NLoed: Overview and Background

Nathan Braniff

January 2021

Contents

1 Introduction

Models used in systems biology are often non-linear, multi-dimensional, and dynamic. Models of this type are difficult to study and fit to data due to the computational cost in simulating them and the lack of analytical tools available for studying their behaviour. However, due to the nature of the biochemical systems of interest, there is often no recourse to model simplification and the applied modeler is forced to contend with an unwieldy model in order to capture the behaviour of interest. Fortunately, increasing availability of novel experimental techniques and high-quality numerical tools can provide some remedy [10]. Calibrating complex models is made more feasible with the availability of richer, more informative datasets. Biology has seen significant advances in experimental techniques including methods for real-time, single-cell, multivariate measurements in dynamic contexts [10]. Also recent decades have seen the community converge on a number of high-level modelling languages such as Python and MATLAB, as well as the emergence of a large variety of specialized computational tools for working with the numerical models that are common in systems biology [36, 20]. However as experimental interests also shift to studying more complex and dynamic systems-level behaviour *in vivo*, modelers will likely need to take a more active role in guiding experimentation. Numerical tools are needed that reverse the traditional flow of information which historically emanated from experimentalists, who collected the data, and was transferred to modelers, who fit models, made predictions, and theorized about mechanisms. Experimental design tools are valuable for systems-level experimentation, as they can take existing understanding of a system, described mathematically, and generate efficient experimental designs for understanding the systems behaviour. In this way, experimental design tools can help to strengthen the feedback between theory and experiment. By including mathematical modelling within the experimental loop, it is hoped that the community can improve the efficiency of data collection and the accuracy of model calibration for more complex systems [10]. This in turn will increase the reliability of model predictions, making models more useful for generating new hypotheses in natural systems and in engineering synthetic ones. To this end, this chapter describes the implementation of the Non-linear Optimal Experimental Design (NLOED) software package, which seeks to provide convenient OED tools, in an open-source Python module, for experimentalists interested in fitting non-linear systems biology models.

1.1 Past Works

There has been a growing interest in experimental design methods in systems biology (i.e. see works cited in [25, 13, 10]), including the previous works contained in this thesis [9, 12, 11]. All of these works have required a custom implementation of the experimental design algorithms in each case. These studies require many common numerical procedures including model simulation, fitting, sensitivity analysis, computing Fisher information matrices, and optimization over the design space. There are still only limited examples of studies that implement numerically designed experiments in the laboratory [5, 35], likely because implementing optimal design algorithms is quite involved, especially for groups that are already focusing on the practical aspects of experiments. Purpose-built software like NLOED can provide significant benefit to model builders and experimentalists, by making design and analysis of experiments for systems biology easier to perform without the need to re-implement algorithms or knowledge of experimental design theory. A modular framework combining many of the common elements of experimental design will make it easier to study optimal designs; study of which will inform modelers about the feasibility of model calibration and help guide experimentalists to improve the efficiency of data collection.

Interest in experimental design spans a number of related fields in statistics [4], chemical engineering [18], control theory [27], pharmacokinetics/dynamics (PKPD) [28] as well as systems biology. Existing software tools have been developed across these various fields. Statistics has a long history of studying optimal design and many established software suites such as SAS and R have experimental design algorithms available in them [19, 3]. R in particular provides access to a wide range of experimental design tools (see the DOE task view page at [19]). The vast majority of these packages are aimed at the traditional static regression models used for empirical modelling in statistics. While these tools are well established and maintained, SAS, R and other statistical languages are less commonly used in systems biology, where MATLAB and Python are the preferred languages for modelling dynamic systems. PKPD researchers have also been interested in optimizing experimental design in the pharmaceutical field, several notable experimental design packages have been developed within this field. These include the PopED package from Nyberg et al. (available in MATLAB) [29] and PFIM 3.0 from Bazzoli et al. (available in R) [7] among others thoroughly compared in later work by Nyberg et al. [28]. These tools are more focused on dynamic models then those in statistics, however they place special emphasis on approximating the FIM for mixed-models (models that include random coefficients that vary between subjects). This emphasis is warranted because subject-specific effects are a common and unavoidable occurrence in pharmaceutical research where animal and human subjects are necessary. Computing the FIM for models with these mixed sources of variability is difficult and these additional sources are not often of interest for systems biologists working with microorganisms. The modelling interests and experimental constraints of microbial systems biology are different enough from PKPD practitioners that a separate set of software tools is desirable. Bayesian design methods are of particular interest to those studying non-linear models, including those in microbial systems biology, as it allows modelers to include prior information on parameter uncertainty rather than relying on purely local analysis. An early example of a Bayesian design package was published by Clyde [14], however this was released in 1993 in the, now defunct, XLispStat language. However, Bayesian design has experienced some renewed interest in recent years; of special interest is the `aceBayes` package by Overstall and Woods [30], which implements Bayesian design optimization for a variety of models. Their very recent work has begun to address dynamic models of biology using similar approaches to that found in `aceBayes` and this work will be of considerable interest to systems biology practitioners going forward [31]. Despite these promising developments in Bayesian design software, Bayesian methods can be both theoretically complex and computationally intensive. For pragmatic reasons a dedicated optimal design package for systems biology, focusing on well established non-Bayesian optimal design methods, is desirable. An accessible package, written in a high-level language, using classic OED techniques will simplify the application of optimal design in systems biology and encourage wider adoption of design optimization.

1.2 Motivation and Objectives

In addition to challenges faced by other experimental design tools non-PKPD systems biology poses a range of unique challenges, especially with respect to developing a general use software package for the field. Biological systems often exhibit strong non-linearity, multi-stability and bifurcations [26]. Interesting dynamic behaviour can occur over a range of timescales, and steady-state relations are often described by implicit functions. Often the systems of interest have relatively small species counts resulting in significant stochastic effects, and replication can be challenging and take on multiple meanings [22]. Compounding this, biological systems often have a large number of separate species, many of which may be inaccessible to measurement. Data from biological experiments can come from a variety assay types, and often exhibit non-normal error distributions. In addition measurements may span multiple physical scales, from individual chemical species counts, to single-cell data, to bulk population measurements all within the same experiment. As such a single software tool will likely not address all systems biology applications equally well, but we have identified several key objectives for a system biology-focused software package like NLOED which are outlined below;

- Develop the package in an established modelling language for users in systems biology (in this case Python) and make the tool open source and accessible. In addition it is desirable that the package interfaces easily with other established numerical libraries (i.e. Numpy, Pandas, Matplotlib) so that users can easily import and export data and results.
- Support optimal design for both static and dynamic nonlinear models, including multi-input and multi-output models. This will allow researchers to study dynamic and steady-state behaviour for a wide variety of systems and experimental protocols within one tool.
- Accommodate non-normal observation distributions (i.e. Poisson, binomial and log-normal etc.) and allow for a variety of observation distributions within the same model. Such heterogeneous distributions are often found in biological experiments with assays such as plate counts and fluorescent measurements.
- Provide not only the design algorithms, but also the fitting, evaluation, simulation and data sampling routines that are useful in an overall workflow for model building and analysis. These features allow the package to serve as a standalone environment for model development with a specific focus on experimental design. These features also ensures asymptotic tools used in design are paired with their appropriate fitting algorithms, decreasing the amount of statistical knowledge the user will need.

In addition to these design objectives, ongoing development in other numerical fields has provided some opportunity to update the experimental design tool set. Optimal experimental design for non-linear models relies heavily on sensitivity analysis in order to compute asymptotic objectives based on the Fisher information matrix. Optimal design also depends significantly on efficient optimization algorithms. We have taken advantage of maturing open-source numerical projects available for sensitivity analysis and optimization. Specifically, while most past software tools have relied on finite difference methods for computing model sensitivities, these can be numerically error prone and computationally inefficient [16]. Automatic differentiation (AD) provides a novel alternative, in which derivatives are computed through code generation created at run-time by specialized libraries, ensuring rapid computation with minimal numerical error [1]. AD tools have matured significantly in recent years, with higher-level tools emerging that are easier to use [1]. The NLOED package has been built on top of CasADi, an AD equipped software tool for rapid prototyping of optimal control problems [1]. CasADi is a natural choice for integration with a systems biology experimental design tool, as it is primarily built for the optimal control community, a field with similar formalisms and objectives to many systems biologists. In addition CasADi provides a convenient interface to specialized third-party open-source optimization frameworks such as IPOPT [39]. Our package therefore uses some of the most cutting-edge numerical tools, which is an improvement on past optimal design packages.

While many researchers would benefit from the above objectives, there remains a huge variety of users, models and design objectives that would identify with the umbrella-term ‘systems biology’. Given that each choice of a numerical method comes with certain limitations, necessary design trade-offs have to be made during software development. Below we outline specific design decision that, while narrowing the package’s use cases, are required to ensure a well-tested and cohesive software tool;

- The package is designed with a programmatic interface, rather than a graphical interface. While written in high-level language and in an object-oriented style, the NLOED package is designed to be modular so that users can generate a wide variety of experiments and simulation studies with the available code. Designs are not generated in a single out-of-the-box function call, and the user requires some familiarity with the package classes and functions. Helper-functions and interfaces can be built on top of the package in future development.

- The package supports deterministic model structures that can be implemented in CasADi's modelling framework, this includes both algebraic and differential equation-based models. The package supports a variety of common observation distribution types, including many from the exponential family.

- The package is primarily focused on local, asymptotic objectives base on the Fisher information matrix. This type of objective benefits most from the use the AD and optimization frameworks available. While not as robust to parameter uncertainty, optimization is comparatively fast for these objectives allowing for multiple local analyses in a comparatively short period of time.

- Optimization objectives in the package are primarily dedicated to parameter accuracy, specifically D-optimal and related objectives (see Chapter **??** for details). Model selection objectives have not been implemented in the initial release, although robustness to model uncertainty has been given some attention.

- The package relies on gradient-based non-linear programming via the IPOPT package. This type of optimization also benefits significantly from the AD tools available in CasADi. This approach is computationally efficient and scales well at the expense of local optima and some sensitivity to starting designs.

- In the initial release we have not implemented multi-shooting or collocation methods which would allow optimal design to scale to more complex and higher-dimensional dynamic experiments. While powerful, these methods can be highly specialized and require a more complex interface and dedicated code for specifying the optimization problem. However these methods could be included in future releases given that CasADi is an ideal tool on which to build related optimal control algorithms.

- In the package we solve a relaxed version of the design optimization problem and then provide the user with rounding methods to generate an exact, implementable design with a specific sample size. This is an optimize-then-discretize approach with respect to the replicate allocation. This approach benefits most from the available AD and optimization tools, and is fast and flexible at potentially some expense to optimality in small sample size designs.

To summarize, our design choices have focused on harnessing the underlying AD and optimization tools as effectively as possible to achieve a fast turnaround in modelling and design, at the expense of using local and relaxed approximations. This means users can quickly, and in some cases interactively, design multiple experiments and explore a range of cases. This is motivated by the philosophy that it is better to explore multiple scenarios quickly and approximately, rather than exactly optimize the "perfect" objective only once – after significant computational cost and delay. Furthermore, in the authors experience, any single optimal design, while valuable, always embodies trade-offs against competing possibilities. Significant value is often obtained in going through the process of experimental design as it allows the user to discover qualitative properties of good design structure for their system. The design process also allows the user to explore model identifiability under various scenarios. In pursuit of these qualitative and exploratory goals, a fast turnaround, with flexibility to examine multiple possibilities is an important attribute. In pursuit of flexibility, the package does not deliver a design in a single function call or through a pre-fixed graphical interface. We have also prioritized flexibility and modularity in the package interface, keeping to the ideal that it is better to show the user how to use a modular set of functions for many things, rather than force a user to solve a single type of problem with a single purpose built function. In a similar fashion, we have focused on supporting as broad a range of models and distributions as possible, believing its better to support many models adequately rather than only a niche set of model types with fully optimized numerics. This flexibility will ideally allow the user to explore a wide variety of models of the a given system (i.e. steady-state and dynamic behaviour) in the same software environment.

2 Workflow and Definitions

The NLOED package is written in Python 3 and can be used in Python scripts or interactively in the interpreter. The package consists primarily of two core classes; the `Model` class and the `Design` class. The `Model` class encodes information about model structure, observation distributions, and model-specific functions such as those used for fitting, simulation and model analysis. The `Design` class accepts models and other design information and can be used to optimize and output experimental designs.

Model equations passed into the `Model` class are created using CasADi symbolics and the CasADi `Function` class. Use of CasADi constructs enables much of the auto-generated functionality within the main two classes. Due to this reliance on CasADi, the core package does require some familiarity with model construction in the CasADi framework. However the modular and object-oriented nature of the core package classes means it is amenable to future extensions, such as wrapping the core classes in more beginner-friendly helper functions or GUI interfaces.

Output from both the `Model` and `Design` classes is primarily returned as Pandas and Numpy data structures. For examples experimental designs, model simulations and predictions are exported as Pandas dataframes. Data, candidate designs, and other user provided information are also passed into class functions as dataframes. This makes it easy to read in raw data from Excel and CSV files, via Pandas functions like `read_csv()`, `to_csv()`, `read_excel()` and `to_excel()`. The use of dataframes also makes it easy to plot model predictions and sample data using third-party plotting packages such as Matplotlib, as well as exporting output to a variety of formats for use in other tools such as MATLAB and R.

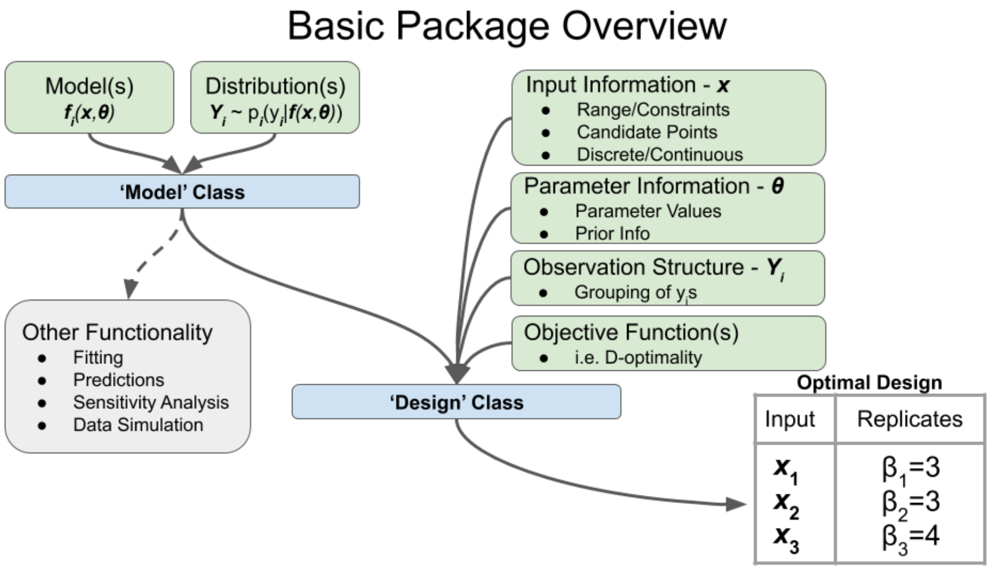


Figure 1: A diagram depicting the archetypal NLOED workflow including model creation using the `Model` class, and design creation using the `Design` class. Here x is the vector of model inputs describing the experimental conditions, y are the observation variables, θ are the unknown model parameters, $f(.)$ is the model function and $p(.)$ is the data distribution, see text for further description.

The majority of use cases for the package center around a simple archetypal workflow, as shown in Fig. 1: 1) *Model creation*; create a model using CasADi symbolics and the NLOED `Model` class, 2) *Design creation*; pass the `Model` instance, along with other design specifications, into the `Design` class constructor to create an optimal design. This simple pattern can be recycled in a variety of ways, i.e. to construct simulation studies, to compare designs across different models, or to generate design variations with a single model. The `Model` class can perform a range of other functions, such as fitting and simulation, which are useful in both real experimental work and simulation studies.

2.1 Model Definition

Before discussing the programmatic details of model creation in the NLOED package, we first define what mathematical structure NLOED accepts as a suitable model. NLOED models generally follow the model specification given in Chapter **??**, which is briefly reviewed here. The model connects the random observation variables, Y_i, with model input vector, x, which quantifies the experimental conditions.

Note that when considering the model declaration the user can ignore the j sub-scripting on the input vectors, x_j, as the number of unique experimental conditions is not addressed until the design phase. Therefore, when not discussing the data or a design, it is often convenient to suppress the jth index on x_j, $\eta_{i,j}$, $Y_{i,j}$ and $y_{i,j}$. Recall that each Y_i is a random variable representing a given type of observation. Each Y_i has a specific parametric distribution, $p_i(.)$, but this distribution's shape is conditional on the model inputs x and the unknown parameter vector θ. Therefore for each possible observation, Y_i, the connection between input and observation is mediated by two components: 1) the conditional observation distribution, $p_i(y_i|\eta_i)$ and 2) the deterministic model function $\eta_i = f_i(x,\theta)$. These two components are combined to create the overall model:

$$Y_i \sim p_i(y_i|\eta_i{=}f_i(x,\theta)). \tag{1}$$

The observation distributions, $p_i(.)$, in NLOED are all parametric distributions and they currently include; the normal, Poisson, binomial, log-normal, Bernoulli, exponential and gamma distributions. The deterministic model, $f_i(.)$, maps the experimental conditions x to the sampling statistics, η_i, of the observation distribution $p_i(.)$. The function $f_i(.)$ must be sufficiently smooth and implementable in CasADi symbolics. This permits a wide range of model types including models based on numerical integration. Recall the sampling statistics are the natural parameters of the parametric distribution $p_i(.)$. For example with the normal distribution the sampling statistics are the mean and variance, for the Poisson distribution the sampling statistic is the λ rate parameter, and for the Bernoulli distribution the sampling statistic is the probability of a success. Therefore in NLOED the model function $f_i(.)$ does not always predict the mean observational response, but rather predicts the appropriate sampling statistics of the random variable, Y_i, assigned to the given observation type. Specifying models in this way allows for much more flexibility in the types of experimental observations NLOED can handle. Each y_i represents a single-dimensional realization of the random observation variable Y_i. NLOED always assumes all observation variables Y_i are independent; this allows NLOED to easily accommodate a variety of distribution types rather than only supporting (possibly correlated) normally distributed data. When creating a model in NLOED the user must specify the vector dimensions of the parameters, θ, and of the model inputs, x, as well as the list of observation variables, $Y_i \in Y$. The user must also indicate the distribution type of $p_i(.)$ for each observation variable, Y_i, and they must provide the model function, $f_i(.)$ using CasADi's symbolic tools. The distribution, $p_i(.)$, and the function $f_i(.)$ for each Y_i are passed as a list so the user can add as many observations as is desired.

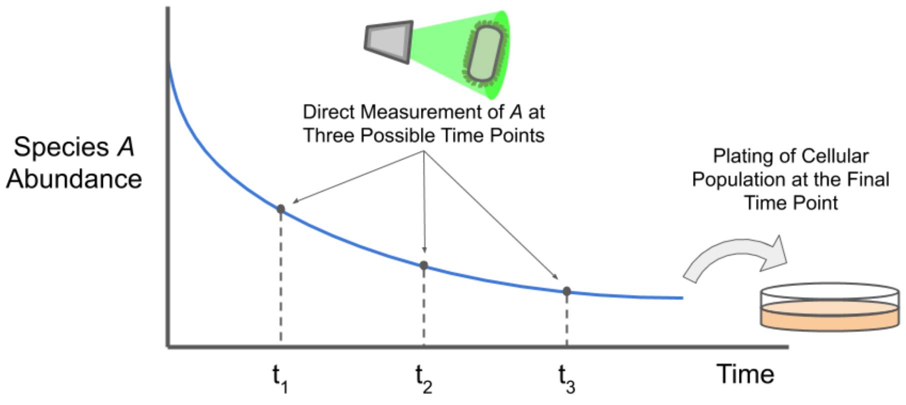

Species A Abundance
Direct Measurement of A at Three Possible Time Points
Plating of Cellular Population at the Final Time Point
t_1
t_2
t_3
Time

Figure 2: A figure depicting an example experimental scenario, involving a decaying cellular species A. The experiment includes both direct measurement of A and a plate count assessing A's potential for conferring resistance to a selective agent.

Specifying models in this way provides flexibility in mixing different observation types. As an example, assume synthesis of a biochemical species A has previously been induced in a cell culture and A's intra-cellular concentration is undergoing exponential decay. This scenario is depicted in Figure 2.

An experimenter wishes to take some replicate measurements of A at any of three possible time points spaced evenly throughout the first three hours after decay begins. Furthermore, assume that A confers some selection resistance and the experimenter will plate the culture on selective plates at the fourth hour and perform a plate count. Assume the experimenter controls the initial induction level as input x_1. In this proposed experiment there are four observation variables; Y_1, Y_2, and Y_3 are observations of A's concentration at one, two and three hours after induction. Observation Y_4 is the plate count from the fourth hour. We assume that $p_1(.)$ $p_2(.)$, and $p_3(.)$ are normally distributed with a known fixed variance of $\sigma^2 = 1$, and $p_4(.)$ is Poison distributed. We assume that the mean concentration of A can be modelled as

$$\mu(t) = \alpha x_1 e^{-\gamma t}, \tag{2}$$

we assume that the Poisson rate for the plate count is modeled as

$$\lambda(t) = \frac{\nu}{\frac{\kappa e^{\gamma t}}{\alpha x_1} + 1}. \tag{3}$$

The model can be defined by listing out the model components as:

$$\begin{aligned}
&x = [x_1], && \theta = [\alpha, \gamma, \nu, \kappa], && Y = [Y_1, Y_2, Y_3, Y_4]\\
&p_1(.) = \text{Normal}, && \eta_1 = [\mu(t{=}1), \sigma^2], && f_1(x,\theta) = \left[\alpha x_1 e^{-\gamma}, 1\right]\\
&p_2(.) = \text{Normal}, && \eta_2 = [\mu(t{=}2), \sigma^2], && f_2(x,\theta) = \left[\alpha x_1 e^{-2\gamma}, 1\right]\\
&p_3(.) = \text{Normal}, && \eta_3 = [\mu(t{=}3)\sigma^2], && f_3(x,\theta) = \left[\alpha x_1 e^{-3\gamma}, 1\right]\\
&p_4(.) = \text{Poisson}, && \eta_4 = [\lambda(t{=}4)], && f_4(x,\theta) = \left[\frac{\nu}{\frac{\kappa e^{4\gamma}}{\alpha x_1} + 1}\right].
\end{aligned} \tag{4}$$

Another possible model structure for this experimental scenario would involve treating the observation time for the A species assay as a second input dimension, x_2. In this scenario there would now be only two observation variables: Y_1 is the abundance of species A at any time point before the fourth hour, and Y_2 is the plate count performed at the fourth hour. Using this encoding the model structure would be

$$\begin{aligned}
&x = [x_1, x_2], && \theta = [\alpha, \gamma, \nu, \kappa], && Y = [Y_1, Y_2]\\
&p_1(.) = \text{Normal}, && \eta_1 = [\mu(t{=}x_2), \sigma^2], && f_1(x,\theta) = \left[\alpha x_1 e^{-\gamma x_2}, 1\right]\\
&p_2(.) = \text{Poisson}, && \eta_2 = [\lambda(t{=}4)], && f_2(x,\theta) = \left[\frac{\nu}{\frac{\kappa e^{4\gamma}}{\alpha x_1} + 1}\right].
\end{aligned} \tag{5}$$

This encoding provides greater flexibility in the sampling schedule of A which may be desirable or problematic depending on the experimental protocol. Choosing the appropriate encoding for a given experimental scenario depends on practical aspects of the protocol and numerical considerations for the optimization. Information and examples provided throughout this chapter will be useful for guiding this decision. Regardless of how the model is encoded, specification of the deterministic model components and the observation distributions fully defines an NLOED model. Receiving this information, the `Model` class constructor auto-generates a large variety of useful code including maximum likelihood fitting functions, parametric sensitivity functions, and model prediction and sampling functions as described below.

2.2 Design Definition

Designs in NLOED generally follow the design definition outlined in Chapter **??**, which is summarized here for convenience. Designs in NLOED must specify two main pieces of information, 1) the set of input conditions (support points) $x_j \in \mathcal{X}$ used in the experiment and 2) the number of replicate observations $\beta_{i,j} \in \mathcal{B}$ taken of each observation variable, Y_i, in each condition x_j. These two properties are common to most design formalisms [2, 17]. NLOED uses two different types of designs in its workflow; *relaxed designs* and*exact designs* (see Chapter **??** for further discussion). In exact designs, each $\beta_{i,j}$ is an integer and the overall sample size for the experiment, N_{Tot}, is the sum of the replicates in each observation and in each condition: $N_{Tot} = \sum_j^M \sum_i^N \beta_{i,j}$. Relaxed designs instead use real valued weights, $\xi_{i,j}$, between 0 and 1 to represent the replicate allocations to each observation and in each condition. The sum of the real valued weights is 1 and therefore relaxed designs do not have a sample size. Instead the weights represent the approximate fraction of an arbitrary sample size that should be allocated to each input

Table 1: An example of the structure used to represent an experimental design in the NLOED package. Here both the continuous sampling weights and the discrete sample numbers are shown. Relaxed designs use the continuous weights where as exact designs use the discrete numbers. Here the exact design has a sample size of $N = 20$.

