Optimal Experimental Design Applied to Models of Microbial Gene Regulation

by

Nathan Braniff

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Applied Mathematics

Waterloo, Ontario, Canada, 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Julio Banga Professor, IIM-CISC
Supervisor:	Brian Ingalls Professor, Applied Mathematics, University of Waterloo
Internal Members:	Matt Scott Assoc. Professor, Applied Mathematics, University of Waterloo Kirsten Morris Professor, Applied Mathematics, University of Waterloo
Internal-External Member:	Trevor Charles Professor, Biology, University of Waterloo

Authour's Declaration

Statement of Contributions

Chapters 1, 2, 7, & 8: I am the sole author of this material, under the supervision of Brian Ingalls. This material has not been previously published.

Chapter 3: This material has been previously published as:
"New opportunities for optimal design of dynamic experiments in systems and synthetic biology" by Nathan Braniff, and Brian Ingalls in *Current Opinion in Systems Biology* 9 (2018): 42-48

I co-authored this manuscript with Brian Ingalls. My contribution was gathering the candidate references and in selecting the initial areas of focus for the article. I also wrote the initial draft of the manuscript, which Brian and I then collaborated on to produce the final text.

Chapter 4: This material has been previously published as:
"Component Characterization in a Growth-Dependent Physiological Context: Optimal Experimental Design" by Nathan Braniff, Matt Scott, and Brian Ingalls in *Processes* 7.1 (2019): 52.

I am the primary author of this manuscript, with edits and intellectual contributions from Matt Scott and Brian Ingalls. The choice of project, the mathematical model and the implementation of the optimal design algorithms were my work, with some guidance and suggestions from Matt and Brian.

Chapters 5: This material has been previously published as:
"Optimal experimental design for characterizing gene expression: sample scheduling" by Nathan Braniff, Max Reed, and Brian Ingalls in *IFAC-PapersOnLine* 51.19 (2018): 48-51.

I am the primary author of this manuscript, with edits and intellectual contributions from Max Reed and Brian Ingalls. My contribution was in developing and implementing the optimal scheduling method; Max Reed investigated some of the case studies used in demonstrating its utility.

Chapters 6: This material has been previously published as:
"Optimal Experimental Design for a Bistable Gene Regulatory Network" by Nathan Braniff, Addison Richards, and Brian Ingalls in *IFAC-PapersOnLine* 52.26 (2019): 255-261.

I am the primary author of this manuscript, with edits and intellectual contributions from Addison Richards and Brian Ingalls. My contribution was developing and implementing the optimal experimental procedure for the bi-stable motif; Addison Richards implemented a stochastic simulation algorithm for assessing the design methodology and approximations used in this work.

Abstract

Microbial gene expression is a comparatively well understood process, but regulatory interactions between genes can give rise to complicated behaviours. Regulatory networks can exhibit strong context dependence, time-varying interactions and multiple equilibrium. The qualitative diagrammatic models often used in biology are not well suited to reasoning about such intricate dynamics. Fortunately, mathematics offers a natural language to model gene regulation because it can quantify the various system inter-dependencies with much greater clarity and precision. This added clarity makes models of microbial gene regulation a valuable tool for studying both natural and synthetic gene regulatory systems.

However models are only as good as the knowledge and assumptions they are built on. Specifically, all models depend on unknown parameters – constant that quantify specific rates and interaction strengths within the regulatory system. In systems biology parameters are generally fit, rather than measured directly, because their values are contextually dependent on state of the microbial host. This fitting requires collecting observations of the modeled system. Exactly what is measured, how many times and under what experimental conditions defines an *experimental design*. The experimental design is intimately linked to the accuracy of any resulting parameter estimates for a model, but determining what experimental design will be useful for fitting can be difficult. Optimal experimental design (OED) provides a set of statistical techniques that can be used make design choices that improve parameter estimation accuracy.

In this thesis I examine the use of OED methods applied to models of microbial gene regulation. I have specifically focused on optimal design methods that combine asymptotic parametric accuracy objectives, based on the Fisher information matrix, with relaxed formulations of the design optimization problem. I have applied these OED methods to three biological case studies. (1) I have used these methods to implement a multiple-shooting optimal control algorithm for optimal design of dynamic experiments. This algorithm was applied to a novel model of transcriptional regulation that accounts for the microbial host's physiological context. Optimal experiments were derived for estimating sequence-specific regulatory parameters and host-specific physiological parameters. (2) I have used OED methods to formulate an optimal sample scheduling algorithm for dynamic induction experiments. This algorithm was applied to a model of an optogenetic induction system – an important tool for dynamic gene expression studies. The value of sampling schedules within dynamic experiments was examined by comparing optimal and naive schedules. (3) I derived an optimal experimental procedure for fitting a steady-state model of single-cell observations from a bistable regulatory motif. This system included a stochastic model of gene expression and the OED methods made use of the linear noise approximation to derive a tractable design algorithm. In addition to these case-studies, I also introduce the NLOED software package. The package can perform optimal design and a number of other fitting and diagnostic procedures on both static and dynamic multi-input multi-output models. The package makes use automatic differentiation for efficient computation, offers a flexible modeling interface, and will make OED more accessible to the wider biological community. Overall, the main contributions of this thesis include: developing novel OED methods for a variety of gene regulatory scenarios, studying optimal experimental design properties for these scenarios, and implementing open-source numerical software for a variety of OED problems in systems biology.

Acknowledgements

I would like to thank my supervisor Brian Ingalls for his generous support and mentorship; I am very lucky to have been able to work with him over these past four years. I would also like to thank Matt Scott for introducing me to bacterial physiology, I very much appreciate the time he took to answer my many questions. I would also like express appreciation for my lab mates, especially Christian Barna, Julian Smith-Roberge and Kwan Lai for their good company and the many interesting discussions that we had together. Finally, I would like to especially thank the undergraduate students who I collaborated with, and from whom I learned so much, these folks include: Melissa Prickaerts, Nick Hon, Leah Fulton, Max Reed, Addison Richards, William Forrest, Michael Astwood, Cody Receno, Davey Lee, Keenjal Mistry, Zixuan Liu and Taylor Pearce.

Dedication

This thesis is dedicated to my parents, for all their love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination. He can perhaps say what the experiment died of."
-Ronald Fisher

Models and data are two sides of the same coin. Without data, models are unmoored imaginations of what might be; without models –even very simple non-mathematical ones– data is not much more than a listing of measurements. Interpreting data is a process of reading between the data points, sometimes literally interpolating and sometimes more figuratively, to glimpse an underlying process that has not yet been understood. In some sense, building models is only a more refined version of visually assessing a trend, but with more precision both in our assumptions and, hopefully, in the predictions and understanding that result. Unlike a scientist's intuition in eye-balling a scatter plot, mathematical models provide researchers with a precise way of writing down assumptions about relationships between different observables. This has made them an appealing tool in dissecting the complex set of interactions that make up biological systems. Especially as the focus in biology has shifted from individual genes and molecules towards interaction and the behaviour of many-component systems, mathematical language has become increasingly valuable for describing relationships and interpreting data.

However, as we increasingly rely on models to encode our assumptions, it is logical that this paradigm will also influence which data we collect. Different assumptions imply different questions which in turn suggest different experimental designs to interrogate nature. As the opening epigraph from Ronald Fisher suggests, data is only valuable if it comes from a well-designed experiment, as the data's ability to illuminate is determined by the design, not the result. Each experiment is a delicately posed question and our intuition is often a poor guide to the exact consequences of our experimental inquiries. Planning for data collection can be a tedious business that requires careful considerations of what uncertainties need to be hedged against and how our assumptions will color our interpretation of the results. In a field such as systems biology, where a system's behaviour is naturally described with mathematics, mathematical tools can also be used to guide our experimental questions. Optimal experimental design is a sub-discipline of statistics that focuses explicitly on this purpose, using mathematical rigour to guide experimentation for improved results. This is a reversal of the traditional flow of data in biology which, perhaps due to historical or cultural factors, has often moved from the experimentalist to the theorist with little feedback being returned. This lack of feedback is in sharp contrast to disciplines like physics and chemistry with their robust dialogue between theory and experiment. However, with each passing decade biology seems increasingly poised to become more like its quantitative cousins. This will hopefully see mathematical biologists getting their hands dirty putting experimental questions to Mother Nature, and experimental biologists using more precise quantitative thinking to interpret their results. With the popular hype about big data and machine learning, especially in biology, it is often easy to forget that the quality of the data we collect can often be much more important than how much we have; asking the right question can be more useful than asking many.

In this thesis I have focused on applying ideas from the field of optimal experimental design (OED) to models in microbial systems biology. Optimal experimental design has a long tradition within the field of statistics, with early work dating back to Ronald Fisher near the beginning of the last century [6, 7]. While much of the theory for experimental design was developed decades ago, its application and implementation still pose challenges. Each new field brings with it unique models, questions, and experimental hurdles. Systems biology, like physics and chemistry, often relies on (pseudo-)mechanistic models, and practitioners can plan direct experimentation rather than rely on observational studies. However, unlike the fundamental sciences and much more like economics or ecology, the parameters in systems biology models are often themselves crude abstractions of more fundamental processes. Also, these parameters can often only be determined via fitting. This unique blend of mechanistic model structure and designed experiments, with a reliance on statistical fitting, has created significant opportunities for OED; but systems biology also presents great challenges. Systems biology has a comparatively large variety of models; including a wide variety of space and time scales, dimensions, non-linearities and sampling distributions. The experimental systems under study run the gamut from molecular reactions to multi-organism interactions. Despite this complexity, recent works have suggested that experimental design can significantly improve model calibration in systems biology [8] but there have been few practical demonstrations (see [9, 10] for isolated examples). Therefore, developing specialized OED methods and more accessible tools for the field is part of an ongoing research program.

This thesis will specifically focus on experimental design for parameter estimation in models of microbial gene expression. Gene expression provides an ideal system of study for experimental design for a number of reasons. Gene expression is relatively well understood, with adequate understanding of the biochemical processes involved. This knowledge makes it suitable for a mechanistic modelling approach as the model structure can be proposed with reasonable confidence. However, as gene expression involves multiple scales, the mechanistic aspects of the model are often combined with pseudo-mechanistic components; phenomenological parameters often lump fine-grained interactions into single constants (i.e. like sigma factor binding, mRNA folding, and initiation stages of transcription and translation). This blend of mechanistic and phenomenological modelling is emblematic of many systems biology models, and this makes gene expression a suitable proving ground for OED techniques. In addition, experimental methods for measuring gene expression are mature, diverse, and widely available. This means many labs may be interested in and capable of implementing designs for studying expression, and that there is fair degree of flexibility in the types of experiments that may be used. This is in contrast to more experimentally constrained systems of study where there may not be as much interest or choice in experimental design.

Applying optimal experimental design to gene expression models may also yield some unique benefits. For example in gene expression models, parameters are often intrinsically interesting, not just for accurate predictions but also as aspects of study in themselves. For example, when estimated accurately, parameters describing transcription factor, polymerase, and ribosome binding strengths, and can provide information about low-level DNA/RNA sequence properties. It is also often possible to manipulate these parameters, through mutation or replacement by synthetic sequences so that the parameters serve to characterize different alterations made to a natural system. Tools for manipulating DNA are now pervasive and building gene expression constructs now plays a central role in many bio-engineering projects. While it has long been a goal to perform model-guided design of biological systems [11, 12], in the past, few models were predictive enough to be used in designing constructs other than as rough a intuitive guide. For example early synthetic gene expression circuits like the Elowitz's repressillator and Gardner's toggle switch were studied mathematically in a *post hoc* or parallel manner rather than being designed in a fully model-driven fashion [13, 14]. In recent years though, gene expression networks have become one of the the first synthetic systems to be wholly designed and implemented successfully *in vivo* using automated model-drive design [12]. This work by Nielsen et al. points the way to future research on genetic part characterization and model-guided design for a wider variety of systems. Many of these research programs will depend on having accurate predictive models of component behaviour and accurate parameter estimates. Here there is potentially great opportunity for optimal experimental design to ensure efficient and accurate part characterization and model calibration.

1.1 A Simple Example

The diversity of models and experimental questions that can be asked is very large, even just for models of gene expression. Exactly what experimental designs should be used in each context depends on the experimental goals and experimenter's assumptions. The assumed model structure encodes many of these assumptions and plays a very crucial role in optimal design for parameter estimation.

To give some intuition for the experimental design process and its relation to the proposed model, we begin here with a simple linear model:

$$E(y) = mx + b. \tag{1.1}$$

The random observation variable y has a conditional mean, $E(y)$, that depends on the experimental input x and the two unknown parameters m and b. Figure 1.1 illustrates a simplified fitting scenario, comparing two different experimental designs, shown with red or green dots respectively for each dataset. Each experimental design has only two unique input points x_1 and x_2 (shown as x_i's for the red dataset). The true and unknown underlying data-generating process is shown as the blue dashed line. The red dataset has x_1' and x_2' placed close together and the green dataset has x_1 and x_2 spaced far apart. The true underlying distributions for y, depicted at each input condition, are shown as blue distributions (shown horizontally as the observation noise occurs vertically in the observation variable y). It can be seen somewhat intuitively that in the green dataset, by placing the two experimental inputs far apart, any feasible fit like the example green line is restricted to be somewhat close to the true underlying trend. In the red dataset we can see that several fits, shown as the pair of red lines, may be feasible depending on the exact realization of the random observations. This implies that in the red datasets the model fitting process will be sensitive to outliers, whereas in the green dataset fitting will be more robust. It is also fairly obvious from this simple illustration that our knowledge of the slope and intercept parameters will be much more precise if we fit the model to the green dataset; the green lines lie much closer to the true blue line than the candidate red

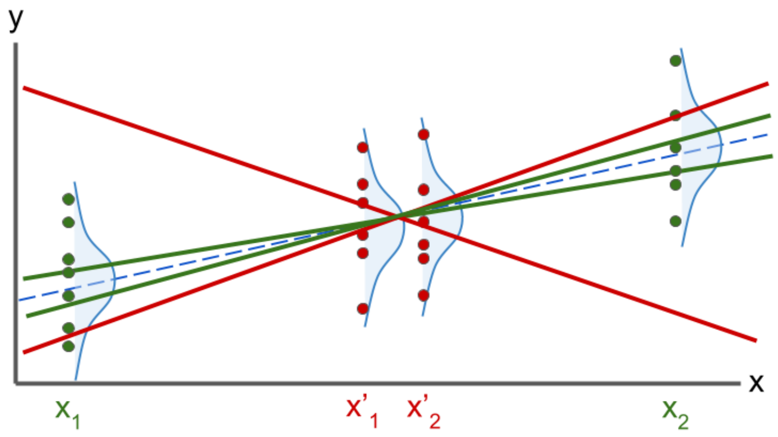


Figure 1.1: A depiction of a linear model ('true' trend: blue dashed line, 'true' error distributions: blue distributions) being fit with two different experimental designs (less optimal: red, more optimal: green). Feasible fits are shown as solid lines with corresponding colors.

fits. This in turn means we will get more accurate predictions when we use the green-fit model to extrapolate and interpolate as the green lines lie closer to the true trend across all x values. The red dataset will yield less accurate estimates of m and b and will also generally produce a model that makes bad predictions. The simplicity of this scenario makes some of the general ideas of optimal experimental design intuitively clear. It is easy to see that the design can improve model fit and robustness to observation noise even if the sample size is held constant. Also, parameter accuracy and prediction accuracy are intrinsically related.

For a linear model like the one illustrated above the optimal design is characterized by always splitting the total sample size over two x values spaced as far apart as possible [15]. However, optimal designs depend heavily on the assumptions encoded in the model they are created for. An optimal design can therefore take a very different structure under alternative assumptions, even for trends that look broadly similar. For example, for a nearly identical process but this time with a fixed zero intercept the optimal design is quite different as shown in Figure 1.2. Here the true trend is again shown as a blue dashed line. As the model has a zero intercept, we have forced the b parameter to be zero. This

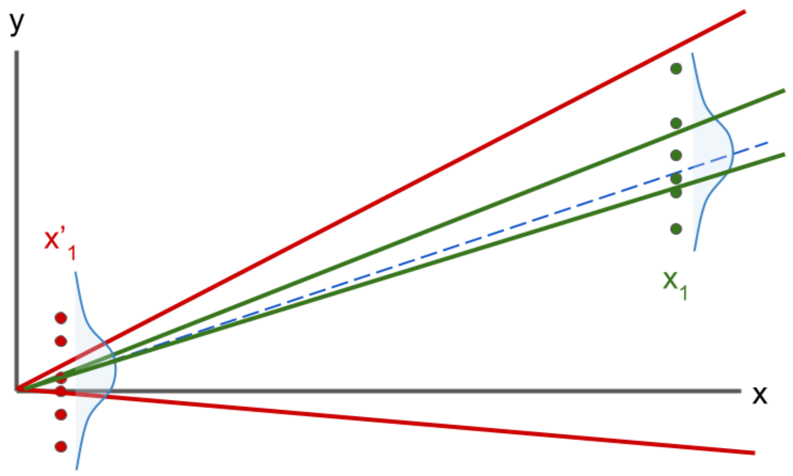


Figure 1.2: A depiction of a linear model with no intercept ('true' trend: blue dashed line, 'true' error distributions: blue distributions) being fit with two different experimental designs (less optimal: red, more optimal: green). Examples of feasible fits are shown as solid red and green lines.

is an assumption made by the experimenter and by assuming it is true, observations taken near $x = 0$ are no longer expected to be useful. In fact we only need to observe a single x value to calibrate the model. If we select the x level near zero, at x'_1 as in the red dataset, the feasible fits shown as the red lines can still vary widely depending on the realization of the data. If instead the data is collected at an x level as far from zero as is possible, such as x_1 in the green design, the estimate of the slope and resulting feasible fit will improve, as is shown by the green lines representing feasible fits for this case. Our assumption of a zero intercept has reduced the model flexibility and therefore changed what design will give optimal results when we perform the fitting. If we have good prior evidence that $b = 0$ but we do not assume this constraint, as in the first model, we will need to spread out our observations over two levels of x to achieve a good fit. However in the end this may be wasted effort, in light of the strong existing evidence for $b = 0$. Our assumptions determine what choices appear optimal for us in the moment by encoding the information that we have at hand. Conversely, poor experimental designs can result from making unjustified assumptions about the model structure. If for example the intercept parameter was actually nonzero but we assumed that $b = 0$, even the more optimal green design in

1.2 would not produce a good fit, and worse we may not ever know it. The optimal design we compute always depends on the assumptions being made but its true performance in turn depends on the accuracy of the assumptions. For low-dimensional, linear models with normal error distributions, good designs can be created through intuition and by thinking through the model geometry. However, when the model is nonlinear and the data comes from a complex distribution this is no longer the case and tools from optimal experimental design can provide some guidance.

1.2 Thesis Overview

This thesis is a synthesis of several related projects; most of the chapters have appeared as journal articles or conference proceedings. The general theme running through each chapter is the design of experiments for studying various aspects of microbial gene expression and regulation. Therefore, in Chapter 2, I give some history and background for the theory of optimal experimental design for parameter estimation and its historical development. Chapter 3 consists of a review article written by myself and Brian Ingalls detailing some of the new opportunities for optimal experimental design appearing in microbial system and synthetic biology. This review specifically details the emergence of new experimental techniques in the field and how these techniques can be paired with appropriate experimental design algorithms. Chapter 4 consists of a journal article, co-authored with Brian Ingalls and Matt Scott, describing a physiologically-aware model of gene expression, and an algorithm for generating optimal experimental designs useful for calibrating the model. In this work we handled the optimal design problem using a model predictive control algorithm implemented with multiple shooting. Chapter 5 consists of a shorter conference paper written with Max Reed and Brian Ingalls on the convex structure of optimal sample scheduling problems for dynamic stimuli experiments; this is then applied to a simple dynamic model of gene expression. Chapter 6 consists of another conference paper, written with Addison Richards and Brian Ingalls, studying optimal experimental design for a bi-stable genetic motif observed at single-cell resolution and at steady state. In Chapter 7, I summarize current work and implementation of the NLOED package – a Python software package for nonlinear model building and optimal experimental design. Work on the package has not yet been published but the software is being used in some current laboratory projects. Finally in Chapter 8 I conclude by summarizing some common themes and lesson shared across the various works.

Chapter 2

Background on Optimal Experimental Design

Optimal experimental design (OED) is a sub-field of statistics that uses mathematical optimization to guide the selection of observations in order to achieve a given statistical objective [15, 16]. In this chapter I give a brief description of the history and mathematical background of OED. However before starting on this material, it is useful to briefly clarify the relation of OED to some other closely related sub-fields, and to justify the terminology used in this work.

Optimal experimental design can be considered a subset of the broader statistical sub-field called design of experiments (DOE) [17]. DOE focuses on the general aspects of experimental design, such as creating standardized, easy-to-implement designs with well-balanced performance in fitting many common regression models [18]. DOE also includes other sub-fields that have less of an emphasis on modelling objectives, such as response surface methodology which deals with optimizing process performance [19]. OED specifically differentiates itself from other areas within DOE with its heavy use of optimization applied to mathematically formalized objectives – especially objectives related to improving the accuracy of parameter estimates and model fitting [15, 16]. Like its parent field, OED has also traditionally focused on regression models, however there has also been a long history of applying the mathematical ideas from OED to nonlinear models found in other scientific disciplines [16, 20]. I generally use the term OED to describe the methods implemented in this thesis, as I feel it is the most accurate term available. Work in this thesis has specifically focused on optimizing experiments using statistical objectives developed within the OED literature. However, within systems biology and bio-engineering, there has emerged another commonly used term, model-based design of experiments (MBDOE), which is often used in a similar manner to OED [21, 22]. In my readings I have come to understand MBDOE as having a broader set of experimental goals, like the parent field of DOE, but researchers working under the MBDOE heading have focused on more complex, mechanistic models found in systems biology and bio-engineering. Therefore, when discussing more broad statistical objectives and novel methods developed specifically for systems biology, I have also used MBDOE in certain chapters (Chapter 3 for example).

2.1 A Short History

Experiments have been done for hundreds, if not thousands of years. It seems reasonable to assume that experimenters have tweaked and optimized their experimental methods to achieve better results for almost as long. These improvements may have involved an improved apparatus, better measurement devices or techniques, or more practice with the prescribed protocol by the experimenters themselves. These practical aspects of experimentation are perhaps the most important for generating useful results. However, optimal experimental design in this thesis deals with a more narrow and abstract aspect of experimental design. Here, experimental design refers specifically to the mathematical characterization of numerical measurements and their relation to proposed mathematical models of the process. This perspective on experimentation is a comparatively recent innovation, and before describing the mechanics of optimal design, it is useful to situate optimal experimental design in a broader scientific context.

Modern statistical theory, on which OED is constructed, began to be formalized around the end of the 19th and into the early 20th century by the likes of Francis Galton and Karl Pearson [23]. These early practitioners of statistics built upon more theoretical work on probability by the likes of Laplace and Gauss in the 18th century, but unlike their predecessors their interests turned away from abstract games of chance and astronomical observations and towards problems in data interpretation found in daily life [23]. Before the modern field of statistics emerged, the idea of observational errors and random variation in data was largely handled in an *ad hoc* manner [23]. Methods for handling data were developed independently for specific fields, like astronomy, without a more general theory for how an experiment, data, and models may be connected [23]. However by the early 20th century, ideas like sampling distributions, model fitting, and parameter estimates, were being formalized for increasingly more general situations [23]. It is only by developing these concepts that early statisticians were first able to formalize exactly what constitutes an experiment, from a mathematical perspective, and what formalized objectives may be pursued in doing them. It is in this context that the first works on the design of experiments began to emerge.

The first modern work on optimal experimental design is generally thought to be Kirstine Smith's 1918 paper on optimal design for polynomial regression models [24]. (Stigler notes that there exists at least one pre-modern work, but it is of more historical rather than scientific interest [25].) Smith's work was in many ways ahead of its times, as many statistical ideas had not yet diffused into general scientific practice. Before optimal experiments could be used in real experiments, the statistical ideas on which they were based needed to mature and reach more widespread adoption. Ronald Fisher, perhaps the most important statistician in history, made ground-breaking contributions to statistical theory and his work helped catalyze the adoption of statistical methods and experimental design by a much wider scientific audience [26]. Fisher wrote, "The Arrangement of Field Experiments" in 1926 [6] and "The Design of Experiments" in 1939 [7], both of which are classic early works on experimental design that explained the statistical aspects of experimental methods to the general scientific community. Fisher also made significant contributions to maximum likelihood estimation and derived the Fisher information matrix, mathematical concepts that are central to OED and this thesis [27]. Fisher's experimental work was influenced by his position at the Rothamsted agricultural research station, where agricultural field trials had been conducted for nearly a half-century at the time of his appointment. This close contact with daily experimental work is evident in Fisher's useful blend of practical and theoretical analysis and much of the modern tradition of experimental design begins with this early work by Fisher[28].

After Fisher, the design of experiments blossomed into a thriving sub-discipline in its own right. Many standard designs in DOE were developed and analyzed in the decades from 1940-1960, including fractional factorial design [29], Placket-Burman designs [30], and orthogonal arrays [31]. George Box, perhaps the most important contributor to experimental design after Fisher, also became active in this period [32]. During this time Box developed response surface methodology (RSM) for designing experiments used to optimize processes in the chemical industry [33]. He also did early work on experimental design for non-linear dynamic models of chemical reactions [20]. This latter work is of special interest as it is the first example of optimal design being applied to a numerically integrated system of nonlinear differentiation equations, a topic which has become a central theme in optimal design for systems biology and in this thesis. Box's nonlinear work was perhaps, like Smith's founding paper, before its time, as the computational tools that would make it practically useful in the hands of everyday experimenters would not appear for several decades.

The standardized designs developed in the DOE literature in this era were easy to generate and gave good performance in fitting simple linear regression models. However they were not necessarily well adapted to more complicated experimental constraints or models. Designs in RSM too were focused on a very simple class of models, and were more focused on finding the optimum operating conditions for a process rather than producing improved parameter estimates or predictions. However, in parallel to work in the general DOE and RSM literature, researchers in the 1950's also began to study exactly what made experiments optimal for statistical goals like parameter estimation and prediction. This began as a more detailed mathematical study of classic regression models, but eventually involved more complex tools from convex analysis. This research program had early contributions from Wald [34], Chernoff [35], and Elfving [36, 37]. But some of the most notable contributions were from Jack Kiefer [38]. Kiefer's early works helped frame and address the main problems of the OED field [38, 39], and his seminal paper with Jacob Wolfowitz first described the general equivalence theorem [40]. The equivalence theorem given in Kiefer and Wolfowitz's paper is important for several reasons. Firstly it showed that designs that were optimal for estimating parameters were also optimal for certain measures of prediction accuracy. Secondly, the paper shows the equivalence between certain easily checked conditions on the model and the optimality of a design, providing useful criterion for checking optimality. Thirdly it provides mathematical tools for constructing numerical algorithms for optimizing designs. Kiefer and other contributors in this era laid the groundwork for generalizing experimental designs to non-standard models and experimental constraints. Their detailed study of design optimization freed experimenters from needing to use standard off-the-shelf designs. Now, instead, methods from OED could provide a tailored experiment for any scenario [41].

After its formative years in the 1950's and 1960's, OED methods have slowly diffused into neighbouring disciplines and application areas. This has been facilitated by the increasing availability of computing power and software needed for optimization [41]. In the field of statistics, optimal design theory and methods have been expanded to increasingly more complex models [42] and data distributions [43]. Ljung and colleagues, working in control theory, have used ideas from optimal design in systems identification, and have extended these methods to frequency domain analysis of linear time-invariant systems [44]. Following in Box's footsteps, chemical engineers have continued to explore applications of optimal experimental design applied to complicated nonlinear chemical and biochemical processes [21]. Optimal design has also found applications in pharmaceutical research and drug development, for improving the efficiency of data collection [45, 16].

2.2 Mathematical Necessities for Optimal Experimental Design

Optimal experimental design centres on choosing an ideal set of experimental conditions in which to take observations. However, in order to define what conditions are useful and to be able to pose this as an optimization problem, some mathematical background needs to be developed. In this thesis I have primarily focuses on optimizing experiments for fitting model parameters. In this subsection I mathematically formalize the general definition of a model, with special attention paid to the connection between the model structure and the sampling distribution of observations. I also formalize the notion of an experimental design for a given model, which determines the structure of a dataset that will be used for fitting. After this I give a brief overview of maximum likelihood estimation (MLE) and its asymptotic properties, which provides a flexible framework for fitting a large variety of models. The asymptotic properties of the MLE parameter estimates allows us to quantify the expected variability of the parameter estimates around the true parameter value. These asymptotics also lead to the Fisher information matrix (FIM), which can be used to compute the expected utility of experiments that have not yet been performed. I conclude by discussing objective functions computed from the FIM and by giving a brief introduction to OED-type optimization problems. It should be noted that the resulting optimal experimental design depends on all of the mathematical components discussed in this chapter. Designs are optimized for a specific model, combined with a specific fitting method, and for specific asymptotic objectives. This means that careful attention should be paid to all model components, estimation methods and assumptions, as changing any aspect of the overall modelling process can alter the structure of the resulting optimal design.

2.2.1 Components of a Data-generating Model

Models are abstraction of a true underlying data-generating process in a given experimental scenario. For the purposes of this thesis a model can be defined as a mathematical structure that connects the observation variables, Y_i, – which are measured during an experiment – with the experimental conditions that are known or can be controlled, $\boldsymbol{x}_j$. Here the observation variables are indexed by i where each i, from 1 to M, corresponds to a unique observation variable (i.e. different measurement types taken of the system). The experimental conditions, $\boldsymbol{x}_j$, are vector-valued, numerical encodings of any of the relevant conditions known or controlled by the experimenter when the observations are taken (i.e.

ambient conditions, treatments, physical or chemical perturbations etc.). Each dimension of the input vector may be subject to various restrictions, such as some inputs being restricted to a real interval or being drawn from a discrete set. (These constraints can be ignored for the purposes of this exposition; see Chapter 7 for further discussion of how these constraints are handled in practice.) As the system can be observed in multiple sets of conditions in a single experiment, the set of condition vectors is index by j, from 1 to N, with a total of N possible experimental condition vectors. As all experimental observations are effected by random errors or uncontrolled variation, the observation variables, Y_i, are random variables. The observation distribution, $p_i(.)$, of each observation variable is conditional on the experimental vector, $\boldsymbol{x}_j$, and so to be formally correct, each observation type in each unique condition is a unique random variable, $Y_{i,j}$. (However, it is convenient when referring to all observations of the same type, regardless of the experimental conditions, to suppress the j subscript and use Y_i.) With the above definitions, a full model can be formalized as

$$Y_{i,j} \sim p_i(y_{i,j}|\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})). \tag{2.1}$$

In this work $p_i(.)$ is generally approximated as a parametric distribution, for example the normal, log-normal or Poisson distributions. Observation variables, Y_i, are assumed to come from one parametric distribution type, such that each $p_i(.)$ always has the same type regardless of the inputs, $\boldsymbol{x}_j$, or parameter values, $\boldsymbol{\theta}$. The support of Y_i depends on the distribution type assumed such that Y_i can be restricted to the real numbers, positive real numbers or positive integers as the experimental situation dictates. Here $\boldsymbol{\eta}_{i,j}$ are the distribution $p(.)_i$'s natural parameters; in this work they are referred to as the *sampling statistics* of the distribution. For example, the normal distribution would have the mean and variance as its sampling statistics; the Poisson distribution would have the λ rate parameter as its sampling statistic. A repeated observation of the same observation variable and experimental conditions is termed a *replicate*. There can be a unique number of replicates of each observation and condition for a total of $\beta_{i,j}$ in each combination. Each replicate results in a realization $y_{i,j}^{(k)}$, of an identically distributed random variable $Y_{i,j}$; here the k subscript indexes the replicates from 1 to $\beta_{i,j}$. The deterministic model components $\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})$ describes the relationship between the experimental conditions, $\boldsymbol{x}_j$, and the sampling distribution statistics, $\boldsymbol{\eta}_{i,j}$, and therefore provides the link between the experimental conditions and the random but conditional variability of the observations. It should be stressed that the function $\boldsymbol{f}_i(.)$ can take the form of any sufficiently-smooth mathematical or numerical expression; it can therefore be a constant, an algebraic expression, a differential equation solution, the root of an implicit function or the solution to an optimization problem. Finally the functional mapping between the experimental conditions $\boldsymbol{x}_j$ and the sampling statistics $\boldsymbol{\eta}_{i,j}$, is parameterized by the unknown parameter vector $\boldsymbol{\theta}$. In this work it is generally the case that $\boldsymbol{\theta}$ needs to be estimated from the data resulting from any designed experiments. Note that in this formulation there is some ambiguity in whether observations of the same type taken at different points in time or space are handled as model input dimensions in $\boldsymbol{x}_j$ – by entering the temporal or spatial location directly as an input dimension value – or as a separate observation variables Y_i – with their own distributions $p_i(.)$ and functions $\boldsymbol{f}_i(.)$. However for the theoretical exposition of parameter estimation and optimal design this distinction has no direct impact, see Chapter 7 for practical considerations.

