current trial, users can update the posterior as follows:

```
365 engine.update(design, response)
```

Simulating responses ADOpy can be run in the simulation mode to assess design quality and experiment efficiency (see next section). Given that the design itself, the model chosen, and the grid resolution of the design space and model parameters all affect how ADO performs, Simulation mode can be useful to fine-tune the aforementioned variables. Using the engine object of the ADOpy
370 package, users can generate simulated responses given true parameters. As a concrete example, let us run the simulation with true parameter values of `k = 0.12` and `tau = 1.5` of the delay discounting model described in Equation (3). To acquire a simulated response, we use the Bernoulli likelihood function for a binary choice response as described below:

```
from scipy.stats import bernoulli
375
def get_simulated_response(model, design):
    """Simulate a response using tau = 1.5 and k = 0.04."""
    # Compute the probability of choosing the LL option
    p_obs = model.compute(t_ss=design['t_ss'], t_ll=design['t_ll'],
380                           r_ss=design['r_ss'], r_ll=design['r_ll'],
                          k=0.12, tau=1.5)

    # Compute a random binary choice response using Bernoulli
    return bernoulli.rvs(p_obs)
```

385 With the functions and objects defined as above, we can now run the simulations with a code block like below:

```python
NUM_TRIALS = 108  # number of trials for the simulation
for trial in range(NUM_TRIALS):
    design = engine.get_design('optimal')  # Design optimization
    response = get_simulated_response(model, design)  # Experiment
    engine.update(design, response)  # Bayesian updating
```

Note that the above code block consists of three parts: design optimization, experimentation, and Bayesian updating, in the same way done in an actual ADO-based experiment as described in Figure 1.

3.6 Practical issues

Users should carefully consider several practical issues when using ADOpy.

Grid-based ADO, which is what is used here, may demand a lot of memory. While pre-computing a look-up table lessens repeated calculation between trials, it requires more and more memory as the grid size increases. Thus, users are advised to first determine the proper number of grid points on each dimension of the model parameters and design variables and to check if computing time with the settings is suitable (i.e., fast enough to prevent boredom between trials). By varying grid resolutions, users should determine how many grids they will use considering its estimation accuracy and the computational cost for calculation. The other option is to use a dynamic gridding algorithm, in which grid space is dynamically adjusted and grid points near posterior means are more finely spaced. Adaptive mesh refinement (AMR: e.g., Berger, 1984) is one such method. ADOpy does not currently support dynamic-gridding-based ADO, but the development is ongoing.

To use ADO for a typical experiment with participants, the computation time on each trial should not exceed approximately one second. Since grid-based ADO requires high-dimensional operations, it may take considerable time. Using linear algebra libraries (e.g., Intel MKL, LAPACK, or BLAS) can be one option, which enables the operations to be executed efficiently and in parallel using multi-core CPUs. In some cases, computation time for ADOpy in a local computer may be too long. One alternative way to address this issue is by using a remote server or a cloud computing system, such as Amazon Web Service. By obtaining an optimal design asynchronously from a fast remote server, users can minimize delay between trials.

ADOpy will eventually start to select the same or similar design on consecutive trials. This is

a sign that not much more can be learned from the experiment (e.g., parameter estimation is quite good). This will happen toward the end of an experiment if there are sufficient trials. One option to address the issue is to dilute their presence by using filler trials, showing randomly chosen or predetermined designs for a trial when ADO picks the same design twice or more in a sequence. Another option is to run the experiment in a "self-terminating mode"; stop the experiment once a specific criterion (e.g., efficiency) is reached, e.g., the standard deviations of posterior distributions go below certain predetermined values.

4 Tasks and Models implemented in ADOpy

Currently, three tasks are implemented in the ADOpy package as in Table 1: Psychometric function estimation (`adopy.tasks.psi`), the delay discounting task (`adopy.tasks.dd`), the choice under risk and ambiguity task (`adopy.tasks.cra`). At least two models are available for each task.

In this section, we describe these tasks and illustrate how to use each task/model in ADOpy and how ADO performs compared to traditional non-ADO (e.g., staircase, random) methods, along with simulated results for the three tasks. In particular, we provide and discuss a complete and full Python script for simulating psychometric function estimation in ADOpy.

Module	Task		Model			Engine
	Class	Designs	Class	Model name	Parameters	
Choice under risk & ambiguity (`adopy.tasks.cra`)	`TaskCRA`	`p_var`, `a_var`, `r_var`, `r_fix`	`ModelLinear`	Linear	`alpha`, `beta`, `gamma`	`EngineCRA`
			`ModelExp`	Exponential		
Delay discounting (`adopy.tasks.dd`)	`TaskDD`	`t_ss`, `t_ll`, `r_ss`, `r_ll`	`ModelExp`	Exponential	`tau`, `r`	`EngineDD`
			`ModelHyp`	Hyperbolic	`tau`, `k`	
			`ModelHPB`	Hyperboloid	`tau`, `k`, `s`	
			`ModelCOS`	Constant Sensitivity	`tau`, `r`, `s`	
			`ModelQH`	Quasi-Hyperbolic	`tau`, `beta`, `delta`	
			`ModelDE`	Double Exponential	`tau`, `omega`, `r`, `s`	
Psychometric function estimation (`adopy.tasks.psi`)	`Task2AFC`	`stimulus`	`ModelLogistic`	Logistic function	`guess_rate`, `lapse_rate`, `threshold`, `slope`	`EnginePsi`
			`ModelWeibull`	Log-Weibull CDF		
			`ModelProbit`	Normal CDF		

Table 1: Tasks and models implemented in the ADOpy package (alphabetized order). For detailed information, see the documentation website for ADOpy (`https://github.com/adopy/adopy`).

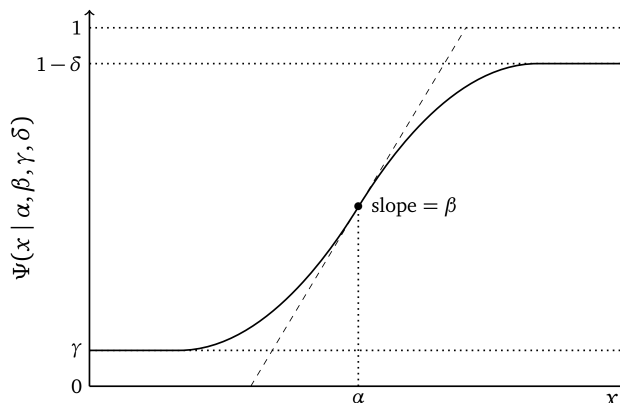

Figure 5: The psychometric function and its parameters defined in Equation (4).

4.1 Psychometric function estimation

Psychometric function estimation is one of the first modeling problems in the psychological sciences in which a Bayesian adaptive framework was applied to improve the efficiency of psy-
435 chophysical testing and analysis (Watson and Pelli, 1983; King-Smith et al., 1994; Kujala and Lukka, 2006; Lesmes et al., 2006). The problem involves a 2-alternative forced choice (2AFC) task in which the participant decides whether a psychophysical stimulus, visual or auditory, is present or absent while the stimulus intensity is varied from trial to trial to assess perceptual sensitivity.

The psychometric function that defines the probability of correct detection given stimulus in-
440 tensity x is given as the following general form (Garcia-Perez, 1998; Wichmann and Hill, 2001):

$$\Psi\left(x \mid \alpha, \beta, \gamma, \delta\right) = \gamma + \left(1 - \gamma - \delta\right) F(x; \alpha, \beta) \tag{4}$$

The participant's response in the psychophysical task is recorded in either $y = 1$ (correct) or $y = 0$ (incorrect). The two-parameter sigmoid function $F(x; \alpha, \beta)$ that characterizes the relationship between the response probability and the stimulus intensity is typically assumed to follow the
445 logistic, cumulative normal, or cumulative log Weibull form (see, e.g., Wichmann and Hill, 2001, for further details). The parameter vector $\theta = (\alpha, \beta, \gamma, \delta)$ of the psychometric function consists of α (threshold), β (slope), γ (guess rate) and δ (lapse rate), as depicted in Figure 5. Note that design variable is stimulus intensity, i.e., $d = x$.