Input Vector x_j	Observation Variable Y_i	Sample Weight $\xi_{i,j}$	Sample Number $\beta_{i,j}$
$x_1 = 0$	Y_1	0.1	2
$x_2 = 10$	Y_1	0.2	4
$x_2 = 10$	Y_3	0.1	2
$x_2 = 10$	Y_4	0.3	6
$x_3 = 5$	Y_4	0.3	6

and observation condition so that $N_{Tot}\xi_{i,j} \approx \beta_{i,j}$. This relation only holds approximately because the weights, $\xi_{i,j}$, may not share the desired sample size as a common denominator, or may even be irrational. NLOED uses both design types in its workflow because it generates designs using an optimize-then-discretize approach. Initially an optimal relaxed design is solved for, after which the relaxed design is then rounded to a discrete exact design with a desired sample size. Optimizing the exact design directly is difficult because, with $\beta_{i,j}$ restricted to discrete integers, the resulting optimization problem is a nonlinear integer programming problem which is difficult to solve efficiently [2, 17]. The term *relaxed* comes from the relaxation of the integer constraint. Relaxed designs can be viewed as a mathematical idealization of an optimal experiment with infinite data which means they can be difficult to implement with small finite sample sizes [2, 17].

While an approximation, the optimize-then-discretize approach naturally splits the workflow into two phases providing greater flexibility for the user. Specifically, the user's ultimate goal may be to choose their sample size to achieve certain accuracy objectives as efficiently as possible; they wish to take as few samples as they can but as many as they must. An optimal design will help to improve estimation accuracy, however the overall sample size is a more important factor. The accuracy of parameter estimates always improves monotonically with increasing sample size and the structure of the design only determines how much marginal improvement each additional measurement contributes. When optimizing an exact design directly the user must specify the sample size before they optimize, which means they do not know the structure or the pre-factor utility of the optimal design when the overall size of the experiment is chosen. Therefore, in direct optimization of the exact design, choosing a minimally sufficient sample size would require multiple runs of a difficult integer programming problem. By optimizing the relaxed design first, the user will achieve an approximately optimal design structure. They can then use efficient rounding procedures to investigate multiple sample sizes and choose the lowest possible experimental burden that achieves their desired accuracy level.

When using the NLOED package, relaxed designs are generally hidden from the user within a `Design` object, however they can be printed as a dataframe if desired. Exact designs are created from existing `Design` objects using a specific rounding function. The rounding function returns a dataframe containing an implementable exact design with the desired sample size. When representing design information in a dataframe, NLOED uses a three column data format. This format consists of 1) a list of input settings, x_j, 2) a list of observation variables, Y_i, indicating which output is measured at each setting, 3) a list of replicate allocations, either $\beta_{i,j}$ or $\xi_{i,j}$. In NLOED's design format, the input list can contain duplicates of support points, x_j, if multiple observation variables are measured at the same point. However each input-observation pair is unique and corresponds to a single replicate allocation. Table 1, shows how NLOED designs are specified for the example model discussed in the previous section, in its initial discrete time formulation. The first column contains a listing of input conditions at which observations are to be taken. The overall design here only has three unique support points: x_1, x_2, and x_3, however five input points are shown because x_2 has been repeated three times. The second column specifies which observation variables are to be observed at each of the input points to their left. In the third and fourth columns, two different types of replicate allocations are shown. The third column contains continuous sampling weights, $\xi_{i,j}$ which sum to one, and indicate the fraction of the total sample size to be made at the input-observation pair specified to the left. The fourth column lists the replication allocation as integer counts, $\beta_{i,j}$, and in this case they sum to a total sample size of 20. In any given design only one of the third and fourth columns is needed; which type is dependent on whether the design is a *relaxed design* or an *exact design*, as previously discussed.

3 The `Model` Class

The `Model` class encapsulates all of the information about the model structure and error distribution needed within the NLOED package. The class is designed to be a minimal but self-sufficient modeling environment providing functions for generating predictions, fitting parameters, simulating data, performing diagnostics like confidence region plots, and doing sensitivity analysis. Figure 3 gives a general overview of the process for creating and interacting with a `Model` class instance. The figure is divided into three main sections: green represents the parts of the process the user controls, blue represents the parts of the process that are automatic. At the top of the figure, the green section labeled *User-Provided Information* illustrates the data a user needs to prepare in order to create an NLOED model. The required information includes the deterministic parts of the model $f_i(x, \theta)$, encoded in a CasADi functions. It also includes the assumed observation distributions, $p_i(.)$ for each observation variable, as well as names for the model inputs and parameters. The user provides this information to the `Model` class constructor which creates the class instance.

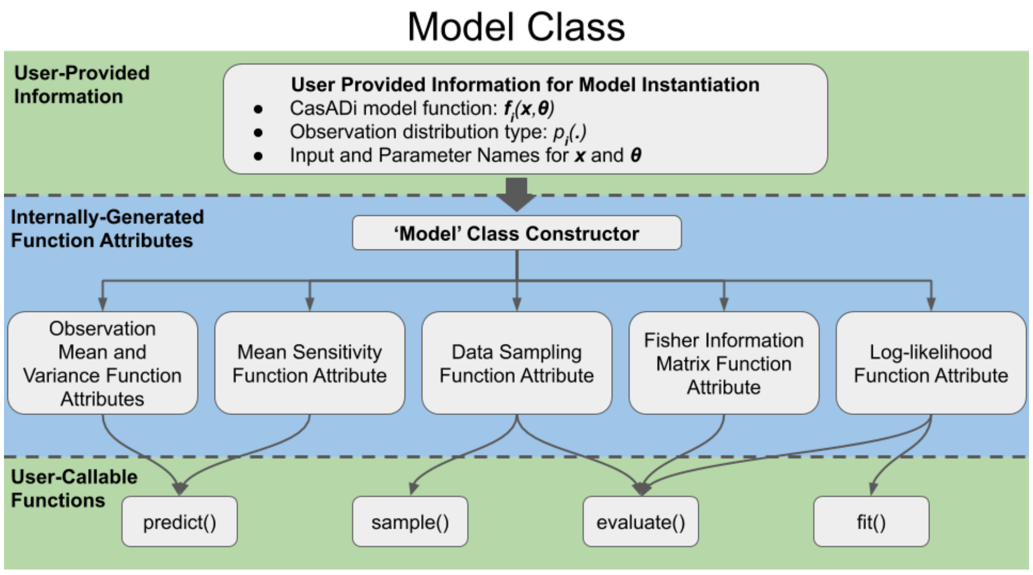

Model Class
User-Provided Information
User Provided Information for Model Instantiation
CasADi model function: $f_i(x,\theta)$
Observation distribution type: $p_i(.)$
Input and Parameter Names for x and θ
Internally-Generated Function Attributes
'Model' Class Constructor
Observation Mean and Variance Function Attributes
Mean Sensitivity Function Attribute
Data Sampling Function Attribute
Fisher Information Matrix Function Attribute
Log-likelihood Function Attribute
User-Callable Functions
predict()
sample()
evaluate()
fit()

Figure 3: A diagram depicting the user-provided arguments, internal function attributes and user-callable functions of the NLOED `Model` class.

Instantiation is normally done in a single line of code and occurs almost instantaneously; however it hides several automated processes that generate internal *function attributes* inside the class instance. These automatic processes are labelled as *Internally-Generated Function Attributes* in Figure 3 and are shown in the blue region in the middle of the figure. During instantiation, the `Model` class uses the CasADi function, $f_i(x, \theta)$, passed by the user, to auto-generate a variety of CasADi function attributes. These function attributes include atomic functions to compute the mean and variance of observations variables, the log-likelihood, the parametric sensitivities, the FIM and simulated data. These expressions are first generated symbolically and then encapsulated in a callable CasADi function stored within the `Model` class instance. The user will rarely interact with them directly, however they are used internally in many scenarios. These scenarios include when the `Model` instance is passed to the `Design` class for optimization or when the user invokes user-callable functions (discussed below) that use these auto-generated function attributes within their implementation.

The final section in Figure 3 is labelled *User-Callable Functions* and it highlights various high-level functions the user can call directly to perform specific tasks using the model. For example the user can ask for predictions from the model using the `predict()` function, or the user can use the `fit()` function to fit the model parameters to a provided dataset. These high-level user-callable functions are designed to provide a simple means to perform common tasks efficiently, while hiding the more mathematical function attributes within the class. Thus the user will generally instantiate a model at the beginning of a script or session and then use that model, and its available high-level functions, to perform various computations such as designing experiments, fitting data and predicting new behaviour. In the following three subsections we go into more detail on 1) how to create a model, 2) the exact role of the auto-generated function attributes, and 3) explaining the usage of the high-level user-callable functions for fitting, predicting, sampling and evaluating designs.

3.1 Creating a `Model` Object

To create a `Model` class instance, the user first needs to encode the deterministic part of their model in CasADi symbolics. Recall, the deterministic part of the model are the mathematical relations, $f_i(x, \theta)$, mapping the inputs, x, and parameters, θ, to the sampling statistics, η_i, of the observation variable, Y_i. Listing 1 demonstrates this process for a simple two-input, two-output, four-parameter model: 6,

$$\begin{aligned} \mu_{Y_1} &= \theta_0 + \theta_1 x_1 + \theta_3 x_1 x_2, \quad \sigma^2_{Y_1} = 0.1, && Y_1 \sim \text{Normal}(\mu_{Y_1}, \sigma^2_{Y_1}), \\ \lambda_{Y_2} &= \exp(\theta_0 + \theta_2 x_2 + \theta_3 x_1 x_2), && Y_2 \sim \text{Poisson}(\lambda_{Y_2}). \end{aligned} \tag{6}$$

This model has a simple linear regression model for Y_1 and a Poisson regression model for Y_2. To begin implementing this model in CasADi symbolics the `Model` class is imported from the NLOED package on line 1 of Listing 1. Lines 3 and 5 declare CasADi symbols for the input vector, x, as variable `x` and the

```
1  from nloed import Model
2  #create casadi symbols for the inputs
3  x = cs.SX.sym('x',2)
4  #create casadi symbols for the parameters
5  theta = cs.SX.sym('theta',4)
6  #define y1 sampling statistics; mean and variance
7  mean_y1 = theta[0] + theta[1]*x[0] + theta[3]*x[0]*x[1]
8  var_y1 = 0.1
9  #define y2 sampling statistics; mean and variance
10 rate_y2 = cs.exp(theta[0] + theta[2]*x[1] + theta[3]*x[0]*x[1])
11 #create a casadi function for y1 stats
12 eta_y1 = cs.vertcat(mean_y1, var_y1)
13 func_y1 = cs.Function('y1',[x,theta],[eta_y1])
14 #create a casadi function for y2 stats
15 eta_y2 = rate_y2
16 func_y2 = cs.Function('y2',[x,theta],[eta_y2])
```

Listing 1: An example encoding a mathematical model in CasADi symbolics and creating a CasADi function, prior to instantiating a `Model` instance.

parameter vector, θ, as variable `theta`. In lines 7 and 8 the mean, μ_{Y_1}, and variance, $\sigma^2_{Y_1}$, of observation variable, Y_1, are defined in terms of the inputs and parameters as `mean_y1` and `var_y1`. In line 10 the Poisson rate, λ_{Y_2}, for observation variable, Y_2, is likewise defined. In line 12, the sampling statistics vector, η_1, for observation Y_1 is defined as `eta_y1`. In line 13 a CasADi function, `func_y1`, mapping inputs and parameters to the sampling statistics for Y_1 is created; this CasADi function implements $f_1(x, \theta)$. Likewise in line 15, the sampling statistic, η_2, for observation variable Y_2 is defined as `eta_y2` and in line 16 a CasADi function `func_y2` corresponding to model function $f_2(x, \theta)$ is also created. From the perspective of NLOED, the CasADi functions implementing $f_i(x, \theta)$ are a computational black box and the user can use a wide range of CasADi modelling methods and different building blocks to construct these functions. Regardless of how they are built up, NLOED will auto-generate any required functionality internally. While this regression model is quite straightforward, symbolic construction can become more nuanced for dynamic models which is covered in the example section 5.

Having constructed the deterministic parts of the model as CasADi functions the user can now instantiate a `Model` object. The general call structure for the `Model` class constructor is;

```
Model(observ_list, input_names, param_names, options={})
```

The first argument, `observ_list`, is a list of tuples, each tuple corresponds to an independent observation variable. The first entry in each tuple is the CasADi function for the sampling statistics (i.e. $\eta_i = f_i(x, \theta)$), the second entry is the name of the distribution assigned to that observation (i.e the type for $p_i(.)$, see Listing 2). The list of observation tuples in `observ_list` can be extended to accommodate dozens of observation variables, which becomes important for dynamic models with multiple states and time points. The second and third arguments to the `Model` constructor, `param_names` and `input_names`, are an input name list and parameter name list. These are both lists of strings, naming the parameters and inputs according to the order they are given to the CasADi function. Input and parameter names are required in NLOED so that any returned dataframes or graphics will be labelled intelligibly. Note that the names for the observation variables, Y_i, are inherited from their corresponding CasADi function strings, assigned as the first argument when the functions are created. In Listing 1 the names `y1` and `y2` were assigned in the creation of the CasADi functions `func_y1` and `func_y2` respectively. These names will be passed into the `Model` constructor via the CasADi functions when they are added to the `observ_list` argument. Listing 2 demonstrate the creation of the required `Model` constructor arguments and the calling of the

```
17  #create observation list
18  observ_list = [(func_y1,'Normal'),(func_y2,'Poisson')]
19  #creat input name list
20  input_names = ['x1','x2']
21  #create parameter name list
22  parameter_names = ['Theta0','Theta1','Theta2','Theta3']
23  #create NLOED Model
24  model_object = Model(observ_list, input_names, parameter_names)
```

Listing 2: An example showing creation of a `Model` instance in NLOED.

`Model` class constructor. In line 18, an observation list, `observ_list`, is constructed with a tuple for each observation variable: y_1 and y_2. The first element of each tuple is a CasADi function: `func_y1` and `func_y2`. The second element of each tuple is the assigned distribution; here we pass `Normal` assigning the normal distribution to Y_1, and `Poisson` assigning the Poisson distribution to Y_2. The other distribution options include; `Lognormal`, `Bernoulli`, `Binomial`, `Exponential`, and `Gamma`. In lines 20 and 22, names are given in lists for the inputs and parameters. In line 24 the `Model` object is created with a call to the `Model` class constructor.

3.2 Function Attributes of the `Model` Class

After instantiating the model, several automatic process occur to create the previously mentioned *function attributes*. The CasADi functions the user passes in the `observ_list` argument implement the deterministic model components (i.e. $f_i(x, \theta)$). Having the deterministic model component expressed as a CasADi function provides a powerful tool for auto-generating new mathematical expressions because CasADi functions have a dual nature as both symbolic and numeric functions. If we pass numerical values to the CasADi function, it will compute a numerical output. On the other hand, if we pass CasADi symbols to a CasADi function, the returned object is a symbolic expression. This means we can use CasADi functions to generate new symbolic expression. The expressions can then be algebraically combined as well as differentiated (via AD) to compute other new quantities symbolically. Any newly generated symbolic expressions can also be encapsulated in a CasADi function so that they can be used to generate numerical results when required. CasADi functions therefore allow this interleaving of symbolic expression building and numeric function generation which is ideal for constructing the mathematical infrastructure needed by the `Model` class. During instantiation of a `Model` object, this process is used to auto-generate and store mathematical function attributes used for experimental design, fitting and other higher-level tasks. These function attributes are available to the user but are not intended to be called directly. However they may be of use if the user wishes to perform certain model analyses that are not available as a higher-level user-callable function. Below we give a brief overview of the auto-generated function attributes to give context for the package architecture and its internal capabilities. Note, there is a function attribute for each observation variable, Y_i, and the class fields where they are stored are therefore lists, indexed by the observation variable order. That is to say each field discussed below is a list of functions indexed by `[i]`, one function for each observation variable.

- **Sampling statistics** The CasADi functions passed by the user, implements $\eta_i = f_i(x, \theta)$ from the model definition. The original CasADi functions are therefore useful for predicting sampling statistics of each observation variable, conditional on the specified input and parameter values. The user provided CasADi functions are therefore stored as function attributes, one for each observation variable, in the `model[i](inputs,parameters)` field for use in other function attributes and user-callable functions.

- **Observation Mean and Variance** Given the conditional sampling statistics, η_i, of each observation variable, it is also possible to compute the observations, Y_i's, expected mean, $E[Y_i]$ and variance, $Var[Y_i]$, algebraically from η_i at any given input and parameter value. (Note, the mean and variance are not the same as the sampling statistics for some non-normal distributions). Function attributes to compute the observation mean and variance for each observation variable, given an input and parameter vector, are auto-generated and stored in the `model_mean[i](inputs,parameters)` and `model_variance[i](inputs,parameters)` fields.

- **Mean Sensitivity** Sensitivity functions for the mean observational responses, $E[Y_i]$, of each observation variable with respect to the parameters are also auto-generated. To do so we make use of CasADi's AD functionality via the `jacobian` function. The resulting function attributes returns a parametric sensitivity vector; the functions attributes are stored in the `model_sensitivity[i](inputs,parameters` field.

- **Observation Sampling** Observations generated by the model are assumed to come from the conditional distribution $p_i(y_i|f_i(x,\theta))$. Sampling from this distribution requires the user-passed CasADi function, (i.e $f_i(x,\theta)$), and the distributional information, (i.e. the type of $p_i(.)$). Given this information, the package can combine the correct distribution-specific random number generation from SciPy or Numpy and the `model[i](inputs,parameters)` function attributes to create a data sampling function attribute for each observation variable. These function attributes are stored in the `observation_sampler[i](inputs,parameters)` field. Each of these function attribute generates a single realization, y_i, of the given random observation variable, Y_i, conditioned on the input and parameter values passed.

- **Log-likelihood** The log-likelihood is needed for performing maximum likelihood fitting and is also useful in calibration diagnostics like profile likelihood intervals and traces (see the `fit()` function later in this section). The log-likelihood for an individual observation is defined as in Chapter **??**, with the indices indicated there, such that

$$l_i(y_i|x,\theta) = \log\left[p_i(y_i|\eta_i{=}f_i(x,\theta))\right]. \quad (7)$$

  Using the type of distribution for $p_i(.)$ and the user passed CasADi function for $f_i(x,\theta)$, the NLOED package will auto-generate a function for computing the log-probability of observing a specific observation value, y_i, given specified input and parameter values. The log-likelihood function attribute is stored in the class field named `loglik[i](observation,inputs,parameters)`, with one function for each observation variable.

- **Fisher Information Matrix** The Fisher information matrix is a key entity used in experimental design. In evaluating a new design, each potential input-observation pair contributes an individual Fisher information matrix to the overall sum for the experiment, see Chapter **??** for a full description. The Fisher information matrix for an individual input-observation pair is defined as

$$I_i(x,\theta) = E_{y_i}[\nabla_\theta l_i(\theta;y_i,x)\cdot\nabla_\theta l_i(\theta;y_i,x)^T]. \quad (8)$$

  The individual matrices, $I_i(x,\theta)$, are additive over input-observation pairs due to NLOED's assumption that all observations are independent. However NLOED uses the chain rule decomposition, discussed in Chapter **??**, to separate the FIM computation into a sensitivity vector and a distribution-specific elemental matrix such that

$$I_i(x,\theta) = \nabla_\theta f_i(x,\theta)\ \Psi(\eta_i{=}f_i(x,\theta))\ \nabla_\theta f_i(x,\theta)^T. \quad (9)$$

  Here $\nabla_\theta f_i(x,\theta)$ is the parametric sensitivity of the sampling statistics η_i. The sensitivity vector can be computed using CasADi's automatic differentiation functionality applied to the user-passed model functions, $f_i(x,\theta)$. The elemental matrix, $\Psi(\eta_i{=}f_i(x,\theta))$, is specific to each distribution [17], and can be computed algebraically from the sampling statistics, η_i. Using this property the package is able to auto-generate a function that can compute the individual Fisher information for a given input-observation pair at a candidate parameter vector instance. The FIM functions are stored in the `fisher_info_matrix[i](inputs,parameters)` field. These functions, like the other function attributes, are a CasADi functions and are therefore capable of both numeric and symbolic computation. The `Model` class uses this dual functionality both to compute evaluation metrics for candidate designs numerically and to construct symbolic expressions for the optimization problem solved in the `Design` class.

3.3 User-callable `Model` Functions

After instantiating a `Model` instance, the user will often follow the basic workflow outlined in Figure 1 and pass the model object into the `Design` class constructor to begin optimizing designs for the given system. However, the `Model` class provides a number of high-level user-callable functions that provide additional model building, calibration and diagnostic tools. Descriptions and usage examples for these functions are outlined below.

The `evaluate()` function The NLOED package exists to optimize experimental designs for various models and objectives. In many workflows, it is reasonable to assume the user may wish to evaluate the performance of many designs for a given model using some common quantitative metrics. While the `Design` class can return the objective for its optimal (relaxed) design, this objective is generally scaled for numerical convenience and is a single number (i.e. the determinant) computed from the total Fisher information matrix. Thus the objective alone may not be interpretable or useful for intuitive comparison. In addition, the user may often wish to evaluate non-optimized designs implemented in existing datasets or prospective designs created using domain-specific knowledge rather than optimization. Motivated by these considerations, the `evaluate()` function accepts any candidate design (optimized or user-created) and can return interpretable comparison metrics for evaluating and comparing design performance.

The `evaluate()` function specifically focuses on comparing designs on their prospective parameter calibration accuracy. NLOED generally assumes that all models are identifiable. Under this assumption, the `evaluate()` function gives approximate metrics about the estimate's distribution, given a design $\mathcal{D}$. Metrics provided by `evaluate()` include the estimate's MSE, covariance and bias as well as the FIM (as discussed in Chapter **??**). Unfortunately it is impossible to compute these first three quantities exactly. However, these metrics can be approximated, either asymptotically or through simulation at a candidate point for the true parameter vector (i.e. a guess or estimate). The `evaluate()` function provides both asymptotic and Monte Carlo methods for computing these metrics for a given design. (Note, these approximations are always local and thus conditional on a nominal parameter estimate, as the true value is never actually known.) First-order asymptotic approximations for the covariance can be be computed as the inverse of the Fisher information matrix [17]. As the asymptotic bias is zero at first order, the diagonal of the inverse of the FIM is an asymptotic approximation for the MSE as well (see discussion in Chapter **??** for details) [34]. The `Design` class performs all optimization using objectives based on the FIM, thus the asymptotic metrics generated by `evaluate()` are computed with similar assumptions to those used in the optimization. These methods are rapid and computationally efficient, especially with CasADi's AD tools, however they compromise on accuracy at smaller sample sizes.

Without using higher order methods, which are significantly more complicated [34], approximating the bias generally requires simulation-based approximations via Monte Carlo methods. The `evalutate()` method therefore also implements a parametric Monte Carlo algorithm for approximating the MSE, covariance and bias terms. The Monte Carlo method may yield greater precision than the asymptotic approach, especially at smaller sample size or with highly nonlinear models. In this method, a 'true' nominal parameter vector is used to simulate a large number of datasets corresponding to the candidate design. These datasets are then each fit independently using maximum likelihood. The resulting set of estimated parameter vectors are then used to compute the MSE, covariance and bias empirically, using the nominal parameter vector as a stand-in for the unknown true values. This then provides a local approximation for the parameter accuracy metrics for the given design, conditional on the assumed true vector values. The Monte Carlo approximation is time consuming and the sample number used may need to be tuned to the design and model being evaluated in order to achieve stable approximations, as the algorithm is necessarily non-deterministic. These considerations make the Monte Carlo approach unsuitable for use in the `Design` class optimization, however Monte Carlo metrics can provide useful benchmarks to assess the suitability of asymptotically derived designs at small sample size. The Monte Carlo metrics are also valuable for comparing amongst several design candidates, allowing the user to accurately differentiate between subtle trade-offs in bias and variance components of parameter error.

All design evaluation methods implemented in `evaluate()` are data-free, meaning they can be used to evaluate a design regardless of whether real data has been collected. This is because all of the metrics discussed here are computed as expectations, taken with respect to the data. Other comparison and diagnostic metric, such as profile likelihood-based methods, require real experimental observations and thus are not suitable for comparing existing datasets and prospective designs. Some data-dependent diagnostic tools, like profile-likelihoods, are implemented in the `fit()` function, described later in this section.

In order to call the `evaluate()` function, the user needs to provide a design. Designs in NLOED are

```
   x1  x2 Variable  Replicates
0   0   1       y1           3
1  -1   1       y1           1
2   2  -1       y1           2
3   3   0       y1           2
4   0   1       y2           3
5  -1   1       y2           1
6   2  -1       y2           2
7   3   0       y2           2
```

Figure 4: An example of a dataframe containing an experimental design.

contained in dataframes which all follow the same format with specific naming conventions; an example of which is shown in Figure 4. The first pair of columns each contain model input values and are named according to the input names passed by the user when the `model_object` was instantiated. Here there are two inputs named `x1` and `x2` from the model shown in equation 6. To the right of the input columns is a column named `Variable` which contains the names of the observation variables to be observed at the input setting listed to the left. Here the observation variables are those from the model shown in equation 6; named `y1` and `y2`. The right most columns is named `Replicates` and it contains the number of observations to be taken at the given input and observation settings listed to the left.