In this work we assume that the experimenter is confident in postulating the sampling distributions, $p_i(.)$, and model structure, $\boldsymbol{f}_i(.)$, for each observation variable. This necessarily implies the experimenter has some knowledge about the system as making choices for these model components imposes significant constraints on the model's flexibility and the resulting optimal design. Information for model specification can come from past data or from mechanistic considerations given knowledge of how the system functions. In this work, it is assumed that the main source of uncertainty for the experimenter is the parameter vector, $\boldsymbol{\theta}$, which needs to be fit to the collected observations. Uncertainty in model structure can be included in a more general optimal design framework [46, 47, 48], however model selection is not a central theme in this work.

2.2.2 Definition of an Experimental Design

An experimental design specifies the structure of the data that is collected. A design must include two aspects to fully specify an experiment, 1) which experimental conditions are used, $\boldsymbol{x}_j$, 2) how many replicates, $\beta_{i,j}$, are taken of each observation variable at the given condition vector. When discussing the set of input conditions used, it is common to refer to the design's *support*, which consists of the unique set of input points used in the experiment [15]. Therefore each $\boldsymbol{x}_j$ from 1 to N must be unique, and the full set of N vectors constitutes the design's support. At each unique support point, a design must also specify a set of replication counts, $\beta_{i,j}$, taken for each observation variable, Y_i, for i from 1 to M. The replicate count, $\beta_{i,j}$, is integer valued and specifies the number of replicates to be taken from the specific random variable $Y_{i,j}$. A design $\mathcal{D}$ is then fully specified by the list of support points $\boldsymbol{x}_j \in \mathcal{X}$ and the list of replication counts $\beta_{i,j} \in \mathcal{B}$. Note here that $\beta_{i,j}$ can have a value of zero indicating that no observations of the ith observation variable are taken for the jth experimental condition vector. The total sample size for the experiment, N_{Tot}, is computed as $N_{Tot} = \sum_i^M \sum_j^N \beta_{i,j}$. A dataset, $\boldsymbol{y}_{\mathcal{D}}$, corresponding to design, $\mathcal{D}$, consists of a listing of the N_{Tot} observations, $y_{i,j}^{(k)}$, for all replicates, k from 1 to

$\beta_{i,j}$, observation variables, i from 1 to M, and experimental conditions, j from 1 to N. A design determines the structure of the dataset that results from it.

2.2.3 Maximum Likelihood Estimation

Maximum likelihood is a well-tested and generally applicable method for fitting probabilistic models [49, 16]. The key idea in maximum likelihood estimation is to seek the parameter values that maximize the likelihood of observing the given data [49]. In order to specify a maximum likelihood estimator for a given model, the model must include a sampling distribution, $p_i(.)$ above. In general, it is not always possible to specify an analytic expression for the observation distribution or to know which parametric distribution may best approximate the observations. However it should be noted that even simpler fitting methods like least-square fitting, can be seen as a special case of MLE when the user is (implicitly) assuming normally distributed errors [50].

In order specify the maximum likelihood estimator, we need to derive an expression for the total data likelihood [49]. The likelihood is directly related to the probability distribution of the data conditioned on the parameter values [49]. Using a single dimensional example, the conditional probability distribution of a random observation variable Y is $p(y|\theta_o)$; this is the distribution for an individual observation y conditioned on a fixed true value of θ_o. The likelihood is functionally identical to the conditional distribution of the observation except with a reversal of the nominal input; for a fixed observation value, y, the likelihood $L(\theta; y) = p(y|\theta)$ is a function of any candidate value of θ. Assuming independence of observations, the total likelihood, $L_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D})$, of all replicates, $y_{i,j}^{(k)}$, in a given experimental dataset with design, $\mathcal{D}$, and observations, $\boldsymbol{y}_{\mathcal{D}}$, can be expressed as

$$L_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D}) = \prod_{j}^{N} \prod_{i}^{M} \prod_{k}^{\beta_{i,j}} p_i(y_{i,j}^{(k)} | \boldsymbol{\eta}_{i,j} = \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})). \tag{2.2}$$

(Note, dependent observations are discussed below.) The design influences the total likelihood by varying both the inputs, $\boldsymbol{x}_j$, and the replicates, $\beta_{i,j}$. The parameter vector, $\boldsymbol{\theta}$, that optimizes the likelihood expression is the maximum likelihood estimate, denoted as $\hat{\boldsymbol{\theta}}$.

While the likelihood is straightforward to define, it is almost always advisable to maximize the log-likelihood, $l_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D}) = \log(L_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D}))$. The $log(.)$ function is monotonic increasing and therefore the maximizing parameter vector for the log-likelihood is identical to that of the likelihood. However the log-likelihood can be numerically simpler to compute and more stable. Also most of the asymptotic results for the MLE are developed using the log-likelihood because, for independent observations, the logarithm converts products involved in a joint observation distribution into sums. Expressing the estimator in terms of a summation of random terms leads more naturally to the application of various central limit theorems and the subsequent asymptotic derivations [16]. For brevity in writing out the complete data log-likelihood we express the log-probability of each data point as

$$l_i(\boldsymbol{\theta}; y_{i,j}^{(k)}, \boldsymbol{x}_j) = \log(p_i(y_{i,j}^{(k)}|\boldsymbol{\eta}_{i,j} = \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}))). \tag{2.3}$$

Then the total log-likelihood can be expressed as a summation such that

$$l_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D}) = \sum_{j}^{N} \sum_{i}^{M} \sum_{k}^{\beta_{i,j}} l_i(\boldsymbol{\theta}; y_{i,j}^{(k)}, \boldsymbol{x}_j). \tag{2.4}$$

For brevity, the dependence of the total log-likelihood, $l_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D})$, on the data, $\boldsymbol{y}_{\mathcal{D}}$, and the design, $\mathcal{D}$, is often suppressed because for a given dataset these values are constants. In these cases the total log-likelihood is abbreviated as $l_{Tot}(\boldsymbol{\theta})$. The additive nature of each data point within the total log-likelihood has important consequence for the structure of optimal experimental design problems. However, not all observation variables are independent. If we admit dependence between various observations, it will be reflected in the joint distribution of various groups of $Y_{i,j}$. Dependence between observations means that the total log-likelihood ceases to be additive because the joint probability of two observations does not factor; $p(Y_{a,b}, Y_{c,d}) \neq p(Y_{a,b})p(Y_{c,d})$. In this case the user must specify a joint distribution for any correlated observations that are taken together, and also specify a marginal distribution for each observation taken alone. In addition analytic multivariate distributions are only available for a subset of the possible observational supports that may be required. For example, expressing correlation between two normal distributions is straightforward but expressing dependence between a normal and Poisson distribution, or between pairs of Poisson distributions is not analytically tractable. If we wish to have an analytically tractable log-likelihood (with consequences later for the optimal design computation), there is a choice in either accommodating multiple data types easily, but with presumed independence, or in handling dependent observations with a much more limited set of observation distributions. In this thesis I have generally focused on models where observations can be approximated as independent.

2.2.4 Quantifying the Accuracy of Parameter Estimates

Observations in a given realization of an experimental design are the result of a random process. This implies that the maximum likelihood estimate, $\hat{\boldsymbol{\theta}}$, is also a random variable. The probability of $\hat{\boldsymbol{\theta}}$ exactly matching the true parameter value, $\boldsymbol{\theta}_o$, is unlikely given randomness inherent in the observations. However despite this variability, we want the estimate to be near to the truth on average, and a key goal in evaluating fitting procedures and experimental designs is in quantifying how close to the truth the estimate is likely to be. At a bare minimum, for a large variety of models, it can be proven that the MLE will recover the true parameter value in the limit of large sample sizes. For example this has been proven for nonlinear regression models [51], generalized linear models [52], and models with non-identically distributed data [53]. (These are all model types that are accommodated by the general formalism outlined previously, however the reader is warned that it is possible to specify models for which the MLE has not been proven to be consistent within the above framework.) For the asymptotic results to apply it is important that the model is *identifiable* [54]. Identifiability requires that for any pair of candidate parameter values, $\boldsymbol{\theta}_a$ and $\boldsymbol{\theta}_b$, the model yields distinguishable prediction for at least one possible input vector $\boldsymbol{x}_j$ and observation variable Y_i; there must exist conditions for which $\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}_a) \neq \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}_b)$ holds. This implies that at least one possible observation variable will have a distinguishable difference in distribution for alternative parameter values under some conditions achievable in an experiment. Many models will conform to this assumption if they are not over-parameterized and a sufficient variety of system measurements can be included in an experimental design. Even in some cases that do not satisfy the identifiability condition, re-parameterization can yield an identifiable model. Given this identifiability assumption and some other mild regularity conditions, the maximum likelihood estimate is both asymptotically non-biased and consistent [55]. For models meeting the required conditions, these properties imply in the limit of large sample sizes the expected difference between the estimate and true parameters goes to zero, and the MLE estimate converges in probability to the true value (i.e. the probability of not generating an exactly true estimate goes to zero in the limit).

The asymptotic non-biasedness and consistency of the estimator suggest that it is at least plausible that the MLE estimate yields reasonable results for identifiable models. However, when dealing with finite sample sizes in real experiments, estimates for non-linear models are almost always biased and can exhibit significant variability in their distribution around the true value [56]. With finite sample sizes, the design of the experiments can have a significant impact on the distribution of the estimate, $\hat{\boldsymbol{\theta}}$ [16]. The experimental design influences $\hat{\boldsymbol{\theta}}$'s distribution by mediating the sensitivity of the estimate to the variability inherent in the observations. Therefore it is desirable to have a more precise measure of the effect of a design $\mathcal{D}$ on the distribution of the estimate about the true value. An ideal measure of accuracy is the expected mean squared error of the estimate,

$$\mathrm{MSE}(\mathcal{D}, \boldsymbol{\theta_o}) = \mathrm{diag}\{E_{\boldsymbol{y}_{\mathcal{D}}}[(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}_o)^T \cdot (\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}_o)|\mathcal{D}, \boldsymbol{\theta}_o]\}, \tag{2.5}$$

(with respect to the data, $\boldsymbol{y}_{\mathcal{D}}$, as distributed under the true parameter vector, $\boldsymbol{\theta}_o$) which is effectively the second central moment of each parameter estimate about its true value [49]. Here diag$\{.\}$ returns the diagonal elements of a square matrix and $\hat{\boldsymbol{\theta}}$ and $\boldsymbol{\theta_o}$ are presumed to be row vectors. For computing the MSE, it can be useful to decompose it as follows:

$$\mathrm{MSE} = \mathrm{diag}\{E_{\boldsymbol{y}_{\mathcal{D}}}[\hat{\boldsymbol{\theta}}^T \cdot \hat{\boldsymbol{\theta}}|\mathcal{D}, \boldsymbol{\theta}_o]\} + E_{\boldsymbol{y}_{\mathcal{D}}}[(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}_o)|\mathcal{D}, \boldsymbol{\theta}_o]^2\}. \tag{2.6}$$

The $E_{\boldsymbol{y}_{\mathcal{D}}}[\hat{\boldsymbol{\theta}}^T \cdot \hat{\boldsymbol{\theta}}|\mathcal{D}, \boldsymbol{\theta}_o]$ term is the covariance matrix of the MLE estimate, more conveniently denoted as $\mathrm{Cov}(\hat{\boldsymbol{\theta}}|\mathcal{D}, \boldsymbol{\theta}_o)$. The estimate's covariance tells us about the raw variability of the estimator, but does not contain any information on its nearness to the true value, $\boldsymbol{\theta}_o$. The $E_{\boldsymbol{y}_{\mathcal{D}}}[(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}_o)|\mathcal{D}, \boldsymbol{\theta}_o]$ term is the estimate's bias, denoted as $\mathrm{Bias}(\hat{\boldsymbol{\theta}}|\mathcal{D}, \boldsymbol{\theta}_o)$ and measures the expected difference between the estimate and the true parameter vector [49]. (Note, the square in the bias term is applied element-wise.) Unfortunately it is impossible to compute the MSE, covariance, or bias exactly because the distribution of the estimator, $p(\hat{\boldsymbol{\theta}}|\mathcal{D}, \boldsymbol{\theta}_o)$ is generally not known or analytically tractable, as it results from a nonlinear transformation of the observation randomness propagated through the fitting process.

However it is possible to compute an asymptotic approximation to the MSE, and the bias and covariance terms. To do so, an expansion is taken in the square root of the inverse of the sample size, N_{Tot}; this analysis is complex and tedious and further details can be found elsewhere [57]. In performing the expansion it can be shown that the bias has no effect until after the first order, whereas the covariance expansion contains a non-zero first order term [56, 57]. This means that the variability of the estimator is generally dominant in the overall error compared to the estimator's bias, at least for reasonably large sample sizes. Therefore in this work, like most optimal design works, we focus on approximations for the estimator's covariance.

Approximating the covariance matrix of the MLE parameter estimate can be done using the Fisher information matrix. This result is often given as part of the proof of the asymptotic normality of the MLE [50, 16]. The Fisher information matrix for a single observation, Y_i, at input, $\boldsymbol{x}_j$, is defined as

$$I_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}}) = E_{Y_{i,j}}[\nabla_{\boldsymbol{\theta}} l_i(\bar{\boldsymbol{\theta}}; y_{i,j}, \boldsymbol{x}_j) \cdot \nabla_{\boldsymbol{\theta}} l_i(\bar{\boldsymbol{\theta}}; y_{i,j}, \boldsymbol{x}_j)^T]. \tag{2.7}$$

Here $\nabla_{\boldsymbol{\theta}}$ is the parametric gradient operator which is treated as a column vector. Note, that the Fisher information would ideally be evaluated at the true parameter value $\boldsymbol{\theta}_o$, however this is unknown. Instead, the FIM is generally computed at the estimate, $\hat{\boldsymbol{\theta}}$ – if initial data is available – or a nominal value, $\bar{\boldsymbol{\theta}}$, based on literature or other considerations. With independent observations, the total Fisher information matrix for an entire experiment can be computed as a weighted sum of the individual FIM's at each replicate:

$$I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}}) = \sum_{j}^{N} \sum_{i}^{M} \beta_{i,j} I_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}}). \tag{2.8}$$

Here the weights, $\beta_{i,j}$, are the number of replicates taken in the experimental conditions corresponding to the individual Fisher information matrix, $I_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}})$. Via the asymptotic expansion, it can be shown that the covariance of the parameter estimate with the given experimental design is approximated by the matrix inverse of the total FIM [16]:

$$\mathrm{Cov}(\hat{\boldsymbol{\theta}} | \mathcal{D}, \bar{\boldsymbol{\theta}}) = I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}})^{-1}. \tag{2.9}$$

To be clear this relation approximates the estimator's covariance assuming the nominal parameter vector, $\bar{\boldsymbol{\theta}}$, is near to the true value of $\boldsymbol{\theta}_o$ (i.e. this is a local approximation, see additional comments below).

The full proof of the FIM's connection to the parametric covariance is tedious and is not reproduced here, however it is possible to gain some intuition for how the FIM is linked to the covariance by observing some of its easily derived properties. By definition the maximum likelihood estimate occurs at the point, $\hat{\boldsymbol{\theta}}$, that maximizes $l_{Tot}(\boldsymbol{\theta})$. It follows that, at the optima, $\nabla_{\boldsymbol{\theta}} l_{Tot}(\hat{\boldsymbol{\theta}}) = 0$. Here the vector valued function $\nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta})$ is called the *score*, and the MLE must occur at a point where all entries in the score are zero (due to first-order optimality conditions) [49]. From the definition of the FIM, we see that it is the matrix of second moments and cross-moments for the score evaluated at our guess of the true parameter vector. It can be further shown asymptotically (at first-order) that the expected value of the score evaluated at the true parameter vector, $E_{\boldsymbol{y}_{\mathcal{D}}}[\nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta}_o)]$, is zero [49], which is consistent with the fact that the MLE is asymptotically non-biased. This expectation implies that the diagonal entries of the FIM are not just the second moments of the score, but the second central moments about its expected value, because

$$\begin{aligned} \mathrm{Var}(\nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta}_o)) &= \mathrm{diag}\{E_{\boldsymbol{y}_{\mathcal{D}}}[\nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta}_o) \cdot \nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta}_o)^T]\} - E_{\boldsymbol{y}_{\mathcal{D}}}[\nabla_{\boldsymbol{\theta}} l_{Tot}(\boldsymbol{\theta}_o)]^2, \\ &= \mathrm{diag}\{I_{Tot}(\mathcal{D}, \boldsymbol{\theta}_o)\} - \mathbf{0}. \end{aligned} \tag{2.10}$$

As the root of the score function fully defines the MLE, we should expect its variability to be strongly connected to the variability of the MLE estimate; in this case the variances

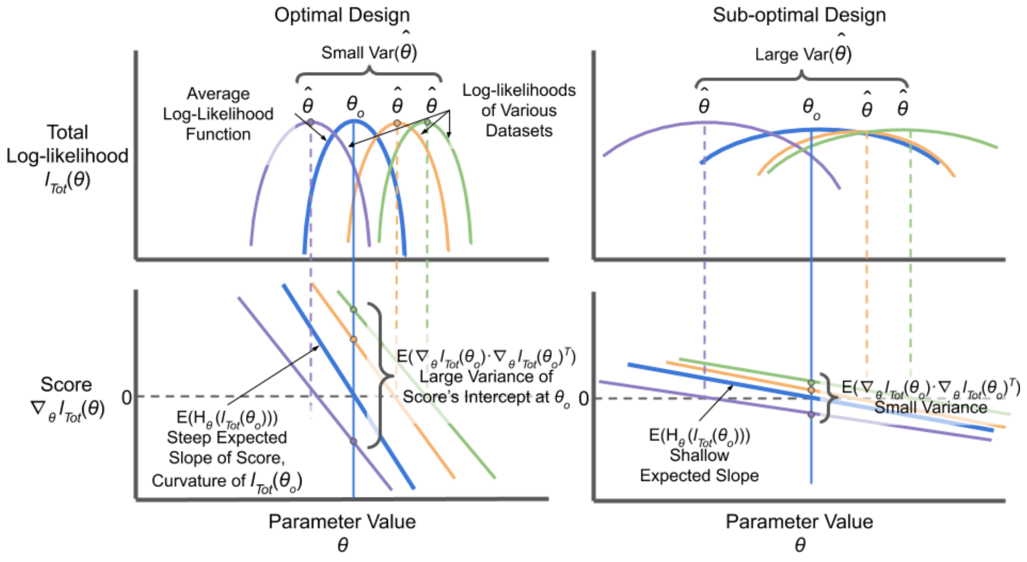

Optimal Design
Small Var($\hat{\theta}$)
Sub-optimal Design
Large Var($\hat{\theta}$)
Total Log-likelihood $l_{Tot}(\theta)$
Average Log-Likelihood Function
Log-likelihoods of Various Datasets
$\hat{\theta}$ θ_o $\hat{\theta}$ $\hat{\theta}$
Score $\nabla_\theta l_{Tot}(\theta)$
0
E($\nabla_\theta l_{Tot}(\theta_o) \cdot \nabla_\theta l_{Tot}(\theta_o)^T$) Large Variance of Score's Intercept at θ_o
E($H_\theta(l_{Tot}(\theta_o))$) Steep Expected Slope of Score, Curvature of $l_{Tot}(\theta_o)$
E($\nabla_\theta l_{Tot}(\theta_o) \cdot \nabla_\theta l_{Tot}(\theta_o)^T$) Small Variance
E($H_\theta(l_{Tot}(\theta_o))$) Shallow Expected Slope
Parameter Value θ

Figure 2.1: A figure comparing the total log-likelihood and score behaviour for multiple realizations of both an optimal design (left) and a sub-optimal design (right). The blue curves correspond to the average log-likelihood (top) and score function (bottom) for the given design – averaged over all possible datasets. The other colors represent the log-likelihood and score functions for specific realizations of the design each with random observations.

are inversely related. If the variability of the score is large over multiple realizations of the experimental design, it implies that the estimate is heavily constrained by the design to fall near its expected true value. This scenario is depicted in Figure 2.1 where, for the optimal design on the left, the intercepts of the individual score functions with the (blue) vertical line through the true parameter value, $\boldsymbol{\theta}_o$, exhibit large variability when the score functions' roots, $\hat{\boldsymbol{\theta}}$s, cluster around the true value. For the sub-optimal dataset on the right, the score function's value taken at the true parameter exhibits less variability and the estimates are therefore more variable.

Intuition for the FIM can be developed further by expressing the FIM in an alternative form in terms of the log-likelihood's second derivatives. For an individual independent observation it can be shown that [50]

$$I_i(\boldsymbol{x}_j, \boldsymbol{\theta}_o) = E_{Y_{i,j}}[\nabla_{\boldsymbol{\theta}} l_i(\boldsymbol{\theta}_o; y_{i,j}, \boldsymbol{x}_j) \cdot \nabla_{\boldsymbol{\theta}} l_i(\boldsymbol{\theta}_o; y_{i,j}, \boldsymbol{x}_j)^T] = -E_{Y_{i,j}}[H_{\boldsymbol{\theta}}(l_i(\boldsymbol{\theta}_o; y_{i,j}, \boldsymbol{x}_j))]. \quad (2.11)$$

Here $H_{\boldsymbol{\theta}}(.)$ is the Hessian operator with respect the parameter vector, such that the latter term above is the matrix of expected values of the second parametric derivatives of the log-likelihood. These two definitions of the FIM are equivalent under mild regularity conditions; to understand why, it is helpful to consider the single dimensional case. For a single observation y and a single parameter θ; the sampling probability is $p(y|\theta)$ and the log-likelihood is $l(\theta; y) = \log(p(y|\theta))$. Under mild regularity conditions, it follows from the identity $\int p(y|\theta)dy = 1$ that $\int \frac{d}{d\theta}p(y|\theta)dy = 0$ and $\int \frac{d^2}{d\theta^2}p(y|\theta)dy = 0$ also hold. It then follows that [58]

$$
\begin{aligned}
E\left[\frac{d^2}{d\theta^2}l(\theta;y)\right] &= E\left[\frac{\frac{d^2}{d\theta^2}p(y|\theta)}{p(y|\theta)}\right] - E\left[\left(\frac{\frac{d}{d\theta}p(y|\theta)}{p(y|\theta)}\right)^2\right], \\
&= \int \frac{d^2}{d\theta^2}p(y|\theta)dy - \int \frac{\left(\frac{d}{d\theta}p(y|\theta)\right)^2}{p(y|\theta)}dy, \\
&= 0 - \int \left(\frac{d}{d\theta}l(\theta;y)\right)^2 p(y|\theta)dy, \\
&= -E\left[\left(\frac{d}{d\theta}l(\theta;y)\right)^2\right].
\end{aligned}
\tag{2.12}
$$

This result helps us to link the variance of the parameter estimate with the FIM – defined as the variance of the score. To see this, note that while $E\left[\frac{d^2}{d\theta^2}l(\theta;y)\right]$ is a measure of the curvature of the log-likelihood, it is also the expected slope of the score function. This is shown in Figure 2.1 where on the left, for the optimal design, the log-likelihood exhibits significant curvature around the optima; therefore the slope of the score, shown below, is very steep. On the right, the sub-optimal design has less curvature around the optima and therefore exhibits shallow slopes in the score below. It follows from the previous paragraph that, for the optimal design, given the diagonal entries of the FIM are large, the variance of the score's intercept with the vertical line at the true parameter value will be large. This also implies that the expected slope of those same score functions will be steep. With steep average slopes, the roots of the score functions will have a smaller variability. As the score's root is the MLE estimate, this implies the MLE has a lower variability in the optimal case. The situation is reversed for the sub-optimal design, where the low variability in the score's value at the true parameter and the score's shallow average slope allow the estimates to vary widely under multiple realizations of the design. These results also hold for multiple parameters and provide some intuition for connecting the FIM to the MLE's variability.

As previously noted, we do not know the true value, $\boldsymbol{\theta}_o$, at which to take the asymptotic approximation and at which to evaluate the FIM. This means that any asymptotic results are inherently local to our nominal parameter vector, $\bar{\boldsymbol{\theta}}$. In practice initial estimates for $\bar{\boldsymbol{\theta}}$ can come from the available literature or from the MLE fit to some initial data. As the underlying model (i.e. $\boldsymbol{f}_i(.)$) must be sufficiently smooth, we can also expect asymptotic measures to hold approximately for parameter values near to the nominal vector. Other strategies have been proposed for dealing with the local nature of the FIM computed at the nominal parameter value $\bar{\boldsymbol{\theta}}$. Perhaps the most common approach is to implement optimal design in an iterative fashion, with each batch of optimized experiments adding to the collected data, improving the parameter estimates for the next round [15]. In this case even though the designs are optimized for a local objective, the parameter estimates tend to become more accurate after each iteration. This accuracy is ensured by the increasing sample size, so even if the initial designs were erroneous and sub-optimal, as the estimates improve, so will the optimality of the iterative designs over the course of the experiment. However, some experimental contexts are not suitable for iterative design. In these case other strategies exist including Bayesian averaging over a parametric prior distribution or min-max optimization – where the design is optimized for a parameter vector that has the worst estimation performance [15, 59]. In some cases, the local analysis alone can be useful if it is used to qualitatively explore optimal design structure under various parametric scenarios within the plausible parameter space; this has been done in some parts of this thesis.

Computing the derivatives and expectation required for the FIM for an arbitrary model may seem daunting. However, this derivation can be done quite simply by observing the following application of the chain rule [16]:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} l_i(\boldsymbol{\theta}; y_{i,j}, \boldsymbol{x}_j) =& \nabla_{\boldsymbol{\theta}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}))), \\ & \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}) \cdot \nabla_{\boldsymbol{\eta}_{i,j}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j})). \end{aligned} \tag{2.13}$$

The above decomposition makes use of the fact that the sampling statistics, $\boldsymbol{\eta}_{i,j}$, of the observation distribution, $p_i(.)$, are directly computed by the deterministic model component such that $\boldsymbol{\eta}_{i,j} = \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})$. The score function can therefore be decomposed into the deterministic model's parametric sensitivity and derivative of the log-probability of the observation with respect to the sampling statistics $\boldsymbol{\eta}_{i,j}$ [16]. As the model sensitivity does not depend on the random observation values, $Y_{i,j}$, this simplifies the computation of the Fisher information matrix at each observation so that we have [16]

$$\begin{aligned} I_i(\boldsymbol{x}_j, \boldsymbol{\theta}) &= E_{Y_{i,j}}[\nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}) \nabla_{\boldsymbol{\eta}_{i,j}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j})) \cdot \nabla_{\boldsymbol{\eta}_{i,j}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j}))^T \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})^T], \\ &= \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}) \; \Psi(\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})) \; \nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})^T. \end{aligned} \tag{2.14}$$

Here $\Psi(\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}))$ is defined as

$$\Psi(\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})) = E_{Y_{i,j}}[\nabla_{\boldsymbol{\eta}_{i,j}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j})) \cdot \nabla_{\boldsymbol{\eta}_{i,j}} \log(p_i(y_{i,j}|\boldsymbol{\eta}_{i,j}))^T]. \tag{2.15}$$

This expectation is effectively just the Fisher information of a standard parametric distribution, such as the normal, Poisson etc., with respect to its sampling statistics. We follow Fedorov in referring to these matrices as elemental FIMs of the given distribution [16]. These elemental FIMs have been computed for the vast majority of common one and two parameter distributions for which the expectation is analytically tractable. A good reference can be found in Fedorov's recent book [16]. Therefore, computing the Fisher information for the overall experimental design, decomposes into computing a parametric sensitivity vector for each observation point, $\nabla_{\boldsymbol{\theta}} \boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta})$, and matrix multiplying it with the observation distribution's elemental FIM; $\Psi(\boldsymbol{\eta}_{i,j}{=}\boldsymbol{f}_i(\boldsymbol{x}_j, \boldsymbol{\theta}))$. The formula for the elemental FIM for common parametric distributions is easily looked-up and computed from the predicted values of $\boldsymbol{\eta}_{i,j}$. The individual FIM's for each observation are then summed to produce the total FIM for the experiment. The total Fisher information matrix can be used to quantify the expected parameter estimate covariance for the given design at the nominal parameter values, $\bar{\boldsymbol{\theta}}$.

2.2.5 Scalar Objectives for Optimizing Designs

The Fisher information matrix captures approximate information about the expected distribution of the MLE estimate given a specified design. However, other than in the single parameter case, the Fisher information is matrix-valued which makes it unsuitable for use as an optimization objective. In order to construct an optimization problem for selecting an optimal design, we need a scalar function, $\Psi(.)$, of the FIM that can be maximized:

$$\max_{\mathcal{D}} \Psi(I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}})). \tag{2.16}$$

The most common scalar objective in practice is the determinant of the Fisher information matrix [15, 59]. This is equivalent to minimizing the determinant of the expected covariance matrix of the parameter estimates. Optimizing a design subject to this objective is known as finding a D-optimal design [15]. The determinant of a covariance matrix is referred to as the generalized variance of the underlying random vector, and as such the D-optimal design minimizes the generalized variance of the MLE estimate [15]. The D-optimal design is commonly used because it has a number of useful properties, including that it is simple to compute, it can be formulated as a convex objective and D-optimal designs are invariant under re-parameterization [15, 59]. As such, D-optimal designs are the main designs studied in this work. Other than D-optimal designs, there exist a number of other scalar design criteria, these are sometime referred to as the alphabetic optimality criteria because they are generally named using single letters; for example, A-optimal, c-optimal, I-optimal etc. [15].

To find an optimal design, $\mathcal{D}$, we seek to find the optimal support points $\boldsymbol{x}_j \in \mathcal{X}$ and optimal replicate counts $\beta_{i,j} \in \mathcal{B}$. Using a scalar objective, the overall optimization problem can be defined as

$$\begin{gathered} \max_{\mathcal{D}} \; \Psi(I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}})), \\ I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}}) = \sum_{j}^{N} \sum_{i}^{M} \beta_{i,j} I_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}}), \\ \mathcal{D} = \{\mathcal{X}, \mathcal{B}\}. \end{gathered} \tag{2.17}$$

This optimization is difficult for a number of reasons. It is non-linear in the support points, $\boldsymbol{x}_j$. It is also an integer programming problem because of the integer restrictions on $\beta_{i,j}$. Also, the optimal number of support points, N, is not generally known. These computational difficulties are already layered on top of those required to compute the FIM, as the FIM also requires computing sensitivities, which can be difficult for certain models.