The module '`adopy.tasks.psi`' included in the ADOpy package provides classes for psychometric
450 function estimation in the 2AFC experimental paradigm (see Table 1). In the module, `Task2AFC` is pre-defined for 2AFC tasks with a single design variable (`stimulus`) and binary responses (0 for incorrect or 1 for correct). Without passing any arguments, users can utilize the pre-defined

`Task2AFC` class as below:

```
from adopy.tasks.psi import Task2AFC
task = Task2AFC()

task.designs    # ['stimulus']
task.responses  # [0, 1]
```

For the task, users can specify the form of the two parameter sigmoid psychometric function $F(x; \alpha, \beta)$ as in Equation (4) from three classes: a logistic function (`ModelLogistic`), a log Weibull CDF (`ModelWeibull`), and a normal CDF (`ModelProbit`). Here, assume that the psychometric function has a logistic form which computes correct detection as:

$$\Psi\left(x \mid \alpha, \beta, \gamma, \delta\right) = \gamma + (1 - \gamma - \delta) \cdot \frac{1}{1 + \exp\left[-\beta(x - \alpha)\right]}. \tag{5}$$

Based on Equation (5), the `ModelLogistic` class in the `adopy.tasks.psi` provides the equivalent model with four parameters (`threshold` α, `slope` β, `guess_rate` γ, `lapse_rate` δ).

```
from adopy.tasks.psi import ModelLogistic
model = ModelLogistic()

model.params  # ['threshold', 'slope', 'guess_rate', 'lapse_rate']
```

As grid resolutions for the task and model, we provide an example code while fixing guess rate to 0.5 and lapse rate to 0.04 as described below. Especially for `stimulus` and `threshold`, users should define them within appropriate ranges for their tasks of interest.

```
import numpy as np

grid_design = {
    'stimulus': np.linspace(20 * np.log10(.05), 20 * np.log10(400), 100)
}

grid_param = {
    'guess_rate': [0.5],
    'lapse_rate': [0.04],
    'threshold': np.linspace(20 * np.log10(.1), 20 * np.log10(200), 100),
```

```
        'slope': np.linspace(0, 10, 100)
    }
```

480 Based on the task object, model object, and grids, `adopy.tasks.psi` provides an Engine class, called `EnginePsi`, pre-implemented for psychometric function estimation. The `EnginePsi` class not only provides an optimal design or randomly chosen design, but also computes a design using the staircase method. The staircase method is probably the most commonly used procedure in adaptive estimation of the psychometric function (e.g., Garcia-Perez, 1998) in which stimulus intensity is 485 adjusted by a fixed and pre-determined amount based on a participant's response on the current stimulus. The following code initializes the engine and computes designs:

```
from adopy.tasks.psi import EnginePsi
engine = EnginePsi(model, grid_design, grid_param)

engine.get_design('optimal')    # Returns the optimal design.
engine.get_design('random')     # Returns a randomly chosen design.
engine.get_design('staircase')  # Returns a design using the staircase method.
```

where `EnginePsi` requires only three arguments (`model`, `designs`, and `params`) since the task is fixed to the psychometric function estimation.

495 The particular up/down scheme of the staircase method implemented in '`EnginePsi`' is as follows:

$$x_{t+1} = \begin{cases} x_t - \Delta & \text{if } y_t = 1 \\ x_t + 2\Delta & \text{otherwise (if } y_t = 0) \end{cases} \tag{6}$$

where Δ is a certain amount of change for every trial. `EnginePsi` has a property called `d_step` to compute Δ, which means the number of steps for a index on the design grid. In other words, the denser the design grid is, the smaller Δ becomes. Initially, `d_step` is set to 1 by default, but users 500 can use a different value as described below:

```
engine.d_step      # Returns 1.
engine.d_step = 3  # Update d_step to 3.
```

Having defined and initialized the required task, model, grids, and engine objects, we are now in a position to generate simulated binary responses. This is achieved by using the mod- 505 ule `scipy.stats.bernoulli`. Here, the data-generating parameter values are set to `guess_rate` = 0.5, `lapse_rate` = 0.04, `threshold` = 20, and `slope` = 1.5:

```
from scipy.stats import bernoulli


def get_simulated_response(model, design):
    # Compute a probability to respond positively.
    p_obs = model.compute(stimulus=design['stimulus'], \
            guess_rate=0.5, lapse_rate=0.04, threshold=20, slope=1.5)

    # Sample a binary response using Bernoulli distribution.
    return bernoulli.rvs(p_obs)
```

Finally, the following example code runs 60 simulation trials:

```
num_trials = 60  # number of trials to simulate
design_type = 'optimal'  # or 'random' or 'staircase'


for i in range(num_trials):
    # Compute a design for the current trial
    design = engine.get_design(design_type)
    # Get a simulated response using the design
    response = get_simulated_response(model, design)
    # Update posterior in the engine
    engine.update(design, response)
    # Print the trial number and posterior means and standard deviations
    print('Trial', i + 1, '-', engine.post_mean, '/', engine.post_sd)
```

We conclude this section with a brief presentation of simulation results, comparing performance among three design conditions: ADO, staircase, and random (see Appendix B.1 for the details of the simulation setup). The simulation results are summarized in Figure 6. As shown in Figure 6A, for all three conditions, the estimation of the threshold parameter α, as measured by root mean square error (RMSE), converges toward the ground truth, with ADO designs exhibiting clearly superior performance over staircase and random designs. As for the slope parameter β, the convergence is much slower (ADO and staircase) or even virtually zero (random). Essentially the same patterns of results are observed when performance is measured by the posterior standard deviation (Figure 6B). In short, the simulation demonstrates the advantage of using ADO designs in psychometric function

estimation.

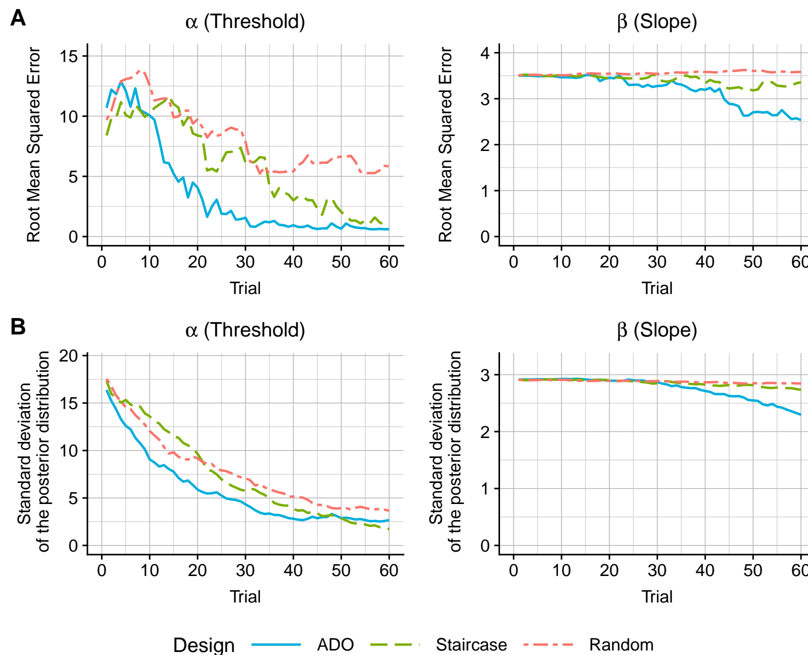

Figure 6: Comparison of ADO, staircase, and random designs in the simulation of psychometric function estimation. Simulations were conducted using the logistic model with parameter values of threshold $\alpha = 20$, slope $\beta = 1.5$, guess rate $\gamma = 0.5$, and lapse rate $\delta = 0.04$. The three designed are compared with root mean squared errors (RMSE; Panel A) and standard deviations of the posterior distribution (Panel B). RMSE represents the discrepancy between true and estimated parameters in that the lower RMSE, the better estimation performance. Standard deviations of the posterior distribution indicate the certainty of a belief on the distribution for model parameters, i.e., the lower the standard deviations is, the higher certainty on the model parameters. Each curve represents an average across 1,000 independent simulation runs.