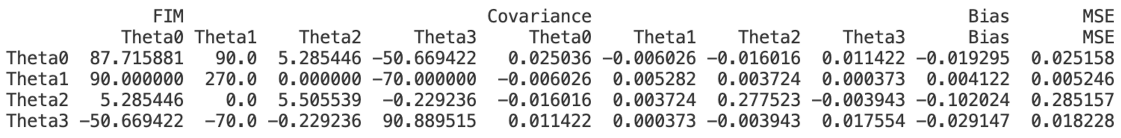
```
1  #define a design
2  design = pd.DataFrame({ 'x1':[0,-1,2,3]*2,
3                          'x2':[1,1,-1,0]*2,
4                          'Variable':['y1']*4 + ['y2']*4,
5                          'Replicates':[3,1,2,2]*2})
6  #set nominal parameter values
7  param = [0.1, 2, 0.4, 1.3]
8  #declare specific options
9  eval_opts={'Method':'MonteCarlo',
10             'FIM':True,
11             'Covariance':True,
12             'Bias':True,
13             'MSE':True,
14             'SampleNumber':100}
15 #call the evaluate() function
16 eval_info = model_object.evaluate(design, param, eval_opts)
17 #print the resulting evaluation
18 print(eval_info)
```

Listing 3: Code example using the `evaluate()` function from the `Model` class.

The general call structure for the `evaluate()` function is:

```
model_object.evaluate(designs, param, options={})
```

Here the `designs` argument is a dataframe containing the candidate design to be evaluated. The `param` argument contains the nominal parameter vector at which the analysis is to take place. The `options` argument is optional and can be used to pass a dictionary of key-value pairs to alter the default `evaluate()` behaviour. An example of a call to the `evaluate()` function is shown in Listing 3. Lines 3-5 show the creation of a design dataframe and line 7 defines the nominal parameter values. In lines 9-13 various options are set in an options dictionary. The `Method` option can take two values; `Asymptotic` or `MonteCarlo` depending on which method is to be used. The `evaluate()` function by default computes metrics asymptotically, and the Fisher information matrix is always computed asymptotically. The options `FIM`, `Covariance`, `Bias`, and `MSE` accept booleans depending on whether or not the user wants the specific metric included in the returned dataframe. By default only the covariance matrix is returned. The `SampleNumber` option accepts an integer value for the samples used to generate the Monte Carlo estimates. In line 15 the `evaluate()` function is called with the previously declared arguments, and in line 17 the returned dataframe is printed.



```
                  FIM                                   Covariance                                        Bias       MSE
               Theta0 Theta1    Theta2     Theta3     Theta0    Theta1    Theta2    Theta3      Bias       MSE
Theta0  87.715881   90.0  5.285446 -50.669422   0.025036 -0.006026 -0.016016  0.011422 -0.019295  0.025158
Theta1  90.000000  270.0  0.000000 -70.000000  -0.006026  0.005282  0.003724  0.000373  0.004122  0.005246
Theta2   5.285446    0.0  5.505539  -0.229236  -0.016016  0.003724  0.277523 -0.003943 -0.102024  0.285157
Theta3 -50.669422  -70.0 -0.229236  90.889515   0.011422  0.000373 -0.003943  0.017554 -0.029147  0.018228
```

Figure 5: An example of the dataframe returned by the `evaluate()` function, containing parameter accuracy metrics for the given design.

An example output for the multi-index dataframe returned by `evaluate()` is shown in Figure 5. Dataframe columns are grouped by the upper-levels `FIM`, `Covariance`, `Bias`, and `MSE`. Rows are named according to the parameter names passed by the user when the given model was instantiated. The `FIM` upper-level contains columns named for each parameter, mirroring the rows. Each entry in these columns gives the Fisher information matrix values for the model in the given experiment. The `Covariance` upper-level likewise contains columns for each parameter and gives the expected covariance matrix entries. The `Bias` upper-level contains a single column with values, in each row, for the bias of each parameter. The `MSE` level likewise contains a single column, with the expected MSE for each parameter with respect to the nominal true value in the given design.

The `evaluate()` function does not return Wald confidence intervals for the parameters. (Wald intervals are asymptotic intervals generally computed using the FIM, see [38]. However, these are easily generated from the root of the diagonal of the covariance matrix, which yields the standard deviation for each parameter. Other similar metrics such as the generalized variance and alphabetic optimality criteria, can be easily computed from the returned metric (see Chapter **??** for some discussion). The `evaluate()` function can also be run in a batch mode, where multiple designs are passed as a list to the `designs` argument. In this case the returned object is a list of dataframes, each structured like the previous example. The batch mode is useful for evaluating many design variants at once and automating the comparison process.

The `sample()` function The `sample()` function is provided in order to generate simulated data from the model. This is useful for generating more detailed diagnostics about specific aspects of a design via simulation studies. Simulation studies involve simulating data for a given experimental design and performing batch fitting and evaluation on the resulting set of fits to study expected statistical properties of the fitting process. To this end, the `sample()` function can generate artificial datasets by sampling observations y_i from the observation distribution $p_i(y_i|f_i(x,\theta))$ for each replicate specified in a given design, and at a given nominal parameter vector.

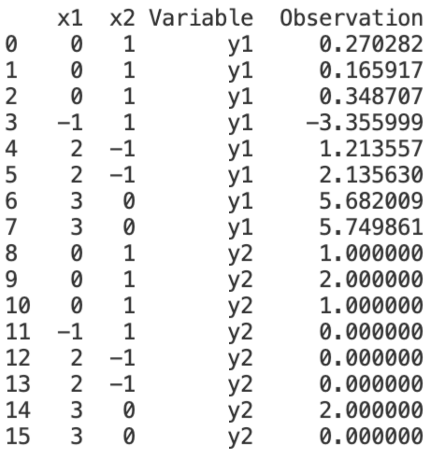
```python
#define a design
design = pd.DataFrame({ 'x1':[0,-1,2,3]*2,
                        'x2':[1,1,-1,0]*2,
                        'Variable':['y1']*4 + ['y2']*4,
                        'Replicates':[3,1,2,2]*2})
#set nominal parameter values
param = [0.1, 2, 0.4, 1.3]
#call the sample() function
dataset = model_object.sample(design, param, design_replicates=1)
#print the resulting dataset
print(dataset)
```

Listing 4: Example using the `sample()` function from the `Model` class.

In order to call the `sample()` function, the user must provide a dataset in a dataframe, using the previously detailed naming convention (see Figure 4). The general call structure for the `sample()` function is:

```
model_object.sample(designs, param, design_replicates=1,options={})
```

The `design` argument accepts a dataframe with format depicted in Figure 4. The `param` argument contains the nominal parameter values at which the samples are to be taken. The `design_replicates` option is an optional argument that specifies the number of replicate datasets to generate from the design. The default value is a single dataset however when doing simulation studies it is useful to be able to generate batches of datasets for a given design and nominal parameter value. The `options` argument is optional and accepts a dictionary of key-value pairs that can modify the default behaviour of the the `sample()` function. An example call to `sample()` is shown in Listing 4. In lines 2-5 an example design dataframe is created, and in line 7 nominal parameter values are listed. In line 8 the `sample()` function is called and a simulated dataset is created as a dataframe; in line 11 the returned dataframe is printed. The `sample()` function returns a dataframe containing the simulated dataset. All datasets in NLOED follow the same structure and naming convention; an example dataset is shown in Figure 6. The first pair of columns correspond to the model input settings and are named according to the string names passed during model instantiation; here they are `x1` and `x2` as per the model given in equation 6. Following the input columns there is the `Variable` column which contains the name of the observation variable, here `y1` and `y2` as per the model in equation 6. Finally, there is the `Observation` column which contains the numeric values of the observed samples.



```
    x1  x2 Variable  Observation
0    0   1       y1     0.270282
1    0   1       y1     0.165917
2    0   1       y1     0.348707
3   -1   1       y1    -3.355999
4    2  -1       y1     1.213557
5    2  -1       y1     2.135630
6    3   0       y1     5.682009
7    3   0       y1     5.749861
8    0   1       y2     1.000000
9    0   1       y2     2.000000
10   0   1       y2     1.000000
11  -1   1       y2     0.000000
12   2  -1       y2     0.000000
13   2  -1       y2     0.000000
14   3   0       y2     2.000000
15   3   0       y2     0.000000
```

Figure 6: An example dataset generated from the `sample()` function.

The `sample()` function can also be run in a batch mode for generating datasets from multiple designs at the same time. To run a batch of designs, each design should be added to a list which is then passed in as the `designs` argument. This can be coupled with the `design_replicates` argument to generate a specified number of replicates of each design. In batch mode the returned object is a list of dataframes for each dataset. When replicates are also included, the returned object is a list of lists, where the first dimension indexes the design and the second indexes the replicate number.

The `fit()` function Model fitting is an important part of the workflow when generating model-based optimal experimental designs. Fitting is needed for estimating parameters from real data; data that can be preliminary or the result of optimal experiments. It is also valuable for performing simulation studies before experiments are run to give users an idea of the true expected utility of their experimental plans. All fitting functionality in the package is provided via the `fit()` function. The `fit()` function is also provided as a matter of convenience, as the type of fitting algorithm to be used for a given optimized design is not arbitrary. The `Design` class performs experimental optimization using various objectives based on the Fisher information matrix. The primary justification for using the FIM is that it provides an asymptotic estimate of the parameter uncertainty that can be achieved after fitting with maximum likelihood. The FIM and other information matrices are asymptotic for the specific model, at the guessed parameter values and *for the given fitting method* [17]. This means that the optimized experiment should be paired with its appropriate fitting method for the best results. Using `fit()` for fitting the user's model ensures the appropriate maximum likelihood fitting algorithm is used in all cases.

In order to fit data using the `fit()` function, the user needs to provide a dataset as an argument. Datasets passed to the `fit()` function follow the same format as other datasets in NLOED, see Figure 6 for an example. The general call structure for the `fit()` function is shown below.

```
model_object.fit(datasets, start_param=None, options={})
```

The `datasets` arguments accept a dataframe, structured as in Figure 6. The `start_param` argument is optional, but can be used to initialize the optimization to a specific initial parameter point. The `options` argument, is a dictionary with various key-value pairs that can be used to override specific default behaviours of the fitting algorithm, some of which are outlined below. The `fit()` function solves for the maximum likelihood estimate with a call to the nonlinear programming package IPOPT via the CasADi interface. In order to set up the log-likelihood optimization for the IPOPT call, `fit()` uses the log-likelihood function attributes that are auto-generated in the `Model` object's instantiation. The overall likelihood objective is constructed by iterating through the rows of the passed dataset, applying the appropriate log-likelihood function attribute to each input-observation pair, and summing the result. This produces a CasADi symbol for the overall fitting objective (see Chapter **??** for further discussion of maximum likelihood),

$$l_{Tot}(\theta; y_{\mathcal{D}}, \mathcal{D}) = \sum_{j}^{N} \sum_{i}^{M} \sum_{k}^{\beta_{i,j}} l_i(y_{i,j}^{(k)}; x_j, \theta). \tag{10}$$

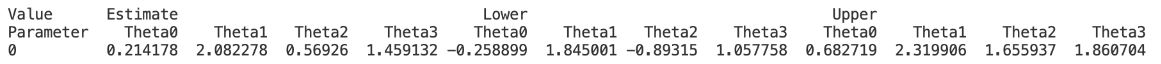
As the objective is a symbolic expression, when it is passed to IPOPT via the CasADi interface, any required derivative information for the interior-points algorithm is automatically generated. This ensures the maximum likelihood fitting implemented in `fit()` occurs rapidly in a few iterations for most models.

```
1 # set specific fitting options
2 fit_opts={'Confidence':'Intervals',
3           'InitParamBounds':[(-1,1),(-1,1),(-1,1),(-1,1)],
4           'InitSearchNumber':7}
5 #call the fit() function
6 fit_info = model_object.fit(dataset, options=fit_opts)
7 #print the fitting information
8 print(fit_info)
```

Listing 5: Example using the `fit()` function from the `Model` class.

A generic call to the `fit()` function is shown in Listing 5, this code assumes `dataset` contains an existing dataset like the one shown in Figure 6. In lines 2-4, several options for the `fit()` function are modified. The `Confidence` option has been set from the default value of `None` to `Intervals` which signals to the fitting algorithm that it should generate profile likelihood-based confidence intervals for all fitted parameters after fitting [24]. The `InitParamBounds` option has been set with a list of tuples specifying a range for each parameter over which a coarse fitting pre-search is performed. The `InitSearchNumber` option specifies the number of evaluations to perform in the `InitParamBounds` ranges; here we have set it to seven but its default value is three. The pre-search procedure is performed to ensure the starting parameter vector is reasonable and the pre-search is a good recourse if a suitable `start_param` cannot be specified *a priori*. If a `start_param` value is specified and the pre-search option `InitParamBounds` is also passed, the `start_param` is appended to the pre-search evaluation list but it may not actually be used as the start value for IPOPT's maximum likelihood optimization as a more suitable starting value may be found during the pre-search. Note, that `InitParamBounds` are not bounds for IPOPT's optimization, only the pre-search, and that `fit()` does not allow bounded or constrained maximum likelihood fitting. The user is expected to use parameter transformations to ensure models are specified properly (examples of this are given in Section 5). As maximum likelihood estimates are invariant under parameter transformation, any required re-parameterization should not effect the resulting fits [32]. In lines 6 the `fit()` function is called for the provided dataset and in line 8 the fitting information is printed to the console.

An example output of the dataframe returned by the `fit()` function is shown in Figure 7. The returned value is a multi-index dataframe, which is organized by the upper level labels `Estimate`, `Lower`, and `Upper`. The `Estimate` level contains columns for each parameter, named according to the names passed when the model was instantiated. These columns contain the maximum likelihood estimates for the model parameters. The `Lower`, and `Upper` levels also contain columns for each parameter with the corresponding upper and lower bounds for the requested likelihood-based intervals. These columns are only returned if the `Confidence` option is set to something other than `None`, such as `Intervals` in this example. By default a 95% confidence interval is returned.



```
Value      Estimate                                  Lower                                        Upper
Parameter    Theta0    Theta1   Theta2    Theta3    Theta0    Theta1   Theta2    Theta3    Theta0    Theta1    Theta2    Theta3
0          0.214178  2.082278  0.56926  1.459132 -0.258899  1.845001 -0.89315  1.057758  0.682719  2.319906  1.655937  1.860704
```

Figure 7: An example of the dataframe output from a call to the `fit()` function.

The intervals returned via a call to the `fit()` function are computed using the profile likelihood with the observed data [24, 32]. Profile likelihood confidence intervals are based on asymptotic properties of the likelihood ratio, defined as $-2L_{Tot}(\theta_o)/L_{Tot}(\hat{\theta})$ [32]. Using properties of the likelihood ratio, it can be shown that

$$-2[l_{Tot}(\theta_o) - l_{Tot}(\hat{\theta})] \leq \chi^2_{p,(1-\alpha)}. \tag{11}$$

Here the expression on the left of the inequality is equivalent to the likelihood ratio. The $l_{Tot}(.)$ function is the overall log-likelihood for the dataset, $\hat{\theta}$ is the MLE parameter estimate and θ_o is the unknown true parameter value. The $\chi^2_{p,(1-\alpha)}$ term is the $1-\alpha\%$ percentile of a Chi-square distribution, with p degrees of freedom where p is the number of dimensions of θ. Effectively, the above inequality states that the likelihood ratio between the unknown true parameter vector and the MLE estimate should be less than $\chi^2_{p,(1-\alpha)}$ with a probability of $1-\alpha\%$ [38, 6]. To construct intervals for each parameter dimension using the above result, each dimension is *profiled.* Profiling each dimension allows the confidence intervals for a single dimension to properly account for uncertainty in the other parameters (see [24, 32] for further discussion). Profiling a given dimension, θ_i, involves incriminating its value away from the MLE value; first in an increasing direction and then a decreasing directions, although the order is arbitrary. At each increment of θ_i the other marginal parameter dimensions are re-optimized to yield a new conditional MLE for the marginal parameters given the fixed incremented value of θ_i. The resulting parameter vector, including the current value of θ_i and the conditionally optimized marginal parameters, is notated $\bar{\theta}(\theta_i)$. As the value, θ_i, of the profiled dimension is adjusted away from the MLE the conditional estimate $\bar{\theta}(\theta_i)$ traces out a curve in parameter space, this is known as the *profile trace* [6]. The likelihood ratio,

$$-2[l_{Tot}(\bar{\theta}(\theta_i)) - l_{Tot}(\hat{\theta})], \tag{12}$$

between the MLE, $\hat{\theta}$, and the conditional vector, $\bar{\theta}(\theta_i)$, will grow until it reaches the Chi-square threshold, $\chi^2_{p,(1-\alpha)}$. The values of θ_i, both in the positive and negative directions, that make the Chi-squared inequality strict are considered the bounds of the the $1-\alpha\%$ confidence interval for dimension θ_i. The value of the log-likelihood, $l_{Tot}(\bar{\theta}(\theta_i))$, for each value of θ_i is known as the *likelihood profile* [25]; its maxima occurs when θ_i is at its MLE value. The profiling procedure can be performed for each parameter dimension in the parameter vector, yielding an interval, trace and profile for every dimension.

When the `Intervals` value is passed as the `Confidence` option, the `fit()` function currently uses a bisection search to find the profile likelihood interval boundary points rather than completing a full incremental profile. Beyond intervals, the `fit()` function can also generate other useful diagnostic information. The `Confidence` field can also be set to the value `Profiles` to generate graphical plots of the likelihood profiles and 2D projections of the profile traces for the given dataset. Examples of the profiles and trace projections are shown in 8. The plots on the diagonal show each parameter's profile likelihood plot, with the log-likelihood values for the profile shown on the y-axis and the the value of the profiled parameter, θ_i, on the x-axis. The MLE value of the profiled parameter occurs at the profile maximum. The red dashed line indicates the log-likelihood value at which the Chi-squared threshold is reached (the default is for a 95% interval). The intercepts between the likelihood profile and the threshold line denotes the confidence interval end points. In the lower triangular plots, a plane for each pair of the parameters is shown in which 2D projections of each parameter's profile trace are plotted as lines; the blue line belongs to parameter in the given column (profile plot above) and the orange line belongs to the parameter in the given row (profile plot to the right). The trace projections are terminated at the confidence interval boundaries, and their intersection marks the MLE estimate. Returning profiles and trace projections is useful because the shape of the log-likelihood profile, when plotted with respect to the corresponding parameter as in Figure 8, should be asymptotically quadratic; significant deviations from this trend can indicate poorly identified parameters [32]. Specifically, a blunt peak to the profile can indicate that the current data is insufficient to constraint the parameter values [25]. In addition, the projections of the profile traces for each parameter should be asymptotically 'X' shaped, with the MLE estimate at the intersection point [6]. Curvature of the trace projections as they move away from the intersection can indicate significant non-linear effects and the breakdown of the asymptotics. In addition, the profile trace

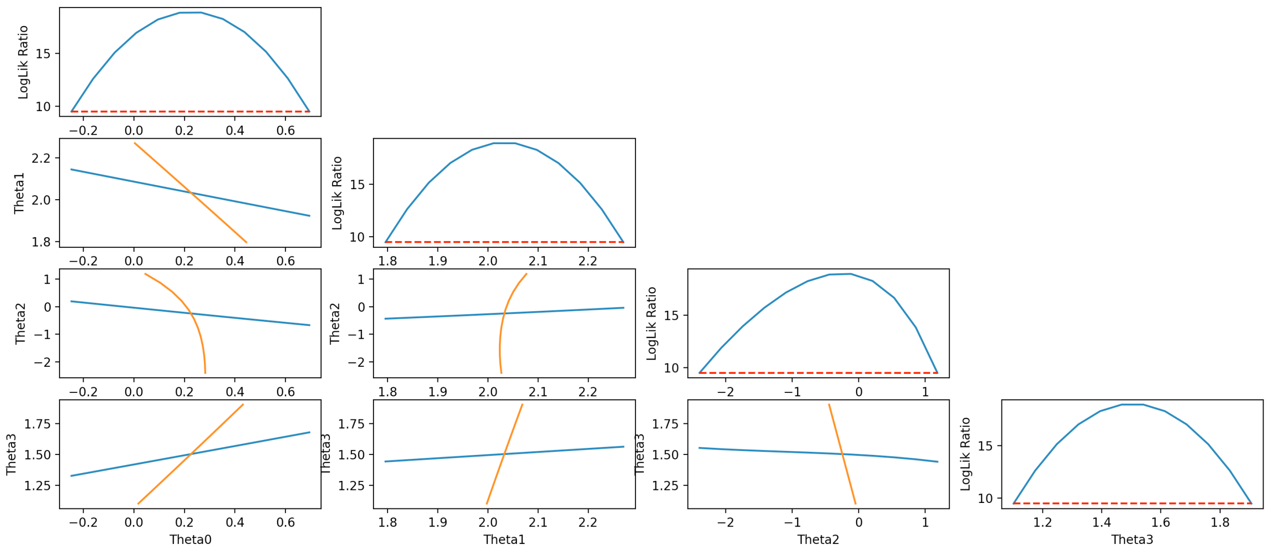


Figure 8: Example of the `fit()` functions graphical output of likelihood profiles and 2D trace projections generated when the `Confidence` option is set to `Profiles`.

projections should ideally intersect at near right angles, with highly oblique angles indicating significant correlation between parameter pairs and possible deficiencies in the experimental design.

The `fit()` function can also compute likelihood *contour projections* by setting the `Confidence` option to `Contours`. Likelihood contour projections consist of a closed curve surrounding the maximum likelihood estimate in a 2D projection of the parameter space [6]. These curves mark the extreme points that the given pair of parameters can extend out from their MLE value before the log-likelihood ratio increases to the asymptotic Chi-square threshold, assuming the marginal parameters have been conditionally optimized (i.e. profiled) [6]. The profiling algorithm used for the confidence interval computation can be modified to find likelihood contour projections in 2D parameter-pair planes by profiling along non-axial parameter vectors rather than along parameter axes. Just like the profile trace, the algorithm begins at the MLE vector, but rather than selecting an individual parameter, a pair are selected; θ_i and θ_j. This pair forms a 2D plane in the parameter space. Next a grid of angles is selected from 0 to 2π radians, each angle corresponds to a direction vector in the parameter-pair plane emanating from the MLE vector. The parameter pair, θ_i and θ_j, are incremented along this vector for each angle in the grid, and the remaining marginal parameters are optimized at each point yielding a conditional MLE vector $\bar{\theta}(\theta_i, \theta_j)$, given the fixed values of θ_i and θ_j. When the a parameter pair is found along the current direction vector that satisfies the equality,

$$-2[l_{Tot}(\bar{\theta}(\theta_i, \theta_j)) - l_{Tot}(\hat{\theta})] = \chi^2_{p,(1-\alpha)}, \tag{13}$$

the position in parameter space is recorded as a member of the $1-\alpha\%$ profile contour projection for that parameter pair. This is repeated for each angle and parameter pair to build a set of 2D contour projections for each parameter pair. Some interpolation of the recorded contour points is used to generate smooth curves during plotting. When the `Confidence` option is set to `Contours`, contour projection plots are added to the profile and trace projection plots previously discussed. An example of the profile and trace plots with added contours is shown in Figure 9. By computing contour projections for each pair of parameters, we can gain an understanding of what parameter sets are feasible given a certain confidence level. The contour projections are asymptotically ellipsoidal but model non-linearity and weak experimental designs can lead to eccentric shapes. Computing the contour projections will generally fail if the region is so eccentric that it contains large concave sections in its boundary. Computing contour projections can be time consuming, especially for large and dynamic models. Currently `fit()` uses a bisection search along each angle to find the contour points, similar to the search used for interval computation.