A common approach for reducing the computational difficulty of optimal design is to relax the integer constraint such that a continuous weighting, $\xi_{i,j}$, over the input-observation pairs is optimized [15]. In this relaxed formulation, each $\xi_{i,j}$ replaces the corresponding $\beta_{i,j}$, however the $\xi_{i,j}$ have non-negative real-values which are constrained so that $0 < \xi_{i,j} < 1$ and $1 = \sum_i^M \sum_j^N \xi i, j$. For a relaxed design the replicate weights, $\xi_{i,j}$, are grouped in the weight set such that $\xi_{i,j} \in \mathcal{Z}$. A relaxed design, $\mathcal{D}_R$, is then defined by the support points and weight set such that $\mathcal{D}_R = \{\mathcal{X}, \mathcal{Z}\}$. The relaxed optimization problem can then be written as

$$\begin{gathered} \max_{\mathcal{D}_R} \; \Psi(I_{Tot}(\mathcal{D}_R, \bar{\boldsymbol{\theta}})), \\ I_{Tot}(\mathcal{D}_R, \bar{\boldsymbol{\theta}}) = \sum_{j}^{N} \sum_{i}^{M} \xi_{i,j} I_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}}), \\ \mathcal{D}_R = \{\mathcal{X}, \mathcal{Z}\}. \end{gathered} \tag{2.18}$$

An exact design, $\mathcal{D}$, specified using integer replicate allocations, $\beta_{i,j}$, can be converted to a relaxed design, $\mathcal{D}$, by dividing the replicate integers by the total sample size: $\beta_{i,j}/N_{Tot} = \xi_{i,j}$. The total FIMs, I_{Tot}, of the two design representations are therefore also related by the sample size so that $I_{Tot}(\mathcal{D}, \bar{\boldsymbol{\theta}})/N_{Tot} = I_{Tot}(\mathcal{D}_R, \bar{\boldsymbol{\theta}})$. Given these relations, the relaxed formulation is effectively normalized so that it does not depend on the overall sample size, N_{Tot}. To convert a relaxed design, $\mathcal{D}_R$, to an implementable exact design, $\mathcal{D}$, with integer replicate allocations, a sample size is selected and a rounding procedure is then applied to generate $\beta_{i,j}$ from the weights $\xi_{i,j}$. Simple rounding or specialized apportionment methods can be used for this conversion [60, 61, 62]. Due to the computational advantages, the relaxed form of the design problem is used for optimization throughout this thesis.

After solving the full design optimization problem, or the relaxed approximation to it, the research will achieve an optimal design with a specified list of support points and replicate allocations. This design will be optimal in a mathematical sense, but only conditionally for the given model structure, nominal parameter values, observation distributions and experimental constraints the research has specified. It is important to remember that optimal designs are generated under these strong assumptions. This means that other sources of uncertainty about model structure and alternative sources of variability are not necessarily hedged for in the optimized design. The optimal design can therefore serve as an idealized guide for a given system, however the design should be combined with good experimental judgment when implementing the prescribed measurements in practice.

Chapter 3

Applications of OED in Systems Biology

Published as "New opportunities for optimal design of dynamic experiments in systems and synthetic biology" by Nathan Braniff, and Brian Ingalls in *Current Opinion in Systems Biology* 9 (2018): 42-48

3.0.1 Summary

Recently developed dynamic experimental techniques offer new opportunities for the use of model-based experimental design in the construction and refinement of predictive models of cellular behaviour. Specifically, novel optogenetic and microfluidic tools have been made accessible by the distribution of low-cost, automated hardware designs that rely on readily available components and inexpensive construction processes. Experimental design methods can be applied to these platforms to identify time-varying input signals that generate maximally informative system responses. We review these developments and illustrate how the convergence of these approaches facilitates the construction of accurate biological models of both natural and engineered cellular systems.

3.0.2 Introduction

Quantitative characterization of the dynamic (time-varying) behavior of cellular systems is central to systems and synthetic biology. In systems biology, mathematical models are used to generate testable hypotheses and to provide insight into the behavior of natural systems. In synthetic biology, predictive models are increasingly needed to guide design, in the tradition of more established engineering disciplines. Unfortunately, biological models are often poorly constrained. This is, in part, due to a lack of appropriate data (e.g. time-series), the collection of which has traditionally required specialized commercial equipment or laborious protocols.

We review a maturing set of accessible techniques for precise dynamic perturbation and observation of cellular systems, and the low-cost, automated equipment that enable their implementation. These techniques can generate a wide range of dynamic excitations not previously achievable, and thus raise the question: what excitation profiles should be applied? Model-based design of experiment (MBDOE) tools can be used to answer this question. These tools provide a framework in which researchers can formalize their experimental goals and identify time-varying stimuli to accomplish them. We begin this brief review with a survey of recent innovations in experimental devices and protocols that facilitate the generation of broad classes of dynamic stimuli. We then provide an overview of model-based experimental design methods and review examples demonstrating their practical use for dynamic experimental design. Together these advances offer an effective work-flow, shown in Figure 3.1, for the design and implementation of dynamic biological experiments.

3.0.3 Time-varying control of cellular systems

Traditional experimental protocols allow for only a limited selection of temporal perturbation patterns (e.g. steps, sometimes pulses). In contrast, optical inputs and microfluidic delivery enable a wide range of time-varying stimuli. In addition, automated culture systems (turbidostats, chemostats) provide precise environmental control, uncoupling dynamic cellular responses from exogenous effects. For a recent review of developments in this area, see [63]. Below, we highlight works that exemplify the flexibility and precision of these dynamic techniques. We then survey a range of community-developed automated and accessible hardware for efficient implementation of these experimental approaches.

Experimental tools for dynamic stimuli The range of available light-induction systems is growing rapidly [64, 65]. Optical stimulus tools have become increasingly popular for precision dynamic perturbation experiments [66]. Toettcher *et al.* [67] used optically-regulated protein-protein interactions to control the translocation dynamics of signaling proteins. They used automated microscopy measurements and light delivery to implement

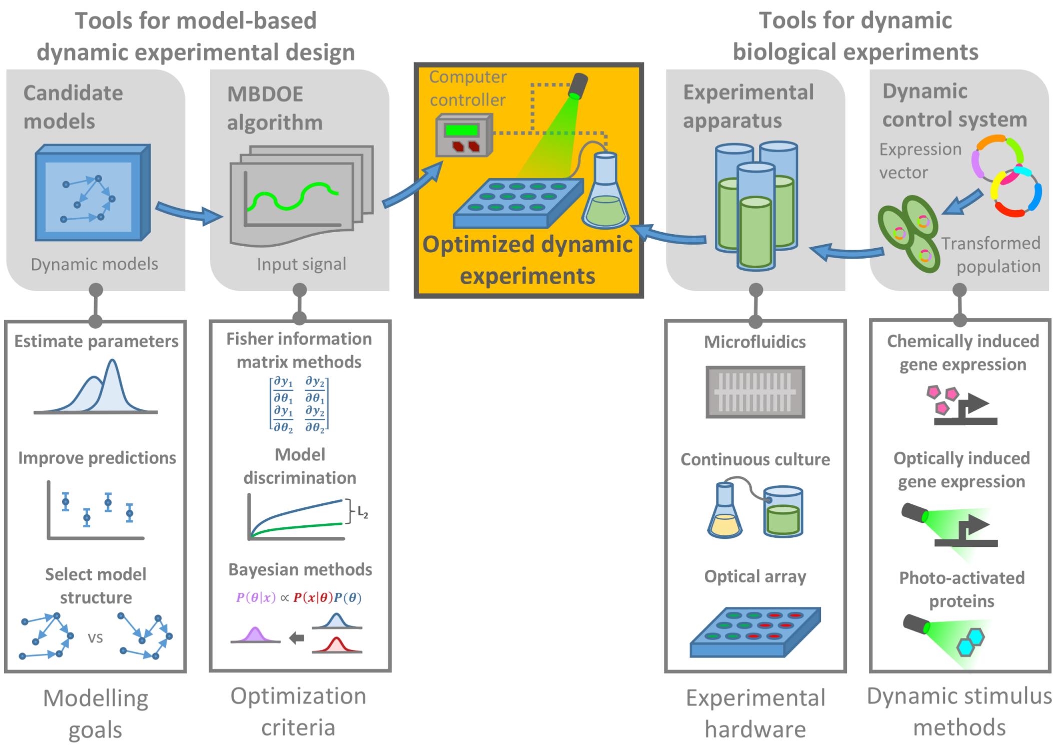


Figure 3.1: *Design and implementation of dynamic biological experiments.* Tools for model-based experimental design (left): Systems biologists often seek to improve a model by more accurately constraining its parameters, its predictions, or its structure. Model-based design of experiment (MBDOE) methods identify time-varying perturbation stimuli to achieve these goals via a range of optimization criteria. Tools for dynamic biological experiments (right): Temporal perturbation profiles can be realized with a range of emerging biological tools. Low-cost and open-source hardware systems can implement these dynamic experiments efficiently, resulting in informative experiments and improved model accuracy.

predefined translocation dynamics, to control population heterogeneity, and to implement robust clamping of targeted intracellular species. Milias-Argeitis *et al.* [68] demonstrated similar results using an optogenetic system and a computerized feedback circuit to control gene expression in real time. This *in silico* control set-up achieved robust, accurate tracking of reference concentration profiles and was used to mediate growth rates via feedback control of an essential metabolic enzyme [69]. In related work, Melendez *et al.* employed optogenetic induction in a chemostat system with microfluidic sampling and microscopy to demonstrate precise control of protein expression in a steady state microbial culture [70].

Several groups have made use of the CcaS/CcaR system for optogenetic control in bacteria. This light-responsive cyanobacterial two-component system was cloned into *Escherichia coli* by Tabor *et al.* [71]. Olson *et al.* have since developed the CcaS/CcaR system into a precise tool for gene expression control using parallelized delivery of light to an array of culture tubes, enabling more complex experiments and improved dynamic characterization [72]. They determined the response of the CcaS/CcaR system to a range of light stimulus profiles, and further demonstrated precise and predictive model-based control of the system in validation experiments. In complementary work, Davidson *et al.* characterized the response of the CcaS/CcaR optogenetic system to pulse width modulations [73]. More recently, Chait *et al.* [74] combined CcaS/CcaR optogenetic induction with microfluidic control of media contents and single-cell microscopy to achieve closed-loop control of single-cell and population-wide gene expression patterns.

Similar optical control strategies have been employed to reveal naturally occurring signaling network behaviour, via a photoactivateable adenylate cyclase stimulating the PKA pathway in yeast [75] and phytochrome-B-PIF activation of Ras/Erk in mammalian cells [76], providing insights into network structure and dynamic response.

Microfluidic approaches allow precise temporal manipulation of media contents [63]. Uhlendorf *et al.* employed a microfluidic chamber with real-time microscopy to implement a computer-in-the-loop controller of gene expression, resulting in tight regulation of variance in protein abundance [77]. A similar apparatus, using a media input system driven by differential hydrostatic pressure, was implemented by Menolascina *et al.* [78] and used by Fiore *et al.* [79] to compare the performance of multiple control strategies. Lugagne *et al.* [80] used microfluidic delivery of chemical inducers and monitoring of single-cell fluorescence to precisely maintain a genetic toggle switch at an unstable equilibrium using closed-loop control of gene expression. Microfluidic control has also been used to decode the behaviour of natural systems, e.g. Msn2 transcription factor signaling dynamics in the *Saccharo cerevisiae* stress response [81, 82].

Together, these recent projects demonstrate robust, accurate performance of closed-loop and model-based control schemes when implemented through optical or precise chemical inputs. This computational control of cellular systems, termed ‘cybergenetics’ [83], can serve as a valuable foundation for the development of maximally informative time-varying perturbation experiments. While chemical and optical induction are the most established methods for dynamic control, novel approaches are an area of active research. Tschirhart *et al.* pioneered an ‘electro-genetic’ system via a redox-sensitive transcription factor. The system was used to control both cell motility and cell signalling in response to electrical signals [84]. The use of more exotic induction systems, such as acoustic and magnetic actuation, has also been explored [85]. This expanding collection of methods could enable an even wider use of dynamic experimental techniques.

Accessible automated experimental hardware Dynamic experiments require time-varying adjustment of experimental conditions and are thus more labor intensive than traditional dose-response approaches. Efficient protocols can be achieved through the use of automated equipment, which enables precise implementation of temporal perturbation and observation profiles. In recent years these automated tools have become increasingly accessible, due to novel technologies such as 3D printing and modular programmable controllers.

Continuous culture systems are attractive tools for dynamic experimentation because they allow for extended experimental runs under consistent, tunable conditions. Toprak *et al.* [86] provided designs for a flexible user-constructed continuous culture ‘morbidostat’. This device, built for the study of evolutionary response to antibiotic treatment, can serve as a chemostat (to maintain constant flow-through rate) or as a turbidostat (to maintain constant growth rate). The complementary design provided by Takahashi *et al.* [87] describes a low-cost, extensible turbidostat, utilizing 3D-printed parts. A similar design, presented by Matteau *et al.* [88], can operate as either a turbidostat or a chemostat, and is supported by a graphical user interface (GUI)-enabled custom software package.

The optogenetic work of Olson *et al.* was implemented through a light tube array (LTA), a custom system enabling parallel delivery of time-varying light signals at varied frequencies to 64 liquid cultures [72]. Follow-up designs for a variation of this system, the light plate apparatus, offer a highly flexible tool for optogenetic experiments [89]. The designs for this tool and its control software are open-source. Wang *et al.* combine ideas from both the LTA and continuous culture systems in a custom-built bioreactor capable of time varying optogenetic stimulation [90]. In addition to light activation, this system employs photosensor readouts of fluorescent protein activity.

Customized microfluidics is becoming increasing accessible [91, 92, 93]. The Metafluidics repository [94] contains a wide variety of custom microfluidic designs, and could enable widespread dissemination of tools necessary for dynamic experiments. Open-source software for automating microscopy data collection allows observations to be collected and analysed efficiently [95, 96]. Microfluidics also offers support for single-cell ‘omics’ data collection, which, when combined with precise dynamic perturbations, could provide improved dynamic characterization of cellular behaviour across the genetic landscape [97, 98].

3.0.4 Model-based design of experiments: dynamic stimuli

Techniques for optimal experimental design were pioneered in the 1950s, notably by Elfving, Kiefer and Box [36, 99, 20]. Much of this work focused on linear factorial models commonly used in statistics [100]. Later work, particularly in the field of systems identification, extended these experimental design techniques to linear dynamic models [44]. These tools are not directly applicable to the nonlinear, (pseudo-)mechanistic models typically employed in systems biology. Such models are better served by model-based design of experiments (MBDOE) [21]. These design approaches, built largely on earlier work in the chemical and bioprocess engineering literature, employ hypothesized models to identify optimal experiments. The results of these experiments then inform model structure and parameterization in an iterative process.

In particular, MBDOE methods have been promoted as a means to resolve problematic identifiability issues in systems biology modelling. Through a series of papers, Sethna and colleagues characterized ‘sloppiness’ in parameter estimation as a common feature of systems biology models (e.g. [101], see also [102]). Sloppiness here is defined by wide variation in the confidence of parameter estimates for a given model. (Formally sloppiness can be thought of as a large ratio between the largest and smallest eigenvalues of the parameter covariance matrix or the Fisher information matrix.) This influential work suggested that sloppiness is perhaps unavoidable, and thus cast doubt on the promise of achieving parametrically accurate models in systems biology. In response, the research community demonstrated the role of experimental design in ameliorating sloppy parameter fits [103, 8] and articulated the important distinction between sloppiness and identifiability [104]. (Sloppiness is concerned with the relative uncertainty of various parameters where as identifiability is concerned with bounding the absolute uncertainty of all parameters of interest.) This more recent work reveals that sloppiness in parameter fits does not necessarily pose challenges for identifiability, especially when optimal experiments have been identified.

Methods for model-based experimental design can be classified according to:

i. Design space - experimental design decisions for which the method provides insight. Examples: selection of targets for external control, choice of time-varying perturbations to be implemented, selection of species or outputs to be measured, and scheduling of sampling times.

ii. Goal - model-based optimality objective. Examples: accuracy of parameter estimation, decidability of model structure, or variance of model predictions.

iii. Optimization approach - how the optimality objectives are measured and evaluated. A range of formulations have been employed, derived from both frequentist and Bayesian statistics, as well as systems identification and control theory.

Several reviews of model-based experimental design have appeared recently: Franceschini and Macchietto [21] provide a comprehensive review of MBDOE methods with clear exposition of the main theoretical points; Kreutz and Timmer [105] enumerate the many aspects of an experiment that can be optimized, and discuss both model discrimination and optimal parameterization approaches; Chakrabarty *et al.* [22] provide a comprehensive survey of the field, describing both Fisher information and Bayesian paradigms; Ryan *et al.* [106] provide a focused review for Bayesian methods. (Fisher information, a measure based on local parametric sensitivities, is very commonly used in this context. Bayesian approaches rely on fewer assumptions, but are more computationally demanding.)

The temporal excitation profiles generated by model-based experimental design techniques yield observations that are optimally useful toward a given modelling goal. To date, most designs have been restricted to step-wise inputs, though arbitrary input profiles have been considered (e.g. control vector parameterizations [107]). Most early treatments of model-based design of dynamic experiments in biology used Fisher information-based optimality criteria with the goal of accurate parameter estimation [108, 109]. (See [110, 111] for workflows.) More recently, Ruess *et al.* extended these tools to a stochastic setting to address single-cell experiments [112].

These Fisher information-based techniques rely on local analysis centered at a nominal point in parameter space. They are thus of limited use when applied at poorly determined parameter estimates. Global experimental design approaches, including Bayesian techniques (e.g. [113]) and sparse-grid approximations (e.g. [114]), can overcome this limitation, but are typically computationally demanding and can be challenging to implement.

Although Fisher information-based techniques have focused primarily on improving model parameterization, experimental design of dynamic stimuli has also been used for model discrimination. The methods of [47] and [115] produce input signals that maximize the distance between model outputs (defined in terms of the L_2 norm and the Kullback-Leibler divergence, respectively). A complementary approach, presented by Apgar *et al.*, involves constructing model-based controllers for candidate models. Model discrimination is then based on the effectiveness of the resulting control inputs. Robust tools for model discrimination are provided in [116, 117].

Tools like the light plate apparatus [89] and automated microfluidics [95, 96] are heralding an era of high-throughput dynamic perturbation experiments. With this equipment, researchers will be able to probe multiple components of a system in parallel or to exhaustively examine the effect of environmental context on the dynamic behavior of cellular systems. Some model-based experimental design techniques have been proposed to address such experiments, e.g. design methods suited to experiments run in parallel ([118, 119]) or in conditions that allow real-time (i.e. online) redesign of experimental inputs ([120, 121]). Further development is needed, in both experimental design methods and experimental techniques, to realize optimal high-throughput experimental approaches.

3.0.5 Optimal dynamic experiments in practice

As methods for dynamic perturbation of cellular systems have caught up to MBDOE theory, examples of their combined utility have appeared. These join a longer history of successful applications of model-based experimental design in bioprocess engineering (e.g. [122, 123, 124]) where temporally controlled inputs are more standard.

An early example of model-based experimental design in the context of cellular systems biology is the work of Bandara *et al.* [9] who used optimal design of time-varying chemical inputs to optimize fluorescence microscopy experiments for a model of PIP_3 signaling in fibroblast cells. Optimization allowed them to realize a 60-fold reduction in the mean variance of parameter estimates over non-optimized experiments. They further note that the optimal experiments improved identifiability by reducing correlation between parameter estimates.

More recently, Ruess *et al.* [10] applied optimal Fisher information-based design techniques using a stochastic model to generate optimized inputs for an optogenetic signaling system in yeast. This was coupled with a Bayesian inference algorithm, used to update the stochastic model parameters. This iterative procedure out-performed randomly chosen experimental designs. The predictive power of the resulting model was illustrated by design of novel control inputs that accurately generated prescribed gene expression patterns.

3.0.6 Future Directions

Implementations of cybergenetic *in silico* control of cellular systems are in their infancy. They are poised to have significant impact on our ability to precisely characterize and manipulate cellular activity. Continued improvement of experimental tools will expand the range of time scales, stimulus targets and cell types for dynamic perturbations [84, 64, 65]. Dissemination of low-cost, open-source designs for the associated hardware can foster widespread adoption [87, 89]. One barrier to generalizing the resulting data and models is the traditional approach of collecting data in relative (i.e. arbitrary) units. The development of standardized unit-full measures [125, 126, 127] will be essential for the community to fully realize the collaborative potential of these new modelling tools.

As for experimental design, a current barrier to adoption is simply the wide variety of model-based approaches available in the literature, many of which seem equally well-suited to any given task. Comparison studies, such as [128, 129], especially those that would compare experimental implementations, can serve as valuable 'consumer guides', and may lead to community consensus as to which tools are most effective in which contexts. To complement the expected increase in application of model-based experimental design, continued theoretical work is needed to further address limitations and constraints commonly faced in cell biology, such as model mismatch [130, 131].

Following the precedents set by [9] and [10], further applications of experimental design tools in time-varying perturbation experiments will increase our understanding of dynamic behavior in natural and engineered biological systems. In synthetic biology, tools for automated and data-driven rational system design have begun to emerge [12, 132]. Optimized experiments hold promise for extension of these ideas to complex dynamic cellular behaviours.

Chapter 4

Optimal Experimental Design for a Physiologically-aware Model of Gene Expression

Published as "Component Characterization in a Growth-Dependent Physiological Context: Optimal Experimental Design" by Nathan Braniff, Matt Scott, and Brian Ingalls in *Processes* 7.1 (2019): 52.

4.1 Summary

Synthetic biology design challenges have driven the use of mathematical models to characterize genetic components and to explore complex design spaces. Traditional approaches to characterization have largely ignored the effect of strain and growth conditions on the dynamics of synthetic genetic circuits, and have thus confounded intrinsic features of the circuit components with cell-level context effects. We present a model that distinguishes an activated gene's intrinsic kinetics from its physiological context. We then demonstrate an optimal experimental design approach to identify dynamic induction experiments for efficient estimation of the component's intrinsic parameters. Maximally informative experiments are chosen by formulating the design as an optimal control problem; direct multiple-shooting is used to identify the optimum. Our numerical results suggest that the intrinsic parameters of a genetic component can be more accurately estimated using optimal experimental designs, and that the choice of growth rates, sampling schedule and input

profile each play an important role. The proposed approach to coupled component-host modelling can support gene circuit design across a range of physiological conditions.

4.2 Introduction

Proposed applications of synthetic biology demand complex synthetic constructs involving dynamic internal regulation. Novel analytic and experimental approaches will be needed to efficiently navigate the corresponding design space [133, 134]. Model-based design approaches promise to (partially) replace costly experiments with computer simulations. A range of theoretical approaches to automated or computer-assisted design have been published [135, 136, 137, 138, 139, 140, 141, 142]. When supported by reliable characterization of genetic regulatory components, automated design algorithms can greatly increase the efficiency of design. As an example, Nielsen et al. demonstrated efficient automated design of very large genetic logic circuits from a carefully designed and characterized regulatory library [12]. Systematic characterization of genetic components can include generation of standardized data sheets [143], use of standardized relative units [144], and predictive characterization of circuit dynamics [145]. This type of characterization is demanding and resource-intensive; to date it has rarely been accomplished. The resulting knowledge deficit is a major bottleneck to wider use of model-based design.

A drawback of standard approaches to characterization is that they fail to distinguish effects due to the host's physiological state from the intrinsic properties of the construct [146]. Here we define intrinsic properties to be those biochemical properties specific to a genetic construct's sequence. The physiological state incorporates, among other features, (i) the DNA quantity and gene copy number, (ii) available RNA polymerases (RNAPs) and ribosomes, and (iii) the cell volume, all of which impact gene expression [147, 2]. Calibration of model parameters without accounting for the cell's physiology results in aggregate parameters that describe lumped effects from both host and component. Such a model cannot be trusted to extrapolate beyond the physiological state in which it was calibrated. Separating host state and component behaviour is a difficult and multivariate problem. Experimentalists cannot directly perturb or even measure many physiological properties. The cell's physiological state can be modulated indirectly by external or internal perturbations including modulation of (i) nutrient sources [2, 148], (ii) antibiotics [148], (iii) gene expression burden [148], (iv) metabolic fluxes [149, 150], as well as temperature, pH and osmolarity. The aggregate effect of each of these perturbations influences growth rate, but predicting host properties from the perturbation or the growth rate is not generally possible. However, for nutrient-limited growth, it has been shown that the exponential growth rate acts as a summary statistic for the physiological state of an *E. coli* host cell [2, 147]. Klumpp and Hwa demonstrated how nutrient-limited growth rates can then be used as an aggregate predictor of the physiological effects on gene expression [147].

In [147], Klumpp and Hwa stop short of proposing an explicit dynamic model for coupling physiology to gene expression. They focus primarily on steady-state behaviour. More recent works have developed coupled models of gene expression, host physiology and growth rates [151, 152, 153]. Here, we focus specifically on the use of a coarse-grained model to empirically predict physiological properties from an observed exponential growth rate. Our model distinguishes intrinsic parameters of the genetic construct from extrinsic parameters of the host's physiological state. This distinction allows for extrapolation across physiological conditions and for reuse of the estimated component parameters. We have aimed to keep the parameter set small to maximize both identifiability and interpretability.

Noise and nonlinearity make precise estimation of the parameters that characterize biomolecular systems a challenging task [101, 102]. Optimal experimental design (OED) tools offer a means to improve the efficiency of data collection for model calibration [8, 103]. Despite its potential to increase experimental efficiency and the precision of parameter estimates, OED has not seen widespread use in laboratory experiments within systems or synthetic biology. Two notable exceptions are reported by Bandara *et al.* and Ruess *et al.*; both groups implemented optimal experimental design for efficient calibration of dynamic biological models [9, 10]. OED techniques are especially promising when coupled with optogenetic or microfluidic techniques, which allow for a broad range of dynamic perturbations *in vivo* [154]. Here, we employ optimal experimental design algorithms originally demonstrated for chemical and bioprocess engineering applications [155, 156, 157] for the characterization of a genetic component from simulated data.

We develop our physiologically-aware model of gene expression for an exponentially growing *E. coli* population, because this is the only host system for which the necessary data has been collected. We use nutrient quality in exponential phase as a predictable controller of the cell's growth rate and relevant physiological state. We use the model to optimally design dynamic induction experiments across a set of growth rates to simulate efficient estimation of the intrinsic parameters of the genetic component. We adopt an optimal design approach that expresses the experimental design as an optimal control problem, which can be efficiently solved using numerical optimal control methods, such as multiple shooting. We specifically focus on multiple shooting because this method provides efficient solutions for the complex experimental control problems addressed in this work, while also being tractable and easily implemented for the large number of states required in optimizing experimental design objectives. We treat the induction profile, growth rate, and sampling schedule as experimental controls. Their selection is simultaneously optimized over multiple sub-experiments. This simultaneous optimization of sampling schedule and multiple experimental perturbations, including growth rates, is an improvement over previously published accounts of OED in synthetic biology (although see [107]). Previously published analyses have either assumed constant sampling rates or have chosen sampling schedules in a secondary optimization step [10, 158], both of which sub-optimal [159]. We use numerical simulations to demonstrate that the optimal experiments outperform intuitively designed experiments: the optimal designs improve both parameter estimation accuracy and out-of-sample prediction accuracy. We further demonstrate that the use of multiple growth rates is important for model identifiability over realistic parameter ranges.

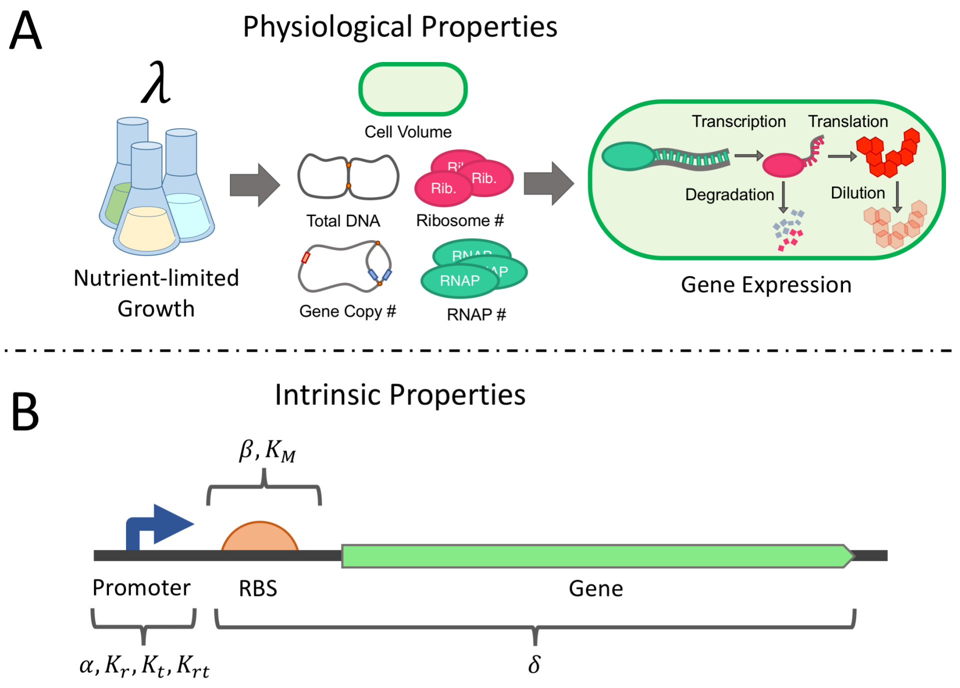


Figure 4.1: (A) Growth media nutrient quality dictates the growth rate of an *E. coli* culture. The observed growth rate can be used to predict many physiological parameters of the host, which in turn influence gene expression. (B) By accounting for physiological parameters, we can optimally estimate intrinsic parameters of a genetic construct, which reflect properties of its sequence. These intrinsic parameters can be reused across growth conditions and can be used to guide changes to the construct sequence.

4.3 Materials and Methods

In this work we design optimal experiments for calibrating a dynamic model of expression of a genomically-integrated gene, induced by an activating transcription factor (TF). For simplicity, we assume that the controlled induction input is the activating transcription factor copy number, u. (More realistically, the controlled input would be some signal that influences the TF abundance.) Our population-averaged model incorporates both gene-specific intrinsic parameters and parameters that characterize aspects of the host physiology. The physiological parameters of the model are dependent on the steady-state exponential growth rate, λ, controlled via nutrient quality. The empirically observed relations and rationale for the connection between a cell's physiological parameters and its growth rate have been discussed in past works [147]. The model describes mRNA copy number, X_{rna}, and protein copy number, X_{prot},

$$\begin{aligned}
\frac{d}{dt}\frac{X_{rna}}{V} &= \alpha\frac{g}{V}\frac{\frac{P_a}{\eta G}K_r + \frac{P_a K_{rt}}{(\eta G)^2}u(t)}{1 + \frac{P_a}{\eta G}K_r + \left(\frac{K_t}{\eta G} + \frac{P_a K_{rt}}{(\eta G)^2}\right)u(t)} - \delta\xi\frac{X_{rna}}{V} \\
\frac{d}{dt}\frac{X_{prot}}{V} &= \frac{\beta\frac{R_f}{V}}{K_M + \frac{R_f}{V}}\frac{X_{rna}}{V} - \lambda\frac{X_{prot}}{V}.
\end{aligned} \tag{4.1}$$

The gene's intrinsic characteristics are captured by the intrinsic parameters α, K_r, K_t, K_{rt}, δ, β and K_M, while expression is also dependent on the physiological parameters: V, g, P_a, G, R_f, and λ, which are growth rate dependent, and η and ξ, which are fixed. The intrinsic parameters characterize each gene's induction behaviour. In contrast, the physiological parameters reflect the state of the host cell (Figure 4.1). These parameters are summarized in Table 4.1, along with nominal values and relevant ranges. Justification for these parameter values is provided in the next section.

Table 4.1: A table describing the symbols, nominal values, and feasible ranges of the intrinsic parameters and physiological properties used in the model.