4.2 Delay discounting task

540 There exists a sizable literature on computational modeling of delay discounting (e.g., Green and Myerson, 2004; Van-DenBos and McClure, 2013; Cavagnaro et al., 2016). As described earlier in Section 3.5, preferential choices between two options, SS (smaller-sooner) and LL (larger-later), are made based on the subjective value of each option, which takes the following form:

$$V = R \cdot D(t) \tag{7}$$

545 where V is the value of an option, R and t are the amount of reward and delay of the option respectively, and $D(t)$ is the discounting factor assumed to be a monotonically decreasing function of delay t.

Various models for the specific form of $D(t)$ have been proposed and evaluated, including the ones below:

$$\begin{aligned} \text{Hyperbolic:} \quad & D(t) = \frac{1}{1+kt} \\ \text{Exponential:} \quad & D(t) = e^{-kt} \\ \text{Hyperboloid:} \quad & D(t) = \frac{1}{(1+kt)^s} \\ \text{Constant Sensitivity:} \quad & D(t) = e^{-(kt)^s} \end{aligned} \tag{8}$$

where the parameter k is a discounting rate and the parameter s reflects the subjective, nonlinear scaling of time (Green and Myerson, 2004). Based on subjective values of options, it is assumed
550 that preferential choices are made stochastically depending on the difference between the subjective values, according to Equation (3). In summary, the models for the delay discounting task assume at most three parameters with $\theta = (k, s, \tau)$, and there are four design variables that can be optimized, i.e., $d = (t_{SS}, t_{LL}, R_{SS}, R_{LL})$. The participant's choice response on each trial is binary in $y = 1$ (LL option) or 0 (SS option).

555 The module '`adopy.tasks.dd`' included in the ADOpy package provides classes for the delay discounting task (see Table 1). `TaskDD` represents the DD task with four design variables (`t_ss`, `t_ll`, `r_ss`, and `r_ll`) with a binary choice response.

```
from adopy.tasks.dd import TaskDD
task = TaskDD()

task.designs   # ['t_ss', 't_ll', 'r_ss', 'r_ll']
task.responses  # [0, 1]
```

560

In addition, the same module '`adopy.tasks.dd`' includes six models (see Table 1): Exponential model (Samuelson, 1937), Hyperbolic model (Mazur, 1987), Hyperboloid model (Green and Myerson,
565 2004), Constant Sensitivity model (Ebert and Prelec, 2007), Quasi-Hyperbolic model (Laibson, 1997), and Double Exponential model (McClure et al., 2007). Here, we demonstrate the Hyperbolic model which has two model parameters (`k` and `tau`) and computes the discounting factor as in Equation (8):

```
from adopy.tasks.dd import ModelHyp
model = ModelHyp()
```

570

```
model.params  # ['k', 'tau']
```

A simulation experiment like that for Psychometric function estimation was carried out with the hyperbolic model, and the results from three designs (ADO, staircase, and random). See Appendix B.2 for the details of the simulation setup and the Python scripts used. The simulation results are presented in Figure 7. As the trial progresses, the discounting rate parameter k converges toward the ground truth for all three design conditions, with the swiftest (almost immediate) convergence with ADO. On the other hand, the inverse temperature parameter τ showed a much slower or even no convergence (staircase), probably due to the relatively small sample size (i.e., 42). In short, the simulation results, taken together, demonstrated the superiority of ADO designs over non-ADO designs.

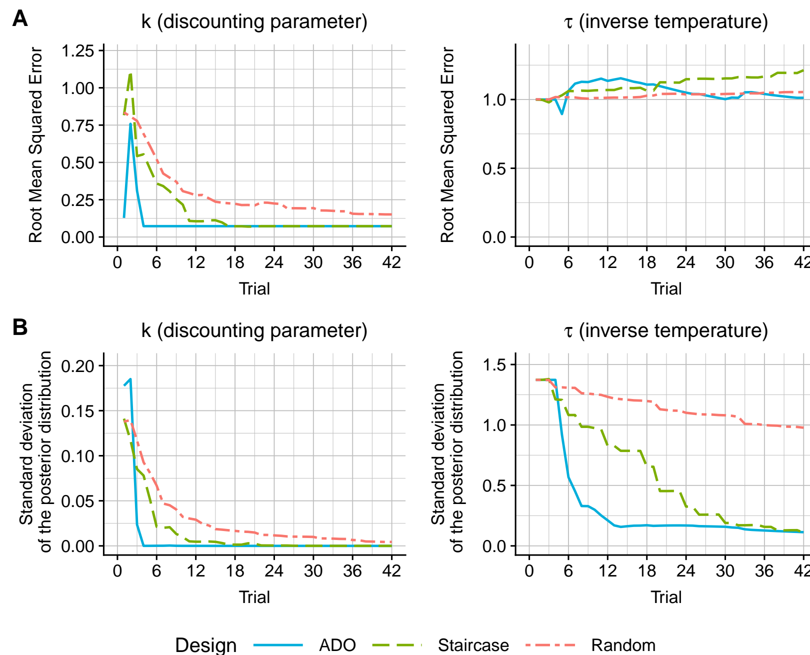

Figure 7: Comparison of ADO, staircase, and random designs in the simulation of the delay discounting task. Simulations were conducted using the hyperbolic model with parameter values of $k = 0.12$ and $\tau = 1.5$. The three designs are compared with root mean squared errors (RMSE; Panel A) and standard deviations of the posterior distribution (Panel B). Each curve represents an average across 1,000 independent simulation runs.

4.3 Choice under risk and ambiguity task

The choice under risk and ambiguity (CRA) task (Levy et al., 2010) is designed to assess how individuals make decisions under two different types of uncertainty: risk and ambiguity. Example stimuli of the CRA task are shown in Figure 8.

The task involves preferential choice decisions in which the participant is asked to indicated a preference between two options: (1) winning either a fixed amount of reward denoted by R_F with a probability of 0.5 or winning none otherwise; and (2) winning a varying amount of reward (R_V) with a varying probability (p_V) or winning none otherwise. Further, the variable option comes in two types: (a) *risky* type in which the winning probabilities are fully known to the participant; and (b) *ambiguous* type in which the winning probabilities are only partially known to the participant. The level of ambiguity (A_V) in the latter type is varied between 0 (no ambiguity and thus fully known) and 1 (total ambiguity and thus fully unknown). As a concrete example, the CRA task of Levy et al. (2010) employed the following values: $R_F = 5$ (reference option); $R_V \in \{5, 9.5, 18, 34, 65\}$, $p_V \in \{0.13, 0.25, 0.38\}$ and $A_V = 0$ (variable options on risky trials); and finally, $R_V \in \{5, 9.5, 18, 34, 65\}$, $p_V = 0.5$ and $A_V \in \{0.25, 0.5, 0.75\}$ (variable options on ambiguity trials).

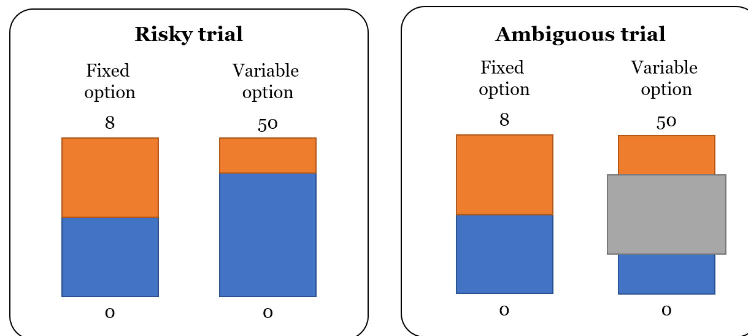

Figure 8: Illustrated scheme of the choice under risk and ambiguity (CRA) task. The participant chooses one of two options on either a risky trial (left) or an ambiguous trial (right). A risky option has the amount of reward and a probability of winning the reward indicated by the upper, brown proportion of the box. For an ambiguous option, the probability to win is not explicitly shown but partially blocked by a gray box. On each trial, a risk or ambiguous option is always paired with a fixed (reference) option whose probability of winning the reward is set to 0.5. .