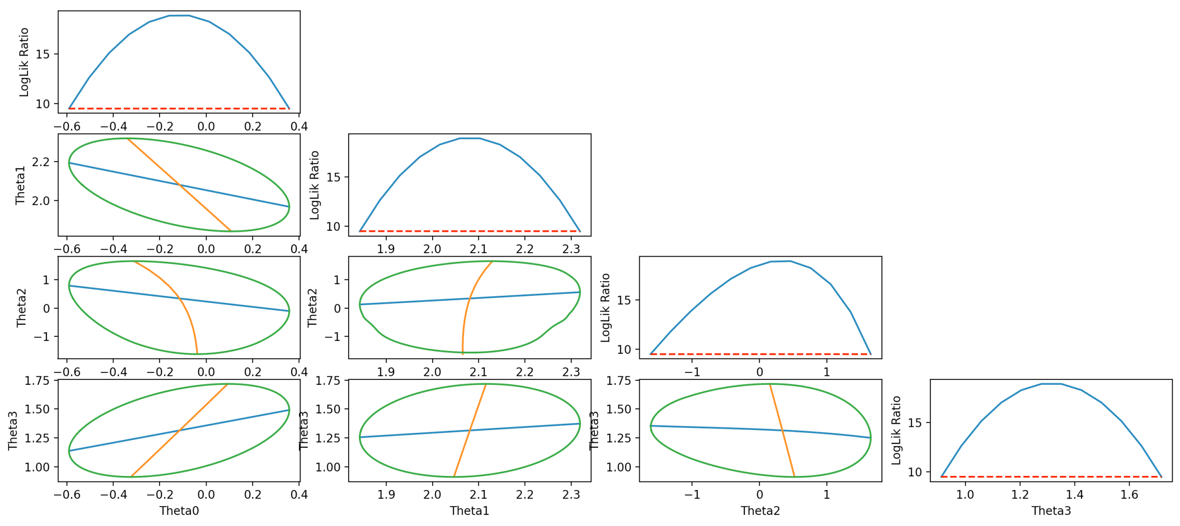


Figure 9: Example of the `fit()` function's graphical output of likelihood contour projections generated when the `Confidence` option is set to `Contours`. The contour projections are shown along with the profiles and trace projections previously shown in figure 8.

```
1  #create three datasets
2  data1, data2, data3 = dataset = model_object.sample(design, param,
   ↪  design_replicates=3)
3  #combine datasets into a single list
4  datasets = [data1, data2, data3]
5  # set up specific options for fitting
6  fit_opts={'Confidence':'Intervals'}
7  #call the fitting procedure
8  fit_info = model_object.fit(datasets,options=fit_opts)
9  #print the fitting information
10 print(fit_info)
```

Listing 6: Example using a batch call of the `fit()` function from the `Model` class.

The intervals and graphical diagnostics generated by the `fit()` function are data-dependent, in that the intervals, profiles and contours require data in order to compute. This is in contrast to the data-free diagnostics that are provided in the `evaluate()` function, and which are used in the `Design` class for experiment optimization. The data-free methods involve an expectation and so are suitable to apply before or after data is collected but they are less useful for generating visual diagnostics and they lack the interpretive richness of the data-required methods implemented in the `fit()` function.

```
Value      Estimate                                      Lower                                          Upper
Parameter    Theta0    Theta1    Theta2    Theta3    Theta0    Theta1    Theta2    Theta3    Theta0    Theta1    Theta2    Theta3
0          0.076718  1.959471 -0.424494  1.124808 -0.396464  1.723006 -2.587254  0.723141  0.545436  2.196488  1.134938  1.526694
1          0.224183  1.987154  0.398126  1.193759 -0.248185  1.750248 -1.143593  0.792909  0.691805  2.224489  1.532261  1.594884
2          0.279028  2.023878 -0.029809  1.413481 -0.192847  1.787084 -1.907175  1.011892  0.745895  2.261125  1.268460  1.815189
```

Figure 10: Example dataframe returned with a batch call to the `fit()` function.

The `fit()` function can be used in batch mode by passing a list of datasets rather than a single dataframe. In this mode each dataset is fit independently yielding its own parameter estimate. This batch fitting is generally not useful for fitting to real experimental data but it is valuable for fitting to collections of simulated datasets in simulation studies. (Note, to fit multiple datasets simultaneously to achieve a single estimate, the user can simply vertically concatenate their respective dataframes and pass them as a single dataset.) Listing 6 shows an example of a batch call with three datasets. In line 2 three datasets are simulated with a call to `sample()` and on line 4 they are bundled into a list. On line 6 options are set to return confidence intervals for each fit and on line 8 the datasets and options are passed to `fit()`. Line 10 prints the resulting dataframe containing all of the estimates and intervals.

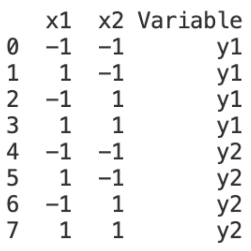
Figure 10 shows an example of the returned dataframe from a batch call to fit. Each additional dataset adds an extra row to the returned frame. In simulation studied we are often interested in checking if true values fall in the intervals near the desired percentile rate or if the estimates exhibit a similar covariance structure to the covariance matrix predicted by the FIM. This information can easily be extracted from the returned dataframe. For example to compute the covariance of all the estimates, the user can use the expression:

```
np.cov(fit_info['Estimate'].to_numpy().T)
```

To check the fraction of datasets for which a given parameter (named `'Par'`) falls within its interval the user can call:

```
sum(fit_info['Estimate','Par'].between(fit_info['Lower','Par'],
↪  fit_info['Upper','Par']))/len(fit_info)
```

The `predict()` function Generating model predictions is often the final goal of the modeling building process. However predictions are also useful during model calibration to understand expected model behaviour in various experimental scenarios as well as to understand how parameter uncertainty and sampling variability propagate to model prediction uncertainty. The `predict()` function can be used to generate model predictions for specific input settings and parameter values. The `predict()` function by default provides predictions for the mean behaviour of the observation variable, $E(Y_i)$. In addition to the mean behaviour the `predict()` function can also provide various confidence intervals for the observation variables, Y_i, given parameter uncertainty. This uncertainty propagation can be done using the delta method [32] or through Monte Carlo simulation. In addition, the `predict()` function can also generate parametric sensitivities for the mean response.



```
   x1  x2 Variable
0  -1  -1       y1
1   1  -1       y1
2  -1   1       y1
3   1   1       y1
4  -1  -1       y2
5   1  -1       y2
6  -1   1       y2
7   1   1       y2
```

Figure 11: Example of an input dataframe used to generate predictions with the `predict()` function.

In order to call the `predict()` function, the user must first create a dataframe containing the input and observation combinations at which they desire predictions. An example of this dataframe is shown in Figure 11 for a two input model with inputs names `x1` and `x2`. The last column is always named `Variable` and specifies for which observation variables predictions should be made. Here eight predictions are asked for at different input settings for one of two observation variables `y1` and `y2`.

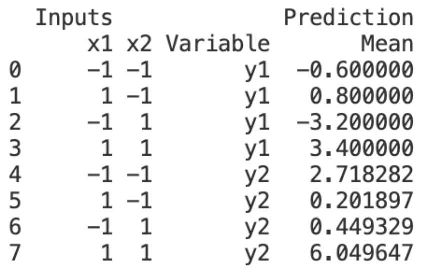
```
#define the inputs for predict()
predict_inputs = pd.DataFrame({ 'x1':[-1,1,-1,1]*2,
                                'x2':[-1,-1,1,1]*2,
                               'Variable':['y1']*4 + ['y2']*4})
#specify the parameter values
params = [0.1, 2, 0.4, 1.3]
#call predict()
predictions = model_object.predict(predict_inputs, params)
#print the predictions
print(predictions)
```

Listing 7: Example of a simple call to the `predict` function from the `Model` class.

The general function call for `predict()` takes the following format;

```
model_object.predict(input_struct, param, covariance_matrix=None, options={})
```

The `input_struct` argument accepts the previously mentioned dataframe, shown in Figure 11, specifying the input and observation variables. The `param` argument accepts a list containing the parameter values at which the prediction is to be made. The `covariance_matrix` argument is an optional argument that can be passed if the user has a parameter covariance matrix generated from an existing fit to data or a theoretical prior, this is used for computing certain prediction intervals. The `options` argument is a Python dictionary of string-value pairs that can be used to adjust default options in the prediction method, some of the more useful options are highlighted below.



```
  Inputs               Prediction
      x1 x2 Variable         Mean
0     -1 -1       y1    -0.600000
1      1 -1       y1     0.800000
2     -1  1       y1    -3.200000
3      1  1       y1     3.400000
4     -1 -1       y2     2.718282
5      1 -1       y2     0.201897
6     -1  1       y2     0.449329
7      1  1       y2     6.049647
```

Figure 12: Example of the returned dataframe from a call to the `predict()` function.

An example of `predict()` function call is shown in Listing 7. In lines 2-4 we define the input dataframe, specifying for which inputs and observations we want predictions. In line 6 we set numerical values for the parameters and in line 8 we call the `predict()` function. In line 10 we print the predictions, the return object `predictions` is a multi-index dataframe, an example of the printed output for which is shown in Figure 12. Here the columns that were passed as part of the input dataframe, specifying the inputs and observation variables, have been grouped under the upper index `Inputs`. The returned predictions for the mean response are listed under the upper index `Prediction` in the columns named `Mean`. This column lists the predicted mean observation response, $E(Y_i)$ for each condition.

```
1  #define a covariance matrix for the parameters
2  cov_mat = np.diag(params*0.05)
3  #specify desired option values
4  predict_opts = {'Method':'MonteCarlo',
5                  'PredictionInterval':True,
6                  'ObservationInterval':True,
7                  'Sensitivity':True}
8  #generate the predictions
9  predictions = model_object.predict(input_frame, params, cov_mat,predict_opts)
10 #display the predicted outputs
11 print(predictions)
```

Listing 8: A more advanced example using the `predict()` function from the `Model` class to return prediction intervals and sensitivity information.

A more advanced call to `predict()` can be used to generate intervals and sensitivity data. Listing 8 gives an example where the user requests both prediction and observation intervals (defined below) via Monte Carlo, as well as sensitivity data. This additional information is requested by using the `options` dictionary, see lines 4-7. Prediction intervals require the user to pass a parametric covariance matrix, which is specified in line 2 as a diagonal matrix with parameter variances set at 5% of their nominal values. This covariance matrix defines a multivariate normal parametric prior over the parameter vector. The mean of this prior distribution is located at the parameter vector passed via the `param` argument.

Both *prediction intervals* and *observation intervals* are requested in Listing 8. Prediction intervals are percentile-based intervals on the mean of the observation variable, $E(Y_i)$, given uncertainty in the parameters. Observation intervals are percentile-based intervals on the random observation variables, Y_i, itself, given both observation variability and, if a prior is provided, parameter uncertainty. For the prediction intervals, $E(y_i)$ is an expectation with respect to the observation variability of Y_i, and therefore the randomness inherent in each observation has been integrated out. However, the mean response, $E(Y_i)$, can still be considered random when parameter estimate uncertainty, in the form of a prior distribution, is considered. To be more precise, let the mean observation response of Y_i as a function of fixed inputs and parameter values be defined as

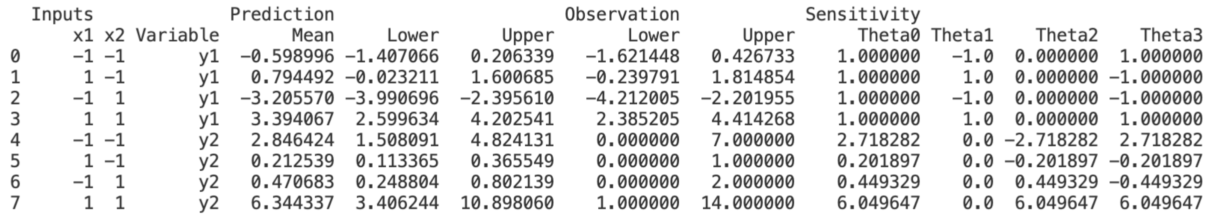
$$\hat{y}_i(x,\theta) = \int_y y_i p(y_i|\eta_i{=}f_i(x,\theta))dy_i. \tag{14}$$

Let it be emphasized that $\hat{y}_i(x,\theta)$ is a deterministic function of x and θ. For all of the parametric distributions used in NLOED, $\hat{y}_i(x,\theta)$ can be computed algebraically from the statistics η_i. However, when uncertainty in the parameter vector, θ, is assumed, the parametric uncertainty propagates to the mean response, $\hat{y}_i(x,\theta)$. The distribution of $\hat{y}_i(x,\theta)$ under the parameter uncertainty then depends on the prior distribution, $p(\theta)$. The `predict()` function defines the prior to be a multivariate normal distribution, centered at the value passed in argument `param` and with covariance matrix passed in argument `covariance_matrix`. Prediction intervals are computed such that interval bounds enclose the true mean response, $\hat{y}_i(x,\theta_o)$, of the true parameter vector, θ_o, with the prescribed confidence probability, by default 95%. This computation obviously assumes the prior correctly reflects the uncertainty about the location of the true parameter vector. Observation intervals, on the other hand, are computed so that their bounds contain realizations, y_i, of the observation variable at the prescribed probability, by default 95%. Observation intervals are computed so as to include both sampling variability, as well as parameter uncertainty if a covariance matrix has been passed. Mathematically the distribution of yet to be observed values of the observation variables, y_i can be expressed as follows:

$$p(y_i|x) = \int_\theta p_i(y_i|f_i(x,\theta))p(\theta)d\theta. \tag{15}$$

The observation interval bounds are determined from the percentiles of this distribution. Without parameter uncertainty this distribution just reduces to the observation distribution $p_i(y_i|f_i(x,\theta))$.

An example of the returned `predictions` dataframe from Listing 8, containing both types of intervals and sensitivity data, is shown in Figure 13. Several additional columns are now included, grouped by upper indices. The upper index `Prediction` now includes columns `Lower` and `Upper` which mark the lower and upper bounds of the prediction interval for each input-observation combination. A new upper index `Observation` has been added, containing `Lower` and `Upper` bound columns for the 95% observation intervals. Note that under parameter uncertainty the predicted mean response, $E_\theta(\hat{y}_i)$, and the mean observation, $E_{y,\theta}(y_i)$, should be the same and so only the predicted mean is returned (i.e. $E_\theta(\hat{y}_i)$). Sensitivities of the prediction mean (taken at the value passed via the `param` argument) are shown in the third upper index grouping named `Sensitivities`. Here each column is named according to the parameter names the user originally passed to the `Model` class during instantiation.



```
  Inputs              Prediction                         Observation              Sensitivity
      x1 x2 Variable        Mean     Lower      Upper        Lower      Upper        Theta0 Theta1    Theta2    Theta3
0     -1 -1       y1   -0.598996 -1.407066   0.206339    -1.621448   0.426733      1.000000   -1.0  0.000000  1.000000
1      1 -1       y1    0.794492 -0.023211   1.600685    -0.239791   1.814854      1.000000    1.0  0.000000 -1.000000
2     -1  1       y1   -3.205570 -3.990696  -2.395610    -4.212005  -2.201955      1.000000   -1.0  0.000000 -1.000000
3      1  1       y1    3.394067  2.599634   4.202541     2.385205   4.414268      1.000000    1.0  0.000000  1.000000
4     -1 -1       y2    2.846424  1.508091   4.824131     0.000000   7.000000      2.718282    0.0 -2.718282  2.718282
5      1 -1       y2    0.212539  0.113365   0.365549     0.000000   1.000000      0.201897    0.0 -0.201897 -0.201897
6     -1  1       y2    0.470683  0.248804   0.802139     0.000000   2.000000      0.449329    0.0  0.449329 -0.449329
7      1  1       y2    6.344337  3.406244  10.898060     1.000000  14.000000      6.049647    0.0  6.049647  6.049647
```

Figure 13: An example of the returned dataframe from the `predict()` function, including prediction and observation intervals as well as sensitivity information.

When computing the mean response, prediction intervals, and observation intervals, the user has a choice of method options: `Exact`, `Delta` and `MonteCarlo`. Only one method can be used in a given call to `predict()`, and all returned information will be computed using the selected method. The `Exact` method is the default method, and it ignores any parameter uncertainty. The `Exact` method returns $E(Y_i)$ computed algebraically from the statistics, η_i, computed at the nominal parameter values passed. The `Exact` method cannot compute prediction intervals as the propagation of a normal prior through a non-linear model cannot be computed exactly, at least for arbitrary models. The observation intervals can be computed exactly, ignoring any parameter uncertainty, by using the cumulative distributions function of the observation probability distribution, $p_i(y_i|f_i(x,\theta))$. The `Delta` method propagates parameter uncertainty using local parametric sensitivities and a normal approximations for the prediction and observation intervals [32]. Using the `Delta` method, the `predict()` function can compute mean, prediction intervals, and observation intervals, but these may be inaccurate for observation distributions that are not well approximated by the normal distribution and for highly nonlinear models. For example, due to the normal approximation in the `Delta` method, interval bounds can be negative even if the observation variable is strictly positive. The `MonteCarlo` method uses Monte Carlo sampling from the parameter prior, $p(\theta)$, and the observation distribution, $p_i(y_i|f_i(x,\theta))$, to compute the mean response and intervals. The Monte Carlo method can be accurate for most scenarios but can be slow and may require the user to increase the default number of samples via the options dictionary to ensure stable estimates.

4 The `Design` Class

The `Design` class is used to generate optimal designs in the NLOED package. Design optimization is conditional on the model and parameter values at which the design is optimized, but also depends on experimental constraints encoded in the optimization, and on how the optimization problem is structured computationally. Each of these factors can alter the resulting optimal design structure. Therefore it may often be the case that the user will want to create multiple optimal designs under various scenarios, and then compare their performance. This motivates having designs encapsulated in a specific class object, so that multiple designs can be created in a modular fashion by instantiating a `Design` object for each scenario the user wishes to consider. Another reason for having designs encapsulated in a `Design` object, is that the output of any design optimization problem in NLOED is a relaxed design which requires additional processing to generate an exact design. This processing is not necessarily unique and involves some user choices. The `Design` class therefore stores relaxed solutions internally and provides users an easy functional interface to generate various exact designs from the relaxed solution.

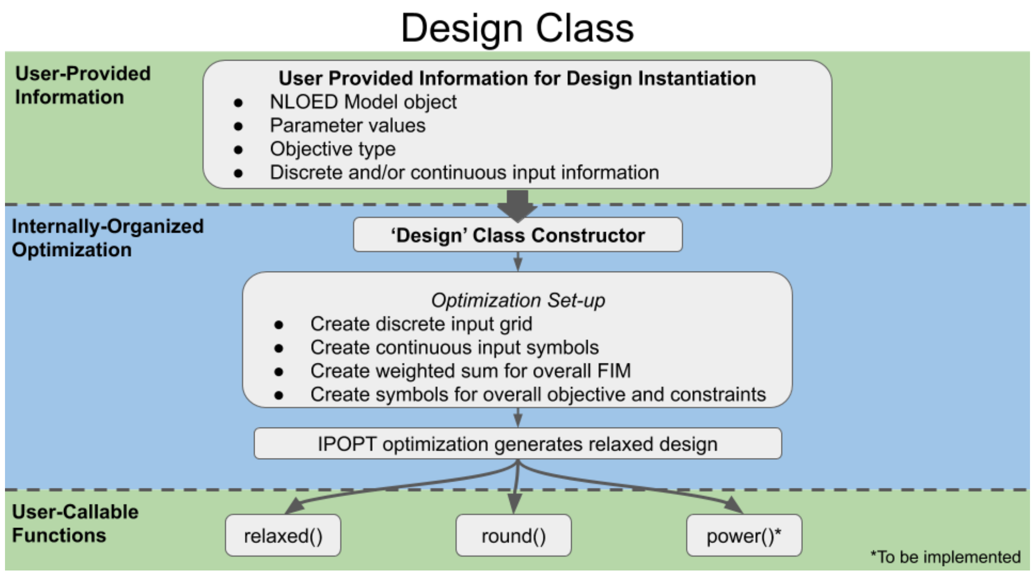

Design Class
User-Provided Information
User Provided Information for Design Instantiation
NLOED Model object
Parameter values
Objective type
Discrete and/or continuous input information
Internally-Organized Optimization
'Design' Class Constructor
Optimization Set-up
Create discrete input grid
Create continuous input symbols
Create weighted sum for overall FIM
Create symbols for overall objective and constraints
IPOPT optimization generates relaxed design
User-Callable Functions
relaxed()
round()
power()*
*To be implemented

Figure 14: Caption text

Figure 14 outlines the basic process for creating a `Design` object. Here, as in Figure 3, the green areas indicate user-controlled passing of data or calling of functions. The blue area indicates automatic processes performed during object instantiation. The upper green area is labelled as *User Provided Information*, this area indicates the various data and options the user needs to provide to the `Design` class constructor for object creation. This information includes which model and parameter value are going to be used, as well as which objective is optimized, how each input variable is to be handled by the optimization algorithm, and choices about sampling flexibility for models with multiple observation variables. Together this information determines the exact nature of the optimization problem and will influence the resulting design considerably. Once this information is properly encoded and passed to the class constructor, instantiation of the Design object commences.

Instantiation of the `Design` class will only take a single line of code to initiate, but as the instantiation process contains a call to IPOPT for optimization, this can take some time to complete. During this process several automated actions are taken within the `Design` class constructor. These automated processes are labelled as *Internally-Organized Optimization* in Figure 14. These processes are primarily split into two parts, 1) optimization set-up and 2) the IPOPT call for optimization that generates the relaxed design. During optimization set-up the specific information passed during instantiation, especially function attributes within the passed models, are used to construct a CasADi symbol for the optimization problem and its constraints. Once this symbolic structure is prepared, it is passed to IPOPT via CasADi's interface; the optimizer then runs and returns an optimal solution which is parsed and stored within the resulting design object.

After optimization completes, the design object will be fully instantiated and the user can interact with it using various user-callable functions. These are shown in the bottom green section and are labeled *User-Callable Functions* in Figure 14. These functions can be used to create exact designs from the optimal relaxed archetype or to compare various exact designs' performance depending on user's choice regarding the sample size. Specifically, the `round()` function can be used to return an exact design as a dataframes in NLOED's default design format. The exact designs can be used directly with the `Model` class's user-callable functions like `sample()` and `evaluate()` for further analysis, or to guide real experimentation. The following subsections give detailed descriptions of each of the three phases outlined in Figure 14, including sample code and call structures where appropriate.

4.1 Creating a `Design` Object

In order to create a `Design` object the user needs to call the `Design` constructor. The general call structure for the `Design` class constructor is;

```
Design(models, parameters, objective, discrete_inputs=None, continuous_inputs=None,
↪  observ_groups=None, fixed_design=None, options={})
```

The first three arguments; `models`, `parameters`, and `objective` are always required. The `models`, argument accepts a model object of the `Model` class created previously by the user. The `parameters` argument accepts a the nominal parameter values at which the optimal design is computed. The `objective` argument accepts a string specifying the objective function type. Currently this argument only accepts `D` for D-optimal designs (the determinant of the Fisher information matrix).

The remaining input arguments: `discrete_inputs`, `continuous_inputs`, `observ_groups`, `fixed_design`, and `options`, are optional to varying degrees. The user exerts significant control over the posing of the design problem in specifying these remaining arguments. The `discrete_inputs` and `continuous_inputs` arguments control how the inputs to the model are handled. At least one of these two input-related arguments must be passed, as all inputs must be treated as either discrete or continuous. Discrete inputs are dimensions for which the optimization algorithm will only consider discrete levels of the input in the design. Continuous inputs are treated as real-valued and thus can be varied accordingly. The `observ_groups` argument accepts information about which observation variables must be sampled together. By default the `Design` class assumes all observations can be replicated with complete flexibility and independence, sometimes this is not possible and `observ_groups` allows the user to force additional structure on which observation variables can be measured together in a given input condition. The argument `fixed_design` allows the user to pass an existing fixed design to the `Design` class. This is useful if the user has certain observations that need to be taken regardless of the optimal design (i.e. based on domain specific knowledge) or to optimize the current design conditionally on past data. The `options` arguments is optional and accepts a dictionary of key-value pairs to modify default settings of the `Design` constructor and optimization.

To help explain the behaviour of the optional arguments of the `Design` constructor, assume an existing model object, `model_object`, has been passed to the `Design` constructor. Also assume that `model_object` has four inputs, and two observation variables. Let the input names be: `x1`, `x2`, `x3`, and `x4` , and the observation variable names be: `y1` and `y2`. In the subsequent paragraphs this example model will be used to explain the usage of the various optional arguments. As the descriptions are quite lengthy, the topics are outlined here for clarity:

- **Discrete Inputs** Using the `discrete_inputs` argument to handle all model inputs.
- **Continuous Inputs** Using the `continuous_inputs` argument to handle all model inputs.
- **Mixed Inputs** Using a both the `discrete_inputs` and `continuous_inputs` argument to handle different model input subsets as either discrete or continuous.
- **Observation Groups** Using the `observ_groups` argument to force observation variables to be sampled together.

- **Fixed Design Aspects** Using the `fixed_design` argument to pass in an existing or fixed aspect of the experimental design for conditional optimization.

Discrete Inputs Both the `discrete_inputs` and `continuous_inputs` arguments are dictionaries, each with multiple fields. The user must create them prior to passing them to the `Design` constructor. To assign all four inputs to be discrete, the user can use the code in Listing 9.

```python
#discrete_input argument creation
discrete_dict ={'Inputs':['x1','x2','x3','x4'],
                 'Grid':[[-1,-3,-6,3],[0,0,0,0],[4,1,2,-1],
                          [3,8,-3,7],[9,7,-7,9],[0.1,0.3,-0.2,1]]}
#declare parameters
param = [2, 3.3, 0.5, 1]
#call Design constructor
design_object = Design(model_object, param, 'D', discrete_inputs=discrete_dict)
```

Listing 9: Example of the `discrete_inputs` argument creation and passage, assigning four model inputs to be handled discretely.