Parameter Label	Intrinsic Parameter	Nominal Value	Feasible Range
Intrinsic Transcription Parameters			
Promoter Escape Rate	α	20 min^{-1}	$[1 - 30]$
RNAP-Promoter Binding	K_r	40	$[10 - 40]$
TF-Promoter Binding	K_t	5×10^5	$[2 \times 10^3 - 1 \times 10^6]$
TF-RNAP Interaction	K_{rt}	1.09×10^9	$[4.02 \times 10^5 - 5.93 \times 10^{10}]$
Intrinsic mRNA Decay Parameters			
mRNA Decay Rate	δ	2.57×10^{-4} $\mu\mathrm{m}^{-3}\mathrm{min}^{-1}$	$[7.7 \times 10^{-5} - 7.7 \times 10^{-4}]$
Intrinsic Translation Parameters			
Max. Initiation Rate	β	4.0 min^{-1}	$[1 - 10]$
Half-saturating Constant	K_M	750 μ m^{-3}	$[750 - 1500]$

Property Label	Physiological Property	Value at μ=0.6 db/hr	Value at μ=3 db/hr
Physiological Properties of Transcription			
Gene Copy Number	g	1.4	5.7
Available RNAP	P_a	1000	4000
Genome-lengths of DNA	G	1.3	4.3
Physiological Properties of mRNA Decay			
RNase Concentration	ξ	900 $\mu\mathrm{m}^{-3}\mathrm{min}^{-1}$	900 $\mu\mathrm{m}^{-3}\mathrm{min}^{-1}$
Physiological Properties of Translation			
Free Ribosomes	R_f	600	7000
General Physiological Properties			
Cell Volume	V	0.4 $\mu\mathrm{m}^{-3}$	2.24 $\mu\mathrm{m}^{-3}$
Growth Rate	λ	0.7×10^{-2} min^{-1}	3.5×10^{-2} min^{-1}

4.3.1 Derivation of the Physiological Gene Expression Model

We develop the model for the case of an exponentially growing *E. coli* population, where growth rate is controlled by nutrient limitation. While some features of the model may generalize to other cell types, it is only in the case of *E. coli* that sufficient data has been collected to provide reasonable estimates of functional relations and parameter values.

We employ two standard measures of growth rate. The doubling rate μ is the inverse of the doubling time τ_μ. The exponential growth rate $\lambda = \ln(2)/\tau_\mu$ is used to describe population growth as $P_0 e^{\lambda t}$, and is thus suitable to use as the dilution rate in a differential equation model. We treat the exponential growth rate, λ, as an independent variable determined by experimental conditions. We also assume the growth rate is constant throughout each experiment, which implies that no nutrient shifts occur.

Cell Volume and Mass, DNA Content and Protein Mass

Prior work in bacterial physiology has established expressions for the physiological parameters V, G, and g in terms of λ. (While each of the parameters is expected to undergo random fluctuations and to vary with the cell cycle, these expressions approximate average values over time.)

Cell volume has been shown to scale with growth rate exponentially [160, 161]:

$$V = V_0 e^{(C+D)\lambda}. \tag{4.2}$$

Here the constant V_0 is the 'initiation volume' measured to be $V_0 = 0.28\ \mu\text{m}^3$ by Si *et al.* [160]. The parameters C and D represent the time periods required to replicate the chromosome and to septate, respectively. We take $C = 40$ min and $D = 20$ min [161].

An expression for the average number of genome equivalent lengths of DNA, G, is given in terms of C, D, and λ by Cooper and Helmstetter [161]:

$$G = \frac{1}{\lambda C}(e^{(C+D)\lambda} - e^{D\lambda}). \tag{4.3}$$

Individual loci vary in copy number based both on growth rate and on their location relative to the origin of replication. (At faster growth rates there are more copies of genes near the origin because the cell must have multiple rounds of DNA replication underway.) We use the constant l_{ori} to indicate the gene's position relative to the origin: $l_{ori} = 0$ at the origin; $l_{ori} = 1$ at the terminus. Bremer and Churchwood [162] provide a simple derivation for the average gene copy number, g, of a specific locus as

$$g = e^{((C+D)-l_{ori}C)\lambda}. \tag{4.4}$$

Total RNA polymerase (RNAP)

We first note that the buoyant density of *E. coli* has been observed as constant across growth rates: Kubitschek *et al.* found $\rho = 1.09$ pg μm^{-3} [163] while Basan *et al.* report $\rho = 0.215$ pg μm^{-3} [164] (average of reported values). We use the dry weight data from [2] with Si *et al.*'s description of volume [160] to select an intermediate density of $\rho = 0.55$ pg μm^{-3} [1].

The average cell mass for a given growth rate is then:

$$M_{Tot} = \rho V = \rho V_0 e^{(C+D)\lambda}. \tag{4.5}$$

This total cell mass M_{Tot} can be partitioned into fractions of protein and other constituents. We fit the protein fraction of the mass, denoted Φ_{pr}, to data from [2] with a linear function (supplemental Figure S1, Appendix A):

$$\Phi_{pr} = \kappa_{pr}\lambda + \Phi_{pr0} \tag{4.6}$$

with $\kappa_{pr} = -6.47$ min and $\Phi_{pr0} = 0.65$.

The total RNAP fraction of the overall protein mass, which we denote Φ_p, exhibits an approximately linear dependence on growth rate, which we fit to data from [2] (supplemental Figure S2, Appendix A) as

$$\Phi_p = \kappa_p\lambda + \Phi_{p0}. \tag{4.7}$$

with $\kappa_p = 0.30$ min and $\Phi_{p0} = 0.0074$. We can then express the total mass of all RNAP protein, M_{RNAP} as

$$M_{RNAP} = \underbrace{(\kappa_p\lambda + \Phi_{p0})}_{\text{RNAP \% of Prot.}} \overbrace{(\kappa_{pr}\lambda + \Phi_{pr0})\underbrace{\rho V_0 e^{(C+D)\lambda}}_{\text{Cell Mass}}}^{\text{Protein Mass}}. \tag{4.8}$$

The total number of RNAPs per cell can be determined by dividing M_{RNAP} by the protein mass per RNAP core enzyme, which we denote m_{rnap} and is estimated as 6.3×10^{-7} pg [165].

[1]The compromise value selected in this work does not take into account the effect that the mass of water has on the two alternative density measurements that are cited. This means the reader should treat the proposed value as a rough approximation subject to error.

Available RNAP

Each RNAP can be classified by its state: freely diffusing; weakly DNA bound at a non-specific site; actively transcribing other genes; paused or non-functioning during transcription; or immature [5, 166, 167]. Below, we describe promoter binding in terms of a thermodynamic model that takes into account the observed rapid equilibrium between non-specifically bound and freely diffusing RNAPs [5]. We therefore define the combined pool of free and non-specifically bound RNAPs as the *available* RNAP pool, with molecular population size P_a. Denoting the fraction of RNAP in this available pool by Φ_a, we estimate, from Bakshi *et al.* [5] and Stracy *et al.* [3] (details in the supplement, Section 1.3, Appendix A):

$$\Phi_a = \kappa_a \lambda + \Phi_{a0} \tag{4.9}$$

with $\kappa_a = -9.3$ min and $\Phi_{a0} = 0.59$. Using this relation we have an expression for the available RNAP:

$$P_a = \frac{\rho V_0}{m_{rnap}} \left(\kappa_a \lambda + \Phi_{a0}\right) \left(\kappa_p \lambda + \Phi_{p0}\right) \left(\kappa_{pr} \lambda + \Phi_{pr0}\right) e^{(C+D)\lambda}. \tag{4.10}$$

Transcription Rate

We describe the initiation of transcription via a thermodynamic equilibrium model of promoter occupancy involving available RNAPs, transcription factors (TFs), promoter copies, and non-specific binding sites along the genomic DNA [168, 169]. At a given growth rate, we assume there are P_a available RNAP copies and T_a active transcription factor copies diffusing along the genomic DNA, and that the DNA contains N_s non-specific binding sites to which these DNA binding proteins may weakly attach and g copies of the regulated promoter of interest. Further we assume that $N_s \gg P_a, T_a, g$ and that each binding of an RNAP or a TF to a non-specific site or a promoter is characterized by an associated binding energy; ϵ_{rn} and ϵ_{rg} for RNAP binding to the non-specific sites and promoters, respectively, and ϵ_{tn} and ϵ_{tg} for transcription factor binding to non-specific sites and promoters, respectively (all ϵ are negative [170]).

We use these species and site counts to enumerate the possible arrangements of RNAP and TF across the genome, and we use the binding energies to derive Boltzmann weights for each arrangement [168]. We denote the differences between the energy involved in binding the promoter and the background non-specific binding as $\Delta\epsilon_t = \epsilon_{tg} - \epsilon_{tn}$ and $\Delta\epsilon_r = \epsilon_{rg} - \epsilon_{rn}$. (Note, $\epsilon_{tn} > \epsilon_{tg}$ and $\epsilon_{rn} > \epsilon_{rg}$ so that $\Delta\epsilon_t$ and $\Delta\epsilon_r$ are both negative [170].) We denote the Boltzmann weights as $K_r = e^{-\frac{\Delta\epsilon_r}{k_BT}}$, $K_t = e^{-\frac{\Delta\epsilon_t}{k_BT}}$ and $K_{rt} = e^{-\frac{(\Delta\epsilon_r+\Delta\epsilon_t+\epsilon_{pt})}{k_BT}}$ (with k_B, Boltzmann's constant, T, temperature in degrees Kelvin) where ϵ_{rt} is the binding energy between RNA polymerase and transcription factor when both are bound to the same promoter. Then, the equilibrium probability of a single promoter being occupied by an RNAP is (further details in the supplement, Section 1.4, Appendix A):

$$p_{bound} = \frac{\frac{P_a}{N_s}K_r + \frac{P_aT_a}{N_s^2}K_{rt}}{1 + \frac{P_a}{N_s}K_r + \frac{T_a}{N_s}K_t + \frac{P_aT_a}{N_s^2}K_{rt}}. \tag{4.11}$$

With g promoter copies, and presuming that open complex and promoter escape occurs at a fixed rate α [171] we have the initiation rate as

$$\text{Initiation rate} = \alpha g \frac{\frac{P_a}{N_s}K_r + \frac{P_aT_a}{N_s^2}K_{rt}}{1 + \frac{P_a}{N_s}K_r + \frac{T_a}{N_s}K_t + \frac{P_aT_a}{N_s^2}K_{rt}} \tag{4.12}$$

We next establish estimates of the parameter values for transcription. To begin, multiplying G by the genomic density of non-specific binding sites, $\eta = 5 \times 10^6$ sites/genome, yields an estimate of the total number of non-specific binding sites N_s as a function of growth rate [168].

From Heyduk *et al.*, we have the rate of open complex formation α for the phage lambda P_R promoter as 19.2 min^{-1} [171]. Assuming P_R exemplifies a strong promoter, we estimate a reasonable range based on analysis of constitutive promoters and mutants of the P_R sequences as $1 - 30$ min^{-1} [172, 173, 174, 171].

The constants K_r and K_t can be interpreted as ratios of dissociation constants for the DNA-binding species (RNAP and TF) binding to non-specific DNA versus the promoter sequence [168]. This provides a convenient method for constraining their feasible values from reported dissociation rates. Expressing them as such yields; $K_r = K^{ns}_{rnap}/K^{prom}_{rnap}$, $K_t = K^{ns}_{tf}/K^{prom}_{tf}$ and $K_{rt} = K_rK_t\exp(\epsilon_{rt}/k_BT)$. Here K^{ns}_{rnap} and K^{ns}_{tf} are dissociation constants of the TF and RNAP with respect to non-specific DNA binding and K^{prom}_{rnap} and K^{prom}_{tf} are dissociation constants for promoter binding. The non-specific dissociation constant for RNAP, K^{ns}_{rnap}, has been observed to be approximately 10000 nM [168]. The promoter-specific dissociation constant, K^{prom}_{rnap}, varies from promoter to promoter. For *lacP1* it is approximately 550 nM and for T7 it is approximately 3 nM [168]. To represent a relatively weak constitutive leak for an inducible promoter, we expect values nearer to the *lacP1* dissociation constant would be reasonable, and so presume K^{prom}_{rnap} could range from 250 nM to 1000 nM. Using the above value for K^{ns}_{rnap} and the range for K^{prom}_{rnap} we estimate a feasible range for K_r to be between 10 and 40 (unitless). We set K_{rnap}^{prom} to be 250 nM and therefore our nominal value for K_r is 40, suggesting a slightly stronger leak than found in uninduced *lacP1*. (Over the growth rates we consider, we found K_r to be practically unidentifiable in simulation studies for various optimized designs, with high variability in its estimated value. We therefore fixed it to the nominal value for our experimental design and fitting.)

To estimate K_t, Stormo suggests a reasonable range for the promoter-specific dissociation constant, K_{tf}^{prom}, of 0.01 nM to 1000 nM [175]. We assume we are working with a relatively strong activator and that K_{tf}^{prom} lies in the range 0.01 nM to 5 nM. Stormo also suggests that non-specific binding dissociation constants are between three and six order of magnitude less than the specific binding constants. For simplicity, we follow Bintu *et al.* and let $K_{tf}^{ns} = 10000$ nM [168]. This yields a range for K_t between 2000 and 10^6. We chose $K_{tf}^{prom} = 0.02$ and $K_t = 5 \times 10^5$ as nominal values.

As noted, the parameter K_{rt} can be expressed as $K_{rt} = K_r K_t \exp(-\epsilon_{rt}/k_B T)$ with ϵ_{rt} representing the energy involved in the RNAP-TF interaction at the promoter. Few estimates exist for such values. Bintu *et al.* use demonstrative values of $\epsilon_{rt}/k_B T$ ranging from -3.5 to -4.5. We allow a wider range from -3 to -5. This allows K_{rt} to range from 4.02×10^5 to 1.61×10^{10}. We take a nominal value of $K_{rt} = 1.09 \times 10^9$.

mRNA Degradation

The data in [2] indicate that the mRNA decay rate is linear in mRNA copy number, with a growth rate-independent rate constant. In [147], the authors hypothesize that this is due to the maintenance of a constant concentration of RNase E, the primary ribonuclease involved in mRNA decay initiation in *E. coli* [176]. (RNase E exhibits auto-regulation which appears to keep its concentration constant [177, 178, 179]. Moreover, RNase E appears to be expressed in excess, resulting in insensitivity to small changes in its abundance [180, 179].) Under this assumption and using mass action we have a simple model for mRNA decay as follows

$$\text{mRNA Decay Rate (copy \# per min)} = \delta \frac{E}{V} X_{rna} \tag{4.13}$$

Here E is the copy number of RNase E. Assuming $\frac{E}{V}$ is constant across growth rates (and therefore $E \propto V$), we can express the mRNA degradation rate as

$$\text{mRNA Decay Rate (copy \# per min)} = \delta \xi X_{rna} \tag{4.14}$$

Here ξ is the constant concentration of relevant degrading enzymes in the host.

The parameter δ represents the susceptibility of the the mRNA transcript to RNase degradation, and acts as a mass action constant. The half-life of mRNAs can range from $1 - 10$ min; Chen *et al.* suggest the mean RNA half life is near to 2.5 min [181, 182, 183]. Based on a constant concentration of RNase E of $\xi = 900\ \mu\text{m}^{-3}$ [184] and taking a nominal half-life of 3 min, we suggest $\delta \approx 2.57 \times 10^{-4}\ \mu\text{m}^{-3}\ \text{min}^{-1}$.

Total and Free Ribosome Populations

A linear relation for the fraction, Φ_r, of protein mass that is composed of ribosomal protein was derived in [148]:

$$\Phi_r = \kappa_r \lambda + \Phi_{r0} \tag{4.15}$$

Fitting this model to data from Bremer [2] yields estimates of $\kappa_r = 5.48$ min and $\Phi_{r0} = 0.030$, (see supplemental Figure S4, Appendix A). The total ribosomal protein mass M_{Rib} is then

$$M_{Rib} = (\kappa_r \lambda + \Phi_{r0})(\kappa_{pr} \lambda + \Phi_{pr0})\, \rho V_0 e^{(C+D)\lambda}. \tag{4.16}$$

Each ribosome has a mass of 2.7 MDa of which 35% is protein [185]. The individual ribosomal protein mass is therefore $m_{rib} = 1.57 \times 10^{-6}$ pg. We then have the number of ribosomes per cell, R_{Tot}, as

$$R_{Tot} = (\kappa_r \lambda + \Phi_{r0})(\kappa_{pr} \lambda + \Phi_{pr0})\, \frac{\rho V_0 e^{(C+D)\lambda}}{m_{rib}} \tag{4.17}$$

Based on results in Dai *et al.* [186], we have assumed that about 10% of the ribosomes are inactive. Assuming that these are free, we denote this fraction as $\Phi_f = 0.1$ and have

$$R_f = \Phi_f R_{Tot} = \frac{\Phi_f \rho V_0}{m_{rib}} (\kappa_r \lambda + \Phi_{r0})(\kappa_{pr} \lambda + \Phi_{pr0})\, e^{(C+D)\lambda}. \tag{4.18}$$

Translation Rate

We employ the Michaelis-Menten model of translation initiation proposed by Borkowski *et al.* [187]. In terms of the free ribosome concentration R_f:

$$\text{Translation Rate (copy \# per min)} = \frac{\beta \frac{R_f}{V}}{K_M + \frac{R_f}{V}} X_{rna} \tag{4.19}$$

In this model, the mRNA's ribosome binding site (RBS) is characterized by two constants: β, the maximal translation initiation rate per mRNA, and K_M, a half-saturating constant specific to the given RBS. A detailed justification for this model is provided in supplemental Section 1.6, Appendix A.

To estimate β, we note that translation initiation rates on the order of $\beta = 4$ min^{-1} have been observed for *lacA* [188]. Additional studies of RBS activity suggest a wide variation in expression levels induced by RBS alterations, ranging over orders of magnitude [189, 190]. We presume that a reasonable range for the maximal translation initiation rate is from 1 min^{-1} to 10 min^{-1}.

The values of K_M reported by Borkowski *et al.* [187] are reported in arbitrary units. They observe a range of saturation levels from near linear behaviour to near constant saturation [187] from which we can propose a range of K_M values based on the observed free ribosome concentration, which in our model is between 1000 μm^{-3} and 3000 μm^{-3}. We let K_M range between 750 μm^{-3} and 1500 μm^{-3}, with a nominal value of $K_M = 750$ μm^{-3}, which is approaching saturation (and hence the near constant translation efficiency observed by Klumpp and Bremer [147, 2]).

4.3.2 Optimal Experimental Design

To illustrate calibration of the intrinsic parameters of the model, we consider a set of dynamic induction experiments conducted over multiple growth rates, where the experimenter can select (i) the (constant) growth rate, (ii) the time-varying induction profile, and (iii) the sampling schedule. We define an experiment as a set of $N = 3$ sub-experiments, each with a constant growth rate λ (possibly repeated), organized into vector $\boldsymbol{\lambda} = [\lambda^{(1)}, \lambda^{(2)}, \lambda^{(3)}]$. (The exponential growth rates, $\lambda^{(i)}$, are reported as doubling rates $\mu^{(i)}$ (db/hr) in the results for interpretability). In each sub-experiment, the population begins at rest and responds to a dynamic induction signal in the form of a time varying transcription factor copy number, $\boldsymbol{u}(t) = [u^{(1)}(t), u^{(2)}(t), u^{(3)}(t)]$. (Such an input $u^{(i)}(t)$ could be implemented by, e.g., a calibrated induction system [72] or closed loop control of fluorescently tagged TFs [191].) For computational efficiency, we have restricted each $u^{(i)}(t)$ to be piecewise constant, with 6 constant control values delivered for 100 min each, for a total duration of $t_f = 600$ min. We have constrained the maximum value of each $u^{(i)}(t)$ to $u_{max} = 1000$ and the minimum to $u_{min} = 0$. This wide range was selected to ensure it spans the un-saturated range of the promoter well in excess. For each sub-experiment we also select sampling schedules for both mRNA and protein species. A single sampling schedule for a specific species is defined as a list of points, $s_{(species)} = \{p_1, ...p_l, ..., p_L\}$. Each $p_l = (\tau_l, c_l)$ consists of a time

$\tau_l \in [0, 600]$ and a positive integer count c_l of the number of samples taken at that time. We have restricted the design such that $c_l \in \{1, 2, 3\}$. The total number of samples for each species, c_{Tot} is constrained to be no more than 12. The sampling schedule for each sub-experiment, $S^{(i)}$, consists of a schedule for each species $S^{(i)} = \{s^{(i)}_{(rna)}, s^{(i)}_{(prot)}\}$. These are collected into an overall schedule: $\boldsymbol{S} = \{S^{(1)}, S^{(2)}, S^{(3)}\}$.

Optimization over the sampling schedules as defined above involves integer programming, which poses significant computational challenges. We avoid these by employing a relaxation approach [157, 192, 193]. We treat the sampling schedule as a continuous sampling density such that $s^{(i)}_{(species)}$ corresponds to a density $w^{(i)}_{(species)}(t)$. Each sub-experiment has sampling density $W^{(i)}(t) = \{w^{(i)}_{(rna)}(t), w^{(i)}_{(prot)}(t)\}$ and the overall sampling schedule is $\boldsymbol{W}(t) = [W^{(1)}, W^{(2)}, W^{(3)}]$. These sampling densities are restricted to be piecewise constant over 48 equal intervals (each of length 12.5 min) during each sub-experiment. Each sampling density $w^{(i)}_{(species)}(t)$ is upper bounded by $w_{max} = \frac{3}{12.5}$, and the integral of the sampling distribution, $\hat{w}_n^{(species)}$ is bounded by the maximal number of samples, c_{Tot}:

$$\hat{w}^{(i)}_{(species)} = \int_0^{t_f} w^{(i)}_{(species)}(t)\, dt \leq c_{Tot} \tag{4.20}$$

The relaxed schedule must be discretized to arrive at an implementable sampling schedule [157]. In practice, the optimal densities are often bang-bang [193] and so can be easily discretized by the sampling interval length (12.5 min) to recover integer sample counts. To account for non-integer values, we applied the Sum-Up-Rounding strategy, a common heuristic in integer programming, to recover integer solutions that approximate the optimal discrete schedule [192]. This discretization strategy effectively rounds the continuous density while respecting the sampling constraints [192]. After generating the sampling counts, c_l, from the sampling densities, we fixed their associated times τ_l to the centre of each of the 12.5 min intervals.

To characterize a genetic construct with the induction experiments as defined above, we seek an experimental design that can accurately estimate the intrinsic parameter set $\boldsymbol{\theta}$. As stated in section 4.3.1, the parameter K_r proved to be practically unidentifiable under the model specifications described above. Consequently, we fixed it to its nominal value listed in Table 4.1, and took the intrinsic parameter set to be estimated as $\boldsymbol{\theta} = [\alpha\ \beta\ K_t\ K_{rt}\ \delta]$. These parameters characterize the regulated promoter, its regulating transcription factor and the downstream gene sequence, independent of growth context. We denote an estimate of the true parameters as $\hat{\boldsymbol{\theta}}$. For our objective, we seek to minimize the determinant of the covariance matrix $\det\left(\text{cov}(\hat{\boldsymbol{\theta}})\right)$, in what is known as a D-optimal design [22]. The determinant of the covariance matrix is also known as the generalized variance [194]; minimizing it is equivalent to minimizing the volume of the confidence ellipsoid defined by the covariance matrix [105]. Because the model is nonlinear and the true parameter vector is, by definition, unknown, it is not possible to estimate the parameter covariance matrix *a priori*. Instead of estimating $\text{cov}(\hat{\boldsymbol{\theta}})$ directly, we use the Fisher information computed at an initial guess $\boldsymbol{\theta_o}$ as a proxy. In linear models, (or in the limit of large sample sizes or low signal to noise ratio) the inverse of the FIM is asymptotically equivalent to $\text{cov}(\hat{\boldsymbol{\theta}})$ [195]:

$$\mathcal{I}(\boldsymbol{\theta}, t_f)^{-1} \approx \text{cov}(\hat{\boldsymbol{\theta}}) \tag{4.21}$$

In this case minimizing $\det\left(\text{cov}(\hat{\boldsymbol{\theta}})\right)$ is equivalent to maximizing the determinant of the Fisher information matrix, $\det(\mathcal{I})$. In the non-linear finite sample regime in which our experiments are conducted, this relation is admittedly tenuous; the use of an initial guess $\boldsymbol{\theta_o}$ also introduces potential errors. However, the process of iterating the local approximation and successively updating $\boldsymbol{\theta_o}$ after each experiment has been numerically demonstrated to yield convergence to the true parameter set in some cases [103, 8]. We thus define our objective as $\Theta_D(\mathcal{I}) = \det(\mathcal{I})$, which is generally known as the D-optimality score [105]. (Other options such as A, E, and E-modified optimality minimize other properties of the covariance [105, 22, 21].)

To determine the optimal set of experiments, we follow an optimal control-based procedure described in [157, 156, 155]. (This approach to OED was demonstrated by application to chemical and bioprocess models. It has seen limited use in systems biology [9] and has not previously been applied to component characterization in a synthetic biology context.) In this approach the control variables include aspects of the experimental design such as the induction profile, the sampling times and the growth rate. To formulate a control problem we must state the objective, the D-optimal score $\Theta_D(\mathcal{I}) = \det(\mathcal{I})$, in terms of the model dynamics. We restate the model in the following generic form

$$\frac{d\boldsymbol{X}}{dt} = \boldsymbol{F}(\boldsymbol{X}, \boldsymbol{\theta}, \lambda^{(i)}, u^{(i)}(t)) \tag{4.22}$$

Here $\boldsymbol{X} = [X_{rna}, X_{prot}]$ and $\boldsymbol{F}$ is the right hand side of the model expressed in equations (4.1). We determine the local sensitivities of X_{rna} and X_{prot} with respect to each parameter θ_i, by solving the following system of sensitivity equations

$$\frac{d}{dt}\frac{\partial \boldsymbol{X}}{\partial \theta_j} = \frac{\partial \boldsymbol{F}}{\partial \theta_j} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}}\frac{\partial \boldsymbol{X}}{\partial \theta_j} \tag{4.23}$$

Because the parameters are dimensioned, their scalings can vary widely, leading to poor conditioning of the Fisher information matrix and associated computations [155]. To rectify this, we scale the sensitivities by the parameter values (i.e. we use logarithmic sensitivities)

$$\bar{\boldsymbol{X}}_{\theta_j} = \frac{\partial \boldsymbol{X}}{\partial \log(\theta_j)} = \theta_i \frac{\partial \boldsymbol{X}}{\partial \theta_j} \tag{4.24}$$

Applying this change of variables yields

$$\frac{d\bar{\boldsymbol{X}}_{\theta_j}}{dt} = \theta_j \frac{\partial \boldsymbol{F}}{\partial \theta_j} + \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{X}} \bar{\boldsymbol{X}}_{\theta_j} \tag{4.25}$$

The initial conditions for the state variables and their sensitivities are constrained to be at steady state with respect to a zero induction input ($u = 0$):

$$\boldsymbol{X}(t=0) = \boldsymbol{X}_{SS}(\boldsymbol{\theta}, \lambda, u=0) \tag{4.26}$$
$$\bar{\boldsymbol{X}}_{\theta_j}(t=0) = \bar{\boldsymbol{X}}_{\theta_i, SS}(\boldsymbol{\theta}, \lambda, u=0) \tag{4.27}$$

Because the steady state depends on both the growth rate and the initialized parameter values, these initial conditions are not constants. They therefore appear as nonlinear constraints in the optimization problem.

We denote the sampling variance for observations of species by $\sigma^2_{species}$. We assume normally distributed errors with variances equal to 5% of the species quantity:

$$\sigma^2_{rna} = (0.05)X_{rna} \tag{4.28}$$
$$\sigma^2_{prot} = (0.05)X_{prot} \tag{4.29}$$

We further assume no covariance between species measurements. The (scaled) Fisher information matrix, $\mathcal{I}(\boldsymbol{\theta}, t)$ can then be written for the relaxed problem as a differential equation, as per [157]:

$$\frac{d}{dt}\mathcal{I}_{jk}(\boldsymbol{\theta}, t) = \bar{X}_{\theta_j} \boldsymbol{\Omega}(t) \boldsymbol{\Sigma}^{-1}(t) \bar{X}_{\theta_k} \qquad \mathcal{I}_{jk}(\boldsymbol{\theta}, 0) = 0 \tag{4.30}$$

where the matrices $\boldsymbol{\Omega}(t)$ and $\boldsymbol{\Sigma}(t)$ are defined as;

$$\boldsymbol{\Omega}(t) = \begin{bmatrix} w_n^{(rna)}(t) & 0 \\ 0 & w_n^{(prot)}(t) \end{bmatrix}, \quad \boldsymbol{\Sigma}(t) = \begin{bmatrix} \sigma^2_{rna} X_{rna}(t) & 0 \\ 0 & \sigma^2_{prot} X_{prot}(t) \end{bmatrix} \tag{4.31}$$

Here $w_n^{(species)}(t)$ are sampling densities, as defined above. Further details on the integration of the Fisher information over continuous sampling densities can be found in [157, 193]. Because samples are assumed to be independent across time points as well as species, the

Fisher information for the experiment is additive across both. (Here we use the Fisher information for a constant error variance, despite the assumption of non-constant variance implicit in the data generating model. We found this simpler formulation performed equivalently and was more tractable.) Therefore we can express the cumulative Fisher information for a given sub-experiment from time 0 to t_f as

$$\mathcal{I}^{(i)}_{jk}(\boldsymbol{\theta}, t_f) = \int_0^{t_f} \bar{X}^{(j)}_{\theta_i} \boldsymbol{\Omega}^{(i)}(t) \boldsymbol{\Sigma}^{-1}(t) \bar{X}^{(i)}_{\theta_k}\, dt. \tag{4.32}$$

The Fisher information for the total experiment, $\mathcal{I}_{Tot}$, is then the sum over all sub-experiments

$$\mathcal{I}_{Tot} = \sum_{i=1}^{N} \mathcal{I}^{(i)}(\boldsymbol{\theta}, t_f) \tag{4.33}$$

The overall objective (D-optimality score) is then

$$\Theta_D\left(\mathcal{I}_{Tot}\right) = \det\left(\sum_{i=1}^{N} \mathcal{I}^{(i)}\right) \tag{4.34}$$

The matrix is symmetric, so only the diagonal and lower triangular elements need to be computed.

The value of $\Theta_D\left(\mathcal{I}_{Tot}\right)$ can vary over many orders of magnitudes for different experiments, and so to improve numerical accuracy we take the logarithm of the determinant as the objective. Finally, because most optimization packages are minimizers, we invert the sign:

$$\min_{\boldsymbol{u}, \boldsymbol{\lambda}, \boldsymbol{W}} -\ln(\Theta_D\left(\mathcal{I}_{Tot}\right)) \tag{4.35}$$

For numerical stability we compute the determinant using QR factorization (details in Section 2 of the supplement, Appendix A).