The linear model (Levy et al., 2010) for the CRA task assumes that choices are based on subjective values of the two options. The subjective values are computed using the following form:

$$\begin{aligned} U_F &= 0.5 \cdot (R_F)^\alpha \\ U_V &= \left[p_V - \beta \left(\frac{A_V}{2} \right) \right] \cdot (R_V)^\alpha \end{aligned} \tag{9}$$

where U_F and U_V are subjective values for fixed and variable options respectively, α is the risk attitude parameter, β is the ambiguity attitude parameter. R_F and R_V are the amounts of reward

600 for fixed and variable options, A_V and p_V are the ambiguity level and the probability to win for a variable option. Both choices are made stochastically based on the difference between the subjective values according to the softmax choice rule:

$$P(V\,over\,F) = \frac{1}{1 + \exp[-\gamma(U_V - U_F)]}. \quad (10)$$

where $P(V\,over\,F)$ represents the probability of choosing the variable option over the fixed one, 605 and the parameter γ represents the inverse temperature that captures the participant's response consistency.

To summarize, the CRA model assumes three parameters, $\theta = (\alpha, \beta, \gamma)$, of α (risk attitude), β (ambiguity attitude), and γ (response consistency). There are four design variables to be optimized: $d = (R_F, R_V, A_V, p_V)$ where $R_F > 0$, $R_V > 0$, $0 < A_V < 1$, and $0 < p_V < 1$ is made up of R_F 610 (reward amount for fixed option), R_V (reward amount for variable option), A_V (ambiguity level) and p_V (winning probability for variable option). The participant's preferential choice on each trial is recorded in either $y = 1$ (variable option) or $y = 0$ (fixed option).

The module '`adopy.tasks.cra`' in the ADOpy package provides classes for the choice under risk and ambiguity task (see Table 1). `TaskCRA` represents the CRA task with four design variables 615 denoted by `p_var` (p_V), `a_var` (A_V), `r_var` (R_V), and `r_fix` (R_F), and a binary choice response.

```
from adopy.tasks.cra import TaskCRA
task = TaskCRA()

task.designs  # ['p_var', 'a_var', 'r_var', 'r_fix']
task.responses  # [0, 1]
```
620

ADOpy currently implements two models of the CRA task: Linear model (Levy et al., 2010) and Exponential model (Hsu et al., 2005). For the linear model in Equation 9, users can define and initialize the model with `ModelLinear` as:

```
from adopy.tasks.cra import ModelLinear
model = ModelLinear()

model.params  # ['alpha', 'beta', 'gamma']
```
625

Now, we briefly discuss results of simulated experiments using the linear model with three design conditions: ADO, fixed, and random design. The fixed design refers to those originally used

630 by Levy et al. (2010). See Appendix B.3 for the details of the simulation setup and code. The results summarized in Figure 9 indicate that two parameters, α (risk attitude) and β (ambiguity attitude), converged to their respective ground truth most rapidly under the ADO condition. On the other hand, the inverse temperature parameter (γ) showed little, if any, convergence for any of the designs, probably due to the relatively small sample size (i.e., 60).

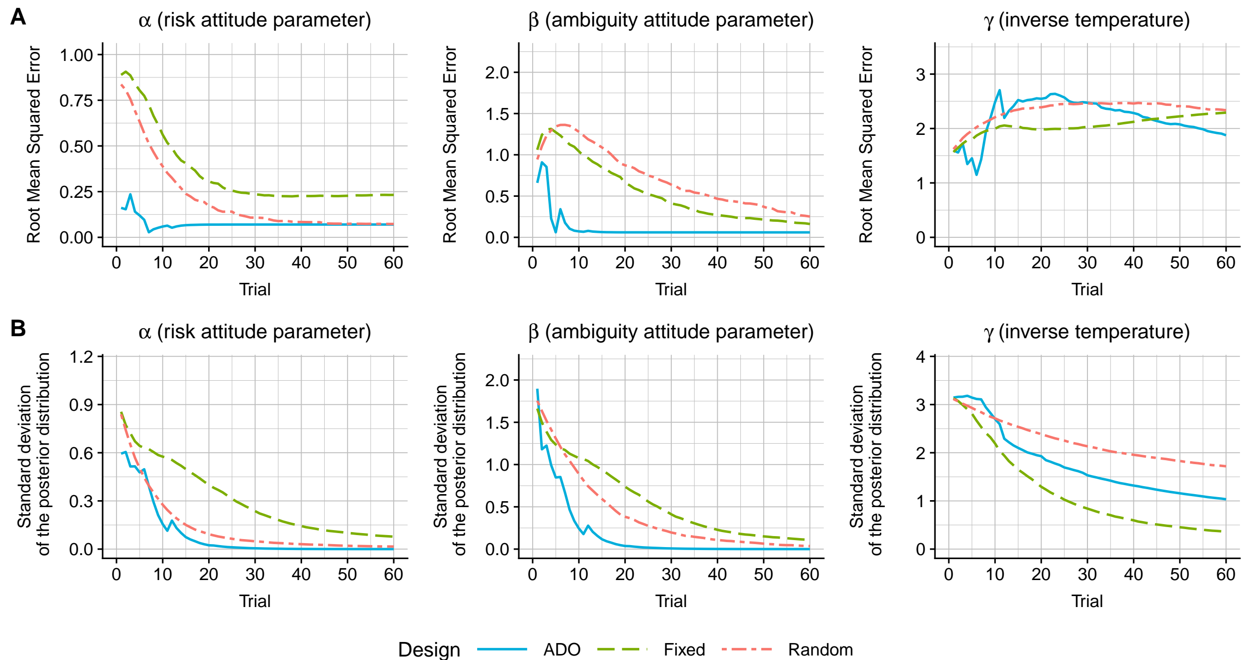

Figure 9: Comparison of ADO, fixed, and random designs in the simulation of the choice under risk and ambiguity task. The fixed design was pre-determined according to Levy et al. (2010). Simulations were conducted using the linear model with parameter values of $\alpha = 0.66$, $\beta = 0.67$, and $\gamma = 3.5$. Three designed are compared with root mean squared errors (RMSE; Panel A) and standard deviations of the posterior distribution (Panel B). Each curve represents an average across 1,000 independent simulation runs.

635

5 Integrating ADOpy with experiments

In this section we describe how to integrate ADOpy into a third-party Python package for conducting psychological experiments, such as *PsychoPy* (Peirce, 2007, 2009), *OpenSesame* (Mathôt et al., 2012), or *Expyriment* (Krause and Lindemann, 2014). Integrating ADOpy into such a experimentation package is accomplished following a two-step procedure described below.

640 First, users should create and initialize an ADOpy `Engine` object. This corresponds to the initialization step illustrated in Figure 3. Users can create their own task and model as described in Section 3 or use pre-implemented tasks and models in ADOpy (see Section 4). Note that the

number of design variables, model parameters, and the grid sizes affect the computation time, so users should ensure the appropriateness of their choice of grid sizes, for example, by running simulations as described in Section 3.6.

Second, users then add code for collecting an observation from a participant using a computed optimal design and updating the engine with collected responses on each trial. 'run_trial(design)' is an experimenter-created function for data collection with the given design values and then records a participant's response on a single trial:

```
def run_trial(design):
    ...  # Collect a response from the participant  using the given design
    return response  # Return the response from a participant
```

This function, 'run_trial(design)', can be used for both simulated and real data. Users can also can run run_trial(design) within a for-loop to conduct an ADO experiment in multiple trials as shown below:

```
for trial in range(NUM_TRIAL):
    design = engine.get_design()  # Design optimization
    response = run_trial(design)  # Experiment
    engine.update(design, response)  # Bayesian updating
```

Note that the three lines inside the for-loop correspond to the three steps in Figure 1.