The `Inputs` key must be passed for `discrete_inputs`, its value is a list of strings naming the inputs to be treated discretely. Here all inputs are listed in the `Inputs` field of the dictionary, which means all inputs are handled discretely and the `continuous_inputs` argument can be ignored. When inputs are discretized the optimization algorithm only considers discrete levels of each input, permuted to create a grid of candidate input points in the discrete input domain. The remaining keys in `discrete_inputs` determine the layout of the discrete grid for the specified inputs. The user can specify the grid in three ways; 1) with `Grid` key, 2) with the `Candidates` key, and 3) with the `Bounds` and `NumPoints` keys. As shown in Listing 9, the user can specify the exact set of grid points using the `Grid` key, which is followed by a list of lists; the outer list contains all of the grid points, the inner list specifies the input values at each point. Values in the inner lists are assumed to be ordered in the same way the discrete input names were passed via the `Inputs` key. Passing the exact set of grid points gives the user complete control over which input combinations are considered, but it can be time consuming to construct. In the case where the user requires specific discrete levels of each input to be considered but faces no other limitation, they can use the `Candidates` key. Here the user passes a list of lists again, but the outer list is the same length as the number of discrete inputs. Each inner list contains the unique candidate values of the corresponding input to be consider. The inner lists can be of different lengths. An example of this type of dictionary is shown below:

```python
#discrete input argument creation
discrete_dict ={'Inputs':['x1','x2','x3','x4'],
                 'Candidates':[[-1,0,1],
                               [1,5,10,15,20],
                               [.1,.7,12,200]],
                               [0,1,2,3,4,5,6,7,8,9,10]}
```

The `Design` constructor will generate all possible permutations of the provided candidate lists and use this permutation set as the grid of potential input points. Finally, in some experiments the user may wish to simply distribute points evenly through some region of the discrete input space. To do so they can use the `Bounds` and `NumPoints` keys. When passing the `Bounds` key, the user gives the lower and upper bounds for each discrete input dimension as a list of tuples. The `Design` constructor then creates a grid of candidate points in this hyper-rectangle defined by the bounds, with the integer passed via the `NumPoints` key determining the number of points per dimension. The points within each dimension are distributed in an equidistant manner and all permutations within the hyper-rectangle defined by the bounds are considered. If `NumPoints` is not passed, the number of values along each dimension defaults to 5. The `Bounds` and `NumPoints` keys are useful for quickly generating an equidistant candidate grid if the exact value of the discrete levels do not matter. An example dictionary using the `Bounds` and `NumPoints` keys is given below:

```python
#discreet input argument creation
discrete_dict = {'Inputs':['x1','x2','x3','x4'],
                         'Bounds':[(-1,1),(-1,1),(-1,1),(-1,1)],
                         'NumPoints':10}
```

There are a number of reasons the user would consider using discrete inputs. Real experimental inputs to a system, when implemented in the lab, are always restricted to discretely distinguished levels by experimental measurement accuracy and equipment limitations. Even if an input such as temperature could in theory be resolved down to an infinitesimal scale, an experimental apparatus in the lab can only maintain temperature consistently in a bounded range, determined by measurement accuracy and the apparatus control resolution. Some experimental inputs cannot be varied from a few discrete levels (i.e. growth rate on various carbon sources), and some inputs are simply numerical encodings of categorical factors that have no ordering (i.e. background strain, antibiotic type etc.). In certain models it can be advantageous to discretize an input, despite its fine experimental resolution, either because the model is relatively insensitive to the given input or greater optimization efficiency can be achieved via discretization. Discrete inputs impose a higher upfront computational cost and a create a larger overall optimization problem (with higher memory overhead) but discretization leads to faster iterations of the optimization solver due to increased convexity and a simpler overall problem structure. The problem size and upfront cost scale with the number of discrete candidate points considered in the input space. As the grid generally grows exponentially with the number of input dimensions, it is generally difficult to handle high-dimensional problems with all inputs treated discretely.

Continuous Inputs In order to treat inputs continuously the user passes the `continuous_inputs` argument, which accepts a dictionary with somewhat similar structure to its discrete counterpart, `discrete_inputs`. An example of a call to the `Design` constructor with all inputs treated continuously is shown in Listing 10;

```
#continuous input argument creation
continuous_dict = {'Inputs':['x1','x2','x3','x4'],
                   'Bounds':[(-1,1),(-1,1),(-1,1)],
                   'Structure':[['x1_lvl1','x2_lvl1','x3_lvl1','x4_lvl1'],
                                ['x1_lvl2','x2_lvl2','x3_lvl2','x4_lvl2'],
                                ['x1_lvl3','x2_lvl3','x3_lvl3','x4_lvl3']]}
#declare parameters
param = [2, 3.3, 0.5, 1]
#call Design constructor
design_object = Design(model_object, param, 'D', continuous_inputs=continuous_dict)
```

Listing 10: Example of the `continuous_inputs` argument creation and passage, assigning four model inputs to be handled continuously.

Here, the `Inputs` key maps to a list of input names that are to be treated continuously. In Listing 10, all four inputs are listed in the `Inputs` field of `continuous_dict` and so the `discrete_inputs` argument can be ignored. Similar to the discrete case, the `Bounds` key for continuous inputs are also specified as a list of tuples containing lower and upper bounds for each input. All continuous inputs must have boundaries specified. These input bounds are necessary as they ensure well-posedness of the optimization problem; the objective value can increase indefinitely with the increase of some combinations of input dimensions for certain models. Also, it is generally infeasible or impractical to vary the input values beyond some natural limits within the laboratory. Bounds can be imposed based on limitations of experimental equipment or based on restrictions on the input ranges for which the model assumptions hold. When discrete inputs are used, the candidate grid points are fixed and the optimization selects inputs by adding them to the solution based on their utility. When using continuous inputs, the input points are treated as the optimization variable themselves, and the user must specify how many unique points the optimizer should consider within the bounded input space. The `Structure` key allows the user to specify how many unique input points to consider and any common dimensional values they may need to share. The `Structure` key must be passed with a list of lists as its value. The outer list corresponds to the number of unique input points that will be considered in the continuous input space. Each inner list has the same number of dimensions as the number of continuous inputs. The inner lists each contain a set of string symbols, one for each continuous input dimension. Each unique string symbols specifies a unique level for the corresponding continuous input.

In the example shown in Listing 10, we have specified three unique points in the continuous input space; each point has complete freedom in each of the four input dimensions. This scenario is encoded with three inner lists, one for each point. The independence of every point and dimension is indicated by the use of a novel symbol string for every list entry (i.e. 'x1_lvl1', 'x2_lvl2', etc.). However using the Structure key we can also specify more constrained experimental design problems. For example, we may consider six input points, but for all six, x1 can only have one unique value. We may also wish to restrict input x2 to one level in the first three points and another in the later three points. Inputs x3 and x4 are assumed to be free in all six input points. An example of the continuous_inputs dictionary encoding such restrictions is shown in Listing 11.

```
#continuous input argument creation
continuous_dict = {'Inputs':['x1','x2','x3','x4'],
                   'Bounds':[(-1,1),(-1,1),(-1,1),(-1,1)],
                   'Structure':[['x1_lvl','x2_lvl1','x3_lvl1','x4_lvl1'],
                                ['x1_lvl','x2_lvl1','x3_lvl2','x4_lvl2'],
                                ['x1_lvl','x2_lvl1','x3_lvl3','x4_lvl3'],
                                ['x1_lvl','x2_lvl2','x3_lvl4','x4_lvl4'],
                                ['x1_lvl','x2_lvl2','x3_lvl5','x4_lvl5'],
                                ['x1_lvl','x2_lvl2','x3_lvl6','x4_lvl6']]}
```

Listing 11: Example of the continuous_inputs argument that uses the Structure field to encode candidate points with shared dimensional values.

Note how the first element of each inner list is now the same string; 'x1_lvl', indicating that input x1 has a single value, shared across all points to be optimized. The second element of each inner list is either 'x2_lvl1' (in the first three) or 'x2_lvl2' (in the latter three), indicting input x2 will only take two unique values after optimization. Lastly, every element corresponding to inputs x3 and x4 has a unique string, indicating unique levels for each point.

The Structure field will often be used as shown in Listing 10, with all points free, and the main choice being how many inputs points to consider. However the more complicated points structure shown in Listing 11 are required in certain experiments. For example consider a time series experiment where the initial conditions are a model input and all replicates share the same initial conditions. In such a case, the initial condition input will need to be restricted to a single value for all replicates, just like x1 above. Another example may be that limited capacity of a laboratory, like the number of incubators, may necessitate certain input dimensions, such as temperate, can only take a finite number of unique values in each experiment, just like input x2 above only has two levels.

When using continuous inputs, the Design constructor will optimize the placement of the input points and will also attempt to optimize the quantity of replicates taken at each point (this is also done for discrete inputs). However for continuous inputs the user may wish to specify that each input point in the structure gets exactly the same number of observations. This option is ideal for optimizing design with smaller sample sizes. To do so the user can pass the option 'LockWeights' as True in the options argument. This option is only available if all inputs are handled continuously.

Choosing to treat inputs continuously rather than discretely depends on the model and the experimental context. Generally, real world experimental levels are discrete up to the tolerance of measurement and control, and thus continuous inputs are an approximation. However, it is generally practical to approximate inputs as continuous if the number of discrete levels achievable in the laboratory is numerous and the experimental control is very fine relative to the model's input sensitivity. Continuous inputs can also be used for computational reasons as they generally lead to smaller problems that are quick to initialize. However continuous inputs generally result in an optimization problem that is more nonlinear leading to more optimizer iterations, more intensive computation on each iteration, and greater chances of getting stuck in local optima. In addition for continuous inputs, the user needs to specify the number of unique input points the algorithm should consider in the design; this information can be difficult to set *a priori*.

Mixed Inputs In certain situations it is desirable to handle some inputs discretely while treating others as being continuous. This can be for both experimental or computational considerations. For example, adding some discrete inputs tends to make a design problem more convex and easier to solve, but adding some continuous inputs can make the overall problem smaller with respect to memory requirements. When dealing with very complex models or input spaces, blending the input types can allow the user to tackle problems that may be difficult to handle with a single input type. Listing 12 shows an example where input `x1` and `x2` have been handled continuously, with similar restrictions to those imposed in Listing 11. Inputs `x3` and `x4` have been discretized. In this case both the `discrete_inputs` and `continuous_inputs` arguments are passed, with the model's four inputs split between both input structures.

```
#continuous input argument creation
continuous_dict = {'Inputs':['x1','x2'],
                   'Bounds':[(-1,1),(-1,1)],
                   'Structure':[['x1_lvl','x2_lvl1'],
                                ['x1_lvl','x2_lvl1'],
                                ['x1_lvl','x2_lvl1'],
                                ['x1_lvl','x2_lvl2'],
                                ['x1_lvl','x2_lvl2'],
                                ['x1_lvl','x2_lvl2']]}
#discrete input argument creation
discrete_dict ={'Inputs':['x3','x4'],
                'Candidates':[[-1,-.5,0,.5,1],
                              [-1,-.5,0,.5,1]]}
#declare parameters
param = [2, 3.3, 0.5, 1]
#call Design constructor
design_object = Design(model_object, param, 'D',
↪   discrete_inputs=discrete_dict,continuous_inputs=continuous_dict)
```

Listing 12: Example calling the `Design` constructor using a mixture of discrete and continuous inputs.

Observation Groups The `observ_groups` argument to the `Design` constructor is optional. By default each observation variable is handled independently meaning that during optimization, outputs `y1` and `y2` could have different replicate quantities assigned to them even in the same input conditions. In certain conditions this is not practical, for example in a costly destructive sampling experiment where a replicate is destroyed on observation. In this case the user will likely measure any relevant observation variables they are able to whenever a replicate is destroyed. In this case it is useful to assign the relevant observation variables to a group that will be sampled together. The `observ_groups` argument accepts a list of lists, each inner list contains the names of observation variables that are measured in an observation group. For example if `y1` and `y2` needed to be measured together in any observation, one could use the example shown in code listing 13.

```
1  #discreet input argument creation
2  discrete_dict = {'Inputs':['x1','x2','x3','x4'],
3                   'Bounds':[(-3,3),(-3,3),(-3,3),(-3,3)],
4                   'NumPoints:10}
5  #declare parameters
6  param = [2, 3.3, 0.5, 1]
7  #observation grouping
8  observ_group_lst = [['y1','y2']]
9  #call Design constructor
10 design_object = Design(model_object, param, 'D', discrete_inputs=discrete_dict,
   ↪   observ_groups = observ_group_lst)
```

Listing 13: Example of the `observ_groups` argument being used to group observation variables `y1` and `y2`.

The `observ_groups` list is constructed in line 8 and is passed in line 10. This will force both observations `y1` and `y2` to be treated as a single unit when the algorithm considers the number of samples to assign to them.

Fixed Design Aspects The fixed_design argument is also optional and can be used to pass fixed design aspects or the design used in existing data. The fixed_design argument accepts a dictionary containing an existing design along with a weight indicating what fraction of the overall sample size dedicated to the fixed design aspects. For example if the user has already collected 5 observations and plans to collect another 15 with an optimal design, the user would pass the fixed_design argument with a weight of 0.25. A coded example using the fixed_design argument is shown in Listing 14. Here in lines 6-11, the initial design is declared. In practice this design can come from a previous optimal design or past datasets. In line 13 the initial design is inserted into a dictionary under the key name Design. In the same dictionary we also include the key name Weight whose value specifies the fraction (between 0 and 1) of the overall sample size dedicated to the initial design. In this case the value of 0.25 is passed as the weight and so the designed experiment is expected to have a sample size three times that of the initial experiment. Passing in fixed aspects of an overall experimental effort is important in order for the optimal design to perform well. Optimal designs created without conditioning on fixed design aspects will ignore information contained in the fixed aspects. This may lead to some observations in the optimal design being inefficiently placed.

```
1  #discrete input argument creation
2  discrete_dict = {'Inputs':['x1','x2','x3','x4'],
3                   'Bounds':[(-3,3),(-3,3),(-3,3),(-3,3)],
4                   'NumPoints:10}
5  #declare initial design
6  init_design = pd.DataFrame({'x1':[-2,-1,1,2]*4,
7                              'x2':[2,-2,-1,1]*4,
8                              'x3':[1,2-2,-1]*4,
9                              'x4':[-1,1,2,-2]*4,
10                             'Variable':['y1']*4+['y2']*4,
11                             'replicates':[3]*8})
12 #create the fixed design dictionary
13 fixed_dict = {'Weight':0.25,'Design':init_design}
14 #declare parameters
15 param = [2, 3.3, 0.5, 1]
16 #observation grouping
17 observ_group_lst = [['y1','y2']]
18 #call Design constructor
19 design_object = Design(model_object, param, 'D', discrete_inputs=discrete_inputs,
    ↪  fixed_design = fixed_dict)
```

Listing 14: Example of a call to the Design constructor with the fixed_design argument used to pass an existing aspect of the overall design.

4.2 Design's Automatic Optimization Set-up

After the Design constructor is called, instantiation of the Design object begins with the automatic organization the optimization problem for passage to IPOPT. The main challenge in setting up the optimization problem is in creating the CasADi symbol for the overall design objective. In order to create the objective symbol, the optimization variables need to be linked to the total Fisher information matrix for the experiment. All objectives used by the NLOED package for optimization are computed as simple algebraic functions from the elements of the experiment's Fisher information matrix; computing the total FIM for a design is therefore the main computational hurdle.

As observations in NLOED are assumed to be independent, the Fisher information matrix for an experiment can be computed as a weighted sum of individual matrices at each observation. For a multi-output model the FIM sum can be written as (see Chapter ?? for details)

$$I_{Tot}(\mathcal{D},\bar{\theta}) = \sum_{j}^{N}\sum_{i}^{M}\beta_{i,j}I_i(x_j,\bar{\theta}). \tag{16}$$

Here there are M observation variables, Y_i, indexed by i, and N support points x_j, indexed by j. The support of the design consists of the set of all unique input vectors such that $x_j \in \mathcal{X}$. The FIM at a given input point, x_j, for the observation variable Y_i is $\mathcal{I}_j(x_i, \bar{\theta})$. The vector $\bar{\theta}$ is the nominal parameter vector at which the design is optimized. Each replicate allocation, $\beta_{i,j}$, corresponds to a support point x_j and an observation variable Y_i. Together the replicate allocations for each observation and input condition form the set $\beta_{i,j} \in \mathcal{B}$, which defines the design's replication structure. For an exact design the weights are restricted to be non-negative integers. The sum of the weights adds to the overall sample size N_{Tot} such that; $N = \sum_i^M \sum_j^N \beta_{i,j}$. The overall information matrix, $\mathcal{I}_{Tot}(\mathcal{D}, \bar{\theta})$, is therefore a function of the design, $\mathcal{D}$, where the design consists of the support point set, $\mathcal{D}$, and weight set, $\mathcal{B}$, such that; $\mathcal{D} = \{\mathcal{X}, \mathcal{B}\}$. The integer constraint makes the above problem very difficult. In the `Design` class the integer constraint on the replicates is relaxed, and the integer replicate allocations, $\beta_{i,j}$, are replaced with real-valued continuous weights, $\xi_{i,j}$, which are constrained so that $1 = \sum_i^M \sum_j^N \xi_{i,j}$. The relaxed design problem can then be written as

$$I_{Tot}(\mathcal{D}_R, \bar{\theta}) = \sum_j^N \sum_i^M \xi_{i,j} I_i(x_j, \bar{\theta}). \tag{17}$$

For the relaxed formulation, all the weights, $\xi_{i,j}$, now form the weight set $\xi_{i,j} \in \mathcal{Z}$, and the relaxed design is defined as $\mathcal{D}_R = \{\mathcal{X}, \mathcal{Z}\}$. The problem in the relaxed form remains nonlinear but it is now possible to pass to a solver such as IPOPT and expect reasonable solution times for many models of interest.

In order to compute the FIM sum for a given experiment, the `Design` constructor uses the FIM function attributes of the `Model` object passed to the `Design` constructor. These function attributes are able to compute $\mathcal{I}_i(x_j, \bar{\theta})$ and can therefore compute the individual FIM's symbolic dependence on each of the candidate support points. The total FIM sum, listed previously, is generated as a CasADi symbolic function with $\xi_{i,j}$ and x_j as the symbolic inputs. The total FIM is then used to generated a CasADi symbol for the overall optimization objective symbol, $\Psi(I_{Tot}(\mathcal{D}_R, \bar{\theta}))$. The resulting objective symbol, $\Psi(I_{Tot}(\mathcal{D}_R, \bar{\theta}))$, effectively encodes the entire optimization problem in a CasADi symbolic structure, linking the objective symbol to symbols for each of the optimization variable symbols; x_j and $\xi_{i,j}$. This whole CasADi symbolic structure can then be passed to IPOPT via the CasADi interface, along with any required constraints. The CasADi interface then uses the symbolic structure to auto-generate objective and constraint derivatives for use in IPOPT's interior points solver. Much of the actual optimization is handled automatically by CasADi and IPOPT. NLOED specifically manages the problem formulation, controlling how the overall FIM is computed and how the optimization problem is structured.

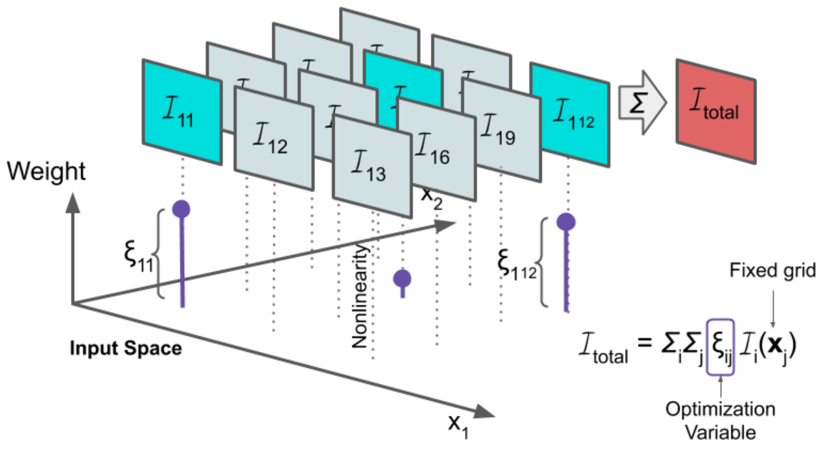


Figure 15: A depiction of NLOED design optimization problem structure with two discretized inputs and a single observation variable.

The input settings the user provides in `discrete_inputs` and `continuous_inputs` during instantiation of the `Design` object significantly influence the optimization problem's computational structure. For example when only discrete inputs are used, the candidate grid specified by the user effectively becomes the support points set $x_j \in \mathcal{X}$. As these points are fixed at specific levels in the grid, the input points,

x_j, are not included as free optimization variables in the optimization problem. Instead, the weights, $\xi_{i,j}$, become the only optimization variables. This results in a large sparse convex optimization problem, because the individual FIMs, $\mathcal{I}_i(x_j, \bar{\theta})$, for each candidate grid point, x_j, can be pre-computed numerically before calling the solver. However, the weights, $\xi_{i,j}$, then control the exact convex combination of these matrices that make up the total FIM. The optimal support points will be indicated by the non-zero replicate weights after optimization. Non-zero replicate weights also indicate which observation variables are to be measured. As each FIM is semi-positive definite, the overall problem is convex (at least for the default D-optimal objective) [2]. However, the discrete handling of inputs requires the `Design` class to compute a FIM for every input grid point, which can be time consuming if there are many inputs and the grid points consist of all permutations of the candidate input levels. Figure 15 visually depicts the optimization problem formulated with a discretized two dimensional input space and a single observation variable. This scenario would correspond to a `discrete_inputs` dictionary structured as;

```
discrete_dict = {'Inputs':['x1','x2'],
                 'Candidates':[[1,2,3],[1,2,3,4]]}
```

In the figure, three levels of `x1` and four levels of `x2` are permuted to create a candidate grid over which the sampling weights serve as optimization variables. In summary, the optimizer has flexibility in assigning sampling weights and observation variables and will easily converge to a global optima, up to numerical precision, but the optimizer has no flexibility to adjust the grid of input points.

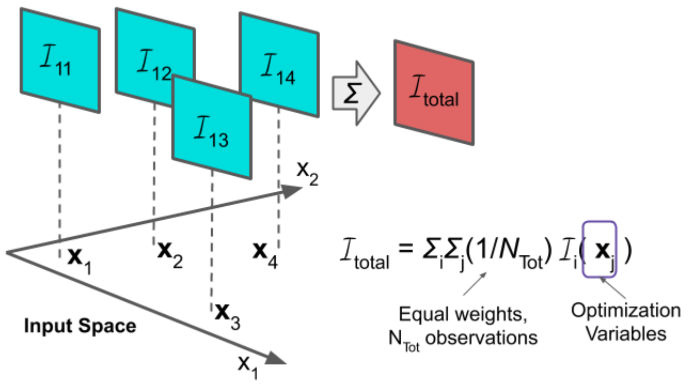


Figure 16: A depiction of an NLOED design optimization problem structure with two continuously handled inputs and the `LockedWeights` constraint activated.

A complimentary scenario to the discretized input grid optimization problem can be achieved, where the weights, $\xi_{i,j}$, are fixed and only the input points, x_j, are optimized. This occurs when only continuous inputs are used, and the `LockWeights` option is set to `True`, so that each input point gets the same fixed weighting. In this case the user controls the number of unique support points, N, by specifying the size of the support set using the `Structure` field of the `continuous_inputs` argument. Figure 16 depicts this optimization set up for a two input scenario. This would correspond to a `continuous_inputs` argument structured as;

```
continuous_dict = {'Inputs':['x1','x2'],
                   'Bounds':[(0,1),(0,1)],
                   'Structure':[['x1_lvl','x2_lvl1'],
                                ['x1_lv2','x2_lvl2'],
                                ['x1_lv3','x2_lvl3'],
                                ['x1_lv4','x2_lvl4']]}
```

In the figure there are four support points free to vary within the bounded domain, each with a fixed identical sampling weight. Here, if the model is nonlinear in its parameters, then the input points,

x_j, are nonlinearly related to the optimization objective. This means the optimization becomes a fully nonlinear programming problem but it tends to be smaller as the number of optimization dimensions is the product of the input dimension size of x and the number of support points, N. Here the optimizer has great flexibility in moving the support points around the input space, but designs may be sensitive to the starting locations of the support points and no flexibility in re-weighting the support points is possible. In this scenario, the only way for the optimizer to replicate a support point is to locate two support points at an identical position, meaning the user will generally need to provide a large number of support points (potentially on the same order as their intended sample size) in order to understand the optimal replication structure. However, the resulting designs derived with locked weights are easily implemented exactly with small sample sizes.

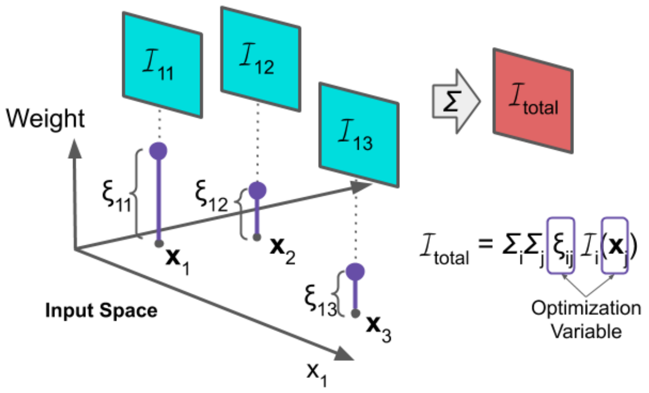


Figure 17: A depiction of NLOED design optimization problem structure with two continuously handled inputs, optimized sampling weights, and a single observation variable.