In summary, we formulate our OED optimal control problem as

Objective:

$$\min_{\boldsymbol{u},\boldsymbol{\lambda},\boldsymbol{W}} -\ln(\Theta_D(\mathcal{I}_{Tot})) = -\ln\left(\det\left(\sum_{i=1}^{N}\mathcal{I}^{(i)}(\boldsymbol{\theta}, t_f)\right)\right)$$

Controls:

$$\boldsymbol{u} = \left\{u^{(1)}(t), ..., u^{(N)}(t)\right\}$$
$$\boldsymbol{\lambda} = \left\{\lambda^{(1)}, ..., \lambda^{(N)}\right\}$$
$$\boldsymbol{W} = \left\{\hat{w}^{(1)}_{rna}(t), \hat{w}^{(1)}_{prot}(t), ...\hat{w}^{(N)}_{rna}(t), \hat{w}^{(N)}_{prot}(t)\right\}$$

Subject to $(\forall i \in \{1, \ldots, N\})$:

$$\begin{aligned}
&\frac{d\boldsymbol{X}^{(i)}}{dt} = \boldsymbol{F}(\boldsymbol{X}^{(i)}, \boldsymbol{\theta}, \lambda^{(i)}, u^{(i)}(t))\\
&\frac{d\bar{\boldsymbol{X}}^{(i)}_{\theta_j}}{dt} = \theta_j \frac{\partial \boldsymbol{F}(\boldsymbol{X}^{(i)}, \boldsymbol{\theta}, \lambda^{(i)}, u^{(i)}(t))}{\partial \theta_j} + \frac{\partial \boldsymbol{F}(\boldsymbol{X}^{(i)}, \boldsymbol{\theta}, \lambda^{(i)}, u^{(i)}(t))}{\partial \boldsymbol{X}} \bar{\boldsymbol{X}}^{(i)}_{\theta_j}, \qquad \forall \theta_j \in \boldsymbol{\theta}\\
&\frac{d\hat{w}^{(i)}_{sp}}{dt} = w^{(i)}_{sp}(t), \qquad \forall sp \in \{rna, prot\}\\
&\frac{d\mathcal{I}^{(i)}_{jk}}{dt} = \bar{X}^{(i)}_{\theta_j} \boldsymbol{\Omega}^{(i)}(t) \left(\boldsymbol{\Sigma}^{(i)}(t)\right)^{-1} \bar{X}^{(i)}_{\theta_k}, \qquad \forall \theta_j, \theta_k \in \boldsymbol{\theta}
\end{aligned} \tag{4.36}$$

With Constraints:

$$\begin{aligned}
&\boldsymbol{X}^{(i)}(t=0) = \boldsymbol{X}^{(i)}_{SS}(\boldsymbol{\theta}, \lambda^{(i)}, u=0)\\
&\bar{\boldsymbol{X}}^{(i)}_{\theta_j}(t=0) = \bar{\boldsymbol{X}}^{(i)}_{\theta_j, SS}(\boldsymbol{\theta}, \lambda^{(i)}, u=0), \qquad \forall \theta_j \in \boldsymbol{\theta}\\
&\mathcal{I}^{(i)}_{jk}(\boldsymbol{\theta}, 0) = 0 \qquad \forall \theta_j, \theta_k \in \boldsymbol{\theta}\\
&\hat{w}^{(i)}_{sp}(0) = 0, \qquad sp \in \{rna, prot\}\\
&\hat{w}^{(i)}_{sp}(t_f) < c_{Tot}, \qquad sp \in \{rna, prot\}\\
&0 < w^{(i)}_{sp}(t) < w_{max}, \qquad sp \in \{rna, prot\}\\
&0 \le u^{(i)}(t) \le u_{max}\\
&\lambda_{min} < \lambda^{(i)} < \lambda_{max}
\end{aligned}$$

This problem formulation matches that used by Telen *et al.* and Hoang *et al.* [157, 155]. We used a multiple shooting algorithm to solve the optimal control problem [156]. We implemented this algorithm in CasADi, a rapid prototyping optimal control toolbox, using its MATLAB interface [196]. CasADi uses algorithmic differentiation to compute first and second derivatives with respect to both the objective and the constraints. This higher-order information is used for optimization in an interior-point barrier method implemented in the nonlinear programming package IPOPT [197]. Further details of the multiple shooting algorithm and optimization settings can be found in Section 2 of the supplement, Appendix A.

4.4 Results

4.4.1 Comparing Lumped and Physiologically-aware Models

Models of gene expression typically use lumped parameters that confound physiological effects with intrinsic features. Such models are usually calibrated against data collected from cells grown in one particular medium (and so at a single growth rate). To illustrate the consequences of neglecting growth effects, we consider a lumped version of our growth dependent model (equation (4.1))

$$\begin{aligned}\frac{d[X_{rna}]}{dt} &= A\frac{B+Cu}{1+B+(D+C)\,u} - E[X_{rna}]\\ \frac{d[X_{prot}]}{dt} &= F[X_{rna}] - \lambda[X_{prot}],\end{aligned} \tag{4.37}$$

where parameters $A = \frac{\alpha g}{V}$, $B = \frac{P_g K_r}{\eta G}$, $C = \frac{P_a K_{rt}}{(\eta G)^2}$, $D = \frac{K_t}{\eta G}$, $E = \delta\xi$, and $F = \frac{\beta R_f}{V(K_M V + R_f)}$ are treated as growth rate-independent constants. (The one exception is the protein decay rate, which is equal to the exponential growth rate.) Of course, this parameter lumping does not impact model behaviour for the growth rate at which the model is calibrated, but accurate extrapolation to other growth rates cannot be assured. This loss of accuracy is illustrated in Figure 4.2. Panels A and B show predictions of the full growth-dependent model and the lumped model, both calibrated to fast growth rates (3 db/hr). As the growth rate drops, significant deviation in the predicted steady state copy number of both protein and mRNA is observed. In Figures 4.2C and D we see similar deviation for the steady state concentrations of these species. (We note that the relative deviations in concentration are comparatively small. Protein concentration is important for, e.g. metabolic enzymes. In contrast, copy number is more important for DNA binding proteins, which tend to disperse along the DNA rather than the total cell volume [191, 5].) A complementary measure of inaccuracy due to lumping appears in Figure 4.2E, which shows how the lumped parameter values vary with growth rate when the full model is used to determine their values (in comparison to their values calibrated at the reference fast growth rate of 3 db/hr) . The divergence in these parameters sets has a significant effect on the model's dynamic behaviour. Figure 4.2F shows the root mean squared error (RMSE) between the two models' response to the input profile depicted in Figure 4.2G. The difference in dynamic behaviour between the lumped and full model under this dynamic induction, at slow growth rate of 0.5 db/hr, is shown in Figure 4.2G.

4.4.2 Null and Optimal Experimental Designs

We next explore how experimental design can improve the estimation of the intrinsic parameters of the full growth-dependent model (equation (4.1)). We have selected a null experiment, shown in Table 4.2, to represent a reasonable non-optimized design. It consists of a set of logarithmically (base 10) distributed pulses, delivered at evenly spaced growth rates, with evenly spaced samples taken every half-hour (mRNA and protein samples are taken at the same time). We propose this as a sensible first experiment for fitting a dynamic model. We designed the null experiments assuming limited information on the dynamics of the specific gene expression system. They were selected to involve a reasonably comprehensive set of perturbations and measurements. For comparison, we examine three variants of the null experiment, also shown in Table 4.2, each with a perturbation to either the growth rates, induction profile, or sampling rate of the null. The growth variant is identical to the null case, except it is performed over a narrower range of growth rates. The sampling variant differs from the null by redistributing the samples so that their rate is halved but the sampling number at each point is doubled (with the same total number of samples). The induction variant uses linearly (rather than logarithmically) spaced strengths of induction pulses. Shown with each design is its predicted optimality score (the negative log determinant of the scaled Fisher information matrix computed at the true parameter set: equation 4.35). As expected, each variant provides less information than the null (as measured by D-optimality).

In contrast, Figure 4.3 depicts an optimal experiment for our model. This design was generated by using the true nominal parameter vector as the algorithm's initialization parameters. This is an artificial scenario (the true parameters are not known initially, by definition), but it serves to demonstrate the difference between intuitive designs and optimal designs selected by our method. In Figure 4.3, each column describes a sub-experiment, labeled with the corresponding optimal growth rate. The top row, (A–C), depicts the optimal input profiles; the middle row, (D–F), shows the system response in both mRNA and protein copy number. The last row, (G–I), shows the sampling densities

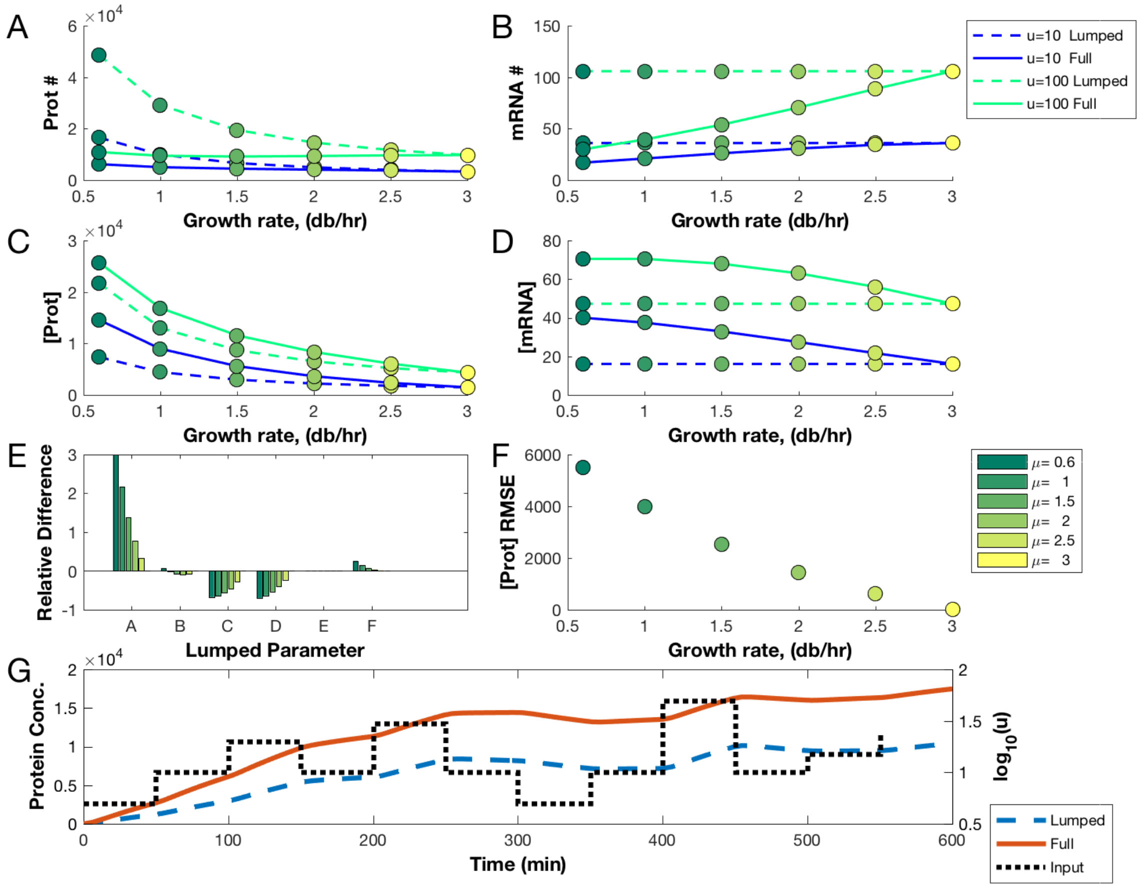
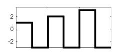
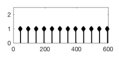
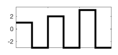
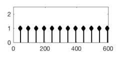
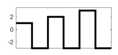
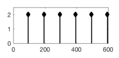
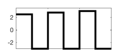
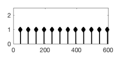


Figure 4.2: **A** and **B**. Steady state protein (A) and mRNA (B) copy number predictions for the lumped and full model across growth rate for two different constant input levels. **C** and **D**. Corresponding protein (C) and mRNA (D) concentrations. **E.** Relative difference in the lumped parameter values, fit at 3 db/hr, compared with the values predicted by the full growth dependent model. **F**. Root mean squared protein concentration error over the dynamic simulation in (G) between the full and lumped model. **G**. Predicted protein concentrations of the lumped and full model over a dynamic simulation with growth rate of 0.5 db/hr.

(in the shaded regions) as well as the results of the Sum-Up-Rounding discretization scheme (depicted by the stem plots, where the height is an integer representing the sample count).

Table 4.2: Null (non-optimal) Experimental Designs

Experiment	Growth Rates (db/hr)	Induction Pattern ($\log_{10}$)	Sampling Schedule	Optimality
Null Experiment	{0.6, 1.8, 3}			-63.8
Growth Variant	{2, 2.5, 3}			-61.5
Sampling Variant	{0.6, 1.8, 3}			-62.6
Induction Variant	{0.6,1.8,3}			-60.5

The optimality score of this design is -69.6. (In this case, the optimal design selected the maximum growth rate twice. We suspect this is due to higher sensitivities at faster growth. More complex models typically demand observations over a wider range of growth rates.)

Comparing the null and optimal designs we see that, while the null experiments have an intuitive appeal because of their wide, even distribution of experimental choices, none achieve a score similar to the optimized design. Further, while null variants exhibit differences in optimality scores, when compared to the optimized experiment these differences are small. This suggests that manually tuning aggregate design measures is less effective than the holistic optimization provided by the OED approach.

4.4.3 Utility of Optimal Designs for Parameter Identification and Prediction

The difference in optimality scores between the null and optimized experiments suggests significant improvements in parameter estimation using the optimized design. However, the statistical theory used to derive the optimality score can only be guaranteed to hold *a priori* for linear models in the limit of small observation variances and large samples [195]. To validate that our optimality scores correspond to improved parameter estimation accuracy, we used simulated experiments to assess the correlation between theoretical and observed variability. To generate simulated data, we simulated the model using the nominal true parameters and added normally distributed observation error with variance equal to 5% of the corresponding species count. We used a multiple shooting approach for parameter estimation; obvious outliers were removed (details in Section 2 of the supplement, Appendix A). Fig. 4.4A shows, on the vertical axis, the logarithm of the generalized variance (determinant of the observed covariance matrix) for the collection of parameter estimates

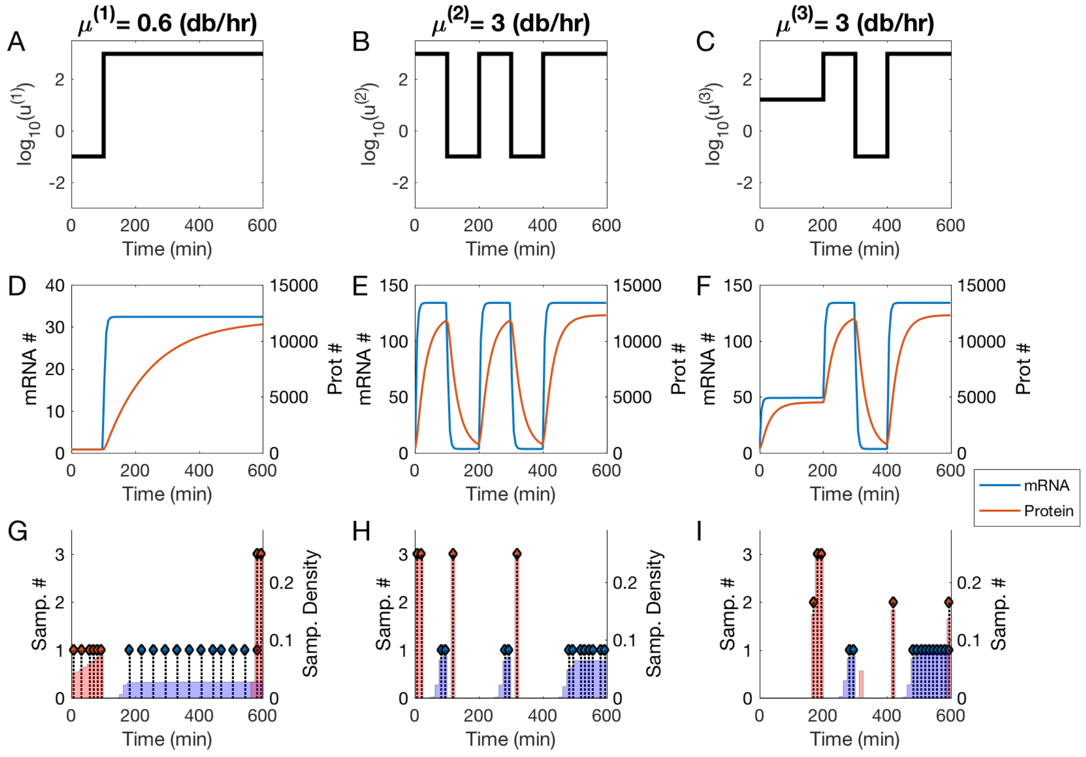


Figure 4.3: An optimal experiment (designed at the true value of $\boldsymbol{\theta}$). Each column depicts one sub-experiment, labelled with optimal doubling rates: $\mu^{(1)}$, $\mu^{(2)}$ and $\mu^{(3)}$ (corresponding to exponential growth rates $\lambda^{(1)}$, $\lambda^{(2)}$ and $\lambda^{(3)}$). The top row (A–C) depicts the optimal inputs, $u^{(1)}(t)$, $u^{(2)}(t)$ and $u^{(3)}(t)$ (on a $\log_{10}$ scale). The middle row (D–F) shows the system response (both mRNA and protein copy number) for each induction profile. The last row shows the sampling densities $w_{rna}^{(i)}$ and $w_{prot}^{(i)}$ (in shaded areas, blue and red, respectively), for both protein and mRNA, as well as the rounded sampling schedule, $s_{rna}^{(i)}$ and $s_{prot}^{(i)}$ (depicted by the stem plot). The optimality score of this experiment was -69.6.

(each corresponding to an experimental design). This measure of fit variability is computed from the collection of parameter fits to independent simulations of the given experiment. From this set of estimates we computed the observed covariance matrix and the generalized variance. The optimality score of the design is shown on the horizontal axis (lower is better), computed at the true parameter value. Fig. 4.4A shows that the optimality score (objective of the OED approach) and the observed parameter covariance (sampled measure of design 'quality') correlate well. In particular, we note that the optimal design achieves both a better optimal score and a smaller generalized parameter variance when compared to the non-optimal designs. This suggests that the objective function (using the homeostatic FIM) is a useful measure of design quality, despite the nonlinearity and heteroskedasticity of the model. We also note that adopting a uniform sampling schedule or reducing the range of growth rates is expected to result in considerably worse performance, as evidenced by comparison between null variants.

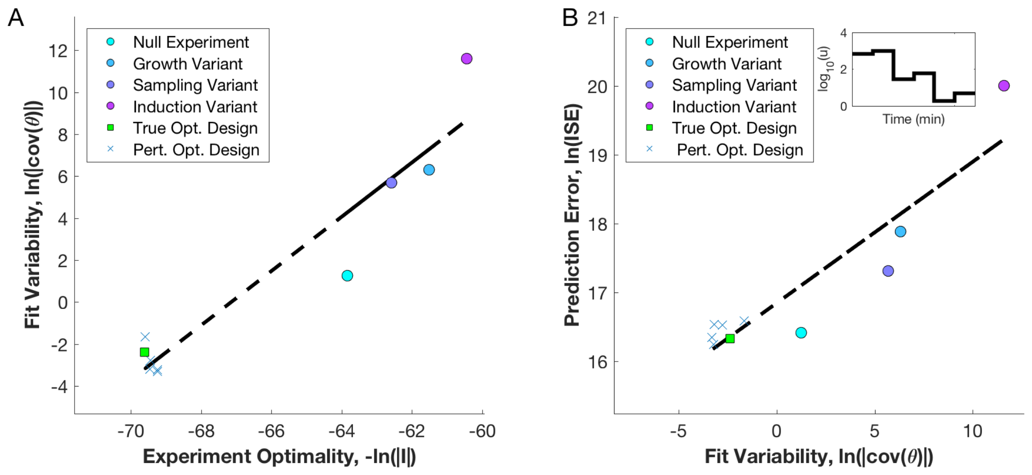


Figure 4.4: **A.** Results from the true optimal (optimal experiment designed at true parameter value) and null experiments (circles): optimality score on the horizontal axis; observed generalized variance of the parameter estimate on the horizontal axis. **B**. The same experiments: generalized parameter variance on the horizontal axis; log integral of the squared error on an out-of-sample prediction experiment (inset) on the vertical axis. Optimal experiments designed with erroneous initial parameter guesses are shown as X's. (Linear trend lines also shown.)

In addition to accuracy of parameter estimates, accurate predictions of the system behaviour for out-of-sample conditions is also important for model-based design. For linear regression models, D-optimal designs that minimize the generalized variance of the parameter estimate are equivalent to designs that minimize the upper bound on prediction variance (General Equivalence Theorem) [40]. This guarantee does not generalize to our dynamic, nonlinear and heteroskedastic model. To verify if prediction accuracy indeed correlates with D-optimality for our model, we ran a second simulation study. For each of the parameter estimates used in calculating the fit variability for the experimental designs, we simulated the model response for a new dynamic experiment. We chose out-of-sample conditions: growth rates and induction levels not used in fitting. We simulated the model with the true nominal parameter values, and computed the integrated squared error (ISE) between the true and fitted values in each case (thus including error at all time points). Figure 4.4B shows the generalized parameter variance along the horizontal and the log ISE on the out-of-sample data set on the vertical axis. The plots shows that the generalized parameter variance correlates reasonably well with prediction accuracy: the optimal design performs better than any of the null designs. This reflects the linear case, in which the D-optimal design sets an upper-bound on the expected prediction variance [40].

So far, we have compared designs in the idealized (and artifical) scenario in which OED is applied to the model parametrized by its true values. We relaxed that assumption by first generating five intrinsic parameter sets from a uniform distribution spanning the intervals specified in the feasible ranges from Table 4.1. We then ran the OED algorithm using models parametrized by these 'perturbed' parameter sets. The optimality scores of these designed experiments, when evaluated against simulated data generated by the true parameter set, are plotted in Figures 4.4A and 4.4B, depicted as X's.

4.5 Discussion

We have proposed a physiological-aware model of gene expression in *E. coli* that accounts for the effects of nutrient limitation on the host physiology. The model is more complex than standard models of gene expression, but this comes with several benefits. The model naturally suggests a partitioning of the parameter set into 1) intrinsic parameters that characterize the genetic component and, 2) a set of empirically derived parameters characterizing the host's physiological state as a function of nutrient-mediated growth rate. The intrinsic parameters can thus be reused across a range of growth conditions. In particular, because the intrinsic parameters can be linked to sequence properties of the component (e.g. promoter affinities, RBS strength, mRNA stability), they could provide insight as to which aspect of a component's DNA sequence could be altered to achieve a desired effect across a variety of contexts. Future work could attempt to link such intrinsic parameters to sequence properties directly. In contrast, the extrinsic parameters can be used to predict the host's context based on the observed growth rate in a range of nutrient conditions.

Accurate estimation of the intrinsic parameter set is more difficult than estimation in context-naive models. We have demonstrated that optimal experimental design can mitigate this challenge by identifying maximally informative experiments. We applied a comprehensive OED platform, optimizing over constant growth rate, time dependent induction profile, and sampling schedule. Many current experimental design approaches in systems biology use general purpose optimization algorithms that scale poorly to such multivariate and non-linear designs [154, 22]. This often leads to non-optimal selection of certain design variables [159]. To achieve computational efficiency in our optimization tasks, we re-interpreted the problem via optimal control, with growth rates, sampling schedules and induction levels as control variables. This optimal control framing of OED problems has so far been underutilized in systems biology; it allows access to the rich tool-set developed by control and process engineers, including direct multiple-shooting and collocation methods [156, 155, 157]. These methods are ideally suited for use with emerging experimental tools such as optogenetic induction systems and automated culture and microfluidic devices [154]. Our numerical results suggest that the multiple shooting algorithm provides an efficient method to optimize experiments, improving both parameter estimation accuracy and prediction accuracy over unoptimized designs.

The optimal experimental design algorithm presented here could be improved in a number of ways. We used the simplest and most tractable form of the Fisher information. More accurate formulations account for the sample variance's parametric dependence [10] and for the constraints imposed by initial conditions [198]. Additionally, our approach provides a design that may be only locally optimal. Current iterative designs are based on the assumption that the iterated algorithm does not become trapped in some local minimum. Past numerical studies of iterative applications of OED have shown consistent convergence to the true parameter set (global optimum), but there is scope for more rigorous investigation [103, 8]. Improved computational efficiency may allow users to address multiple scenarios and larger models, or even implement online experimental design algorithms that can update the design in real-time [199, 200]. Our algorithm was implemented with CasADi, which provides rapid-prototyping capabilities for control problems, algorithmic differentiation and interfaces to powerful non-linear programming solvers like IPOPT [196, 197]. This tool facilitates rapid implementation, but requires some mathematical expertise. Further packaging of OED tools for use by experimentalists would no doubt increase adoption in systems and synthetic biology.

The experiments proposed in this work are demanding, requiring sampling of multiple species and control of inputs over extended time periods. In practice it is not currently possible to precisely control the transcription factor count *in vivo*. It would suffice to map the experimental input (e.g. light, chemical inducer), to the expected average copy number of TF per cell. This could be implemented through precise calibration experiments, achievable with current techniques [72]. Future experiments may be able to implement closed-loop control using real-time measurements to ensure robust tracking of desired inputs. It also may be possible to modify the OED algorithm to account for the effects of input variability (an errors-in model). In this work we have assumed time-series measurements of both mRNA and protein quantities. Emerging experimental tools, combined with assays like RNA-seq [201], will likely enable these complex dynamic experiments in the future [154].

For the model considered here, observing only one species results in a structurally unidentifiable parameter set. Any parameter lumping to alleviate unidentifiability leads to a loss of modularity in characterization. Even with observations of both species, the parameter K_r, which characterizes promoter leak, had to be excluded from the analysis because it was practically unidentifiable over reasonable parameter ranges. In practice it may be possible to ignore promoter leakiness in many cases. However for strong leaks it is important to identify the parameter K_r; fixing it, as we have, may introduce significant bias in other parameter estimates. In addition to these specific identifiability concerns, the reusability of parameter estimates is limited by the use of relative units, which are specific to the measurement instrument they are calibrated on. The methods used in this work should ideally be implemented using absolute units, which will allow for comparison between parameter values calibrated on different instruments and in different labs [127, 126, 202].

Our description of physiological state was restricted to a single case: exponentially growing *E. coli* host cells, with growth rate determined by nutrient quality. Our model formulation was made possible by significant experimental efforts into characterizing precisely this physiological response [2]. Beyond nutrient limitation, several other relevant growth perturbations are of interest to synthetic biologists, including translation-inhibiting antibiotics, gene expression burden and metabolic knockdowns. Phenomenological growth laws and coarse-grained proteome models have been extended to some of these conditions [148, 164, 149]. Recent results suggest that certain metabolic perturbations may effect proteome partitioning (and possibly RNAP and ribosomal fractions) in a manner similar to nutrient limitation [149, 150]. Certain physiological properties, such as the ribosome-to-protein ratio, can also be predictably linked to the growth rate controlled by expression burden or antibiotic dosage [164, 148]. The linear ribosome-to-protein ratios may even generalize across species (Figure S1 of [148]). The details necessary to link these coarse-grained models to the specifics of gene expression, as originally proposed in Klumpp and Hwa [147] (and further elaborated on here) are still lacking. While the nutrient-limited growth theories have taken decades to build, modern techniques can likely accelerate this host characterization process, allowing generalization across strains, growth inhibition conditions and potentially even microbial species.

Extensions of phenomenological growth theories to heterologous protein expression burden and modification of metabolic fluxes could have significant impact on metabolic engineering. These effects are especially relevant to the emerging sub-discipline of dynamic metabolic engineering, where synthetic gene expression circuits dynamically modulate both fluxes and heterologous enzyme expression [203, 204, 205]. Design of such systems is challenging; the engineered regulatory components modulate expression burden and enzymatic activity, which in turn affects the growth rate, broader cell physiology and the regulatory component behaviour itself [206, 207, 208]. An extended growth theory combined with the optimal experimental design algorithm and gene expression models presented here could be valuable for guiding future work in this area.

We have proposed the coupling of physiologically-aware modelling and OED techniques to address limitations in the accuracy and generalizability of component characterization. Current gene expression models often fail to account for the host or environmental state, and are calibrated with poorly constrained parameter estimates using *ad hoc* experimental designs. These shortcomings contribute to the limited use of model-based design in synthetic biology. Poor parameter estimates result in lackluster predictive accuracy, and context-naive model predictions cannot generalize, resulting in recalibration for each new design and circumstance, and thus little advantage of model-based design over trial-and-error approaches. Our model has yet to be validated *in vivo*, but wider use of such coarse-grained, context-aware models and optimal experimental designs will ideally maximize researchers' return on experimental investment, and aid efforts to rationally design gene expression circuits.

Chapter 5

Optimal sample scheduling for dynamic gene expression experiments

Published as "Optimal experimental design for characterizing gene expression: sample scheduling" by Nathan Braniff, Max Reed, and Brian Ingalls in *IFAC-PapersOnLine* 51.19 (2018): 48-51.

5.1 Summary

Recent developments in experimental methods, such as optogenetic induction and microfluidic culture devices, enable precise time-varying control of gene expression. These tools provide new opportunities for dynamic experiments. However the complexity of these experiments poses a challenge to traditional experimental design. In this work, we propose a method for optimal sample scheduling and use simulations to compare it with uniform sampling schedules. We show that optimal scheduling improves the informativeness of results toward parameter estimation and identifiability. In addition, we show that uniform sampling may impose obstacles for the application of experimental design methods for the selection of dynamic inputs.

5.2 Introduction

A number of techniques for precise time-varying control of gene expression have been developed in the last decade, including methods of optogenetic induction and microfluidic delivery of chemical inducers [63]. Accompanying these novel perturbation methods have been increasingly accessible technologies for automating experiments to generate rich time-series measurements of cellular dynamics. Together these tools have enabled precise external control of cellular systems and have enabled the nascent field of 'cybergenetics' [80]. This precise dynamic control unlocks new possibilities for experimentally decoding cellular dynamics.

5.2.1 Model-based design of experiments

Model-based Design of Experiment (MBDOE) methods provide a framework for improving the accuracy of model selection, parameterization and prediction [22].These tools have been used recently to optimize dynamic input profiles that can then be implemented via optogenetics or time-varying chemical induction [10, 9].

However, optimization problems resulting from the MBDOE approach are frequently unwieldy. Optimality criteria are typically conditioned on a wide array of factors, including model structure, nominal parameter values, measurement targets, sampling times, error model, and the nature and target of experimental perturbations. A general solution thus requires solving a hierarchy of interdependent subproblems over the variable experimental design parameters, while integrating over uncertainty in system behaviour and experimental protocols. However, both theoretical and experimental results suggest that even a sub-optimal design can provide significant improvement in the informativeness of experiments [8, 112, 10].

Given the wide array of features that enter into MBDOE analysis, simplifying assumptions must be made. A commonly employed simplification is to assume continuous observation or temporally uniform sampling schedules of the measured species [101, 9, 102, 8]. The specific subproblem of sample time selection has received relatively little attention, despite early evidence of its value [158]. Erguler *et al.* have demonstrated that the information content of samples from certain time intervals in a system's response, such as transient or steady-state, differ significantly from one another [102]. They further show that selecting when to sample is a challenge, as it depends on the qualitative dynamics of the system. Recently, Reuss *et al.* incorporated sample time selection into a full MBDOE framework to select optimal dynamic perturbations in both a theoretical study and an experimental implementation [112, 10]. They first selected optimal time-varying perturbations based on a prescribed uniform sampling schedule and then optimized time points for a chosen perturbation. This approach is robust to uncertainty, but may impose some specific problems on the search for optimal inputs, as we show below.

In this work we propose an efficient method for time point selection for a given dynamic experiment. Our approach is inspired by optimal experimental design in linear statistical models [209]. We further demonstrate that neglecting time point selection when analysing optimality of dynamic perturbation experiments can introduce spurious complications which have been largely overlooked in past experimental design works.

5.3 Methods

We are pursuing the use of MBDOE methods for characterizing the dynamics of light-induced gene expression. We chose a simple mechanistic model of regulated expression to illustrate our approach. Figure 5.1 depicts the system, in which a light-inducible promoter controls expression of a transcription factor (TF) which in turn activates a florescent reporter (GFP).