Below are snippets from fully working programs that illustrate how to use ADOpy in the DD task; they are available for download from the example directory in https://github.com/adopy/adopy/.Users new to ADO should be able to run these two Psychopy programs without any modification of the code after ADOpy has been installed. The first is for a non-ADO experiment (dd_psychopy_non−ado.py), and the second is for its ADO counterpart (dd_psychopy_ado.py). A short description of each is provided, noting the differences between the two.

Non-ADO version In the non-ADO implementation, the staircase method is used to determine experimental designs. The staircase method runs 6 trials for each delay to estimate the indifference point of each delay (see Green and Myerson, 2004). While t_{SS} is fixed to 0, it starts with R_{SS} of \$400 and R_{LL} of \$800. If a participant chooses the SS option, R_{SS} is increased by 50%; if the participant chooses the LL option, R_{SS} is decreased by 50%. After five such trials, the program proceeds to another delay value. In this example, we include seven delays for the LL option (lines

261-262 in `dd_psychopy_non-ado.py`):

```
261 # Delays
262 delays = [1, 2, 4.3, 26, 52, 156, 520] # weeks
```

675

To run the DD task, we define a function `run_trial(design)` (lines 189-219) that conducts an experiment using a given design on a single trial. Note that `run_trial(design)` is identical for both ADO and non-ADO versions. See lines 325-341 in `dd_psychopy_non-ado.py` for the implementation of a non-ADO version:

680

```
325 # Shuffle the order of delay values
326 np.random.shuffle(delays)
327
328 # Run the main task
329 for trial in range(n_trial):
330     if trial % 6 == 0:  # Use new t_ll value on every 6 trials
331         design = {
332             't_ss': 0,
333             't_ll': delays[trial // 6],
334             'r_ss': 400,
335             'r_ll': 800
336         }
337     else:  # Adjust r_ss based on the previous response
338         design['r_ss'] *= 0.5 if response == 1 else 1.5
339
340     # Run a trial using the design
341     is_ll_on_left, key_left, response, rt = run_trial(design)
```

685

690

695

ADO version The ADO-based version is similar to the non-ADO one. We first need to load the ADOpy classes (lines 8-9 in `dd_psychopy_ado.py`), the DD task and the model of our choice (hyperbolic in this case). We could have chosen a different model or defined one ourselves and used it:

700

```
8 from adopy import Engine
9 from adopy.tasks.dd import TaskDD, ModelHyp
```

Then, for the initialization step, `Task`, `Model` and `Engine` objects should be initialized. As in Section 4.2, users can use the implemented task and models for the DD task as described below (lines 262-281 in `dd_psychopy_ado.py`):

```
262 # Create Task and Model objects for the DD task
263 task = TaskDD()
264 model = ModelHyp()
265
266 # Generate grid for design variables and model parameters
267 grid_design = {
268     't_ss': [0],
269     't_ll': [0.43, 0.714, 1, 2, 3, 4.3, 6.44, 8.6, 10.8, 12.9,
270              17.2, 21.5, 26, 52, 104, 156, 260, 520],
271     'r_ss': np.arange(12.5, 800, 12.5),  # [12.5, 25, ..., 787.5]
272     'r_ll': [800]
273 }
274
275 grid_param = {
276     'k': np.logspace(-5, 0, 50),
277     'tau': np.linspace(0, 5, 50)
278 }
279
280 # Initialize an engine
281 engine = Engine(task, model, grid_design, grid_param)
```

Once the engine above is created, the code to run the ADO-based version is actually simpler than the non-ADO version (lines 334-343 in `dd_psychopy_ado.py`):

```
334 # Run the main task
335 for trial in range(n_trial):
336     # Get a design from the ADOpy Engine
337     design = engine.get_design()
338
339     # Run a trial using the design
```

```
340      is_ll_on_left, key_left, response, rt = run_trial(design)
341
342      # Update the engine
343      engine.update(design, response)
```

6 Conclusion

ADOpy is a toolbox for optimizing design selection on each trial in real time so as to maximize the informativeness and efficiency of data collection. The package implements Bayesian adaptive parameter estimation for three behavioral tasks: psychometric function estimation, delay discounting, and choice under risk and ambiguity. Each task can be run in an ADO-based mode or a non-ADO-based mode (random, fixed, staircase depending on the task). Default parameter and design values can be used, or the user can customize these settings, including the number of trials, the parameter ranges, and the grid resolution (i.e., number of grid points on each parameter dimension). Furthermore, in addition to conducting an actual experiment with participants, the package can be used to run parameter recovery simulations to assess ADO's performance. Is it likely to be superior (i.e., more precise and efficient) to random and other (staircase, fixed) designs? By performing a comparison as described in the preceding section, a question like this one can be answered. Causes for unsatisfactory performance can be evaluated, such as altering grid resolution or the number of trials. More advanced users can conduct Bayesian sensitivity analysis on the choice of priors.

The need to tune ADO to a given experimental setup might make readers leery of the methodology. Shouldn't it be more robust and work flawlessly in any setting without such fussing? Such lofty expectations might one day be a reality, but it is typical of machine-learning methods to require parameter tuning to maximize performance. ADOpy's simulation mode is an easy and convenient way to explore how changes in the design and grid resolution alter ADO's performance. Experimenter-informed decisions about the properties of the design space will result in the greatest gains in an ADO experiment.

Use of ADOpy is not limited to the models that come with the package. Users are can define their own model using the Model class. Specification of the likelihood function is all that is required along with the parameters, including any changes to the design space, as mentioned above. For example, it would be straightforward to create ADO-based experiments for other behavioral tasks,

such as the balloon analog risk task (BART: Lejuez et al., 2002; Wallsten et al., 2005)) for assessing
765 risk-taking propensity.

The ADOpy package, as currently implemented, has several limitations. ADOpy cannot optimize the selection of design variables that are not expressed in the likelihood function of the model. For example, if a researcher is interested in learning how degree of distractibility (low or high level of background noise) impacts decision making, unless this construct were factored into
770 the model as a design variable, ADOpy would not optimize on this dimension. This limitation does not prevent ADO from being used by the researcher; it just means that that the experiment will not be optimized on that stimulus dimension.

Another limitation that users must be sensitive to is the memory demands of the algorithm. As discussed earlier, the algorithm creates a pre-computed look-up table of all possible discretized
775 combinations of the outcome variable, the parameters, and the design variables. For example, for 100 grid points defined on each outcome variable, three parameters, and three design variables, the total memory demand necessary to store the look-up table would be 10^{14} bytes ($= 100^{1+3+3}$), i.e., 100 terabytes, assuming one byte allotted for storing each data point. This is clearly well beyond what most desktops or servers can handle. In short, as the dimensionality of the ADO problem
780 increases linearly, the memory demand of the grid-based ADO algorithm grows exponentially, sooner or later hitting a hardware limitation. In other words, grid-based ADO does not scale up well, technically speaking. The good news is that there is a scalable algorithm that does not tax memory. It is known as sequential Monte Carlo (SMC) or particle filter in machine learning (Doucet et al., 2001; Andrieu et al., 2003; Cappe et al., 2007).

785 In conclusion, the increasing use of computational methods for analyzing and modeling data is improving how science is practiced. ADOPy is a novel and promising tool that has the potential to improve the quality of data-based inference in experiments. This is accomplished by exploiting the predictive precision of computational modeling in conjunction with the power of statistical and machine learning algorithms to perform better inference. It is our hope that ADOpy will empower
790 more researchers to harness this technology, one outcome of which should be more informative and efficient experiments that collectively accelerate advances in psychological science and beyond.