When continuous inputs are used in the default manner, without locked weights, both the weights, $\xi_{i,j}$, and the input points, x_j, are treated as optimization variables. This scenario is depicted in Figure 17 and would equate with a `continuous_inputs` argument structured as:

```
continuous_dict = {'Inputs':['x1','x2'],
                   'Bounds':[(0,1),(0,1)],
                   'Structure':[['x1_lvl','x2_lvl1'],
                                ['x1_lv2','x2_lvl2'],
                                ['x1_lv3','x2_lvl3']]}
```

This setup gives the optimizer flexibility in both the location of the support points and in the distribution of samples amongst support points as both points and weights are optimization variables. In the figure there are three support points, each with its own sampling weight. The resulting problem is still highly nonlinear in the support points but the use of the weights means that user can allot less candidate support points and still discover the optimal replication structure. This can lead to a smaller optimization problem but the resulting design may be more suited to a large sample size for implementation.

Mixing continuous and discrete inputs can lead to complicated structures which are difficult to visualize. In the continuous subspace of the overall input space, a certain number of candidate support points are free to vary during optimization, as specified by the `Structure` key in the `continuous_inputs` argument. (Recall these 'points' are technically in a lower dimensional sub-space of the overall input space which consists of only the continuously assigned dimensions). Along the remaining discretely handled input dimensions, a candidate set of levels is available via the discrete grid. An example of this scenario is depicted in Figure 18. This scenario would correspond to the `discrete_inputs` and `continuous_inputs` arguments structured as;

```
#discreet_inputs argument
discrete_dict ={'Inputs':['x2'],
```

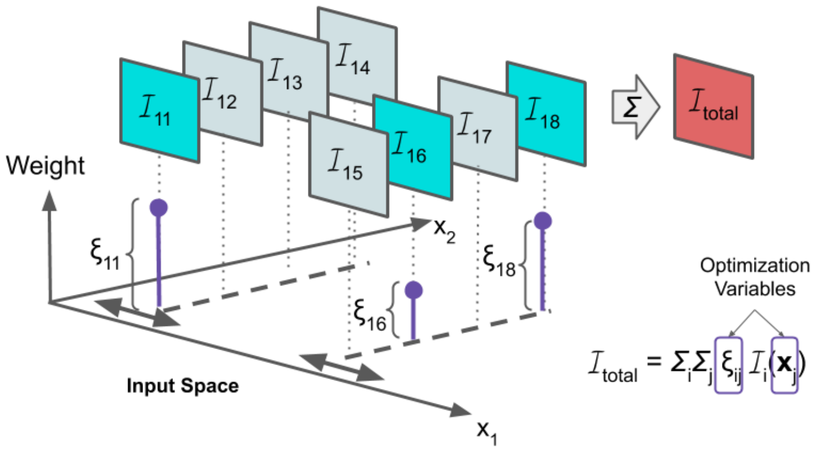


Figure 18: A depiction of NLOED's design optimization problem structure with the first input dimension handled continuously and the second input dimension handled discretely, and with a single observation variable.

```
                  'Candidates':[[1,2,3,4]]}
#continuos_inputs argument
continuous_dict = {'Inputs':['x1'],
                   'Bounds':[(0,1)],
                   'Structure':[['x1_lvl1'],
                                ['x1_lvl2']]}
```

Here two unique levels of input `x1` are available to the optimizer. At each of these two levels, in the second dimension, `x2`, a four level grid is available for selection via the sampling weights. The optimal set of support points that are selected is a function of the location of the continuous dimension points and which discrete grid locations receive non-zero weights. This type of problem formulation allows the user to accommodate specific experimental limitations, for example if `x2` can only be set to four discrete levels in the available equipment, and if `x1` can only be run with at most two unique levels in the given round of experimentation. The mixed formulation can also decrease the non-linearity of a problem, as the input dimensions that are handled continuously are the only ones that are fully nonlinear with respect to the objective.

For models with multiple observation variables, output selection is performed in a similar manner to discrete input selection. Non-zero weights, $\xi_{i,j}$, are used to decide which outputs are measured. Figure 19 shows the default handling of multiple observation variables for a two input model on the left. In this case each of the outputs, `y1` and `y2`, has its own FIM with its own weight but they both share the same underlying input points, regardless of whether inputs are handled continuously, discretely, or in a mixed fashion. When the user groups outputs together using the `observ_struct` argument, the FIMs are automatically summed within the group before they are weighted, implying that they are always observed as a group. This situation is shown on the right in Figure 19 for a two output model where both outputs, `y1` and `y2` have been grouped together.

Based on the provided inputs to `Design` constructor NLOED will implement the appropriate optimization scenario. Each scenario results in different symbolic structures, which are then passed to IPOPT. Some experimentation may be necessary with a given experimental design problem in order to find a structure which works well in IPOPT and satisfies experimental constraints. After optimization, the relaxed design is parsed from IPOPT's output and stored within the `Design` object. The user can then view the relaxed design with its continuous weights or generate an exact design using the objects user-callable functions described in the next section. While NLOED provides extensive flexibility in formulating the optimal design problem, not all experimental limitations can be implemented. In these scenarios, the optimal relaxed design can often provide qualitative information about which input conditions provide the most desirable information about the model parameters. The user can then use the `Model` class's

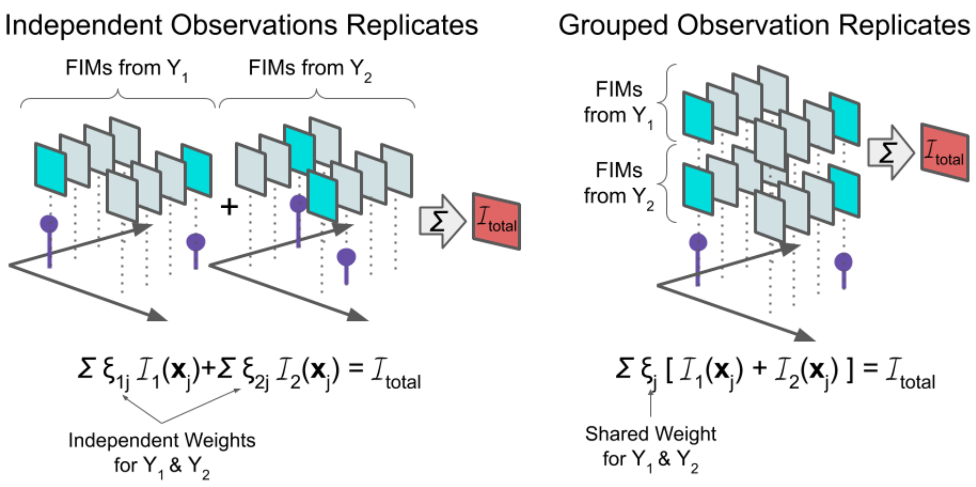

Independent Observations Replicates
FIMs from Y1
FIMs from Y2
+
Σ
$\mathcal{I}_{total}$
$\Sigma\, \xi_{1j}\, \mathcal{I}_1(\mathbf{x}_j) + \Sigma\, \xi_{2j}\, \mathcal{I}_2(\mathbf{x}_j) = \mathcal{I}_{total}$
Independent Weights for Y1 & Y2
Grouped Observation Replicates
FIMs from Y1
FIMs from Y2
Σ
$\mathcal{I}_{total}$
$\Sigma\, \xi_j\, [\, \mathcal{I}_1(\mathbf{x}_j) + \mathcal{I}_2(\mathbf{x}_j)\,] = \mathcal{I}_{total}$
Shared Weight for Y1 & Y2

Figure 19: A depiction of NLOED's method for observation selection as part of its optimization structure. On the left, the default method is used, where observation variables `y1` and `y2` are each given a separate set of weights over the same input structure. On the right, the `observ_struct` has specified that `y1` and `y2` must be observed together in any given input conditions.

`evaluate()` function and simulation tools to assess design modifications and find a reasonable design.

4.3 User-callable `Design` Functions

Once the user has instantiated a `Design` object instance, they can use the object to examine the optimal design structure and to generate exact designs that can be implemented in practice or be simulated by the `Model` class. The user can perform these actions by calling functions available within the `Design` class object. Here we explain the calling procedure for the available functions and discuss some planned extensions.

The `relaxed()` function The `relaxed()` function can be used to return the relaxed optimal design as a dataframe. Relaxed designs resemble exact designs however instead of a `Replicates` column containing the integer count of replicate allocations, $\beta_{i,j}$, they have a `Weight` column that contains the real valued weights, $\xi_{i,j}$. The sum of the `Weight` column values will be one, up to the numerical precision of the optimization algorithm. The `relaxed()` function does not accept any input arguments. An example call is shown in Listing 15.

```
1  #extract the relaxed design from the design object
2  relaxed_design = design_object.relaxed()
3  #print the relaxed design
4  print(relaxed_design)
```

Listing 15: An example call to the `Design` class's `relaxed()` function.

Line 2 returns the relaxed design dataframe and in line 4 the dataframe is printed to the output. An example of a relaxed design that has been printed is shown in Figure 20. Here we can see that there are only four unique support points, with different input values of `x1` and `x2`, each with even weighting across both observation variables `y1` and `y2`. This result is typical for linear models with normal errors and symmetric bounds on the input domain. Returning the relaxed design is not generally necessary but as the relaxed design is used to generate all exact designs via rounding, it can be useful for visualizing the underlying relaxed structure. In addition, depending on how the design optimization problem was structured, the relaxed design may be guaranteed to satisfy certain equivalence theorems [2], and thus

```
   x1  x2 Variable  Weights
0  -1  -1       y1    0.125
1  -1  -1       y2    0.125
2  -1   1       y1    0.125
3  -1   1       y2    0.125
4   1  -1       y1    0.125
5   1  -1       y2    0.125
6   1   1       y1    0.125
7   1   1       y2    0.125
```

Figure 20: An example of a dataframe containing an relaxed design returned by the `relaxed()` fucntion.

can be used to verify a global optima has been achieved. This will only work in special cases, for more information the user may refer to [2].

The `round()` function In order to create an exact design with a finite sample size, N_{Tot}, the user must discretize the exact design in some manner, converting real-valued weights, $\xi_{i,j}$ to integer valued allocations, $\beta_{i,j}$, so that $N_{Tot} = \sum_j^N \sum_i^M \beta_{i,j}$. There are a number of rounding methods available, each with various trade-offs [33, 40, 21]. The `round()` function implements the Adam's apportionment rounding procedure as a default method for performing this task. Adam's apportionment has been noted in previous works as having a number of ideal properties for rounding experimental designs [33, 40, 21]. The general call structure for the `round()` function is;

```
round(sample_size, options={})
```

The `sample_size` argument accepts the desired number of sample size for the experiment. The `options` argument is optional and can be used to override the default behaviour of the rounding algorithm. An example call to the `round()` function is shown in Listing 16.

```
#set the sample size
sample_size = 10
#generate the rounded design
exact_design = design_object.round(sample_size)
#print the resulting exact design
print(exact_design)
```

Listing 16: An example call to the `Design` class's `round()` function.

In line 2, the sample size, N_{Tot}, is set to 10, and in line 4 the exact design is generated with a call to the `round()` function of the `design_object`. In line 6 the exact design is printed to the console output. An example of a exact design returned by the `round()` function for the relaxed design shown in Figure 20 is shown in Figure 21. The `round()` function here is called with a sample size of 10, however, the

```
   x1  x2 Variable  Replicats
0  -1  -1       y1          1
1  -1  -1       y2          1
2  -1   1       y1          1
3  -1   1       y2          2
4   1  -1       y1          1
5   1  -1       y2          1
6   1   1       y1          2
7   1   1       y2          1
```

Figure 21: An example of an exact design dataframe returned by the `round()` function.

relaxed design has 8 unique input-observation pairs, each with equal weighting. As an equal weighting of 8 points is not achievable with 10 observations, the rounding procedure will allocate these additional points randomly. As in the above case Adam's method does not always yield a unique apportionment, in which case currently a random selection is made. In future versions of the package the `rounding()`

function will be extended to allow more rounding methods as well as further analysis in cases where non-unique rounding occurs.

The `power()` function Often the user's ultimate goal is to constrain parameter values within reasonable confidence bounds of a prescribed width. Experimental equipment or cost may place some upper bound on the sample size in a given experiment, and in cases where this is quite restrictive the user will use the maximum sample size as the input for the `round()` function. If the design achieves the desired accuracy than the task is achieved, and if not the experimenter may be forced to iterate over multiple rounds of experimentation.

However, in some cases there may not be a restrictive upper bound on the sample size, but rather observations may be costly in either time or resources and the user may prefer smaller sample sizes but has no firm upper limit. In this context the user needs a method to examine trade-offs in sample size and confidence interval width across a range of feasible sample sizes. This type of analysis is similar to *power analysis* done for traditional regression models in empirical studies for social and medical sciences [15]. Power analysis can demonstrate how confidence intervals or other diagnostic metrics converge as the sample size increases.

The `power()` function is nor currently implemented but its intended role is to allow the user explore rounding of the optimal relaxed design for multiple sample sizes within a range. This process will allow the user to understand performance trade-offs across this range for the given design and to determine at what sample size threshold certain accuracy objectives are expected to be achieved. It is important to note that the same underlying relaxed design is used for each sample size to generate an exact design, however the exact designs will differ in replicate allocation. As the sample size increases the exact design can better approximate the optimal weights in the relaxed design. This generally results in a monotonic increase in performance for exact designs created with larger sample sizes; a larger sample size is almost always better. However, for some relaxed designs, certain integer sample sizes more accurately approximate the relaxed weights than other nearby, even larger, sample sizes. The larger sample sizes will always perform better, but the gains may be marginal relative to the cost for the experimenter. The `power()` function will provide graphical and quantitative methods to assess these trade-offs.

5 Examples

In this section, we give a description of the package workflow using specific models. These examples include code snippets and sample output, and specifically focus on how models are encoded in CasADi's symbolics, and how the experimental constraints are passed to the `Design` class. These example, among other, will serve as prototypes for first-time users in the package documentation so that those new to the package have several working examples to start from and modify when they seek to implement their own projects.

5.1 Optimal Design for Static Models

Here we begin by describing the NLOED package applied to a static model; one that does not require any numerical integration and therefore does not involve ODEs. As a first example we begin with a simple optogenetic dose response curve. Recall from previous chapters that an optogenetic system is one where gene expression can be activated by a the light intensity of a specific color. Here a Hill function is used to describe the expression of GFP as a function of the light intensity such that

$$GFP = \alpha_0 + \alpha \frac{Light^n}{K^n + Light^n}. \tag{18}$$

Here α_0, α, K and n are the parameters of interest, *Light* is the single experimental input and *GFP* is the observed expression of GFP and the single model observable. We assume some past experimental data has been collected for this model but that the user wishes to improve the accuracy of the confidence intervals as efficiently as possible. We assume the user is also confident enough to assert that the distribution of the GFP expression level is normally distributed with a standard deviation of about 5% of the mean expression level.

The user begins by starting their preferred Python interface and importing the required packages. Listing 17 shows the the required commands in lines 1-6.

```python
1  import numpy as np
2  import pandas as pd
3  import casadi as cs
4  import matplotlib.pyplot as plt
5  from nloed import Model
6  from nloed import Design
```

Listing 17: Import statements for using the NLOED package, including the `Model` and `Design` class as well as other common Python numerical libraries.

Next in Listing 18, the user creates CasADi symbols for the input and parameters. This occurs in lines 8 and 9, note that the names here can be arbitrary. These lines make use of CasADi's `cs.SX.sym()` function to create two symbol vectors. The call to `cs.SX.sym()` for creating the experimental input has a single dimension (second argument omitted). The second call to `cs.SX.sym()` for the parameters has four dimensions, one for each parameter of interest. In lines 11-14 we define the named parameters used in the model definition to be the exponentiated values of the parameters in the `parameters` vector. This is a log transformation and it ensures that the named parameter values are always positive. This type of transformation is an important consideration when working with bounded parameter ranges. Asymptotic expressions for parameter variability, like the Fisher information matrix, are more cumbersome to compute for bounded parameter domains and thus NLOED assumes the user either transforms their parameters to avoid the need for bounding or that the feasible parameter range is so distant from the bounds they can safely be ignored during model calibration. Here we have chosen a simple transformation to avoid the need for bounds. A log transformation can also sometimes improve the numerical performance of the design and fitting algorithms. The named parameters are given for clarity but the user could choose to perform the transformation and model definition together for brevity.

```python
7  #define input and parameter symbols
8  inputs = cs.SX.sym('inputs')
9  parameters = cs.SX.sym('parameters',4)
10 #log-transormation of the parameters
11 alpha0 = cs.exp(parameters[0])
12 alpha = cs.exp(parameters[1])
13 n = cs.exp(parameters[2])
14 K = cs.exp(parameters[3])
15 #define the deterministic model for the GFP mean
16 gfp_mean = alpha0 + alpha*inputs**n/(K**n+inputs**n)
17 #assume some hetroskedasticity, std_dev 5% of mean expression level
18 gfp_var = (0.05*gfp_mean)**2
19 #link the deterministic model to the sampling statistics (here normal mean and variance)
20 gfp_stats = cs.vertcat(gfp_mean, gfp_var)
21 #create a casadi function mapping input and parameters to sampling statistics (mean and var)
22 gfp_model = cs.Function('GFP',[inputs,parameters],[gfp_stats])
```

Listing 18: An example of building a CasADi function for the deterministic component of the optogenetic dose-response model, before making a call to the `Model` constructor.

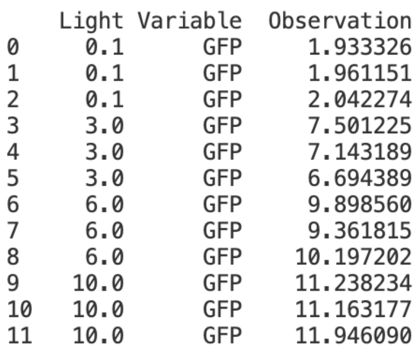
In line 16, the mean GFP response, `gfp_mean`, is defined in terms of the named parameters and the input. In line 18 the variance of the GFP observations, `gfp_var`, is defined to be the square of 5% of the mean GFP expression. In line 20 the mean and variance are concatenated into a single vector, this is mainly done for clarity and it could be merged into the following line. In line 22, a CasADi function, `gfp_model`, is defined using CasADi's `cs.Function()` constructor, this function maps the input and parameters to the GFP sampling statistics. Up to this point the naming of all variables and CasADi symbols was arbitrary, however now the string passed to the `Function()` constructor (in this case '`GFP`') will become the name of the observation variable within the NLOED Model.

Having created a CasADi symbol for the GFP observation variable, the user can now construct an NLOED `Model` instance. Listing 19 demonstrates this process. In line 24, the `gfp_model` function is tupled with the label `Normal` indicating that it describes the sampling statistics of a normal random variable. This tuple is placed in the list `observ_list`. If the model had more observation variables they would also be entered as tuples in the same list, however as the current model has a single observation dimension, the list is a singleton. In lines 26 and 28, names for the input and parameters are given in vectors `input_names` and `parameter_names` respectively. In line 30, the NLOED model constructor is called with the three preceding lists as arguments. The `model_object` variable now contains an instance of an NLOED `Model` class encoding the dose-response model.

```
23  # create observation list, add model function with 'Normal' label as tuple
24  observ_list = [(gfp_model,'Normal')]
25  #create names for inputs
26  input_names = ['Light']
27  #create names for parameters
28  parameter_names = ['log_Alpha0','log_Alpha','log_n','log_K']
29  #instantiate nloed model class
30  model_object = Model(observ_list,input_names,parameter_names)
```

Listing 19: Code showing the instantiation of an NLOED `Model` instance for the optogenetic dose-response model.

We assume the user has some preliminary data describing the GFP-intensity does response relationship, shown as a dataframne in Figure 22. Here there are triplicate observations at four different light intensities 0.1, 3.0, 6.0 and 10.0. (We use this dataset as a stand in for real experimental data, however it was actually generated using the `sample()` function from the `Model` class using parameter vector $[2, 10, 2, 3]$ which we pretend we do not know for this analysis.) We assume this initial data is contained in the dataframe `init_data`. The user can perform an initial fit to the preliminary data using



```
    Light Variable  Observation
0     0.1      GFP     1.933326
1     0.1      GFP     1.961151
2     0.1      GFP     2.042274
3     3.0      GFP     7.501225
4     3.0      GFP     7.143189
5     3.0      GFP     6.694389
6     6.0      GFP     9.898560
7     6.0      GFP     9.361815
8     6.0      GFP    10.197202
9    10.0      GFP    11.238234
10   10.0      GFP    11.163177
11   10.0      GFP    11.946090
```

Figure 22: An example dataset for the optogenetic dose-response model, with triplicate measurements of GFP taken at four different light levels.

the `Model` class's `fit()` function. Listing 20 demonstrates this procedure. In lines 32-33, fit options are set, specifying a range for the parameter pre-search. Here a 7^4 element grid of initial parameters vectors are distributed over the region in parameter space specified by the bound tuple list. The pre-fitting search evaluates each of these points for a good candidate starting point for the maximum likelihood optimization. Using the pre-search is ideal when an initial parameter guess is not possible, as in this case. In line 35, the `fit()` function of the `model_object` is used to fit the model to the data in `init_data`. In line 37, we extract the fit parameter values into a Numpy array, numerically they correspond to $[1.87, 12.20, 1.31, 3.72]$

```
31 #set options to use a simple initial search
32 fit_options={'InitParamBounds':[(-1,2),(1,3),(-1,2),(-1,2)],
33              'InitSearchNumber':7}
34 #fit the model to the initial data
35 fit_info = model_object.fit(init_data, options=fit_options)
36 #extract the parameter values
37 fit_params = fit_info['Estimate'].to_numpy().flatten()
```

Listing 20: Code showing the optogenetic dose-response being fit to an initial dataset using the `fit()` function.

To get a sense for the initial uncertainty in the parameter values we can use the `evaluate()` function in the `Model` class to generate the asymptotic covariance matrix for the inital data's design. Listing 21 shows this process, in line 39-41 we enter the design information for the initial dataset. In lines 43-44, the covariance matrix is requested and the method for computing the matrix is specified in the options dictionary. In line 46 the `evaluate()` function is called from the `model_object`, using the fit parameters values stored in the `fit_params` array. In lines 48-49, we compute the approximate Wald confidence interval bounds.

```
38 # enter the initial design information
39 init_design = pd.DataFrame({'Light':[.1,3,6,10],
40                             'Variable':['GFP']*4 ,
41                             'replicates':[3]*4})
42 #request the asymptotic covariance matrix
43 eval_options={'Method':'Asymptotic',
44               'Covariance':True}
45 # call evaluate() to compute the asymptotic covariance
46 asymptotic_covariance = model_object.evaluate(init_design,fit_params,eval_options)
47 #compute the asymptotic upper and lower 95\% bounds
48 asymptotic_lower_bound = fit_params - 2*np.sqrt(np.diag(asymptotic_covariance))
49 asymptotic_upper_bound = fit_params + 2*np.sqrt(np.diag(asymptotic_covariance))
```

Listing 21: Code showing the use of the `evaluate()` function to generate the asymptotic covariance matrix and Wald confidence interval bounds for the initial dataset.

The resulting confidence intervals are printed in Figure 23. As we know the 'true' values from which the data was generated, we can see the intervals contain the data-generating parameter vector, however the point estimate could certainly be improved. The user, who only has experimental data, would not know the true error, however the width of the confidence intervals is an indicator that accuracy could be improved.

```
           Lower   Estimate      Upper
Alpha0  1.500220   1.870615   2.332458
Alpha   7.020268  12.203428  21.213385
n       0.539345   1.307552   3.169942
K       1.649340   3.720461   8.392341
```

Figure 23: A print out of the returned 95% Wald confidence bounds for the optogenetic dose-response model fit to the initial dataset.

The parameter uncertainty captured in the asymptotic covariance matrix can also be used to approximate how much prediction uncertainty there is conditioned on our uncertainty in the parameter values and our knowledge of the sampling statistics. Listing 22 demonstrates how this is done using the `Model` class's `predict()` function; in line 51 the asymptotic covariance matrix is converted to a Numpy matrix. In line 53-54, we specify the model predictions that are desired for plotting. We request 100 light levels linearly spaced between 0.1 and 10, all of which are of the `GFP` observation variable. In lines 56-57, we specify that the `predict()` function should return both prediction and observation intervals in the options dictionary. In lines 59-62, the `predict()` function is called at the estimated parameter values.

```
#convert the covariance matrix to a Numpy array
covariance_matrix = asymptotic_covariance.to_numpy()
#select prediction intputs
prediction_inputs = pd.DataFrame({'Light':np.linspace(0.1,10,100),
                                  'Variable':['GFP']*100})
#request prediction and observation intervals
prediction_options = {'PredictionInterval':True,
                      'ObservationInterval':True}
#call predict()
predictions = model_object.predict(prediction_inputs,
                                   fit_params,
                                   covariance_matrix = covariance_matrix,
                                   options=prediction_options)
```

Listing 22: Code showing the use of the predict function to generate the mean dose response given the parameter estimates, along with 95% prediction and observation intervals using the asymptotic covariance matrix.