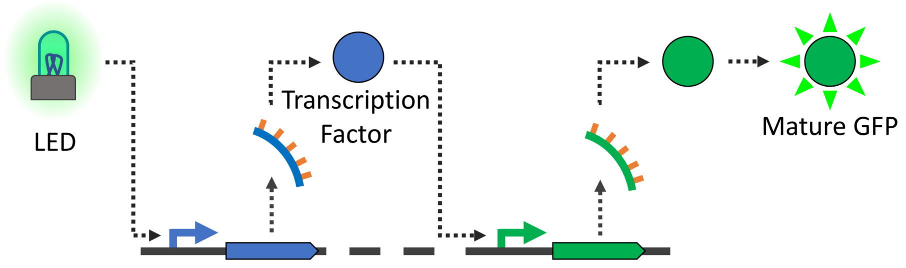


Figure 5.1: Regulated gene expression model: Optogenetic control of a light-inducible promoter activates a transcriptional activator which in turn activates the transcription of a fluorescent protein.

We model transcription and translation of both proteins explicitly. The transcription rate of the light-induced promoter is assumed to be directly controlled via light induction; no model of light mediated signaling is included. This formulation is motivated by previous demonstrations of successful model-based control by [72]. We include an explicit maturation step for the fluorescent protein, the importance of which has been demonstrated [210].

The model is governed by the following equations:

$$\frac{dT_r}{dt} = u(t) - \delta_0 T_r, \qquad (5.1) \qquad \frac{dG_r}{dt} = \alpha \frac{T^n}{K^n + T^n} - \delta_1 G_r, \qquad (5.2)$$

$$\frac{dT}{dt} = T_r - \gamma T, \qquad (5.3) \qquad \frac{dG}{dt} = \beta G_r - (\gamma + \nu) G, \qquad (5.4)$$

$$\frac{dG_m}{dt} = \nu G - \gamma G_m. \qquad (5.5)$$

The state variables are the concentrations of transcription factor mRNA, T_r, transcription factor protein, T, GFP mRNA, G_r, immature (non-fluorescing) GFP protein, G, and mature GFP protein, G_m. The input $u(t)$ is the expression rate of the light-induced transcription factor gene.

Parameters were set to realistic values based on previous dynamic characterization of related systems; $\gamma = 2 \times 10^{-4}$ sec^{-1}, $\delta_0 = 6 \times 10^{-3}$ sec^{-1}, $\delta_1 = 4.8 \times 10^{-3}$ sec^{-1}, $\beta = 0.4$ RiPS,[1] $\alpha = 6.7$ PoPS,[2] $K = 4.17 \times 10^5$ sec, $n = 1.6$, and $\nu = 1.8 \times 10^{-5}$ sec^{-1} [143, 210]. We assume that observations can be made of the mature GFP and both mRNA species.

5.3.1 Measuring experimental optimality

MBDOE techniques begin with specification of an optimality criterion. The quality of parameter estimates can be quantified by an experimentally obtained parameter covariance matrix $\mathbf{\Sigma}_\theta$. A model-derived parameter covariance matrix can therefore be used as a model-based optimality measure. In general the covariance matrix can be approximated by the inverse of the Fisher information matrix (FIM), $\mathbf{I}_\theta$, which in turn can be computed from the sensitivity matrix, $\mathbf{Q}$, and the covariance matrix of the measurements, $\mathbf{\Sigma}_y$ [22]:

$$\mathbf{\Sigma}_\theta = \mathbf{I}_\theta{}^{-1} \qquad \text{where} \qquad \mathbf{I}_\theta = \mathbf{Q}^T \mathbf{\Sigma}_y^{-1} \mathbf{Q} \qquad (5.6)$$

The sensitivity matrix, $\mathbf{Q}$ is defined by

$$\mathbf{Q}_{j,i} = \frac{dy_q(\theta, u(t_k), t_k)}{d\theta_i} \qquad (5.7)$$

where the index j runs over the model observables y_q and the corresponding sampling times times t_k.

[1]RiPS, ribosomes per second

[2]PoPS, polymerases per second

For efficient comparison, a scalar measure of total uncertainty can be derived from the covariance matrix. Here, we use D-optimality, which is the determinant of the Fisher information matrix; maximizing D-optimality is equivalent to minimizing the volume of the confidence region ellipsoid in parameter space. We show minimizing D-optimality also results in improvements of both E-optimality and modified E-optimality (ME-optimality) scores. E-optimality is defined as maximizing the smallest eigenvalue of the FIM and can by thought of as gauging the practical identifiability of the model. ME-optimality corresponds to the condition number of the FIM and can be equated with 'sloppiness' as defined by [101]. These additional measures are subject to ongoing discussion in the community as to what constitutes a well-parameterized model ([104]). For our system, D-optimality correlates with all three measures listed above and it is easily posed as a convex problem (unlike E-optimality which requires some reformulation [209]).

To predict the FIM for a given experiment we need to specify the measurement covariances and nominal parameter values. To simplify the analysis, we assume each measurement is independent with error equal to 5% of the dynamic range of the respective species.

5.3.2 Optimal time point selection

As Erguler *et al.* have previously noted, the informativeness of samples that are uniformly distributed in time is unevenly distributed amongst the observations [102]. Thus experiment protocols can be improved by distributing the samples over the available times and observation variables in an optimal manner. This includes the possibility of repeatedly sampling at some time points.

To arrive at a discrete optimization problem, we specify a fine mesh of potential samples times over the experimental time-frame. Then, for a given input $u(t)$ and initial condition, we can pose the D-optimal time point selection problem as a maximization problem defined over the rows of the sensitivity matrix $\mathbf{Q}$;

$$\underset{\lambda \in Z^+}{\text{maximize}} \qquad \det(\mathbf{Q}^T \mathbf{I} \lambda \mathbf{\Sigma}_y^{-1} \mathbf{Q}) \tag{5.8}$$

$$\text{subject to} \qquad \sum_{l=1}^{N} \lambda_l = N, \tag{5.9}$$

where the total number of observations is fixed as N. The solution to this optimization problem is an integer-valued vector λ that assigns the N total samples to each of the potential time points and observation variables. Solving the integer version of this problem is challenging, but in the limiting case where N is much larger than the number of parameters, the solution to the non-integer relaxation approaches the exact solution and the resulting non-integer vector $\hat{\lambda}$ can be rounded to yield a good approximate solution [209]. The relaxed version is a convex optimization problem because the objective can be written as a convex function of the sum of a set of semi-positive definite matrices. Such convex problems can be solved quickly with general purpose solvers such as CVX, which we used here [211].

5.4 Results

As demonstrated below, an optimized sampling schedule achieves significant improvement in expected parameter estimation accuracy and model identifiability, as well as a decrease in sloppiness, when compared to a uniform schedule. Furthermore, we show that a uniform schedule generates problematic artifacts in optimality measures for dynamic inputs.

5.4.1 Optimal sample scheduling

We consider the response of system (5.1-5.5) to a single input pulse. We initialize the system at steady-state with respect to a constant low input of 0.05 $PoPS$. After 20 minutes the transcription rate is raised to its maximal value of 7.6 $PoPS$ for 20 min before returning to the starting value for the remainder of the experiment. The total experimental time is 8 hours. We compare two sampling schedules. A uniform sampling schedule consists of samples every 15 minutes for GFP mRNA and mature GFP. The transcription factor mRNA relaxes quickly to steady state, so we assign uniform sampling every 2 minutes for only the first hour. This results in a total of $N = 97$ samples taken during the uniform schedule. We solved the optimization scheme (5.8-5.9) with $N = 97$ to arrive at an optimal sampling schedule. Figure 5.2 shows the system response and the set of optimal sampling times. The optimal number of replicates taken at each of these time points ranges between 1 and 12.

Table 5.1 shows the relative improvements provided by the optimal sampling schedule. Improvement in D-optimality, E-optimality and ME-optimality are achieved (despite the fact that this method seeks only to improve the D-optimality criterion). The volume of the eight-dimensional parametric confidence region is reduced by over 97%. Furthermore we determined that only 56 optimally placed samples would be required to achieve an equivalent D-optimality result as the 97 uniform samples.

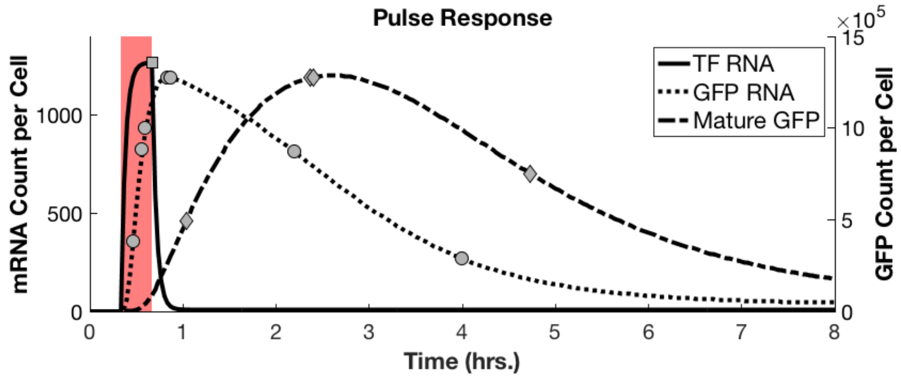


Figure 5.2: System response to a 20 minute pulse of maximal transcription rate (applied over the shaded interval). The optimal sampling points for each species are indicated.

5.4.2 Improved criteria for selection of dynamic inputs

We next demonstrate that uniform sampling schedules can generate spurious effects that may confound optimization over dynamic inputs. Consider the case of system (5.1-5.5) responding to a square wave input rising from $u = 0.76\ PoPS$ to $u = 7.60\ PoPS$ (covering 90% of the system's dynamic range) and back. We suppose the system is observed over 24 hours. We will consider a range of pulse lengths, each with midpoint at 12 hours (Figure 5.3A). In each case we compare two sampling schedules. A uniform sampling schedule is defined by: every 15 minutes for the GFP mRNA and the mature GFP and every two minutes for the TF mRNA. This results in $N = 915$ samples. Alternatively, we solve the optimization scheme (5.8-5.9) using $N = 915$ to find the optimized sampling schedule.

We consider a range of pulse durations from 4 hours to 20 hours. The value of the D-optimality criterion for each sampling approach is shown in 5.3B. As shown, the uniform sampling strategy suggests that D-optimality is maximized at a pulse duration of 12 hours. However, this peak is absent when applying optimal sampling. This spurious effect occurs because uniform sampling seeks to balance the information provided by dwelling at the high and low input values. Optimal sampling (which allows for re-sampling at any time-point) is under no such constraint.

Next consider the case where system (5.1-5.5) is forced by a sinusoidal input with a range $u \in [0, 7.60]\ PoPS$, delivered over a 72 hour experiment (Figure 5.4A). We will sweep over a range of frequencies. For each choice of input we again compare two sampling

Table 5.1: Improvement of Optimal Sampling Relative to Uniform Sampling

Criteria	Increase
D-optimality (square-root[1])	3690%
E-optimality (Identifiability)	272%

Criteria	Decrease
ME-optimality (Sloppiness)	63.6%
Confidence Ellipsoid Volume	97.3%
Parameter Confidence Intervals	from 18.0% to 50.8%
Samples Required[2]	42.3%

[1] D-optimality percentage change is reported after taking the square-root because D-optimality scales with the inverse square of the confidence region volume.

[2] Samples required for the optimal sampling schedule to achieve an equivalent D-optimality result as that of the pre-specified uniform schedule.

approaches: (i) the uniform sampling schedule is used as in the pulse duration experiments above (resulting in $N = 2739$ samples); and (ii) the optimized sampling scheme with $N = 2739$. Figure 5.4B shows the D-optimality value for both sample schedules over a range of input frequencies. Again we see that spurious local maxima are generated by the uniform sampling approach.

5.5 Conclusion

We demonstrated the successful application of a convex optimization approach for time point selection in dynamic experiments. Further we demonstrated that adherence to a uniform sampling schedule can produce spurious extrema when assessing the quality of dynamic inputs in experimental design. These artifacts would likely confound the already challenging optimization task of searching over a high-dimensional input space.

The analysis presented here did not account for parameter and experimental uncertainty. Nevertheless, these results suggest that non-uniformity in sample time distribution is ideal. Further refinement and integration of optimal sampling techniques within the larger MBDOE framework will provide improved experimental protocols and more effi-

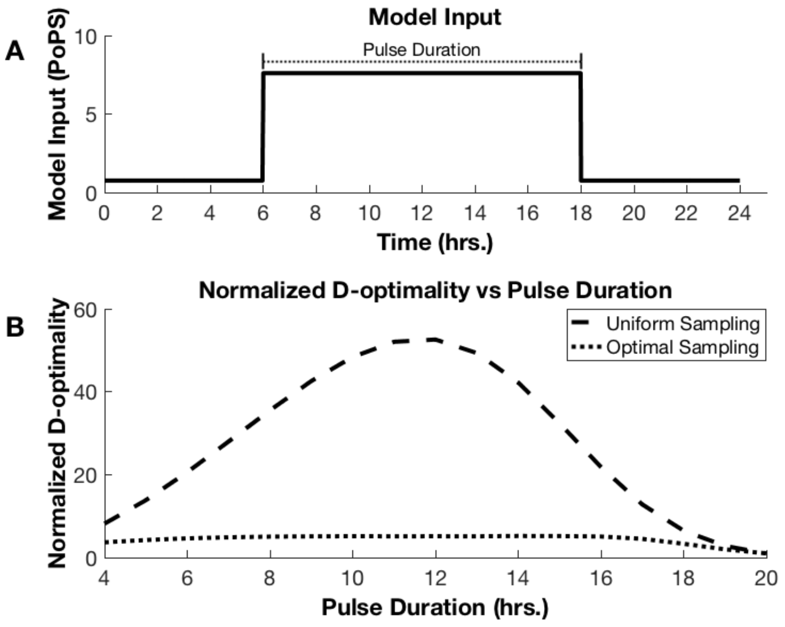


Figure 5.3: A. Square wave input centered at 12 hours. B. Relative D-optimality criteria for optimized and uniform sampling schedules as function of the pulse duration (normalized to their respective minima over the inputs considered).

cient use of emerging dynamic experimental techniques.

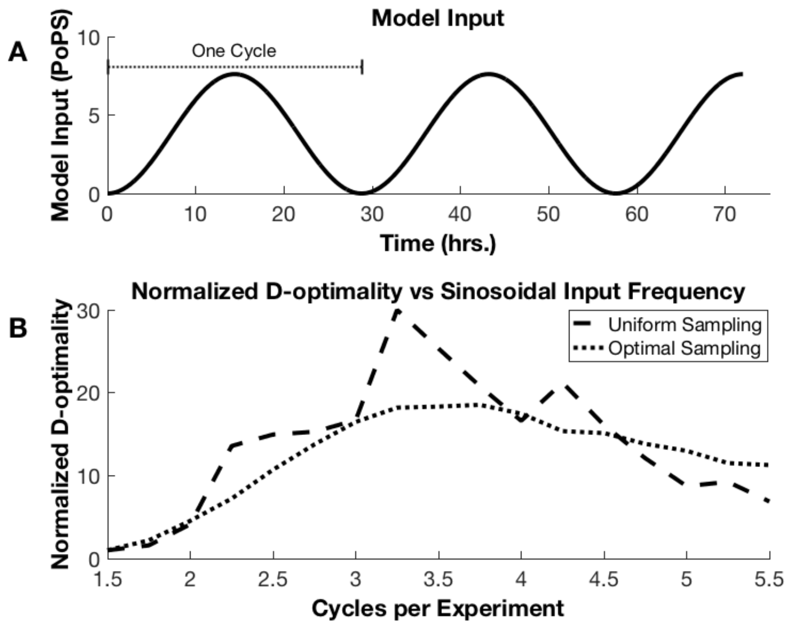


Figure 5.4: A. Sinusoidal input. B. Relative D-optimality criteria for optimized and uniform sampling schedules as function of the pulse duration (normalized to their respective minima over the inputs considered).

Chapter 6

Optimal Experimental Design for a Bistable Gene Regulatory Network

Published as "Optimal Experimental Design for a Bistable Gene Regulatory Network" by Nathan Braniff, Addison Richards, and Brian Ingalls in *IFAC-PapersOnLine* 52.26 (2019): 255-261.

6.1 Summary

Accurate model calibration is essential for model-based design of synthetic gene regulatory networks. Optimal experimental design (OED) techniques can be used to efficiently decrease parameter uncertainty. However, many biological networks of interest exhibit multimodal response functions due to multistability. These models are incompatible with traditional OED approaches that have been developed for models with mono-modal error distributions. In this work we propose an OED approach for a gene expression model that exhibits bistability via a saddle-node bifurcation with respect to an experimental input. We demonstrate construction of an approximate likelihood and derive the corresponding Fisher information across the monostable and bistable regimes. We use the linear noise approximation for the local error model and apply logistic regression to capture the switching probabilities between the stable equilibria. We then use this Fisher information matrix to generate locally optimal experimental designs for this system. This leads to a simple, qualitative approach to optimal experimental design based on experimental detection of bimodality.

6.2 Introduction

Many synthetic biology projects aim to construct designer gene regulatory circuits. These circuits often involve feedback and nonlinear dynamics, including bifurcations and multistability [212]. Model-based design can be a critical tool in the development of such systems. However, biological models have traditionally suffered from sloppy parameter estimates and poor predictive accuracy, limiting the effectiveness of modeling in biological design [102, 101].

Optimal experimental design (OED) provides tools by which synthetic biologists can improve the efficiency of precise model calibration [154, 22]. However, OED approaches have traditionally been applied to monostable systems, which exhibit monomodal error distributions [10] that are often assumed to be Gaussian [103, 8]. Many synthetic regulatory systems of interest exhibit bifurcations and multistability [212], which can result in multimodal error distributions. Limited attention has been given to these more complex error distributions in the OED literature. Examples have been developed for linear models [213], but they are not directly applicable for modelling biological systems.

In this work we develop an OED procedure for characterizing expression of an inducible auto-activating gene that exhibits bistability over a subset of its induction range. We use a stochastic model to account for random jumping between the two equilibria in the multistable regime. This jumping results in a bimodal error model. We use the linear noise approximation (LNA) to form a Gaussian mixture approximating the multimodal likelihood function. We then apply OED to this approximation, using the Fisher information to identify optimal experiments. Specifically, we identify (i) an optimal set of induction input values, and (ii) the fraction of the overall number of steady-state single-cell observations to be taken at each input. The LNA incurs approximation error when applied to multistable systems [214]. However the LNA is analytically differentiable with respect to both inputs and parameters, and it is computationally tractable, both of which are essential for performing optimization over the experimental space.

Past work has investigated parameter estimation procedures for multistable switches [215] suggesting the tracing of hysteresis loops as a means to detect bistable regions. This procedure requires complex experiments with time-varying inputs and careful consideration of the relaxation timescales as part of the experimental design. Our approach uses a more direct strategy for the identification of bistability, focusing exclusively on steady state observations and bimodality detection. Additionally, we provide a simple procedure for avoiding irregularities caused by bifurcations. We find that, for our example system, the optimal experiments follow a consistent qualitative pattern. Further investigation will address the question of whether this pattern generalizes to a simple heuristic procedure for designing optimal experiments for other bistable systems.

6.3 Methods

We employ a simple one-dimensional model to illustrate the method. Equation 6.1 depicts the deterministic rate equation for the mean concentration of the protein product x of an auto-activating gene:

$$\frac{d}{dt}x(t) = \alpha_0 + \alpha\frac{(u + x(t))^n}{K^n + (u + x(t))^n} - x(t). \tag{6.1}$$

Here, u is the concentration of protein provided via a separate, experimentally inducible, source. We assume u and x are functionally identical, but x can be differentiated during measurement, e.g. via a tag or co-expressed reporter. We take the nominal parameter set as $\alpha_0 = 0.5$ $\alpha = 3$ $K^n = 9$, and $n = 3$. The parameters α_0, α, and K are in arbitrary units of concentration; the time-scale is chosen so the rate of degradation is unity. For this parameterization the deterministic system exhibits bistability for inputs u in the range $u \in [u_L, u_R] = [0.07, 0.22]$, and undergoes saddle-node bifurcations at these points, see Figure 6.1B. We seek the inputs that provide maximally informative observations for the purpose of accurately estimating the parameter vector $\theta = [\alpha_0, \alpha, K, n]$. We assume the experiments generate a noisy set of observations, $D = [y_1, ..., y_N]$, taken in long-time equilibrium at input values $U = [u_1, ..., u_N]$. Further we assume the observations y_i are measurements of single-cell concentrations (via a high-throughput instrument such as an automated microscope). Below, we derive the distribution of observations y about the mean x and the resulting likelihood function. We use the likelihood to construct the Fisher information matrix (FIM), and use a scalar function of the FIM as our optimization objective, following classic optimal experimental design theory [59].

The primary challenge with the proposed approach is that gene expression is noisy, and at long-time equilibrium there will inevitably be jumping between equilibria when the system is operating in the bistable regime. Thus the observations y will be bimodally distributed for inputs $u \in [u_L, u_R]$. We can simulate this jumping behaviour by using the deterministic rate law in equation (6.1) to construct a corresponding Master equation model, with reaction propensities as follows:

$$\begin{aligned} \text{Production}: &\quad \Omega \cdot \left(\alpha_o + \alpha \frac{(u + \frac{X}{\Omega})^n}{K + (u + \frac{X}{\Omega})^n} \right), \\ \text{Decay}: &\quad X \end{aligned} \tag{6.2}$$

Here, X is the discrete (random) protein count and Ω is the system size. Simulating this system using the stochastic simulation algorithm (SSA) at a nominal system size of $\Omega = 90$ [216], we see the bimodal distribution for intermediate values of u as shown in Figure 6.1A.

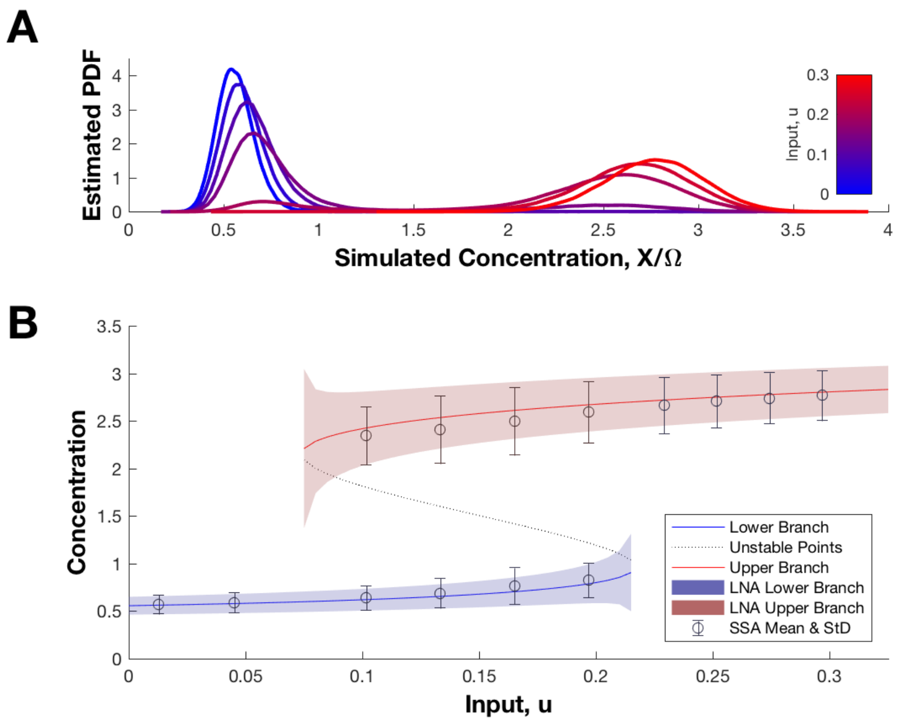


Figure 6.1: (A) An empirical density plot of the simulated species concentration X/Ω for various inputs u, simulated using the SSA with $\Omega = 90$. (B) The bifurcation curve (stable, solid; and unstable, dashed) for the steady state protein concentration x^* from the deterministic model, with the LNA-derived standard error (shaded). The simulated mean values (open circles) and standard errors (error bars) were computed from the SSA simulation.

The bimodal distribution of the observations y cannot be described by normally-distributed homoskedastic or heteroskedastic error models around a single deterministic mean. We instead assume y can be modeled by a Gaussian mixture and propose an approximate log-likelihood for the system of the form:

$$\begin{aligned}\ell(\theta|D,U) = \sum_i \log\{&\rho(u_i)\cdot\varphi_T(y_i|u_i,\theta) \\ &+ [1-\rho(u_i)]\cdot\varphi_B(y_i|u_i,\theta)\}\end{aligned} \tag{6.3}$$

Here $\varphi_T(y_i|u_i,\theta)$ and $\varphi_B(y_i|u_i,\theta)$ are the local distributions of y_i around the top and bottom bifurcation branches, respectively, while ρ represents the probability that a single-cell observation y_i will have originated from $\varphi_T(y_i|u_i,\theta)$ for a given input u_i. Thus $[1-\rho]$ is the probability that the observation originated from $\varphi_B(y_i|u_i,\theta)$

To define $\varphi_T(y_i|u_i,\theta)$ and $\varphi_B(y_i|u_i,\theta)$, we apply Van Kampen's system size expansion (SSE) to the Master equation, using the *ansatz* $X = \Omega x + \sqrt{\Omega}\zeta$, where ζ represents the random fluctuations in species concentration around the deterministic mean concentration x [217]. At first order in the SSE, we recover the deterministic rate equation (6.1) for mean concentration x. At steady state the this rate law yields an equation for the steady state mean x^* in terms of the input u,

$$0 = g(x,u,\theta) = \alpha_0 + \alpha\frac{(u+x)^n}{K^n+(u+x)^n} - x \tag{6.4}$$

which has multiple solutions in the bistable region (i.e. for $u \in [u_L, u_R]$). Over that range we denote the lower root (stable fixed point) as x_B^*, the middle root (unstable saddle point) as x_M^*, and the upper root (stable fixed point) as x_T^*. In the monostable region where $u < u_L$, the only root is x_B^*; for $u > u_R$, only x_T^* appears. Thus x_B^* and x_T^* define the bottom and top branches of the bifurcation curve respectively. The stable upper and lower branches are shown in figure 6.1B along with the means and standard deviations generated from SSA simulations. (To compute the means and standard deviations we used the position relative to the unstable branch to assign each SSA-simulated concentration to either the upper or lower basin of attraction.)

The next order in the SSE yields the linear noise approximation (LNA) [217]. From the LNA we have an expression for the local variance of y about x^*;

$$\sigma^2(x^*,u,\theta) = \frac{\left(a_0 + a\frac{(u+x^*)^n}{K^n+(u+x^*)^n} + x\right)}{-2\Omega\frac{\partial}{\partial x}g(x,u,\theta)|_{x=x^*}}, \tag{6.5}$$

We evaluated this variance function, $\sigma^2(x^*, u, \theta)$, at the roots x_B^* and x_T^* to compute the local variance around each bifurcation branch [218]. Figure 6.1B shows the reasonable agreement between the SSE and SSA for both the means and the standard deviations, at a system size of $\Omega = 90$. Using the means and variance derived from the SSE we define the local distribution $\varphi_B(y_i|u_i, \theta)$ such that $\{y_i|\rho = 0\} \sim \mathcal{N}(x_B^*, \sigma^2(x_B^*, u))$ and $\varphi_T(y_i|u_i, \theta)$ such that $\{y_i|\rho = 1\} \sim \mathcal{N}(x_T^*, \sigma^2(x_T^*, u))$.

To describe the branch probability ρ, we use an alternate framing of the system dynamics as stochastic diffusion within a bistable potential. Focusing on the observation y, define $V(y) = \int_{y_0}^{y} -g(\nu, u, \theta)\, d\nu$ as a potential function for the vector field of the deterministic rate equation. Thus, as a somewhat *ad hoc* approximation, we can write $dy(t) = -\nabla V(y) + \sqrt{2\epsilon} dW$. Here ϵ represents the strength of a constant noise source and dW are increments from the Wiener process. This approximation does not correspond directly to either the linear noise approximation (which has a constant noise source at steady state but a monostable potential about the deterministic mean) or the Kramers-Moyal expansion (which has a nonlinear diffusion term). However, it allows us to use Kramers' expression for the the escape times between two wells in a potential [219];

$$\begin{aligned} \tau_{BT} \approx & \frac{2\pi}{\sqrt{V''(x_B^*)|V''(x_M^*)|}} e^{2\left(|V(x_M^*) - V(x_B^*)|\right)/\epsilon_B}, \\ \tau_{TB} \approx & \frac{2\pi}{\sqrt{V''(x_T^*)|V''(x_M^*)|}} e^{2\left(|V(x_M^*) - V(x_T^*)|\right)/\epsilon_T}. \end{aligned} \tag{6.6}$$

Here τ_{BT} and τ_{TB} are the expected waiting times for the first crossing of the unstable equilibria x_M^* from the bottom-to-top, or top-to-bottom respectively, assuming the system is initialized at equilibrium in the corresponding potential basin. Here ϵ_B and ϵ_T represent the noise strength within each potential. Assuming that the well-to-well transition rates are inversely proportional to these expected waiting times, we can write an equilibrium constant for switching between wells as $K_e = \tau_{BT}/\tau_{TB}$. Recall that we defined ρ as the probability of being in the well around x_T^*. We can then write $\rho = K_e/(1 + K_e)$, i.e.

$$\begin{aligned} \rho(u, \theta) \approx & \operatorname{logistic}\Big(\frac{|V(x_M^*) - V(x_B^*)|}{\epsilon_B} - \frac{|V(x_M^*) - V(x_T^*)|}{\epsilon_T} \\ & + \frac{1}{2}\log\left(V''(x_T^*)\right) - \frac{1}{2}\log\left(V''(x_B^*)\right)\Big), \end{aligned} \tag{6.7}$$

where $\operatorname{logistic}(z) = (1 + e^{-z})^{-1}$. This is essentially the Erying-Kramers (EK) law for reaction rates [219], where the logistic argument involving the potential is a function of the input u. While this *ad hoc* approximation provides some mathematical insight, we cannot hope to generalize it directly to higher dimensional systems for which potentials cannot be readily constructed. Moreover, it relies on the assumption of constant noise that is small relative to the depth of the potential well. While these drawbacks make this expression unsuitable for direct application, the functional form of the EK approximation for $\rho(u, \theta)$ suggests a logistic function may provide a satisfactory approximation for the branch probability. As such we have used the approximation:

$$\rho(u, c_0, c_1) \approx \frac{1}{1 + \exp(-(c_0 + c_1 u))} \tag{6.8}$$

Here, the nonlinear expression involving the potential has been replaced with a linear expression in u. To assess the accuracy of this approach, we used the SSA to numerically determine the probability that the state lies in the basin of attraction of either the top or bottom stable fixed points after long simulation times, see figure 6.2A. As shown, the SSA results are fit well by a logistic function, over a range of system sizes. These logistic curves are specified by parameters c_0, c_1, which we organize into vector $c = [c_0 \;\; c_1]$. These are *auxiliary* parameters, in the sense that, while having some underlying connection to mechanistic parameters θ, they are not the direct target of our experiment, and serve primarily to account for variability that would otherwise make the model intractable.