Acknowledgements

The research was supported by National Institute of Health Grant R01-MH093838 to M.A.P. and J.I.M, and the Basic Science Research Program through the National Research Foundation (NRF) of Korea funded by the Ministry of Science, ICT, & Future Planning (NRF-2018R1C1B3007313 and NRF-2018R1A4A1025891) to W.-Y.A. Portions of this paper are published in the *Proceedings of the 41st Annual Meeting of the Cognitive Science Society* held in July, 2019.

Bibliography

Ahn, W.-Y., Gu, H., Shen, Y., Haines, N., Hahn, H., Teater, J. E., Myung, J. I., and Pitt, M. A. (2019). Rapid, precise, and reliable phenotyping of delay discounting using a Bayesian learning algorithm. *bioRxiv*.

Ahn, W.-Y., Haines, N., and Zhang, L. (2017). Revealing neurocomputational mechanisms of reinforcement learning and decision-making with the hbayesdm package. *Computational Psychiatry*, 1:24–57.

Amzal, B., Bois, F. Y., Parent, E., and Robert, C. P. (2006). Bayesian-optimal design via interacting particle systems. *Journal of the American Statistical Association*, 101(474):773–785.

Andrieu, C., DeFreitas, N., Doucet, A., and Jornan, M. J. (2003). An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43.

Aranovich, G. J., Cavagnaro, D. R., Pitt, M. A., Myung, J. I., and Mathews, C. A. (2017). A model-based analysis of decision making under risk in obsessive-compulsive and hoarding disorders. *Journal of Psychiatric Research*, 90:126–132.

Atkinson, A. and Donev, A. (1992). *Optimum Experimental Designs*. Oxford University Press.

Berger, M. J. (1984). Adaptive meth refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512.

Bickel, W. K. (2015). Discounting of delayed rewards as an endophenotype. *Biological psychiatry*, 77(10):846–847.

Cappe, O., Godsill, S. J., and Moulines, E. (2007). An overview of existing methods and recent advances in sequential Monte Carlo. *Proceedings of the IEEE*, 95(5):899–924.

Cavagnaro, D. R., Aranovich, G. J., McClure, S. M., Pitt, M. A., and Myung, J. I. (2016). On
820 the functional form of temporal discounting: An optimized adaptive test. *Journal of Risk & Uncertainty*, 52:233–254.

Cavagnaro, D. R., Gonzalez, R., Myung, J. I., and Pitt, M. A. (2013a). Optimal decision stimuli for risky choice experiments: An adaptive approach. *Management Science*, 59(2):358–375.

Cavagnaro, D. R., Myung, J. I., Pitt, M. A., and Kujala, J. V. (2010). Adaptive design optimization:
825 A mutual information based approach to model discrimination in cognitive science. *Neural Computation*, 22(4):887–905.

Cavagnaro, D. R., Pitt, M. A., Gonzalez, R., and Myung, J. I. (2013b). Discriminating among probability weighting functions using adaptive design optimization. *Journal of Risk and Uncertainty*, 47:255–289.

830 Cavagnaro, D. R., Pitt, M. A., and Myung, J. I. (2011). Model discrimination through adaptive experimentation. *Psychonomic Bulletin & Review*, 18(1):204–210.

Chaloner, K. and Verdinelli, I. (1995). Bayesian experimental design: A review. *Statistical Science*, 10(3):273–304.

Cohn, D., Atlas, L., and Ladner, R. (1994). Improving generalization with active learning. *Machine*
835 *Learning*, 15(2):201–221.

Cornsweet, T. N. (1962). The staircase-method in psychophysics. *The American Journal of Psychology*, 75(3):485–491.

Cover, T. M. and Thomas, J. A. (1991). *Elements of Information Theory*. John Wiley & Sons, Inc., Hoboken, New Jersey.

840 DiMattina, C. and Zhang, K. (2008). How optimal stimuli for sensory neurons are constrained by network architecture. *Neural Computation*, 20:668–708.

DiMattina, C. and Zhang, K. (2011). Active data collection for efficient estimation and comparison of nonlinear neural models. *Neural Computation*, 23:2242–2288.

Doucet, A., de Freitas, N., and Gordon, N. (2001). *Sequential Monte Carlo Methods in Practice*.
845 Springer.

Ebert, J. E. and Prelec, D. (2007). The fragility of time: Time-insensitivity and valuation of the near and far future. *Management science*, 53(9):1423–1438.

Farrell, S. and Lewandowsky, S. (2018). *Computational Modeling of Cognition and Behavior*. Cambridge University Press, Cambridge, UK.

850 Feeny, S., Kaiser, P. K., and Thomas, J. P. (1966). An analysis of data gathered by the staircase-method. *The American Journal of Psychology*, 79(4):652–654.

Garcia-Perez, M. A. (1998). Forced-choice staircases with fixed step sizes: asymptotic and small-samples properties. *Vision Research*, 38:1861–1881.

Green, L. and Myerson, J. (2004). A discounting framework for choice with delayed and probabilistic
855 rewards. *Psychological Bulletin*, 130:769–792.

Gu, H., Kim, W., Hou, F., Lesmes, L., Pitt, M. A., Lu, Z.-L., and Myung, J. I. (2016). A hierarchical Bayesian approach to adaptive vision testing: A case study with the contrast sensitivity function. *Journal of Vision*, 16(6):15, 1–17.

Hou, F., Lesmes, L., Kim, W., Gu, H., Pitt, M. A., Myung, J. I., and Lu, Z.-L. (2016). Evaluating
860 the performance of the quick CSF method in detecting contrast sensitivity function changes. *Journal of Vision*, 16(6):18, 1–19.

Hsu, M., Bhatt, M., Adolphs, R., Tranel, D., and Camerer, C. F. (2005). Neural systems responding to degrees of uncertainty in human decision-making. *Science*, 310(5754):1680–1683.

King-Smith, P. E., Grigsby, S. S., Vingrys, A. J., Benes, S. C., and Supowit, A. (1994). Efficient
865 and unbiased modifications of the quest threshold method: Theory, simulations, experimental evaluation and practical implementation. *Vision Research*, 34:885–912.

Kontsevich, L. L. and Tyler, C. W. (1999). Bayesian adaptive estimation of psychometric slope and threshold. *Vision Research*, 39:2729–2737.

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

Kujala, J. V. and Lukka, T. J. (2006). Bayesian adaptive estimation: The next dimension. *Journal of Mathematical Psychology*, 50(4):369–389.

Laibson, D. (1997). Golden eggs and hyperbolic discounting. *The Quarterly Journal of Economics*, 112(2):443–478.

875 Lee, M. D. and Wagenmakers, E.-J. (2014). *Bayesian Cognitive Modeling: A Practical Course*. Cambridge University Press, Cambridge, U.K.

Lejuez, C. W., Read, J. P., Kahler, C. W., Ramsey, J. B., Stuart, G. L., and et. al. (2002). Evaluation of a behavioral measure of risk-taking: the balloon analogue risk task (bart). *Journal of Experimental Psychology: Applied*, 8(2):75–85.

880 Lesmes, L. A., Jeon, S.-T., Lu, Z.-L., and Dosher, B. A. (2006). Bayesian adaptive estimation of threshold versus contrast external noise functions: The quick *TvC* method. *Vision Research*, 46:3160–3176.

Levy, I., Snell, J., Nelson, A. J., Rustichini, A., and Glimcher, P. W. (2010). Neural representation of subjective value under risk and ambiguity. *Journal of Neurophysiology*, 103:1036–2047.

885 Lewi, J., Butera, R., and Paninski, L. (2009). Sequential optimal design of neurophysiology experiments. *Neural Computation*, 21:619–687.

Lindley, D. V. (1956). On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, 27(4):986–1005.

Lorenz, R., Pio-Monti, R., Violante, I. R., Anagnostopoulos, C., Faisal, A. A., Montana, G., and
890 Leech, R. (2016). The automatic neuroscientist: a framework for optimizing experimental design with closed-loop real-time fmri. *Neuroimage*, 129:320–334.