The result of the call to the `predict()` function is shown in Figure 24. Here the predicted mean GFP level at the parameter estimates is shown in dark blue. The blue region surrounding the prediction indicates the asymptotic approximation of the 95% confidence region for the mean GFP response given the parameter uncertainty. Here we can see that a large amount of uncertainty regarding model behaviour is concentrated in the light intensity ranges between 0 and 2. This indicates that the constraints the initial data places on the model leaves this region subject to great uncertainty and future experiments will ideally provide better constraints on this region. The orange region indicates the approximate 95% bounds on the data, meaning that given uncertainty in both the parameters and sampling error we would expect, approximately, that 95% of the data would fall in this region.

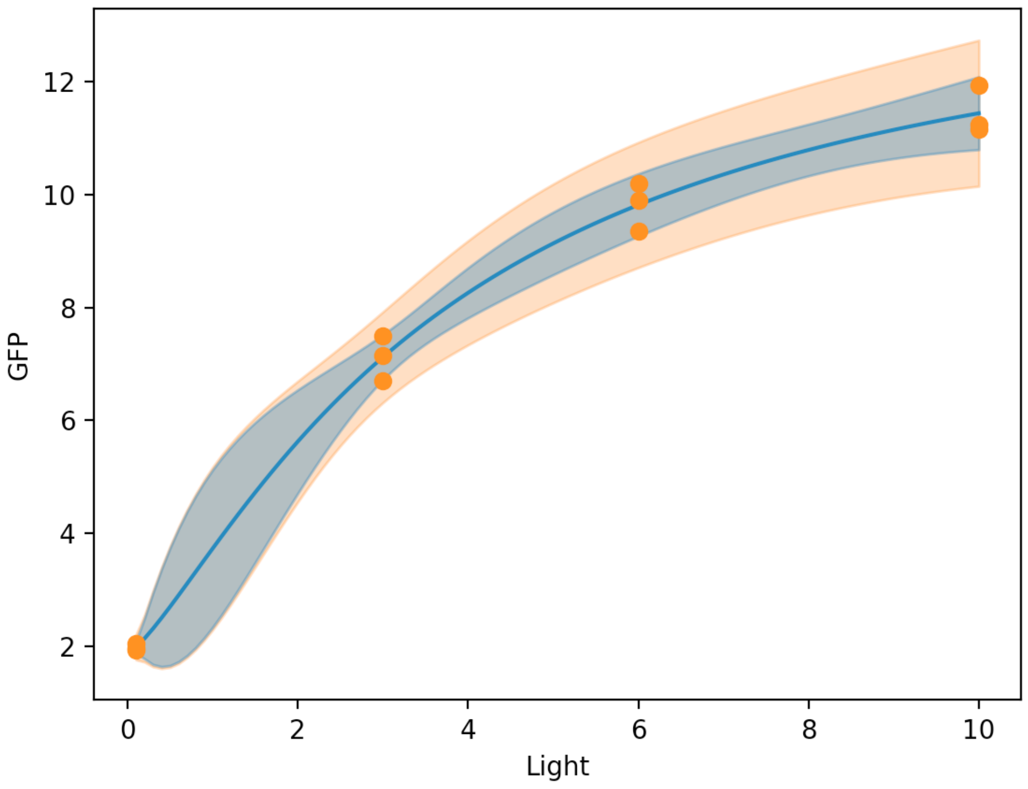


Figure 24: Mean GFP response (blue line), 95% prediction intervals (blue region) and 95% observation intervals (orange region) for the optogenetic model after fitting to the initial dataset (orange dots).

Given the analysis of the initial uncertainty in the parameter estimates and the prediction accuracy, the user will likely wish to improve model fit in the next round of planned experiments. Listing 23 shows the process of defining a `Design` class instance to generate an optimal design for the next set of measurements. Here we choose to treat the light level as continuous as the intensity can be varied to a fine degree in the lab relative the model's sensitivity over that range. In lines 64-67, we initialize the continuous input options so that we consider four unique light levels, just like in the initial experiment, that are free to be set between the light intensity bounds of 0.1 to 10. We assume these bounds are the limits of the experimental equipment. In line 68, we specify the initial design as having a weight of 0.33. Given the initial dataset contained 12 measurements, this assumes the next round of experiments will make approximately 24 GFP measurements. Here `init_design` contains the design for the initial dataset; it is a dataframe similar to the `init_data` but specifying replicate counts instead of observations. In lines 70-72, the `Design` class constructor is called and an instance is returned named `design_object`.

```
63  #set 'Light' as a continuous input with 4 unique levels
64  continuous_inputs={'Inputs':['Light'],
65                     'Bounds':[(.01,10)],
66                     'Structure':[['x1'],['x2'],['x3'],['x4']]}
67  #set fixed design dictionary with init design and weight
68  fixed_dict ={'Weight':0.33,'Design':init_design}
69  # generate the optimal design object
70  design_object = Design(model_object,fit_params,'D',
71                         fixed_design = fixed_dict,
72                         continuous_inputs = continuous_inputs)
73  #extract the relaxed design structure
74  relaxed_design = design_object.relaxed()
75  #set the sample size to 30
76  sample_size = 24
77  #generate a rounded exact design
78  exact_design = design_object.round(sample_size)
```

Listing 23: Code showing the process of creating an optimized `Design` object for the optogenetic dose-response model. Here the relaxed design is also returned and an exact design is generated through rounding.

In line 76, we use the `relaxed()` function of the `Design` class to return the relaxed design. This dataframe is shown in Figure 25. We can see that the optimal relaxed design consists of four unique light levels with asymmetric weights. It is not surprising that approximately 40% of the sampling weight is concentrated at a light level of 0.66, an input that is within the uncertainty bulge in the prediction interval plot in Figure 24. In line 76 of Listing 23, we set the sample size to 24 and in line 78 the `round()` function of

```
       Light Variable   Weights
0  10.000000      GFP  0.141446
1   0.659775      GFP  0.435226
2  10.000000      GFP  0.141500
3   2.698935      GFP  0.281828
```

Figure 25: Output of the relaxed design from the `relaxed()` function for the optogenetic dose-response model.

the `Design` class is used to generate an exact design. The exact design is shown in Figure 26.

```
       Light Variable  Replicats
0  10.000000      GFP          4
1   0.659775      GFP         10
2  10.000000      GFP          4
3   2.698935      GFP          6
```

Figure 26: Output showing the exact design for the optogenetic dose-response model generated from the `round()` function.

Before implementing the exact design in the laboratory it is useful to assess what the optimal designs expected utility is when combined with the initial data. This can be done by concatenating the initial and optimal design dataframes as follows:

```
combined_design = pd.concat([init_design, exact_design], ignore_index=True)
```

The `combined_design` dataframe can then analysed using the same code shown in Listings 21 and 22. The resulting expected asymptotic confidence intervals for the combined data are shown in Figure 31. Here we see that the intervals are expect to shrink considerably after adding the optimal data. In addition,

```
            Lower   Estimate      Upper
Alpha0   1.802369   1.870615   1.941445
Alpha   10.936137  12.203428  13.617575
n        1.186161   1.307552   1.441365
K        3.038102   3.720461   4.556078
```

Figure 27: Expected 95% confidence intervals for the optogenetic dose-response model parameters computed using the `evaluate()` function after combining the initial and optimal designs.

we can generate prediction and observation intervals with the expected asymptotic convariance matrix of the combined data in a similar manner to that used for the initial data. The prediction intervals are also expected to shrink considerably after implementing the optimal design, as shown in Figure 28

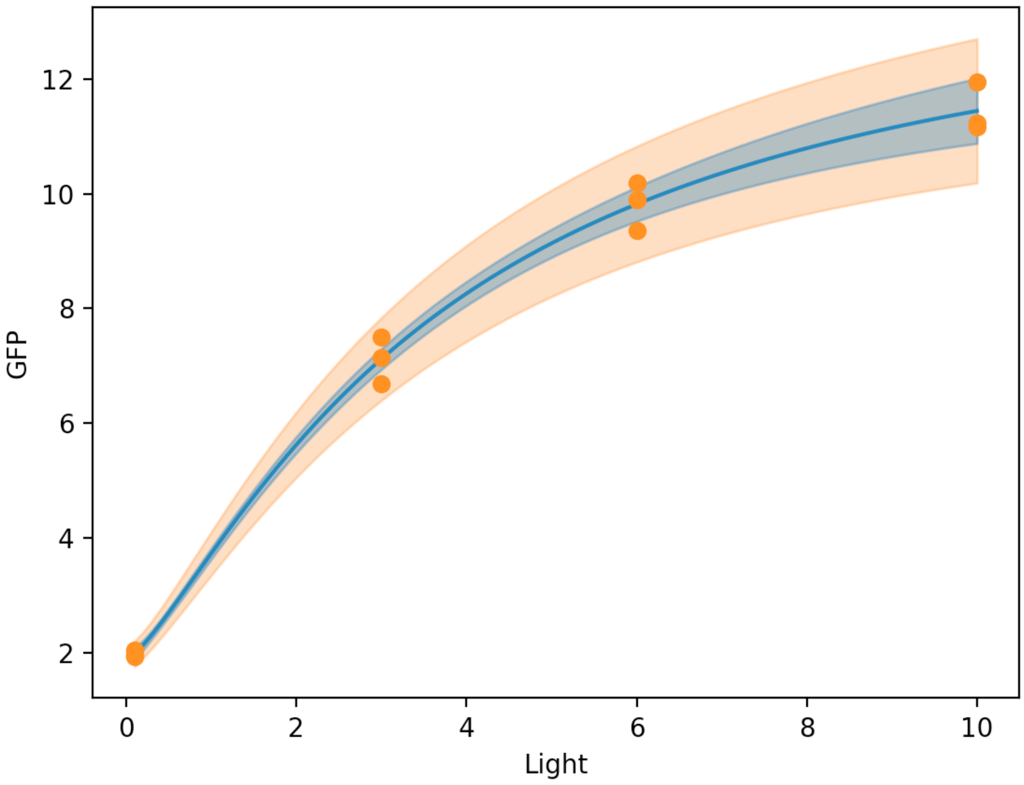


Figure 28: A plot of the 95% prediction and observation intervals under the asymptotic covariance matrix from the combined initial and optimal designs, computed with the `predict()` function.

At this point the user would perform the optimal experiment and return with the new data. Assuming the user has imported these observations into a dataframe named `optimal_data`, a combined dataset can be generated and the model can be re-fit. This process is shown in Listing 24. In line 81 the initial dataset and optimal dataset are combined into a single dataframe. In line 83, an options dictionary is created to request the likelihood contours from the fitting algorithm so that we can visualize the likelihood profiles and 2D contour projections as diagnostics for the final fit. In line 85, the `fit()` is called to fit the model to the combined dataset. In line 87, the fit parameters are extracted from the from the returned dataframe into a Numpy array. The resulting parameter vector is $[1.96, 10.04, 1.93, 2.98]$.

```
#combine the initial and optimal design
combined_data = pd.concat([init_data, optimal_data], ignore_index=True)
#request contours and use a simple initial search
fit_options={'Confidence':'Contours'}
#fit the model to the initial data
fit_info = model_object.fit(combined_data,start_param = fit_params,
↪  options=fit_options)
#extract the parameter values
fit_params = fit_info['Estimate'].to_numpy().flatten()
```

Listing 24: Code showing the concatination of the initial and optimal datasets, and their combined fitting to the optogenetic dose-response model. Fitting is called with a request for likelihood confidence contours for diagnostic purposes.

During fitting the `fit()` function generates the likelihood profiles, trace projections and contour projections shown in Figure 29. Here the profiles (blue curves along the diagonal plots) are nearly parabolic and the profile traces make ‘X’ shapes (blue and orange curves in the lower triangular plots). Some irregularity is detectable in the eccentricity of the 95% contour traces which are not quite elliptical, and some curvature can be seen in the profile traces which deviate from linearity at their endpoints. Overall this diagnostics looks reasonable and the data appears to constrain the model parameters well suggesting the approximations being used to assess parameter accuracy are themselves accurate. Likelihood based

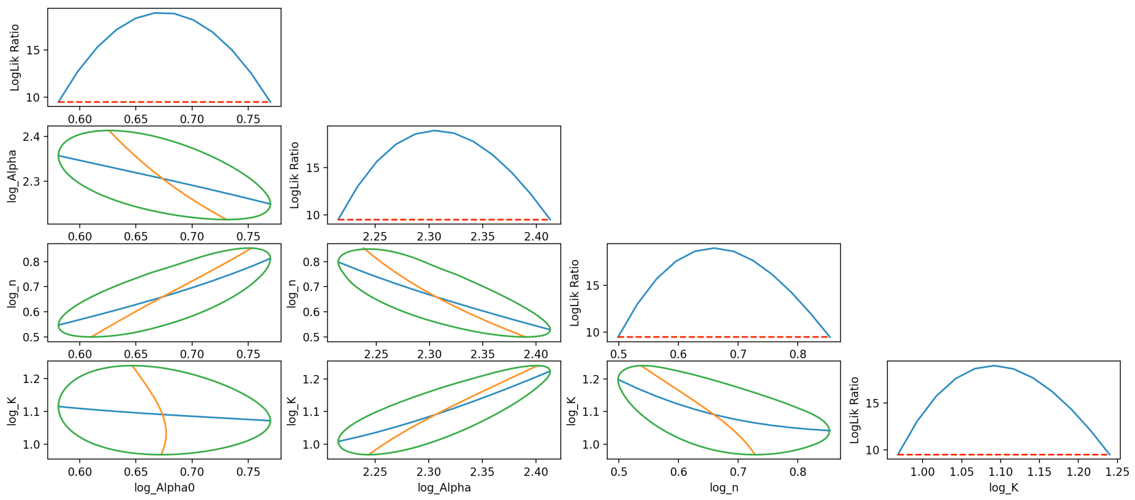


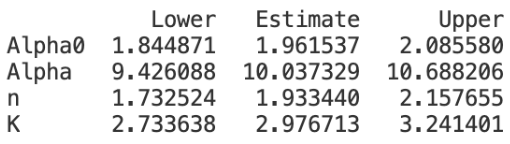
Figure 29: Diagnostic plots generated by the `fit()` function for the optogenetic dose-response model, including likelihood hood profiles, trace projections and contour projections.

intervals are also generated during fitting and are given in the returned dataframe. The likelihood intervals for this example are shown in Figure 30. Wald-type intervals can also be generated by using the

```
           Lower   Estimate      Upper
Alpha0  1.787926   1.961537   2.159234
Alpha   9.166230  10.037329  11.166391
n       1.647012   1.933440   2.348894
K       2.632790   2.976713   3.456342
```

Figure 30: Output of the 95% parameter confidence intervals generated using the profile likelihood functionality of the `fit()` function.

`evaluate()` function as shown in Listing 21. The Wald intervals for this example are shown in Figure 30. These intervals differ from those shown in Figure 27 only in that the parameter estimate has now shifted with the new data; this means the center of the interval moves but their widths are nearly the same



```
            Lower   Estimate      Upper
Alpha0   1.844871   1.961537   2.085580
Alpha    9.426088  10.037329  10.688206
n        1.732524   1.933440   2.157655
K        2.733638   2.976713   3.241401
```

Figure 31: Updated 95% Wald confidence intervals for the optogenetic dose-response model computed at the parameter estimates generated from the combined initial and optimal datasets.

under an exponential transformation (needed due to the log-transformed parameters). Comparing the likelihood-based and Wald-type intervals suggests reasonably good agreement in the intervals with the likelihood based intervals being slightly wider and more conservative. This lends support to the argument that the model has been calibrated with reasonable accuracy and that we are in a signal-to-noise regime where the approximations used in the design and diagnostics are accurate. In fact here, as we know the 'true' data-generating parameters, we can confirm that the resulting estimate is very near the true value.

5.2 Optimal Design for Dynamic Models

Encoding dynamic models in CasADi symbolics is somewhat more complicated than static models. The simplest method for doing so is to encode the numerical procedure itself as a symbolic structure. This has the advantage that the entire algorithm can be differentiated both for sensitivity and FIM computation, as well as within the optimizer. However this process can be confusing for first-time users and its scalability is limited to smaller dynamical systems. CasADi also offers external interfaces to third-party integration and sensitivity analysis packages such as CVODES as well as tools for creating differential-algebraic models [1]. In theory these can also be used in NLOED however they currently need further testing as of the date of writing.

As a first example of a dynamic system we will address a two-state model of gene expression, including states for both mRNA and protein levels. The dynamic system can be written as

$$\begin{aligned}\frac{d[\text{RNA}]}{dt} &= \frac{\alpha}{1+\frac{K}{[\text{Inducer}]}} - \delta[\text{RNA}],\\ \frac{d[\text{Protein}]}{dt} &= \frac{\beta}{1+\frac{L}{[\text{RNA}]}} - \gamma[\text{Protein}].\end{aligned} \tag{19}$$

Here expression of mRNA from the promoter is assumed to be controlled by an experimentally controlled inducer whose concentration is [Inducer] above. The state variable [RNA] is the mRNA concentration, the presence of which initiates translation of the corresponding protein. The concentration of the protien, [Protein], is the second state variable. Here α, β, K, L, δ and γ are parameters to be estimated. We assume the user is going to perform time series experiments on the proposed model and is looking to design such experiments for improved parameter estimation.

```
1  #create state variable vector
2  states = cs.SX.sym('states',2)
3  #create control input symbol
4  inducer = cs.SX.sym('inducer')
5  #create parameter symbol vector
6  parameters = cs.SX.sym('parameters',6)
7  #log-transformed parameters
8  alpha = cs.exp(parameters[0])
9  K = cs.exp(parameters[1])
10 delta = cs.exp(parameters[2])
11 beta = cs.exp(parameters[3])
12 L = cs.exp(parameters[4])
13 gamma = cs.exp(parameters[5])
14 #create symbolic RHS
15 rhs = cs.vertcat(alpha*inducer/(K + inducer) - delta*states[0],
16                  beta*states[0]/(L + states[0]) - gamma*states[1])
17 #create casadi RHS function
18 rhs_func = cs.Function('rhs_func',[states,inducer,parameters],[rhs])
```

Listing 25: Code showing the creation of a CasADi function for the RHS of a two-state ODE model for the mRNA-protein dynamic model.

Encoding a dynamic model in CasADi symbolics for NLOED begins with the same import commands as the static models, see Listing 17. Following this the user begins by creating a symbolic expression and CasADi function for the right-hand side (RHS) of the dynamic system. Listing 25 shows this process. In line 2, symbols are created for the state variables, mRNA and protein concentration. In line 4 a symbol is created for the inducer concentration. Line 6 shows the creation of a symbol vector for the model parameters. In lines 8-13, we perform a similar log transformation as that done in the static model, this ensures non-negativity of the named parameter values and improves numerical performance. In line 15-16, a symbol for the RHS of the system is created, containing symbolic expression for the equations given above. In line 18, a CasADi function is created mapping the current state and inducer levels as well as the parameters to the derivatives expressed by the RHS.

```
19 #time step size
20 dt = 1
21 # Create symbolics for RK4 integration, as shown in Casadi examples
22 k1 = rhs_func(states, inducer, parameters)
23 k2 = rhs_func(states + dt/2.0*k1, inducer, parameters)
24 k3 = rhs_func(states + dt/2.0*k2, inducer, parameters)
25 k4 = rhs_func(states + dt*k3, inducer, parameters)
26 state_step = states + dt/6.0 * (k1 + 2*k2 + 2*k3 + k4)
27 # Create a function to perform one step of the RK integration
28 step_func = cs.Function('step_func',[states, inducer, parameters],[state_step])
```

Listing 26: Code showing the creation of a symbolic implementation of a fourth-order Runge-Kutta integrator using the RHS function for the mRNA-protein dynamic model.

Recall that CasADi functions can be used to return numeric values or new symbolic expressions depending on the inputs provided to them. The CasADi function for the RHS can therefore be used to create new symbols for numerical time-stepping of the state vector by a given integration method. In Listing 26, imitating examples given in the CasADi documentation, we implement a fourth-order Runge-Kutta algorithm (RK4) [1]. In line 20, the time step-size is set, and in lines 22-24, the four incremental slopes are computed using the RHS CasADi function. The inputs to the RHS function are symbols: `states`, `inducer`, `parameters`, and therefore the slopes: `k1`, `k2`, `k3`, and `k4` are also symbols. In line 26, the incremental slopes are combined to create a symbol for a full time step of the length set in line 20. In line 28, a CasADi function is created mapping the current state, inducer and parameter values to the next state values.

```
29 # create a symbol for the initial inducer level
30 initial_inducer = cs.SX.sym('init_inducer')
31 #define the steady state initial states in terms of the initial inducer
32 init_mrna = (alpha/delta)*initial_inducer/(K+initial_inducer)
33 ini_prot = (beta/gamma)*init_mrna/(L+init_mrna)
34 # zip the initial states into a vector
35 initial_states = cs.vertcat(init_mrna, ini_prot)
36 #create a vector for inducer levels in each control interval
37 inducer_vector = cs.SX.sym('inducer_vec',3)
38 #merge all inducer levels into a single experimental inputs vector
39 inputs = cs.vertcat(initial_inducer,inducer_vector)
```

Listing 27: Code showing how the initial conditions for the mRNA-protein model's steady states are encoded symbolically. Composition of the experimental input vector is also illustrated.

The RK4 time-stepping CasADi function, `step_func()`, allows us to build up integrated solution curves of the model system as symbolic expressions. At this point we have a fair degree of flexibility in how the problem will be posed to NLOED via the `Model` class. We specifically need to choose how the initial conditions and inducer concentration throughout a single time series are encoded as NLOED model inputs, and how the state variables at various time points throughout the series are encoded as model observation variables. An example of our chosen formulation of the problem is shown in Listing 27 and 28. We have opted to have the system start at steady state with respect to a variable input level; in Listing 27, lines 30-35, the initial conditions are implemented symbolically. In line 30, a symbol, `initial_inducer`, is created for the initial inducer level. In lines 32-33 we compute the steady state values of the two state variables in terms of the initial inducer concentration. In line 35, we concatenate the initial state variables into a single vector, `initial_states`, containing the initial states of the system. In line 37, we create a vector, `inducer_vector`, containing the three inducer levels, one for each of the three control intervals over the duration of the experiment (see further discussion below). In line 39 we concatenate the `initial_inducer` and `inducer_vector` to form the overall set of inputs for the designed time-series experiment. This means each experiment has four experimental inputs, the initial inducer concentration to which the system starts in steady state, and three other inducer concentrations applied sequentially through the experimental duration.

```
40 #3 samples per cntrl interval, 3 cntrl intervals
41 #control intervals are 1+2+3=6 steps long:
42 #  cntrl_int1    cntrl_int2    cntrl_int3
43 #|-1---2-----3||-1---2-----3||-1---2-----3|
44 #set number of control intervals
45 num_cntrl_intervals = 3
46 #define a sample pattern to apply in each control interval
47 sample_pattern= [1,2,3]
48 #lists to store symbols for each sample point, and times of each sample
49 sample_list, times = [], []
50 # set the initial states and initialize the step counter
51 current_state, step_counter = initial_states, 0
52 #loop over control invervals
53 for interval in range(num_cntrl_intervals):
54   # loop over sample pattern
55   for num_stps in sample_pattern:
56     #iterate steps indicated by sample pattern
57     for k in range(num_stps):
58       #propagate the state variables via integration
59       current_state = step_func(current_state, inducer_vector[interval], parameters)
60       step_counter+=1
61     #save the state symbols and times of each sample
62     sample_list.append(current_state)
63     times.append(step_counter*dt)
```

Listing 28: Code showing how the symbolic integration is used to implement a specific sampling and observation pattern, and how observation variables for various states and time points are collected.