Substituting the expression for $\rho(u, c)$ into (6.3) results in an appropriate mixture likelihood function on the bistable region. However, it cannot be evaluated as written on the monostable branches, where only one of the two distributions is defined. One approach to resolve this issue would be to modify the behaviour of ρ at the bifurcation points in a piecewise fashion. But that would introduce points of non-differentiability (with respect to θ) in ρ, which would invalidate use of the Fisher information [220]. As a tractable alternative, we define a *guaranteed bistability* region, as follows. We presume that preliminary experiments have identified a region of bimodality. This can be achieved by, e.g. a screening experiment, using a grid or bisection search with a bimodality statistic, such as the the Hartigan Dip Statistic (HDS) [221], shown in Figure 6.2B. (The HDS is a min-max, non-parametric comparison of the observed distribution with the family of unimodal distributions, see [221] for details.) The HDS can detect a subinterval of the bistable region where ρ deviates significantly from 0 or 1. Using an appropriate threshold, this data can be used to define a region of guaranteed bistabilty $u \in [B_L, B_R] \subset [u_L, u_R]$. For values of u outside $[B_L, B_R]$, we set ρ to either 0 or 1, thus approximating the system as monostable. By imposing the conservative bounds B_L and B_R, we have traded precision at the bifurcation points for tractability (and smoothness with respect to the parameters) of the resulting likelihood. To make use of this construction we restrict the permissible parameter vectors θ so that model's bifurcation points fall outside the guaranteed bistable interval:

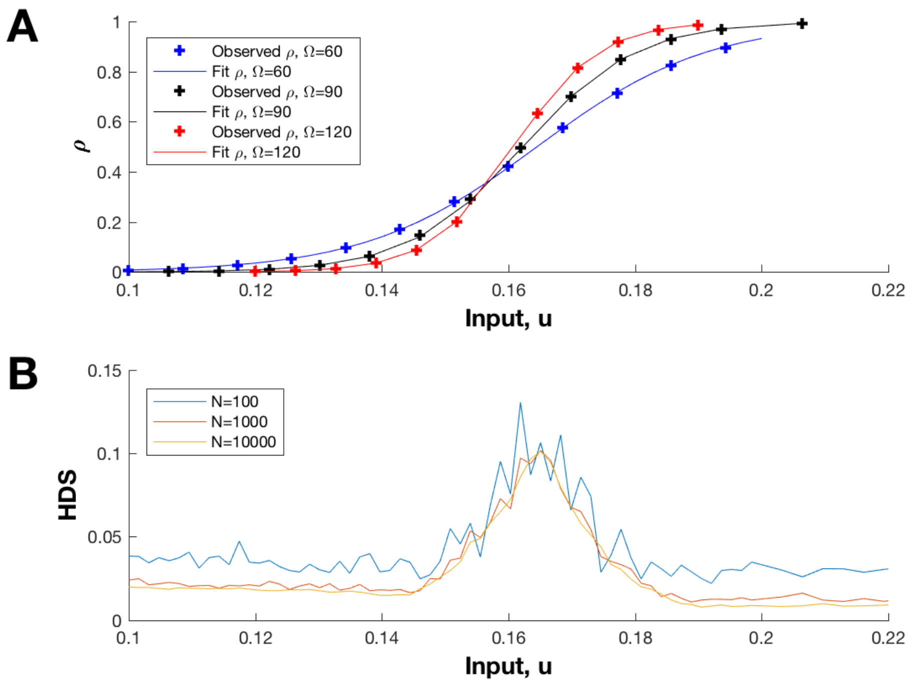


Figure 6.2: (A) Probability that the system is within the basin of attraction of the upper equilibrium point in long-time equilibrium for $\Omega = \{60, 90, 120\}$; SSA (circles), corresponding logistic fit (curve). The fit values are; $c = [-18.1\ 111.7]$ for $\Omega = 90$, $c = [-24.2\ 150.3]$ for $\Omega = 120$ and $c = [-12.1\ 73.8]$ for $\Omega = 60$. (B) The Hartigan dip statistic computed for various inputs using $N = 100$, 1000, or 10000 observations. The algorithm was taken from [1].

$\Theta \in \{\theta | u_L(\theta) < B_L\ \&\ B_R < u_R(\theta)\}$. The likelihood function can then be written as:

$$\begin{aligned}\ell(\theta, c|D, U) = \sum_i \log\{&\rho(u_i, c) \cdot \varphi_T(y_i|u_i, \theta) \\ &+ [1 - \rho(u_i, c)] \cdot \varphi_B(y_i|u_i, \theta)\} \\ \text{Where:}& \\ \rho(u, c) = &\begin{cases} 0 & u \le B_L \\ \text{logistic}(c_0 + c_1 u) & B_L < u < B_R \\ 1 & B_R \le u \end{cases}\end{aligned} \tag{6.9}$$

Note that $\rho(u, c)$ depends only on the auxiliary parameters c while the local distributions φ_B and φ_T depend only on the reaction parameters θ.

We will use the Fisher information to quantify experimental optimality and thus identify the optimal set of inputs. The inverse of the Fisher information matrix is equivalent to the asymptotic covariance of the parameter estimates computed using a maximum likelihood estimator [222]. Calculation of the Fisher information for this nonlinear system relies on a nominal parameterization: $\hat{\theta}$, $\hat{c}$. Consequently our results are only guaranteed to be locally optimal. The initial screening experiments used to define the guaranteed bistability bounds can provide these nominal (preliminary) estimates. The Fisher information for a single sample u_i is;

$$I(u_i|\hat{\theta}, \hat{c}) = E_{y_i}\left[\nabla_{\theta,c}\ell(\theta, c|y_i, u_i)^T \nabla_{\theta,c}\ell(\theta, c|y_i, u_i)\right] \tag{6.10}$$

where y_i is the observed steady state protein concentration resulting from input u_i. For inputs where the response is approximated as monostable (i.e. $u_i < B_L$ or $B_R < u_i$), the Fisher information has an analytic expression. It can be computed as

$$\begin{aligned} I(u_i|\hat{\theta}) =& \frac{\nabla_\theta x^*(u_i, \hat{\theta})^T \nabla_\theta x^*(u_i, \hat{\theta})}{\sigma^2(x^*, u_i, \hat{\theta})} \\ &+ \frac{1}{2}\frac{\left(\nabla_\theta \sigma^2(x^*, u_i, \hat{\theta})^T \nabla_\theta \sigma^2(x^*, u_i, \hat{\theta})\right)}{(\sigma^2(x^*, u_i, \hat{\theta}))^2} \end{aligned} \tag{6.11}$$

Here x^* correspond to x^*_B or x^*_T depending on the input. The sensitivity of the branch mean x^* is

$$\nabla_\theta x^*(u_i, \theta) = \frac{\frac{\partial g(x^*, u_i, \theta)}{\partial \theta}}{\left(1 - \frac{\partial g(x^*, u_i, \theta)}{\partial x^*}\right)}, \tag{6.12}$$

For the variances, we can compute the derivatives analytically as

$$\begin{aligned} \nabla_\theta \sigma^2(x^*, u_i, \theta) =& \frac{\partial \sigma^2(x^*, u_i, \theta)}{\partial \theta} \\ &+ \frac{\partial \sigma^2(x^*, u_i, \theta)}{\partial x^*} \nabla_\theta x^*(x^*, u_i, \theta). \end{aligned} \tag{6.13}$$

To evaluate each of eq. 6.11-6.13, we solve for the roots of $g(x^*, u_i, \theta)$, x^*_B and x^*_T, numerically.

For inputs in the guaranteed bistable region $u \in [B_L, B_R]$, calculation of the Fisher information is more complicated due to the mixed Gaussian response. We computed the sensitivity $\nabla_{\theta,c} l(\theta, c|y_i, u_i)$ using the algorithmic differentiation capabilities of CasADi [196]. The expectation integral in the Fisher information (equation 6.10) is then computed numerically using MATLAB's globally adaptive quadrature function `integral`.

In this work we solve for what is known in the OED literature as the approximate optimal design [223]. Using this approach the optimal design consists of a set of unique inputs $\bar{U} = \{u_1, ..., u_S\}$ and a corresponding set of sampling weights, $\bar{\xi} = \{\xi_1, ..., \xi_S\}$, such that $\sum_i \xi_i = 1$. The weight ξ_i reflects the fraction of the total number of observations that should be made with the given input u_i. (The reason these designs are called approximate is that, with a finite number of total samples in the experiment, one may only approximate the optimal weighting unless the solution happens to have relatively simple fractional weights.) The total Fisher information for the experiment is additive over inputs u (assuming uncorrelated observations), therefore for an approximate design the total FIM is a weighted average, with weights $\bar{\xi} = [\xi_1, ..., \xi_S]$:

$$I_{Tot}(\bar{U}, \bar{\xi}|\hat{\theta}, \hat{c}) = \sum_{i}^{S} \xi_i I(u_i|\hat{\theta}, \hat{c}). \tag{6.14}$$

Next, we select a scalar function of the total information matrix as the OED objective. We use two different optimality measures. The first is the standard D-optimal objective, defined as the determinant of the total Fisher information $|I_{Tot}(\bar{U}|\hat{\theta}, \hat{c})|$ [15]. Maximizing this objective is equivalent to minimizing the volume of the asymptotic confidence ellipsoid for the entire parameter set (θ, c). Because our primary interest is in accurately estimating the reaction parameters θ, we also consider D_s-optimality, which aims to specifically minimize the confidence ellipsoid of θ, without consideration for the accuracy in c [15]. To define D_s-optimality, first recall that the asymptotic covariance matrix, C, is equal to the inverse of the total FIM, I_{Tot}. We block partition both matrices as follows;

$$I_{Tot} = \begin{bmatrix} I_{\theta,\theta} & I_{\theta,c} \\ I_{\theta,c}^T & I_{c,c} \end{bmatrix}, \quad C = \begin{bmatrix} C_{\theta,\theta} & C_{\theta,c} \\ C_{\theta,c}^T & C_{c,c} \end{bmatrix}, \tag{6.15}$$

A D_s-optimal design minimizes the determinant of $C_{\theta,\theta}$, thus minimizing the confidence ellipsoid for θ. This is equivalent to maximizing the determinant [15]:

$$|I_{\theta,\theta} - I_{\theta,c} I_{c,c}^{-1} I_{\theta,c}^T| = \frac{|I_{Tot}|}{|I_{c,c}|}. \tag{6.16}$$

Having defined the D-optimal and D_s-optimal objectives, we can identify optimal experimental designs $\{\xi_i, u_i\}$. To determine the D-optimal designs we use the CVX package over an adaptively refined grid of candidate u_i's [224]. For the D_s-optimal designs we use CasADi and the IPOPT package for optimization, again using an adaptive grid for u [196, 197].

6.4 Results

We begin by investigating the manner in which the optimal designs depend on the nominal values of the logistic parameters c, with θ fixed at the nominal vector $[0.5, 3, \sqrt[3]{9}, 3]$, and with $\Omega = 90$, $B_L = 0.1$ and $B_R = 0.2$. For this analysis, we reparameterize the function ρ such that $c_0 = -c_c c_1$. The new parameter c_c corresponds to the u-value for which $\rho = 0.5$. This allows us to independently set the midpoint and slope of the logistic. Figure 6.3A depicts ρ for c_1 fixed and c_c varying; Figure 6.3B shows ρ for varying c_1 values with c_c constant. The parameter ranges were chosen centered on the fit values of $c = [-18,\ 111]$ at $\Omega = 90$ (shown in figure 6.2A). The corresponding optimal designs are depicted beneath: optimal inputs, u_i (stem locations); optimal weights, ξ_i (stem height). Both the D_s and D optimal designs are shown.

These results reveal that the optimal design in each case involves (i) a modest experimental effort to characterize the monostable response at the extremes of the feasible input range, and (ii) a much heavier sampling within the bistable region. For D_s-optimal experiments, the optimal design clusters the bimodal samples near the mid-point of the logistic function (where $\rho \approx 0.5$). For D-optimal experiments, which aim to characterize the full parameter set (θ, c), optimal samples consistently track a pair of percentiles of ρ above and below the mid-point.

We next investigated the sensitivity of optimal sample placement to the nominal θ vector, by holding the logistic parameters constant at $c = [-18.0,\ 111]$ and randomly selecting ten θ vectors from a normal distribution with mean $[0.5, 3, \sqrt[3]{9}, 3]$ and standard deviations of 10% of the mean values. To ensure all θ were consistent with the assumptions used to derive the likelihood, we rejected parameter sets that did not result in deterministic bistability at $u = B_L - \epsilon$ and $u = B_r + \epsilon$, with $\epsilon = 0.05$, as well as rejecting parameters sets that resulted in bistability at $u = 0$ and $u = 0.3$ (the upper and lower bounds of the feasible range for u). Figure 6.4A depicts stable branches of the ten random bifurcation curves for the chosen θ parameter sets. Figures 6.4B and C depict the corresponding D_s-optimal and D-optimal designs respectively. With the logistic parameters held constant, the optimal samples for both D_s and D optimality show a similar placement and weighting, despite the variability in the model parameterization. This finding suggests that, at this systems size ($\Omega = 90$), the shape of the logistic function (which characterizes the bimodality)

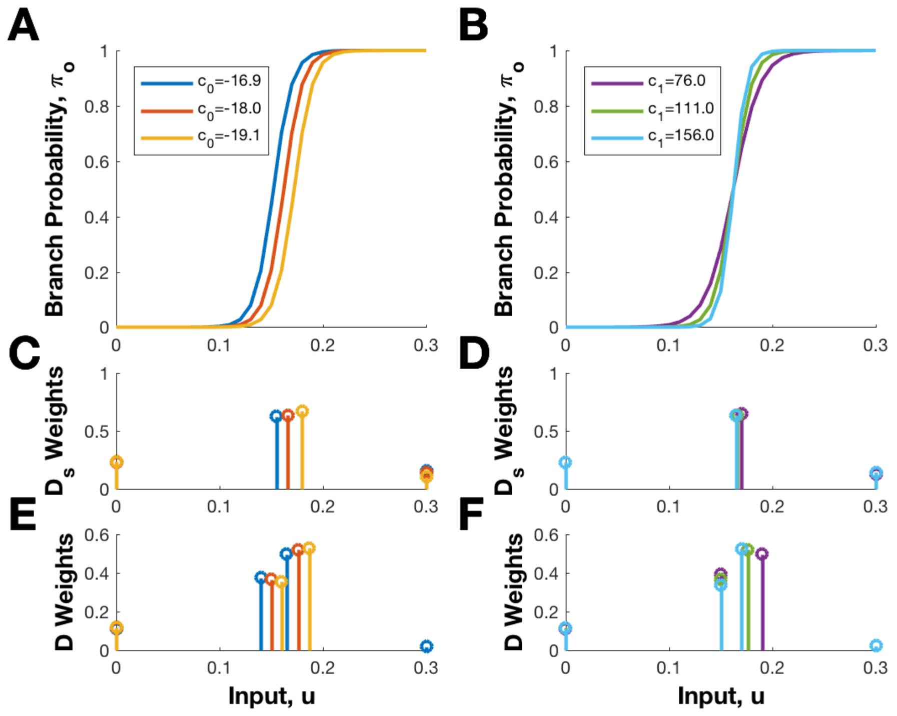


Figure 6.3: (A) Plots of ρ for $c_c = 0.152,\ 0.162,\ 0.172$ ($c_0 = -16.9, -18.0, -19.1$), with $c_1 = 111$. (B) Plots of ρ for $c_1 = 76,\ 111,\ 156$ with $c_c = 0.162$ ($c_0 = -12.33,\ -18.00,\ -25.30$). (C, D) Corresponding D_s-optimal designs. (E, F) Corresponding D-optimal designs. Input values u_i correspond to stem locations. Weights ξ_i correspond to stem heights. Note the short stems at both ends of each feasible input range.

has a strong effect on the optimal sample placement. These also implies that preference between the D or D_s objectives will depend predominantly on the experimental plans. If the experimenter will only perform a single experiment, the D_s design yields an optimal estimate for the primary parameters θ. However, if the experimenter plans to perform a series of iterated experiments, refining their parameter estimates and re-optimizing their experimental design after each iteration, the D-optimal design will yield a sequence of improved estimates for both θ and c. As the optimal design appears to depend largely on the values of c, the D-optimal objective is ideal for this case.

Referring to Figures 6.3 and 6.4, we note that the D_s-optimal designs tended to sample from single mid-point near the middle of the bistable region at a percentile near $\rho = 0.5$. Likewise the D-optimal designs appear to select a pair of optimal inputs that map to

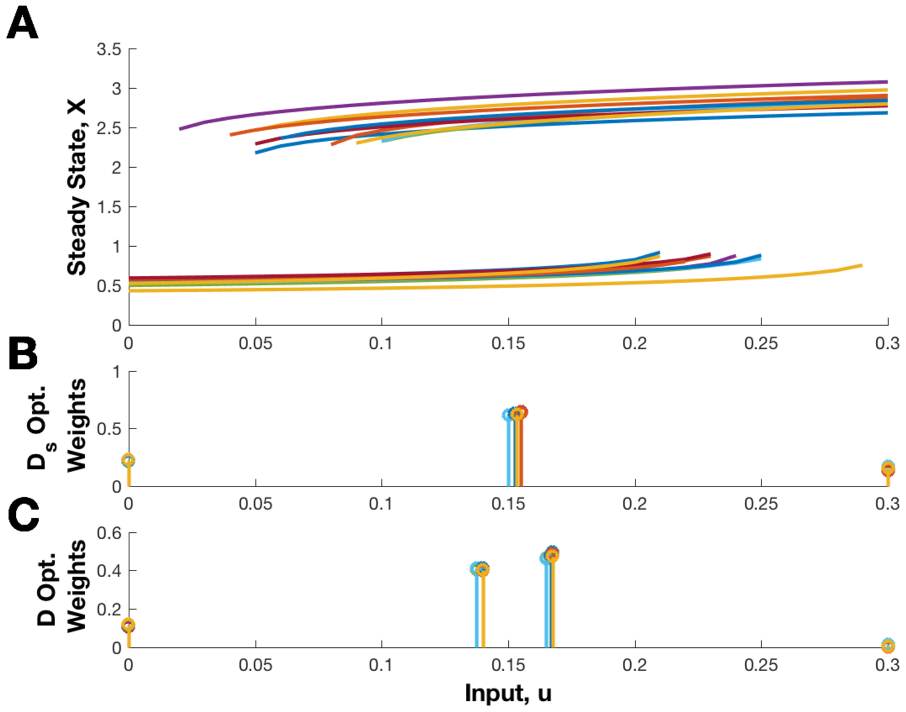


Figure 6.4: (A) Stable branches of the bifurcation curves generated from ten random parameter sets θ. (B) D_s-optimal sample weights and inputs for the chosen θ parameters sets, with logistic parameters c held constant. (C) Corresponding D-optimal designs.

relatively consistent and symmetric percentiles on the ρ logistic curve. To further assess this trend, we generated 500 fully randomized parameters sets. Vectors θ was chosen as above; values of c_1 were selected from a uniform distribution over [91 141], while c_c was selected from a uniform distribution over [0.125 0.175]. This ensured the mid-point $\rho = 0.5$ did not fall too close to the boundaries of the guaranteed bistable region $B_L = 0.1$ and $B_R = 0.2$. Figure 6.5 shows histograms of the D_s and D-optimal sampling locations in the bistable region across all of the randomly generated parameter sets, where the optimal inputs u have been mapped against percentiles of the logistic ρ. The figure shows clear trends: the D_s-optimal input, for use when applying a single round of OED, lies near the 60th percentile of ρ; the D-optimal inputs, which are suited to an iterative OED application, occur near the 25th and 85th percentiles of ρ. Referring to Figure 6.2, we note that the 60th percentile of ρ is approximately where the HDS peaks, while the 25th and 85th percentiles occur where the HDS rises above the baseline. These results suggest

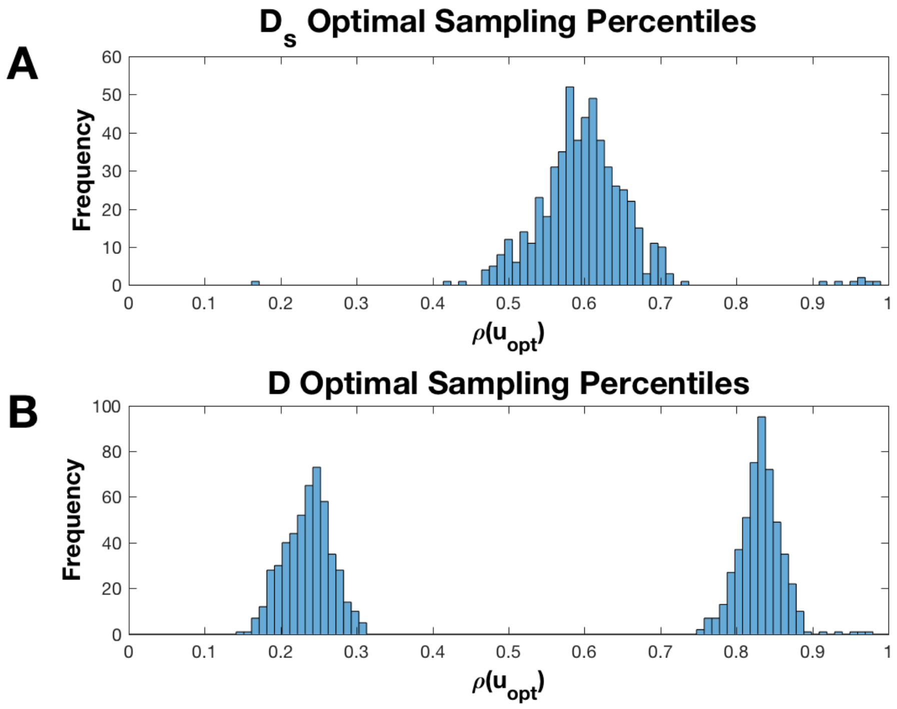


Figure 6.5: (A) Histogram for 500 random parameters sets; shown are the frequency of D_s-optimal input values that fall within the bistable region, plotted according to the percentiles of the logistic ρ to which they correspond. (B) Corresponding histogram for D-optimal inputs.

a simple heuristic for performing OED in this bistable system by using the HDS to do a preliminary search for these locations. After these critical values have been located, the experimenter can use the existing data to provide initial estimates of $\hat{\theta}$ and $\hat{c}$ and perform a D or D_s optimal design. For a crude approximation of the optimal design, the logistic could be fit using only the peak and bounds of the elevated HDS signal to approximate the percentile locations. These critical HDS locations could also be used to select the inputs u directly. The sampling weights ξ could then be approximated based on an average of weights across many θ values, as shown in Figure 6.4. As shown in Figure 6.4B and C, θ appears to have a limited effect on the optimal design once the logistic is known, with the majority of samples being taken in the biomodal input range.

6.5 Conclusion

In this work we applied OED to an inducible and auto-activating gene expression motif that exhibits bistability. We used a stochastic model to derive an approximate likelihood function, based on the LNA, and a logistic approximation for stochastic switching between stable points motivated by Eyring-Kramers' law. We defined a likelihood to avoid irregularities caused by the bifurcations. We then used D_s and D-optimality criteria to determine optimal inputs and sampling proportions for steady state experiments. We showed that, for the given nominal parameter set and system sizes, the logistic switching function strongly influences the optimal input placement and that optimal inputs roughly correspond to certain percentiles on the logistic curve. Our results suggest that bimodality statistics, like the HDS, may provide a convenient method to perform optimal experiments in the absence of accurate initial estimates for the model parameters. These results can be further investigated by additional simulation studies, to examine a wider range of system sizes and parameters. Additional tests can incorporate maximum likelihood fitting to SSA-generated data. We chose the somewhat artificial one-dimensional system used here because it provided a simple, tractable example, allowing us to more thoroughly illustrate the connection between optimal designs and bistability. Extending our analysis to higher-dimensional systems will reveal whether the simple design heuristic we discovered may be representative of a more general property of OED for bistable systems. An obvious next step is to extend our analysis to a more realistic two-dimensional toggle switch, a direction which we are currently pursuing. Another possible line of inquiry would involve using model and dimensionality reduction techniques to map higher dimensional systems into a one or two dimensional observation space. These reduction techniques may be easier to apply at steady state, rather than in a dynamic context, which is an attractive feature of targeting steady state experiments. Finding an appropriate model reduction procedure would ideally make it straightforward to validate and apply our heuristic design rule to a much broader class of systems.

Chapter 7

A Software Package for Optimal Experimental Design in Systems Biology

7.1 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 [154]. 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 [154]. 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 [225, 226]. 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 [154]. 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.

7.1.1 Past Works

There has been a growing interest in experimental design methods in systems biology (i.e. see works cited in [105, 22, 154]), including the previous works contained in this thesis [159, 227, 228]. 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 [9, 10], 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 [229], chemical engineering [21], control theory [44], pharmacokinetics/dynamics (PKPD) [230] 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 [231, 232]. R in particular provides access to a wide range of experimental design tools (see the DOE task view page at [231]). 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) [233] and PFIM 3.0 from Bazzoli et al. (available in R) [234] among others thoroughly compared in later work by Nyberg et al. [230]. 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 [235], however this was released in 1993 in the, now defunct, XLisp-Stat language. However, Bayesian design has experienced some renewed interest in recent years; of special interest is the `aceBayes` package by Overstall and Woods [236], 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 [237]. 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.

7.1.2 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 [238]. 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 [239]. 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 [240]. 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 [196]. AD tools have matured significantly in recent years, with higher-level tools emerging that are easier to use [196]. The NLOED package has been built on top of CasADi, an AD equipped software tool for rapid prototyping of optimal control problems [196]. 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 [197]. 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 2 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.

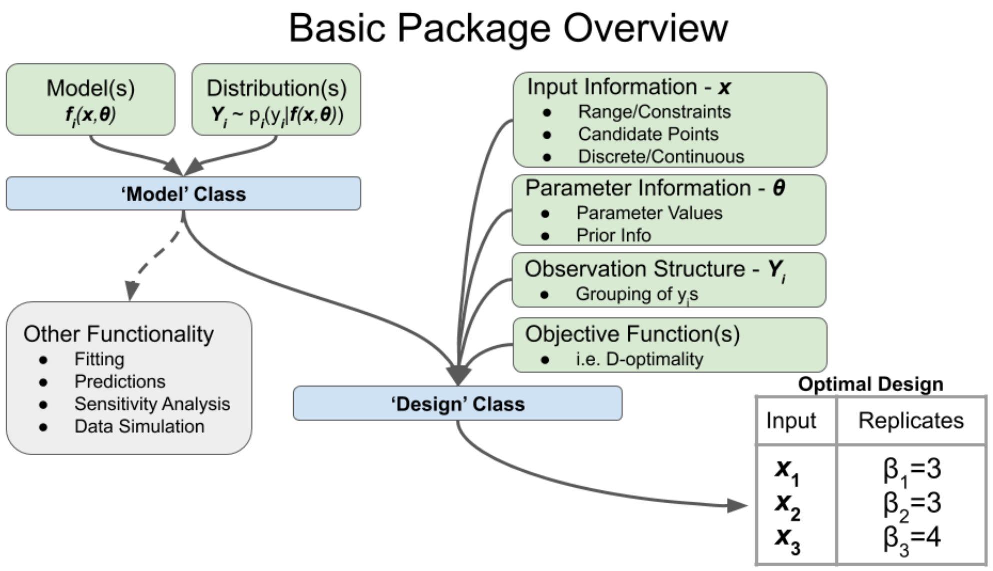
7.2 Workflow and Definitions

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.



Basic Package Overview
Model(s) $f_i(\boldsymbol{x},\boldsymbol{\theta})$
Distribution(s) $Y_i \sim p_i(y_i|\boldsymbol{f}(\boldsymbol{x},\boldsymbol{\theta}))$
'Model' Class
Other Functionality
Fitting
Predictions
Sensitivity Analysis
Data Simulation
Input Information - $\boldsymbol{x}$
Range/Constraints
Candidate Points
Discrete/Continuous
Parameter Information - $\boldsymbol{\theta}$
Parameter Values
Prior Info
Observation Structure - Y_i
Grouping of y_is
Objective Function(s)
i.e. D-optimality
'Design' Class
Optimal Design
Input
Replicates
$\boldsymbol{x}_1$ $\beta_1=3$
$\boldsymbol{x}_2$ $\beta_2=3$
$\boldsymbol{x}_3$ $\beta_3=4$

Figure 7.1: A diagram depicting the archetypal NLOED workflow including model creation using the `Model` class, and design creation using the `Design` class. Here $\boldsymbol{x}$ is the vector of model inputs describing the experimental conditions, $\boldsymbol{y}$ are the observation variables, $\boldsymbol{\theta}$ are the unknown model parameters, $\boldsymbol{f}(.)$ 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. 7.1: 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.

7.2.1 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 2, which is briefly reviewed here. The model connects the random observation variables, Y_i, with model input vector, $\boldsymbol{x}$, which quantifies the experimental conditions. Note that when considering the model declaration the user can ignore the j sub-scripting on the input vectors, $\boldsymbol{x}_j$, 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 $\boldsymbol{x}_j$, $\boldsymbol{\eta}_{i,j}$, $Y_{i,j}$ 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 $\boldsymbol{x}$ and the unknown parameter vector $\boldsymbol{\theta}$. Therefore for each possible observation, Y_i, the connection between input and observation is mediated by two components: 1) the conditional observation distribution, $p_i(y_i|\boldsymbol{\eta}_i)$ and 2) the deterministic model function $\boldsymbol{\eta}_i = \boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$. These two components are combined to create the overall model:

$$Y_i \sim p_i(y_i|\boldsymbol{\eta}_i{=}\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})). \tag{7.1}$$

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, $\boldsymbol{f}_i(.)$, maps the experimental conditions $\boldsymbol{x}$ to the sampling statistics, $\boldsymbol{\eta}_i$, of the observation distribution $p_i(.)$. The function $\boldsymbol{f}_i(.)$ 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 $\boldsymbol{f}_i(.)$ 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, $\boldsymbol{\theta}$, and of the model inputs, $\boldsymbol{x}$, as well as the list of observation variables,

$Y_i \in \boldsymbol{Y}$. The user must also indicate the distribution type of $p_i(.)$ for each observation variable, Y_i, and they must provide the model function, $\boldsymbol{f}_i(.)$ using CasADi's symbolic tools. The distribution, $p_i(.)$, and the function $\boldsymbol{f}_i(.)$ for each Y_i are passed as a list so the user can add as many observations as is desired.

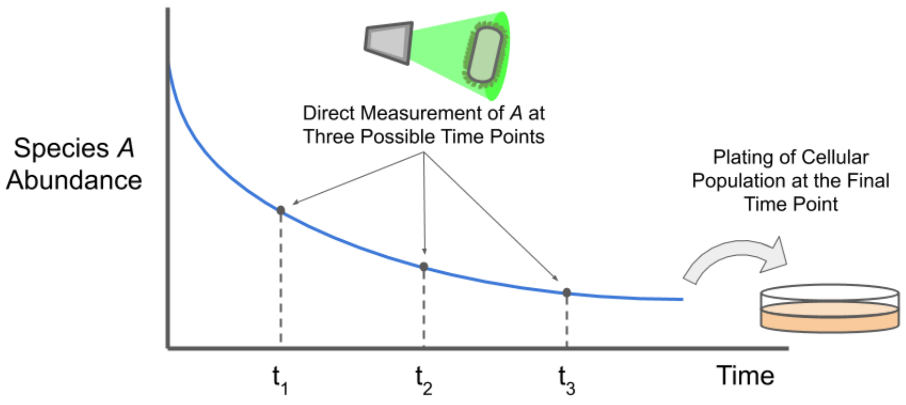

Direct Measurement of A at Three Possible Time Points
Species A Abundance
Plating of Cellular Population at the Final Time Point
t1
t2
t3
Time

Figure 7.2: 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 7.2. 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

$$\mu(t) = \alpha x_1 e^{-\gamma t}, \tag{7.2}$$

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

$$\lambda(t) = \frac{\nu}{\frac{\kappa e^{\gamma t}}{\alpha x_1} + 1}. \tag{7.3}$$

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

$$\begin{aligned}
&\boldsymbol{x} = [x_1], && \boldsymbol{\theta} = [\alpha, \gamma, \nu, \kappa], && \boldsymbol{Y} = [Y_1, Y_2, Y_3, Y_4] \\
&p_1(.) = \text{Normal}, && \boldsymbol{\eta}_1 = [\mu(t{=}1), \sigma^2], && \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\alpha x_1 e^{-\gamma}, 1\right] \\
&p_2(.) = \text{Normal}, && \boldsymbol{\eta}_2 = [\mu(t{=}2), \sigma^2], && \boldsymbol{f}_2(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\alpha x_1 e^{-2\gamma}, 1\right] \\
&p_3(.) = \text{Normal}, && \boldsymbol{\eta}_3 = [\mu(t{=}3)\sigma^2], && \boldsymbol{f}_3(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\alpha x_1 e^{-3\gamma}, 1\right] \\
&p_4(.) = \text{Poisson}, && \boldsymbol{\eta}_4 = [\lambda(t{=}4)], && \boldsymbol{f}_4(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\frac{\nu}{\frac{\kappa e^{4\gamma}}{\alpha x_1} + 1}\right].
\end{aligned} \tag{7.4}$$

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

$$\begin{aligned}
&\boldsymbol{x} = [x_1, x_2], && \boldsymbol{\theta} = [\alpha, \gamma, \nu, \kappa], && \boldsymbol{Y} = [Y_1, Y_2] \\
&p_1(.) = \text{Normal}, && \boldsymbol{\eta}_1 = [\mu(t{=}x_2), \sigma^2], && \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\alpha x_1 e^{-\gamma x_2}, 1\right] \\
&p_2(.) = \text{Poisson}, && \boldsymbol{\eta}_2 = [\lambda(t{=}4)], && \boldsymbol{f}_2(\boldsymbol{x}, \boldsymbol{\theta}) = \left[\frac{\nu}{\frac{\kappa e^{4\gamma}}{\alpha x_1} + 1}\right].
\end{aligned} \tag{7.5}$$

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.