Mathôt, S., Schreij, D., and Theeuwes, J. (2012). Opensesame: An open-source, graphical experiment builder for the social sciences. *Behavior research methods*, 44(2):314–324.

Mazur, J. E. (1987). An adjusting procedure for studying delayed reinforcement. *Commons, ML.;*
895 *Mazur, JE.; Nevin, JA*, pages 55–73.

McClure, S. M., Ericson, K. M., Laibson, D. I., Loewenstein, G., and Cohen, J. D. (2007). Time discounting for primary rewards. *Journal of neuroscience*, 27(21):5796–5804.

Müller, P. (1999). Simulation-based optimal design. In Berger, J. O., Dawid, A. P., and Smith, A. F. M., editors, *Bayesian Statistics*, volume 6, pages 459–474, Oxford, UK. Oxford University
900 Press.

Müller, P., Sanso, B., and De Iorio, M. (2004). Optimal Bayesian design by inhomogeneous Markov chain simulation. *Journal of the American Statistical Association*, 99(467):788–798.

Myung, I. J. (2003). Tutorial on maximum likelihood estimation. *Journal of Mathematical Psychology*, 47:90–100.

905 Myung, J. I., Cavagnaro, D. R., and Pitt, M. A. (2013). A tutorial on adaptive design optimization. *Journal of Mathematical Psychology*, 57:53–67.

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 neuroinfor-*
910 *matics*, 2:10.

Rose, R. M., Teller, D. Y., and Rendleman, P. (1970). Statistical properties of staircase estimates. *Perception & Psychophysics*, 8(4):199–204.

Samuelson, P. A. (1937). A note on measurement of utility. *The review of economic studies*, 4(2):155–161.

915 Settles, B. (2009). Active learning literature survey. *University of Wisconsin-Madison Computer Sciences Technical Report TR1648 (http://digital.library.wisc.edu/1793/60660).*

Van-DenBos, W. and McClure, S. E. (2013). Towards a general model of temporal discounting. *Journal of the Experimental Analysis of Behavior*, 99:58–73.

Vandekerckhove, J., Rouder, J. N., and Krushke, J. K. (2018). Editorial: Bayesian methods for
920 advancing psychological science. *Psychonomic Bulletin & Review*, 25:1–4.

Vincent, B. T. (2016). Hierarchical Bayesian estimation and hypothesis testing for delay discounting tasks. *Behavior Research Methods*, 48:1608–1620.

Wallsten, T. S., Pleskac, T. J., and Lejuez, C. W. (2005). Modeling behavior in a clinically diagnostic sequential risk-taking task. *Psychological Review*, 112(4):862–880.

925 Watson, A. B. and Pelli, D. G. (1983). Quest: A Bayesian adaptive psychometric method. *Perception & Psychophysics*, 33(2):113–120.

Wichmann, F. A. and Hill, N. J. (2001). The psychometric function: I. fitting, sampling, and goodness of fit. *Perception & Psychophysics*, 63(8):1293–1313.

Appendices

A Defining Grids for Delay Discounting Task

As the first example, suppose that the delay discounting task has two constraints on its designs: the delay of SS option should be smaller than that of LL option (`t_ss < t_ll`), and the amount of reward of SS option should be smaller than that of LL option (`r_ss < r_ll`). Considering seven delays (i.e., right now, two weeks, a month, six months, a year, three years, and ten years) and 79 possible rewards (from $12.5 to $787.5 with an increment of $12.5), users can make a grid for design variables by executing the following lines:

```
# Initialize the possible values of delay and reward variables
tval = [0, 2, 4.3, 26, 52, 104, 520]  # Delays in a weekly unit
rval = np.arange(12.5, 800, 12.5)  # [12.5, 25, ..., 775, 787.5] as reward values

# Make a 2d matrix with rows of [t_ss, t_ll]
t_joint = []
for t_ss in tval:
    for t_ll in tval:
        if t_ss < t_ll:
            t_joint.append([t_ss, t_ll])
t_joint = np.array(t_joint)

# Make a 2d matrix with rows of [r_ss, r_ll]
r_joint = []
for r_ss in rval:
    for r_ll in rval:
        if r_ss < r_ll:
            r_joint.append([r_ss, r_ll])
r_joint = np.array(r_joint)

grid_design = {('t_ss', 't_ll'): t_joint, ('r_ss', 'r_ll'): r_joint}
```

As an another example, if users want to use the amount of reward of the SS option (`r_ss`) and

the delay of the LL option (`t_ll`) while fixing `t_ss` to 0 and `r_ll` to $800, define a grid as shown below:

```
grid_design = {
    # t_ss: [Now]
    't_ss': [0],
    # t_ll: [2 weeks, 1 month, 6 months,
    #        1 year, 2 years, 10 years] in a weekly unit
    't_ll': [2, 4.3, 26, 52, 104, 520],
    # r_ss: [$12.5, $25, ..., $775, $787.5]
    'r_ss': np.arange(12.5, 800, 12.5),
    # r_ll: $800
    'r_ll': [800]
}
```

For model parameters, users should define a grid object containing grid points on a proper range for each parameter. For example, a grid for the hyperbolic model (Mazur, 1987) with two parameters (k and τ) can be defined as below:

```
grid_param = {
    # k: 20 points on [10^-5, 1] in a log scale
    'k': np.logspace(-5, 0, 20),
    # tau: 20 points on [0, 5] in a linear scale
    'tau': np.linspace(0, 5, 20)
}
```

B ADOpy Simulations

B.1 Psychometric function estimation

Simulations for psychometric function estimation were conducted for a simple 2-alternative forced choice (2AFC) task with one design variable. With an assumption that the psychometric function has a logistic function shape, we ran 1,000 simulations for three designs: (a) ADO design, (b) staircase design, and (c) randomly chosen design. For each simulation, responses were simulated for a total of 60 trials, using `Task2AFC` and `ModelLogistic` in the module `adopy.tasks.psi`.

Simulated responses were generated with true parameter values of threshold $\alpha = 20$, slope $\beta = 1.5$, guess rate $\gamma = 0.5$, and lapse rate $\delta = 0.04$. The simulation for psychometric function estimation used 100 grid points for the design variable (`stimulus`) and two model parameters (`threshold` and `slope`) each, and the guess and lapse rates were fixed to 0.5 and 0.04, respectively. The grid settings were given as follows:

Design variable

- `stimulus`: 100 grid points from $20 \log_{10} 0.05$ to $20 \log_{10} 400$ in a log scale.

Model parameters

- `threshold`: 100 grid points from $20 \log_{10} 0.1$ to $20 \log_{10} 200$ in a log scale.
- `slope`: 100 grid points from 0 to 10 in a linear scale.
- `guess_rate`: fixed to 0.5.
- `lapse_rate`: fixed to 0.04.

B.2 Delay discounting task

Assuming the hyperbolic model, simulations for the delay discounting (DD) task were conducted using `TaskDD` and `ModelHyp` in the module `adopy.tasks.dd`. We compared three designs: (a) ADO design, (b) staircase design, and (c) randomly chosen design. The staircase method runs 6 trials for each delay to estimate the discounting rate. While t_{SS} is fixed to 0, it starts with R_{SS} of \$400 and R_{LL} of \$800. If a participant chooses the SS option, the staircase method increases R_{SS} by 50%; if the participant chooses the LL option, it decreases R_{SS} by 50%. After repeating this 5 times, it proceeds to another delay value.

One thousand independent simulations were performed for each design condition, each for a total of 108 trials. Simulated data were generated using the true parameter values of $k = 0.12$ and $\tau = 1.5$. Grid resolutions used for the simulations were as follows:

Design variables

- `t_ss`: fixed to 0, which means 'right now'.
- `t_ll`: 18 delays (3 days, 5 days, 1 week, 2 weeks, 3 weeks, 1 month, 6 weeks, 2 months, 10 weeks, 3 months, 4 months, 5 months, 6 months, 1 year, 2 years, 3 years, 5 years, 10 years) in a unit of a week.