Having prescribed the model inputs we now need to collect the observation variables (at multiple time points) into an observation list for the `Model` constructor. To do so we must increment over the time series using the RK4 time stepping, implementing the inducer control scheme and collecting time points through the looping process. This procedure is shown in Listing 28. The current time-stepping algorithm is ideal for a step-wise input in the inducer concentration and we choose to implement three control intervals, meaning that each time course can have three different levels of inducer concentration, occurring sequentially each for the same length of time. In line 45, we therefore set the number of control intervals to three. In our case, the control intervals are assumed to be relatively long compared to the relaxation time of the system, as such we will want the option to select several observation time points within each sampling interval. Here we choose to place candidate observation points at one, two and three time steps after the beginning of the control interval. In line 47 we create a list containing the number of time steps to advance before taking an observation. Both the number of control intervals, and the time steps before observation, can be used to construct the looping structure to iterate over the experiment. In line 49, we create lists to store CasADi symbols for the observed state at various time points as well as the times at which these occur. In line 51, we initialize the system state to the steady state values in the `initial_state` variable and set the step counter to 0. Beginning on line 53, we loop over the three control intervals; each pass through this loop integrates a single control interval over which there is a constant inducer level. On line 57 we loop over the `sample_pattern` array, such that the loop variable `num_stps` will contain the number of integration steps that need to be incremented. This loop effectively iterates over observations within each control interval. As there are three possible observation within each control interval, this loop will repeat three times for each control interval; however, note that the time between observations is not the same, as set by the `sample_pattern` array. In line 57, we enter the time stepping and iterate over the specified number of steps using `step_func`. In the time stepping loop, we apply the RK4 step function in line 59 to the current state vector in `current_state` with the prescribed inducer level specified by `inducer_vector[interval]`. The step counter is also incremented in line 60, to track the number of RK4 steps taken and thus the time elapsed. In line 62 we store the state vector, which at this point corresponds to an observation time point; the time is also computed and recorded in line 63.

```
64 # create list for observation structure
65 observation_list= []
66 #create list to store response names
67 observation_names, observation_type, observation_times = [], [], []
68 # loop over samples (time points)
69 for i in range(len(sample_list)):
70   #create a unique name for mrna and prot samples
71   mrna_name = 'mrna_'+'t'+"{0:0=2d}".format(times[i])
72   prot_name = 'prot_'+'t'+"{0:0=2d}".format(times[i])
73   #create mean and var tuple for mrna and prot observ.
74   mrna_stats = cs.vertcat(sample_list[i][0], 0.005)
75   prot_stats = cs.vertcat(sample_list[i][1], 0.005)
76   #create casadi function for mrna and prot stats
77   mrna_func = cs.Function(mrna_name,[inputs,parameters],[mrna_stats])
78   prot_func = cs.Function(prot_name,[inputs,parameters],[prot_stats])
79   #append the casadi function and distribution type to obs struct
80   observation_list.extend([(mrna_func,'Normal'), (prot_func,'Normal')])
81   #store observation names, useful for plotting
82   observation_names.extend([mrna_name,prot_name])
83   #store observation type
84   observation_type.extend(['RNA','Prot'])
85   #store observation time
86   observation_times.extend([times[i]]*2)
```

Listing 29: Code showing the assembly of the `observ_list` argument for use in the `Model` class constructor call for the mRNA-protein dynamic model.

After completing the loop structure shown in Listing 28, the result is a list, `sample_list`, containing symbols for the each state variable at the observation time points under consideration. It remains for us to construct CasADi functions for each of these symbols, before they can be passed to the `Model` constructor via an observation list. We also need to specify the distribution type assumed for each state variable; here we assume a normal distribution with constant variance to keep the model simple for illustrative purposes. Listing 29 provides a loop in which the symbols list is used to construct CasADi functions each of which is then stored in an observation list for passage to the `Model` constructor. In line 65, the `observation_list` list is declared and in lines 67 several other lists are initialized for storing the observation name, type (RNA or protein) and time; these are valuable for plotting and organizing data later. On line 69, the loop for building up the observation list begins; this loop iterates over each element in the sample list containing the observation symbols. In lines 71-72, each state's observation is given a unique string name, marking its type and time. On lines 74-75, the state observation statistics (mean and variance) are concatenated together. On lines 77-78 we construct CasADi functions mapping experimental inputs (the various inducer levels) and the model parameters to the observation statistics. On line 80, we append the CasADi functions in tuples with the `Normal` label, to the observation list. In lines 83-86 we also store the auxiliary information for plotting and data export in the previously declared lists.

```
87 #list the inpit and parameter names
88 input_names = ['Init_Inducer','Inducer_1','Inducer_2','Inducer_3']
89 parameter_names = ['log_Alpha','log_K','log_Delta','log_Beta','log_L','log_Gamma']
90 #instantiate the model object
91 model_object = Model(observation_list, input_names, parameter_names)
```

Listing 30: Code showing the creation of an NLOED `Model` object for the mRNA-protein dynamic model.

Following the creation of the observation structure in the previous loop, we are ready to create the NLOED model. Shown in Listing 30, in lines 88 and 89 we declare the input and parameter names and on line 91 we create an NLOED model object named `model_object` using the NLOED `Model` class constructor.

```
          Lower  Estimate     Upper
Alpha  1.458430  1.900728  2.477161
K      0.807291  1.407532  2.454068
Delta  0.953771  0.991747  1.031235
Beta   1.833569  2.404468  3.153122
L      0.208740  0.430277  0.886932
Gamma  0.478522  0.539838  0.609012
```

Figure 32: Output of the 95% Wald confidence intervals under the initial dataset for the mRNA-protein dynamic model, computed using the `evaluate()` function.

We can use code similar to code listings 20, 21 and 22 in the static model to for fitting, generating Wald-type confidence intervals and prediction bounds. In this example, we assume an initial dataset with a single observation at each possible time point during a single time course. We assume the time course begins with an initial inducer concentration of 0 and the subsequent three inducer levels are set to 1, 0 and 3 respectively. Fitting is done as in the static model using the `model_object` and the `fit()` function. Wald intervals are shown in Figure 32. The size of these bounds relative to the magnitude of the parameters is reasonably large and could be improved. Prediction and observation intervals can also

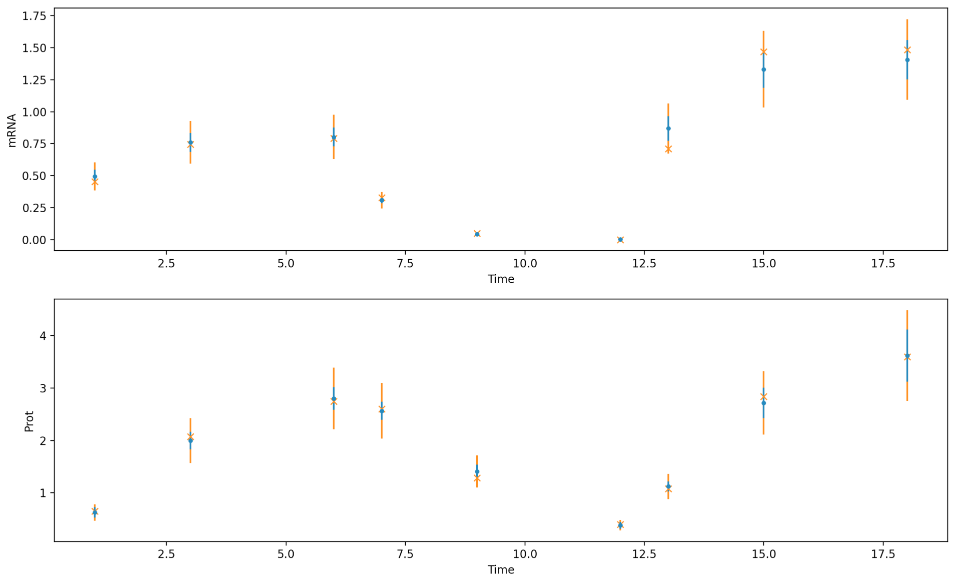


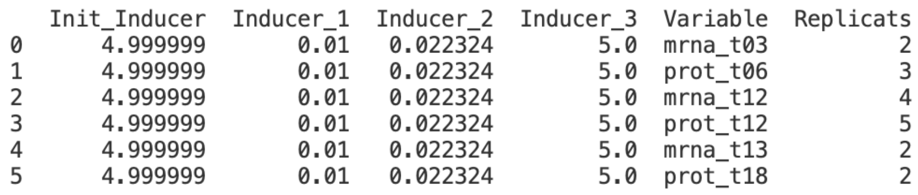
Figure 33: Prediction of the mean mRNA and protein levels, along with the 95% prediction and observation intervals generated with the `predict()` function for the mRNA-protein dynamic model.

be generated using calls to the `evaluate()` function to generate parameter covariances, and calls to the predict function to generate the mean and interval information. Figure 33 depicts the predicted mean response at each observation point in the initial experiment (shown as blue dots). Data from a simulation of the initial experiment is also shown (orange x's), along with prediction intervals for the mean under parameter uncertainty (blue bands) and observation intervals for the data under both sampling and parameter uncertainty (orange bands).

```python
92  #set all inducer inputs as continuous
93  continuous_inputs={'Inputs':['Init_Inducer','Inducer_1','Inducer_2','Inducer_3'],
94                     'Bounds':[(.01,5),(.01,5),(.01,5),(.01,5)],
95                     'Structure':[['I0','I1','I2','I3']]}
96  #create the fixed design dictionary
97  fixed_dict ={'Weight':0.5, 'Design':init_design}
98  # generate the optimal design object
99  design_object = Design(model_object,fit_params,'D',
100                        fixed_design = fixed_dict,
101                        continuous_inputs = continuous_inputs)
102 #set the sample size to 18
103 sample_size = 18
104 #generate a rounded exact design
105 exact_design = design_object.round(sample_size)
```

Listing 31: Code showing the instantiation of a `Design` object for the mRNA-protein model, as well as the generation of an exact design with a sample size of 18.

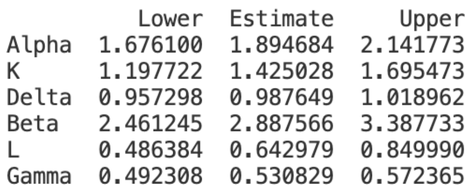
To generate an optimal experiment we need to instantiate a `Design` instance; Listing 31 shows this process. In lines 93-95 we specify that all inputs will be treated as continuous, bounded between 0.1 and 5 and that a single inducer profile will be optimized. In line 97, we create a `fixed_design` dictionary and include the initial design (contained in the dataframe `init_desgin`) and weight it at 0.5. This weighting of the initial design implies the optimal design will be implemented with the same number of observations, 18, as the initial data. In lines 99-101, we call the `Design` constructor and instantiate the `design_object`. in line 103, we set the sample size to 18, and in line 105 we generate an exact design with the `round()` function.



```
   Init_Inducer  Inducer_1  Inducer_2  Inducer_3  Variable  Replicats
0      4.999999       0.01   0.022324        5.0  mrna_t03          2
1      4.999999       0.01   0.022324        5.0  prot_t06          3
2      4.999999       0.01   0.022324        5.0  mrna_t12          4
3      4.999999       0.01   0.022324        5.0  prot_t12          5
4      4.999999       0.01   0.022324        5.0  mrna_t13          2
5      4.999999       0.01   0.022324        5.0  prot_t18          2
```

Figure 34: Output of the optimal exact design generated for the mRNA-protein dynamic model.

The optimal design generated from this process is shown in Figure 34. The resulting design sets an initial inducer concentration of 5 and then drops it to the minimum of 0.01 in the first control interval. In the second control interval, the inducer is increased slightly to 0.022 and in the final interval it is again increased to the maximum of 5. Various protein and RNA measurements are taken at different



```
          Lower  Estimate     Upper
Alpha  1.676100  1.894684  2.141773
K      1.197722  1.425028  1.695473
Delta  0.957298  0.987649  1.018962
Beta   2.461245  2.887566  3.387733
L      0.486384  0.642979  0.849990
Gamma  0.492308  0.530829  0.572365
```

Figure 35: Updated 95% confidence intervals for the mRNA-protein model, generated using the `evaluate()` function and the combined initial and optimal designs.

times (coded in the `Variable` column) and replicate numbers. This type of design is possible if multiple biological replicates can be run with a shared inducer level. For example light or temperature induced systems may be suitable for this type of input and replication structure. In these situations multiple cultures can be grown (and thus multiple replicates sampled) in the same inducer sequence within their incubator or light box. If the user instead wishes to optimize an experiment where each replicate can have its own unique inducer course but multiple replicate observations cannot be made at each time point,

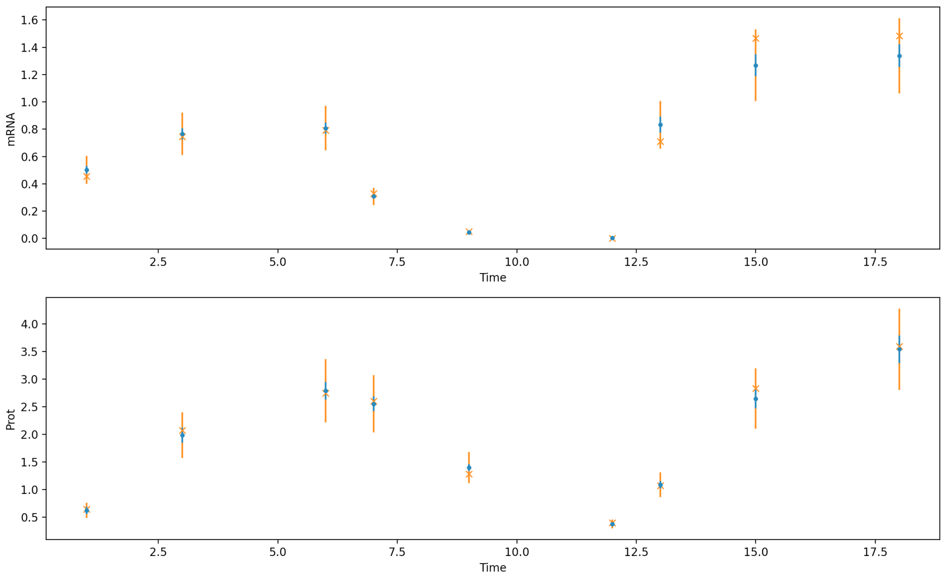


Figure 36: Updated 95% prediction and observation intervals for the mRNA-protein model, generated using the `predict()` function under the covariance found using `evaluate()` applied to the combined intitial and optimal designs.

the user can use the `LockWeights` option of the `Design` class and a different continuous input structure. However use of the `LockWeights` option prohibits time point selection and the user must use a fixed sampling schedule. After implementing the optimal design we can update the Wald intervals, shown in Figure 35. Here we see that interval sizes have shrunk. Figure 36 shows the new prediction and intervals for the original dataset. Here we see that the prediction intervals (blue bands) have shrunk somewhat from the initial data collection. Further improvements would be expected if the sample size was increased for the optimal design.

6 Discussion

The current version of NLOED implements much of the core functionality and many important supplemental features, however I intend to add a number of additional features before the initial release. Currently more testing and further user-interface design is needed before the NLOED package can support optimal design over model or parameter uncertainty. Specifically, in future releases the `models` argument to the `Design` class will accept a list of `Model` instances as along as they have the same input structure. With a list of `Model` instances passed, the resulting design is optimized for an average objective across all of the models. This model averaging provides designs that can be used to fit any of the candidate models, providing robustness to model uncertainty (but not necessarily addressing model selection). In future releases, the local design optimization at a nominal parameter vector, $\bar{\theta}$, will also be supplemented with a pseudo-Bayesian approach which will allow for design optimization over a weighted average of several candidate parameter points [37]. To accomplish this the `parameters` argument to the `Design` class will accept either mean and covaraiance information for a normal prior, or a list of weighted candidate parameter vectors. Much of the numerical structures for these model and parameter uncertainty features have already been implemented but they need further testing and documentation. Also, while the D-optimal objective is widely used and useful, support for D$_s$-optimal designs is also planned for the initial release as the D$_s$ objective facilitates designs targeting specific parameters sets or even model selection in nested models [2]. The `power()` function of the `Design` class remains to be implemented; while not required for optimal design it provides important tools for assessing the sensitivity of confidence intervals to the overall sample size. I also plan to provide greater flexibility and more user feedback within the `round()` function in the `Design` class so the user has more guidance and control over the generation of exact designs. Lastly, I hope to improve the numerical stability of model fitting, likelihood profiling and the design optimization algorithms through additional testing and tuning of the default algorithm settings.

I also have several secondary objectives for package extensions. These include adding diagnostics based explicitly on the general equivalence theorem, which in special cases can be used as a powerful tool for checking optimality and understanding design structure [23, 2]. I also aim to improve the user experience in implementing dynamic models, including helper functions for implementing the CasADi symbolic integration and for extracting data and design information at various time points. While NLOED already supports a large variety of experimental constraints, in future version I plan to allow the user to place non-linear constraints on continuous inputs as well as the sampling weights. I also hope to integrate other powerful numerical tools available through CasADi's interfaces, including other optimization, numerical integration and sensitivity analysis packages (i.e. CVODES), and tools for constrained and differential-algebraic systems. Using these tools I would like to expand support and code optimization for models involving implicit functions, optimization and differential-algebraic systems in the future.

Optimal experimental design has a long history of research; algorithms for optimizing designs for non-linear, dynamic models dates back to the 1950's, see work by Box and Lucas [8]. In over a half-century of study, the area has been well researched and the theory, as well as numerical procedures, for optimal design thoroughly explored. In many ways it is a mature field, with few prospects for revolutionary new ideas. However, it remains a challenge to translate the existing body of knowledge into practice for an evolving experimental discipline like systems biology; updated tools that focus on usability and easy-adoption are needed. While systems biology makes use of (psuedo-)mechanistic models and designed experimentation, model parameters are often only accessible via fitting. This situation poses unique challenges as the models can have complicated non-linear structures with a variety of observables, like in more fundamental sciences, but model calibration and testing often require statistical tools more common in econometrics and other medical or social sciences. Systems biology also includes a wide range of practitioners, from experimentalists with non-quantitative backgrounds from many areas of biology, to more quantitative experts from fields like physics, engineering, computer science, and mathematics. Translating OED ideas into a set of tools that is easy to use yet flexible for such a wide range of models and users is difficult, especially as even the quantitative practitioners may lack familiarity with the advanced statistical diagnostics and fitting concepts on which OED relies.

The NLOED package attempts to address some of the opportunities and challenges listed above. While other packages exist, NLOED supports a unique blend of models and numerical algorithms specifically suited to systems biology models; with special consideration for the diversity of model structures (non-linear, dynamic, multi-input/output) and numerical implementations (numerically integrated and even implicitly defined models). NLOED also incorporates these tools with an updated toolset, making use of automatic differentiation and state-of-the-art open-source optimization via CasADi. This provides a flexible and extensible platform on which to build future capabilities. The package has been written in open-source object-oriented Python with special attentions being paid to interoperability with other third-party packages like Numpy and Pandas. Providing an OED package in Python is ideal as Python blends the open-source licensing and graphical support of languages like R, with the engineering and dynamic systems toolset found in MALAB, while also bringing its own strengths as a fully-functioning programming language. For user convenience and easy-adoption, NLOED supplements the core design capabilities with a flexible set of supporting functions for fitting, diagnostics and simulation – with equal attention paid to optimization as there is to qualitative assessment and modularity. These combined features differentiate NLOED from other software tools, giving it a modern implementation, well suited to systems biology, and good prospects for future extensibility.

References

[1] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation (In Press)*, 2018.

[2] Anthony Atkinson and Alexander Donev. *Optimum Experimental Designs*. Oxford University Press, 1992.

[3] Anthony Atkinson, Alexander Donev, Randall Tobias, et al. *Optimum experimental designs, with SAS*, volume 34. Oxford University Press, 2007.

[4] Anthony C Atkinson. The usefulness of optimum experimental designs. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):59–76, 1996.

[5] Samuel Bandara, Johannes P Schlöder, Roland Eils, Hans Georg Bock, and Tobias Meyer. Optimal experimental design for parameter estimation of a cell signaling model. *PLoS Computational Biology*, 5(11):e1000558, 2009.

[6] Douglas M Bates and Donald G Watts. *Nonlinear regression analysis and its applications*, volume 2. Wiley New York, 1988.

[7] Caroline Bazzoli, Sylvie Retout, and France Mentré. Design evaluation and optimisation in multiple response nonlinear mixed effect models: PFIM 3.0. *Computer methods and programs in biomedicine*, 98(1):55–65, 2010.

[8] George EP Box and HL Lucas. Design of experiments in non-linear situations. *Biometrika*, 46(1/2):77–90, 1959.

[9] N Braniff, M Reed, and B Ingalls. Optimal experimental design for characterizing gene expression: sample scheduling. *IFAC-PapersOnLine*, 51(19):48–51, 2018.

[10] Nathan Braniff and Brian Ingalls. New opportunities for optimal design of dynamic experiments in systems and synthetic biology. *Current Opinion in Systems Biology*, 2018.

[11] Nathan Braniff, Addison Richards, and Brian Ingalls. Optimal experimental design for a bistable gene regulatory network. *IFAC-PapersOnLine*, 52(26):255–261, 2019.

[12] Nathan Braniff, Matthew Scott, and Brian Ingalls. Component characterization in a growth-dependent physiological context: optimal experimental design. *Processes*, 7(1):52, 2019.

[13] Ankush Chakrabarty, Gregery T Buzzard, and Ann E Rundell. Model-based design of experiments for cellular processes. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 5(2):181–203, 2013.

[14] Merlise A Clyde. An object-oriented system for bayesian nonlinear design using xlisp-stat. Technical report, University of Minnesota, 1993.

[15] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[16] Dirk JW De Pauw and Peter A Vanrolleghem. Avoiding the finite difference sensitivity analysis deathtrap by using the complex-step derivative approximation technique. *International Congress on Environmental Modelling and Software.*, 24, 2006.

[17] Valerii V Fedorov and Sergei L Leonov. *Optimal design for nonlinear response models*. CRC Press, 2013.

[18] Gaia Franceschini and Sandro Macchietto. Model-based design of experiments for parameter precision: State of the art. *Chemical Engineering Science*, 63(19):4846–4872, 2008.

[19] Ulrike Groemping. CRAN task view: Design of experiments (DoE) & analysis of experimental data, 2020.

[20] Michael Hucka, Frank T Bergmann, Claudine Chaouiya, Andreas Dräger, Stefan Hoops, Sarah M Keating, Matthias König, Nicolas Le Novère, Chris J Myers, Brett G Olivier, et al. The systems biology markup language (sbml): language specification for level 3 version 2 core release 2. *Journal of integrative bioinformatics*, 16(2), 2019.

[21] Lorens Imhof, Jesús Lopez-Fidalgo, and Wing-Keung Wong. Efficiencies of rounded optimal approximate designs for small samples. *Statistica Neerlandica*, 55(3):301–318, 2001.

[22] Mads Kaern, Timothy C Elston, William J Blake, and James J Collins. Stochasticity in gene expression: from theories to phenotypes. *Nature Reviews Genetics*, 6(6):451–464, 2005.

[23] Jack Kiefer and Jacob Wolfowitz. The equivalence of two extremum problems. *Canadian Journal of Mathematics*, 12(363-366):234, 1960.

[24] Clemens Kreutz, Andreas Raue, Daniel Kaschek, and Jens Timmer. Profile likelihood in systems biology. *The FEBS journal*, 280(11):2564–2571, 2013.

[25] Clemens Kreutz and Jens Timmer. Systems biology: experimental design. *The FEBS Journal*, 276(4):923–942, 2009.

[26] Michel Laurent and Nicolas Kellershohn. Multistability: a major means of differentiation and evolution in biological systems. *Trends in biochemical sciences*, 24(11):418–422, 1999.

[27] Lennart Ljung. *System Identification: Theory for the User.* Prentice Hall PTR, 1999.

[28] Joakim Nyberg, Caroline Bazzoli, Kay Ogungbenro, Alexander Aliev, Sergei Leonov, Stephen Duffull, Andrew C Hooker, and France Mentré. Methods and software tools for design evaluation in population pharmacokinetics–pharmacodynamics studies. *British journal of clinical pharmacology*, 79(1):6–17, 2015.

[29] Joakim Nyberg, Sebastian Ueckert, Eric A Strömberg, Stefanie Hennig, Mats O Karlsson, and Andrew C Hooker. PopED: an extended, parallelized, nonlinear mixed effects models optimal design tool. *Computer methods and programs in biomedicine*, 108(2):789–805, 2012.

[30] Antony M Overstall and David C Woods. Bayesian design of experiments using approximate coordinate exchange. *Technometrics*, 59(4):458–470, 2017.

[31] Antony M Overstall, David C Woods, and Ben M Parker. Bayesian optimal design for ordinary differential equation models with application in biological science. *Journal of the American Statistical Association*, pages 1–16, 2020.

[32] Yudi Pawitan. *In all likelihood: statistical modelling and inference using likelihood.* Oxford University Press, 2001.

[33] Friedrich Pukelsheim and Sabine Rieder. Efficient rounding of approximate designs. *Biometrika*, 79(4):763–770, 1992.

[34] Paul Rilstone, Virendra K Srivastava, and Aman Ullah. The second-order bias and mean squared error of nonlinear estimators. *Journal of Econometrics*, 75(2):369–395, 1996.

[35] Jakob Ruess, Francesca Parise, Andreas Milias-Argeitis, Mustafa Khammash, and John Lygeros. Iterative experiment design guides the characterization of a light-inducible gene expression circuit. *Proceedings of the National Academy of Sciences USA*, 112(26):8148–8153, 2015.

[36] Jan Schellenberger, Richard Que, Ronan MT Fleming, Ines Thiele, Jeffrey D Orth, Adam M Feist, Daniel C Zielinski, Aarash Bordbar, Nathan E Lewis, Sorena Rahmanian, et al. Quantitative prediction of cellular metabolism with constraint-based models: the cobra toolbox v2. 0. *Nature protocols*, 6(9):1290, 2011.

[37] R Schenkendorf, A Kremling, and M Mangold. Optimal experimental design with the sigma point method. *IET systems biology*, 3(1):10–23, 2009.

[38] George AF Seber and Christopher John Wild. Nonlinear regression. hoboken. *New Jersey: John Wiley & Sons*, 62:63, 2003.

[39] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[40] Adalbert Wilhelm. How to obtain efficient exact designs from optimal approximate designs. In *COMPSTAT*, pages 495–500. Springer, 1996.