– `r_ss`: 63 points from \$12.5 to \$787.5 with an increment of \$12.5.

– `r_ll`: fixed to \$800.

Model parameters

– `k` (discounting rate): 20 grid points from 10^{-5} to 1 in a log scale.

1020 – `tau` (inverse temperature): 20 grid points from 0 to 5 in a linear scale.

What follows below are Python code examples to run the simulations using ADOpy.

As for grid settings, note that we manipulated just two design variables of the DD task: the amount of reward in the SS option (R_{SS}, `r_ss`) and the delay in the LL option (t_{LL}, `t_ll`). The Other two design variables, R_{LL} (`r_ll`) and t_{SS} (`t_ss`), were fixed to \$800 and 0, respectively. In
1025 short, the following code blocks show the grid settings used for the design variables and the two parameters:

```
grid_design = {
    't_ss': [0],
    't_ll': [0.43, 0.714, 1, 2, 3, 4.3, 6.44, 8.6, 10.8, 12.9,
             17.2, 21.5, 26, 52, 104, 156, 260, 520]  # 18 delays
    'r_ss': np.arange(12.5, 800, 12.5),
    'r_ll': [800]
}

grid_param = {
    'k': np.logspace(-5, 0, 20),  # 20 points on [10^-5, 1] in a log scale
    'tau': np.linspace(0, 5, 20)  # 20 points on [0, 5] in a linear scale
}
```

Now, to initialize an engine, we can use either the basic `Engine` class or `EngineDD` provided in the
1040 module `adopy.tasks.dd`. Below are some sample codes for initializing the engine and computing designs:

```
from adopy.tasks.dd import EngineDD
engine = EngineDD(model, grid_design, grid_param)
# Same to Engine(task, model, grid_design, grid_param)
```

1045

```
engine.get_design('optimal')    # Returns the optimal design.
engine.get_design('random')     # Returns a randomly chosen design.
```

The next step is to generate simulated responses. For this. we use `scipy.stats.bernoulli` to write a likelihood function to randomly sample a binary response, as shown below:

```
1050  from scipy.stats import bernoulli

      def get_simulated_response(model, design):
          # Compute a probability to respond positively.
          p_obs = model.compute(t_ss=design['t_ss'], t_ll=design['t_ll'],
1055                            r_ss=design['r_ss'], r_ll=design['r_ll'],
                                k=0.12, tau=1.5)

          # Sample a binary response using Bernoulli distribution.
          return bernoulli.rvs(p_obs)
```

1060 Lastly, putting all these together, here we provide example code to run a total of 42 simulation trials under the ADO condition:

```
      num_trials = 42  # number of trials to simulate
      design_type = 'optimal'  # or 'random'

1065  for i in range(num_trials):
          # Compute a design for the current trial
          design = engine.get_design(design_type)
          # Get a simulated response using the design
          response = get_simulated_response(model, design)
1070      # Update posterior in the engine
          engine.update(design, response)
          # Print the trial number and posterior means and standard deviations
          print('Trial', i + 1, '-', engine.post_mean, '/', engine.post_sd)
```

B.3 Choice under risk and ambiguity task

1075 In simulating this CRA task, we assume the linear model and considered three methods for experimental designs in the simulation study: (a) ADO design, (b) 'fixed' design of Levy et al. (2010), and (c) random design.

The fixed design was set as follow. The the reward of the fixed option (R_F) to 5 and the rewards of the variable option (R_V) to 5, 9.5, 18, 34, 65. In risky trials, ambiguity (A_V) is set to 0 but the
1080 probability of winning for the variable option (P_V) is chosen among 0.13, 0.25, and 0.38. On the other hand, in ambiguous trials, the probability p_V is set to 0.5 but the ambiguity A_V is chosen from 0.25, 0.5, and 0.75. The total number of combinations is 30: 15 of which are for risky trials, and the rest of which are for ambiguous trials.

Grid settings for the four design variables and the three model parameters were set as follows:

1085 **Design variables**

- `p_var` and `a_var` in risky trials: there are 9 probabilities to win for `p_var` (0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45), and `a_var` was fixed to 0.
- `p_var` and `a_var` in ambiguous trials: there are 6 levels of ambiguity for `a_var` (0.125, 0.25, 0.375, 0.5, 0.625, 0.75), and `p_var` was fixed to 0.5.
- 1090 `r_var` and `r_fix`: based on 10 reward values (10, 15, 21, 31, 45, 66, 97, 141, 206, 300), rewards pairs such that `r_var > r_fix` were used.

Model parameters

- `alpha` (risk attitude parameter): 11 grid points from 0 to 3 in a linear scale.
- `beta` (ambiguity attitude parameter): 11 grid points from -3 to 3 in a linear scale.
- 1095 `gamma` (inverse temperature): 11 grid points from 0 to 5 in a linear scale.

One thousand independent simulations were performed for each design condition, each for a total of 60 trials, with 30 risky and 30 ambiguous trials. Simulated data were generated using the true parameter values of $\alpha = 0.66$, $\beta = 0.67$, and $\gamma = 3.5$ based on Levy et al. (2010). What follows below are Python code examples to run simulations using ADOpy.

1100 Firstly, the code blocks below were used to define the grid settings for the design variables and the model parameters:

```
# p_var & a_var for risky & ambiguous trials
pval = [.05, .10, .15, .20, .25, .30, .35, .40, .45]
aval = [.125, .25, .375, .5, .625, .75]

# risky trials: a_var fixed to 0
pa_risky = [[p, 0] for p in pval]
# ambiguous trials: p_var fixed to 0.5
pa_ambig = [[0.5, a] for a in aval]
pr_am = np.array(pa_risky + pa_ambig)

# r_var & r_fix while r_var > r_fix
rval = [10, 15, 21, 31, 45, 66, 97, 141, 206, 300]
rewards = []
for r_var in rval:
    for r_fix in rval:
        if r_var > r_fix:
            rewards.append([r_var, r_fix])
rewards = np.array(rewards)

grid_design = {('p_var', 'a_var'): pr_am, ('r_var', 'r_fix'): rewards}

grid_param = {
    'alpha': np.linspace(0, 3, 11),
    'beta': np.linspace(-3, 3, 11),
    'gamma': np.linspace(0, 5, 11)
}
```

Secondly, to initialize an engine, we have a choice between the basic `Engine` class and `EngineCRA` provided in the module `adopy.tasks.cra`. Following codes are for initializing the engine and computing designs:

```
from adopy.tasks.cra import EngineCRA
engine = EngineCRA(model, grid_design, grid_param)
# Same to Engine(task, model, grid_design, grid_param)
```

```
engine.get_design('optimal')    # Returns the optimal design.
engine.get_design('random')     # Returns a randomly chosen design.
```

Now, we define a likelihood function to generate simulated binary responses by using `scipy.stats.bernoulli`, as shown below:

```
from scipy.stats import bernoulli


def get_simulated_response(model, design):
    # Compute a probability to respond positively.
    p_obs = model.compute(p_var=design['p_var'], a_var=design['a_var'],
                          r_var=design['r_var'], r_fix=design['r_fix'],
                          alpha=0.66, beta=0.67, gamma=3.5)

    # Sample a binary response using Bernoulli distribution.
    return bernoulli.rvs(p_obs)
```

By putting these together, below we include example code to run a total of 60 simulation trials of the CRA task under the ADO condition:

```
num_trials = 60  # number of trials to simulate
design_type = 'optimal'  # or 'random'

for i in range(num_trials):
    # Compute a design for the current trial
    design = engine.get_design(design_type)
    # Get a simulated response using the design
    response = get_simulated_response(model, design)
    # Update posterior in the engine
    engine.update(design, response)
    # Print the trial number and posterior means and standard deviations
    print('Trial', i + 1, '-', engine.post_mean, '/', engine.post_sd)
```