7.2.2 Design Definition

Designs in NLOED generally follow the design definition outlined in Chapter 2, which is summarized here for convenience. Designs in NLOED must specify two main pieces of information, 1) the set of input conditions (support points) $\boldsymbol{x}_j \in \mathcal{X}$ 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 $\boldsymbol{x}_j$. These two properties are common to most design formalisms [15, 16]. NLOED uses two different types of designs in its workflow; *relaxed designs* and*exact designs* (see Chapter 2 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 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 [15, 16]. 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 [15, 16].

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

Table 7.1: 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 $\boldsymbol{x}_j$	Observation Variable Y_i	Sample Weight $\xi_{i,j}$	Sample Number $\beta_{i,j}$
$\boldsymbol{x}_1 = 0$	Y_1	0.1	2
$\boldsymbol{x}_2 = 10$	Y_1	0.2	4
$\boldsymbol{x}_2 = 10$	Y_3	0.1	2
$\boldsymbol{x}_2 = 10$	Y_4	0.3	6
$\boldsymbol{x}_3 = 5$	Y_4	0.3	6

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, $\boldsymbol{x_j}$, 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, $\boldsymbol{x}_j$, 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 7.1, 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: $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$, however five input points are shown because $\boldsymbol{x}_2$ 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.

7.3 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 7.3 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

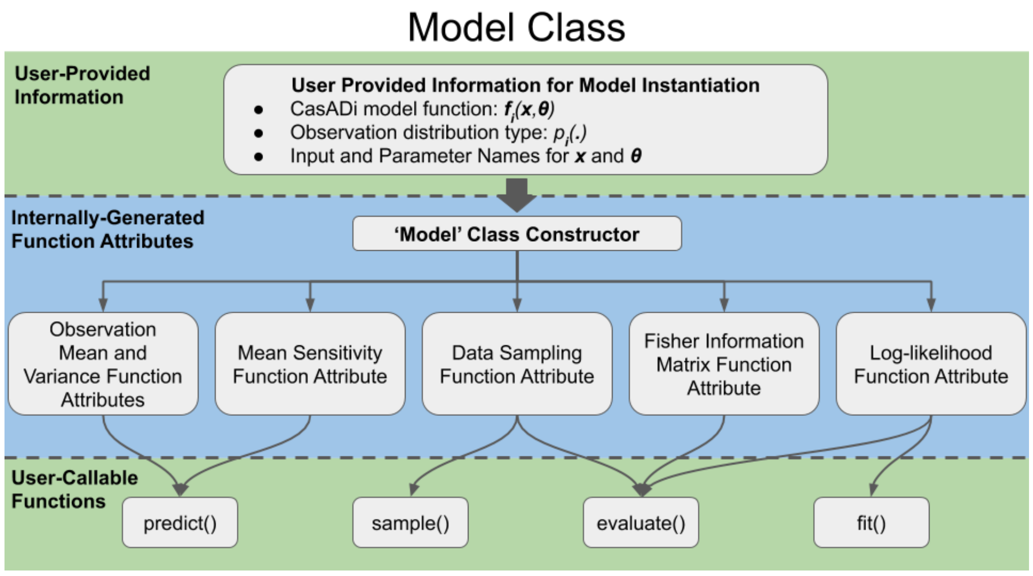

Model Class
User-Provided Information
User Provided Information for Model Instantiation
CasADi model function: $f_i(x,\theta)$
Observation distribution type: $p_i(.)$
Input and Parameter Names for x and θ
Internally-Generated Function Attributes
'Model' Class Constructor
Observation Mean and Variance 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 7.3: A diagram depicting the user-provided arguments, internal function attributes and user-callable functions of the NLOED `Model` class.

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 $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$, 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.

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 7.3 and are shown in the blue region in the middle of the figure. During instantiation, the `Model` class uses the CasADi function, $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$, 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 7.3 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.

7.3.1 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, $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$, mapping the inputs, $\boldsymbol{x}$, and parameters, $\boldsymbol{\theta}$, to the sampling statistics, $\boldsymbol{\eta}_i$, of the observation variable, Y_i. Listing 1 demonstrates this process for a simple two-input, two-output, four-parameter model: 7.6,

$$\begin{aligned} \mu_{Y_1} &= \theta_0 + \theta_1 x_1 + \theta_3 x_1 x_2, \quad \sigma^2_{Y_1} = 0.1, && Y_1 \sim \text{Normal}(\mu_{Y_1}, \sigma^2_{Y_1}), \\ \lambda_{Y_2} &= \exp(\theta_0 + \theta_2 x_2 + \theta_3 x_1 x_2), && Y_2 \sim \text{Poisson}(\lambda_{Y_2}). \end{aligned} \tag{7.6}$$

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

```
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.

the input vector, $\boldsymbol{x}$, as variable `x` and the parameter vector, $\boldsymbol{\theta}$, 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, $\boldsymbol{\eta}_1$, 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 $\boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{\theta})$. Likewise in line 15, the sampling statistic, $\boldsymbol{\eta}_2$, for observation variable

Y_2 is defined as `eta_y2` and in line 16 a CasADi function `func_y2` corresponding to model function $\boldsymbol{f}_2(\boldsymbol{x}, \boldsymbol{\theta})$ is also created. From the perspective of NLOED, the CasADi functions implementing $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$ 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 7.5.

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. $\boldsymbol{\eta}_i = \boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$), 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 `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.

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

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

7.3.2 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. $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$). 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 $\boldsymbol{\eta}_i = \boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$ 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, $\boldsymbol{\eta}_i$, 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 $\boldsymbol{\eta}_i$ 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 $p_i(y_i|\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta}))$. Sampling from this distribution requires the user-passed CasADi function, (i.e $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta})$), 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 2, with the indices indicated there, such that

$$l_i(y_i|\boldsymbol{x},\boldsymbol{\theta}) = \log\left[p_i(y_i|\boldsymbol{\eta}_i{=}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta}))\right]. \tag{7.7}$$

Using the type of distribution for $p_i(.)$ and the user passed CasADi function for $\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta})$, 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 2 for a full description. The Fisher information matrix for an individual input-observation pair is defined as

$$I_i(\boldsymbol{x},\boldsymbol{\theta}) = E_{y_i}[\nabla_{\boldsymbol{\theta}} l_i(\boldsymbol{\theta};y_i,\boldsymbol{x})\cdot\nabla_{\boldsymbol{\theta}} l_i(\boldsymbol{\theta};y_i,\boldsymbol{x})^T]. \tag{7.8}$$

The individual matrices, $I_i(\boldsymbol{x},\boldsymbol{\theta})$, 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 2, to separate the FIM computation into a sensitivity vector and a distribution-specific elemental matrix such that

$$I_i(\boldsymbol{x},\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta})\ \Psi(\boldsymbol{\eta}_i{=}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta}))\ \nabla_{\boldsymbol{\theta}}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta})^T. \tag{7.9}$$

Here $\nabla_{\boldsymbol{\theta}}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta})$ is the parametric sensitivity of the sampling statistics $\boldsymbol{\eta}_i$. The sensitivity vector can be computed using CasADi's automatic differentiation functionality applied to the user-passed model functions, $\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta})$. The elemental matrix, $\Psi(\boldsymbol{\eta}_i{=}\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta}))$, is specific to each distribution [16], and can be computed algebraically from the sampling statistics, $\boldsymbol{\eta}_i$. 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.

7.3.3 User-callable `Model` Functions

After instantiating a `Model` instance, the user will often follow the basic workflow outlined in Figure 7.1 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.

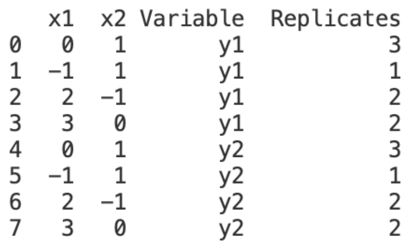
The `evaluate()` function The NLOED package exists to optimize experimental designs for various models and objectives. In many workflows, it is reasonable to assume the user may wish to evaluate the performance of many designs for a given model using some common quantitative metrics. While the `Design` 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 2). 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 [16]. 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 2 for details) [57]. 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 [57], 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 contained in dataframes which all follow the same format with specific naming conventions; an example of which is shown in Figure 7.4. 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 7.6. 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



```
   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 7.4: An example of a dataframe containing an experimental design.

shown in equation 7.6; 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.

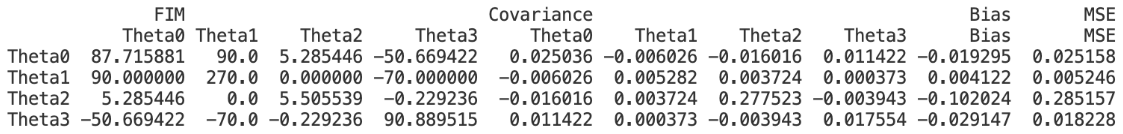
```python
#define a design
design = pd.DataFrame({ 'x1':[0,-1,2,3]*2,
                        'x2':[1,1,-1,0]*2,
                        'Variable':['y1']*4 + ['y2']*4,
                        'Replicates':[3,1,2,2]*2})
#set nominal parameter values
param = [0.1, 2, 0.4, 1.3]
#declare specific options
eval_opts={'Method':'MonteCarlo',
           'FIM':True,
           'Covariance':True,
           'Bias':True,
           'MSE':True,
           'SampleNumber':100}
#call the evaluate() function
eval_info = model_object.evaluate(design, param, eval_opts)
#print the resulting evaluation
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 7.5: 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 7.5. 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 [50].

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 2 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 $p_i(y_i|\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta}))$ for each replicate specified in a given design, and at a given nominal parameter vector.

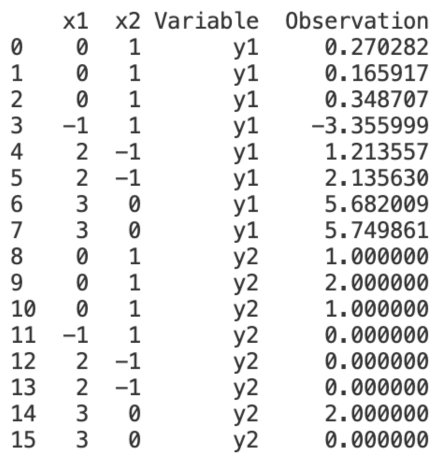
```
#define a design
design = pd.DataFrame({ 'x1':[0,-1,2,3]*2,
                        'x2':[1,1,-1,0]*2,
                        'Variable':['y1']*4 + ['y2']*4,
                        'Replicates':[3,1,2,2]*2})
#set nominal parameter values
param = [0.1, 2, 0.4, 1.3]
#call the sample() function
dataset = 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 7.4). 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 7.4. 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 7.6. 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 7.6. 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 7.6. 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 7.6: 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* [16]. 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 7.6 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 7.6. 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 2 for further discussion of maximum likelihood),

$$l_{Tot}(\boldsymbol{\theta}; \boldsymbol{y}_{\mathcal{D}}, \mathcal{D}) = \sum_{j}^{N} \sum_{i}^{M} \sum_{k}^{\beta_{i,j}} l_i(y_{i,j}^{(k)}; \boldsymbol{x}_j, \boldsymbol{\theta}). \tag{7.10}$$

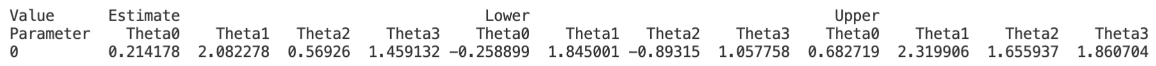
As the objective is a symbolic expression, when it is passed to IPOPT via the CasADi interface, any required derivative information for the interior-points algorithm is automatically generated. This ensures the maximum likelihood fitting implemented in `fit()` occurs rapidly in a few iterations for most 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 7.6. 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 [241]. 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 7.5). As maximum likelihood estimates are invariant under parameter transformation, any required re-parameterization should not effect the resulting fits [49]. 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 7.7. 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 7.7: 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 [241, 49]. Profile likelihood confidence intervals are based on asymptotic properties of the likelihood ratio, defined as $-2L_{Tot}(\boldsymbol{\theta}_o)/L_{Tot}(\hat{\boldsymbol{\theta}})$ [49]. Using properties of the likelihood ratio, it can be shown that

$$-2[l_{Tot}(\boldsymbol{\theta}_o) - l_{Tot}(\hat{\boldsymbol{\theta}})] \leq \chi^2_{p,(1-\alpha)}. \tag{7.11}$$

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 $\boldsymbol{\theta}$. 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 $1-\alpha\%$ [50, 242]. 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 [241, 49] 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 $\bar{\boldsymbol{\theta}}(\theta_i)$. As the value, θ_i, of the profiled dimension is adjusted away from the MLE the conditional estimate $\bar{\boldsymbol{\theta}}(\theta_i)$ traces out a curve in parameter space, this is known as the *profile trace* [242]. The likelihood ratio,

$$-2[l_{Tot}(\bar{\boldsymbol{\theta}}(\theta_i)) - l_{Tot}(\hat{\boldsymbol{\theta}})], \tag{7.12}$$

between the MLE, $\hat{\boldsymbol{\theta}}$, and the conditional vector, $\bar{\boldsymbol{\theta}}(\theta_i)$, 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, $l_{Tot}(\bar{\boldsymbol{\theta}}(\theta_i))$, for each value of θ_i is known as the *likelihood profile* [105]; 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 7.8. 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 7.8, should be asymptotically quadratic; significant deviations from this trend can indicate

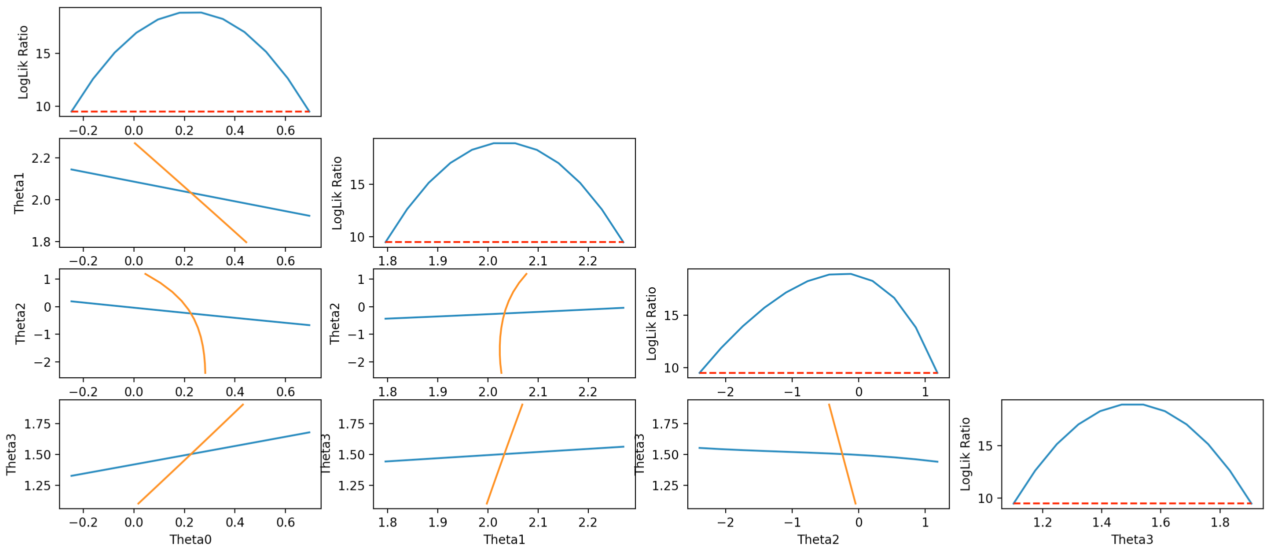


Figure 7.8: Example of the `fit()` functions graphical output of likelihood profiles and 2D trace projections generated when the `Confidence` option is set to `Profiles`.

poorly identified parameters [49]. Specifically, a blunt peak to the profile can indicate that the current data is insufficient to constraint the parameter values [105]. In addition, the projections of the profile traces for each parameter should be asymptotically 'X' shaped, with the MLE estimate at the intersection point [242]. 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 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 [242]. 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) [242]. 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 $\bar{\boldsymbol{\theta}}(\theta_i, \theta_j)$, 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,

$$-2[l_{Tot}(\bar{\boldsymbol{\theta}}(\theta_i, \theta_j)) - l_{Tot}(\hat{\boldsymbol{\theta}})] = \chi^2_{p,(1-\alpha)}, \tag{7.13}$$

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 7.9. By computing contour projections for each

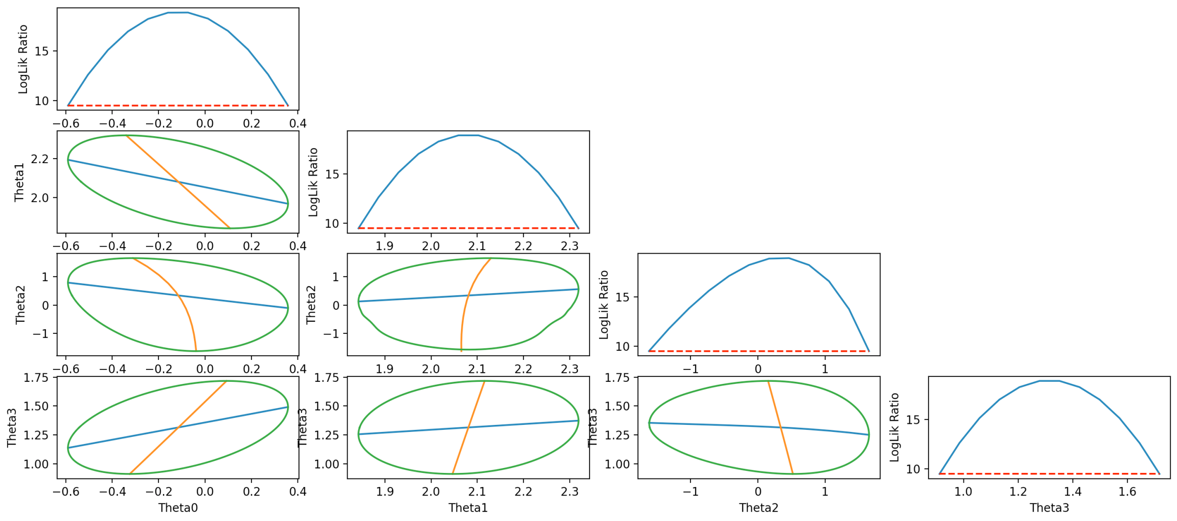


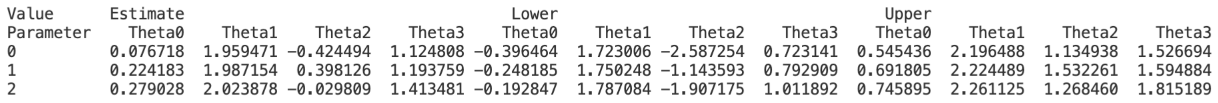
Figure 7.9: Example of the `fit()` function's graphical output of likelihood contour projections generated when the `Confidence` option is set to `Contours`. The contour projections are shown along with the profiles and trace projections previously shown in figure 7.8.

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.

```python
#create three datasets
data1, data2, data3 = dataset = model_object.sample(design, param,
    design_replicates=3)
#combine datasets into a single list
datasets = [data1, data2, data3]
# set up specific options for fitting
fit_opts={'Confidence':'Intervals'}
#call the fitting procedure
fit_info = model_object.fit(datasets,options=fit_opts)
#print the fitting information
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 7.10: 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 7.10 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:

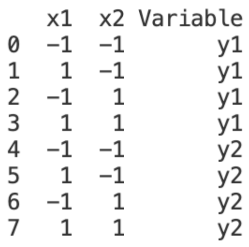
```
np.cov(fit_info['Estimate'].to_numpy().T)
```

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 [49] or through Monte Carlo simulation. In addition, the `predict()` function can also generate parametric sensitivities for the mean response.

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 7.11 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`.



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

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

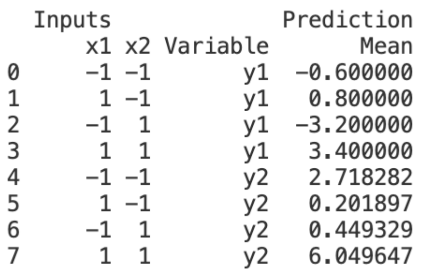
Listing 7: Example of a simple call to the `predict` function from the `Model` class.

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 7.11, 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.

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.



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

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 7.12. 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.

```python
#define a covariance matrix for the parameters
cov_mat = np.diag(params*0.05)
#specify desired option values
predict_opts = {'Method':'MonteCarlo',
                'PredictionInterval':True,
                'ObservationInterval':True,
                'Sensitivity':True}
#generate the predictions
predictions = model_object.predict(input_frame, params,
↪  cov_mat,predict_opts)
#display the predicted outputs
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

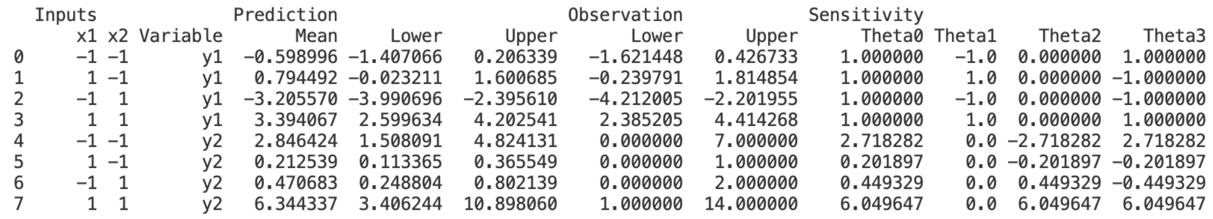
$$\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta}) = \int_y y_i p(y_i|\boldsymbol{\eta}_i{=}\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta}))dy_i. \tag{7.14}$$

Let it be emphasized that $\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta})$ is a deterministic function of $\boldsymbol{x}$ and $\boldsymbol{\theta}$. For all of the parametric distributions used in NLOED, $\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta})$ can be computed algebraically from the statistics $\boldsymbol{\eta}_i$. However, when uncertainty in the parameter vector, $\boldsymbol{\theta}$, is assumed, the parametric uncertainty propagates to the mean response, $\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta})$. The distribution of $\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta})$ under the parameter uncertainty then depends on the prior distribution, $p(\boldsymbol{\theta})$. 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, $\hat{y}_i(\boldsymbol{x}, \boldsymbol{\theta}_o)$, of the true parameter vector, $\boldsymbol{\theta}_o$, 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:

$$p(y_i|\boldsymbol{x}) = \int_{\boldsymbol{\theta}} p_i(y_i|f_i(\boldsymbol{x}, \boldsymbol{\theta}))p(\boldsymbol{\theta})d\theta. \tag{7.15}$$

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

An example of the returned `predictions` dataframe from Listing 8, containing both types of intervals and sensitivity data, is shown in Figure 7.13. 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, $E_{\boldsymbol{\theta}}(\hat{y}_i)$, and the mean observation, $E_{y,\boldsymbol{\theta}}(y_i)$, should be the same and so only the predicted mean is returned (i.e. $E_{\boldsymbol{\theta}}(\hat{y}_i)$). 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 7.13: 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, $\boldsymbol{\eta}_i$, 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, $p_i(y_i|\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{\theta}))$. The `Delta` method propagates parameter uncertainty using local parametric sensitivities and a normal approximations for the prediction and observation intervals [49]. 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, $p(\boldsymbol{\theta})$, and the observation distribution, $p_i(y_i|\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{\theta}))$, 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.

7.4 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.

Figure 7.14 outlines the basic process for creating a `Design` object. Here, as in Figure 7.3, 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

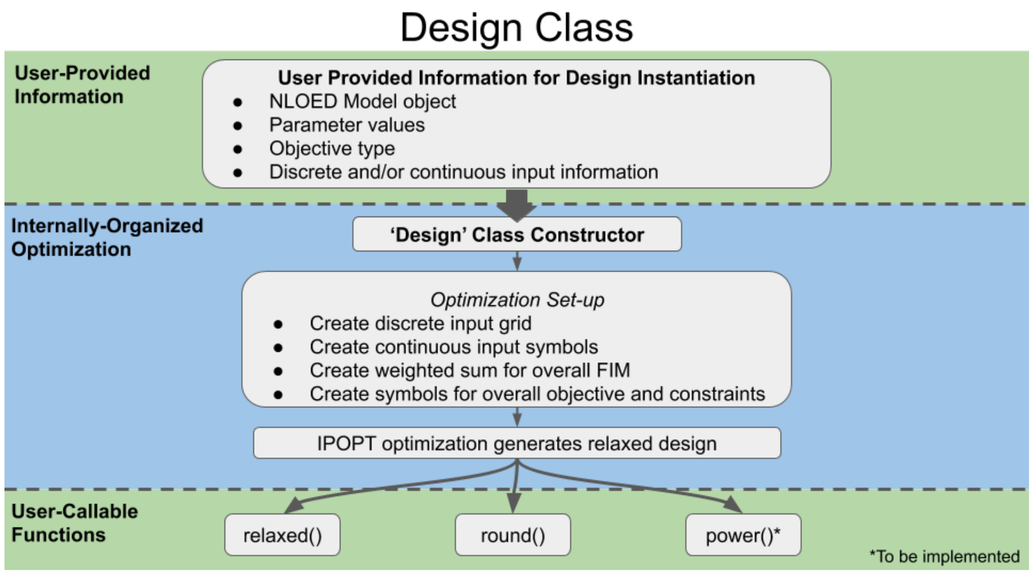

Design Class
User-Provided Information
User Provided Information for Design Instantiation
NLOED Model object
Parameter values
Objective type
Discrete and/or continuous input information
Internally-Organized Optimization
'Design' Class Constructor
Optimization Set-up
Create discrete input grid
Create 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 7.14: Caption text

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 7.14. 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 7.14. 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 7.14, including sample code and call structures where appropriate.

7.4.1 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.

```
#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:

```
#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:

```
#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.

```python
#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.

```python
#discrete input argument creation
discrete_dict = {'Inputs':['x1','x2','x3','x4'],
                 'Bounds':[(-3,3),(-3,3),(-3,3),(-3,3)],
                 'NumPoints:10}
#declare initial design
init_design = pd.DataFrame({'x1':[-2,-1,1,2]*4,
                            'x2':[2,-2,-1,1]*4,
                            'x3':[1,2-2,-1]*4,
                            'x4':[-1,1,2,-2]*4,
                            'Variable':['y1']*4+['y2']*4,
                            'replicates':[3]*8})
#create the fixed design dictionary
fixed_dict = {'Weight':0.25,'Design':init_design}
#declare parameters
param = [2, 3.3, 0.5, 1]
#observation grouping
observ_group_lst = [['y1','y2']]
#call Design constructor
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.

7.4.2 `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 2 for details)

$$I_{Tot}(\mathcal{D},\bar{\boldsymbol{\theta}}) = \sum_{j}^{N}\sum_{i}^{M}\beta_{i,j}I_i(\boldsymbol{x}_j,\bar{\boldsymbol{\theta}}). \tag{7.16}$$

Here there are M observation variables, Y_i, indexed by i, and N support points $\boldsymbol{x}_j$, indexed by j. The support of the design consists of the set of all unique input vectors such that $\boldsymbol{x}_j \in \mathcal{X}$. The FIM at a given input point, $\boldsymbol{x}_j$, for the observation variable Y_i is $\mathcal{I}_j(\boldsymbol{x}_i,\bar{\boldsymbol{\theta}})$. The vector $\bar{\boldsymbol{\theta}}$ is the nominal parameter vector at which the design is optimized. Each replicate allocation, $\beta_{i,j}$, corresponds to a support point $\boldsymbol{x}_j$ 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, $\mathcal{I}_{Tot}(\mathcal{D},\bar{\boldsymbol{\theta}})$, 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

$$I_{Tot}(\mathcal{D}_R,\bar{\boldsymbol{\theta}}) = \sum_{j}^{N}\sum_{i}^{M}\xi_{i,j}I_i(\boldsymbol{x}_j,\bar{\boldsymbol{\theta}}). \tag{7.17}$$

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 $\mathcal{I}_i(\boldsymbol{x}_j,\bar{\boldsymbol{\theta}})$ 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 $\boldsymbol{x}_j$ as the symbolic inputs. The total FIM is then used to generated a CasADi symbol for the overall optimization objective symbol, $\Psi(I_{Tot}(\mathcal{D}_R, \bar{\boldsymbol{\theta}}))$. The resulting objective symbol, $\Psi(I_{Tot}(\mathcal{D}_R, \bar{\boldsymbol{\theta}}))$, effectively encodes the entire optimization problem in a CasADi symbolic structure, linking the objective symbol to symbols for each of the optimization variable symbols; $\boldsymbol{x}_j$ 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.

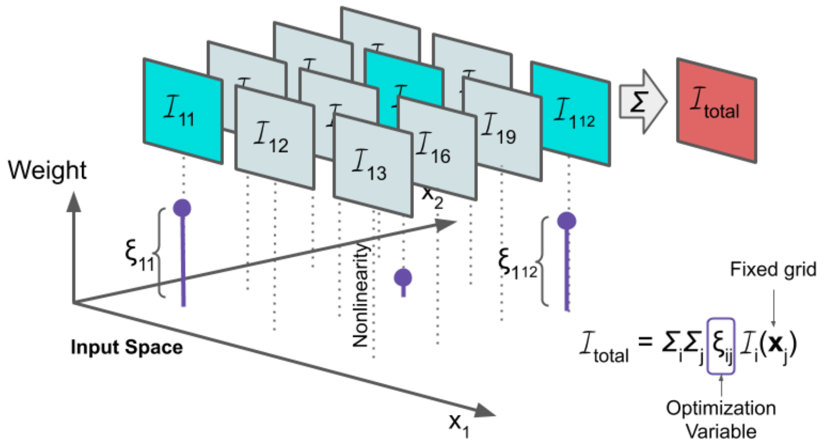


Figure 7.15: A depiction of NLOED design optimization problem structure with two discretized inputs and a single observation variable.

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 $\boldsymbol{x}_j \in \mathcal{X}$. As these points are fixed at specific levels in the grid, the input points, $\boldsymbol{x}_j$, 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, $\mathcal{I}_i(\boldsymbol{x}_j, \bar{\boldsymbol{\theta}})$, for each candidate grid point, $\boldsymbol{x}_j$, 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) [15]. 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 7.15 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.

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, $\boldsymbol{x}_j$, 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 7.16 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']]}
```

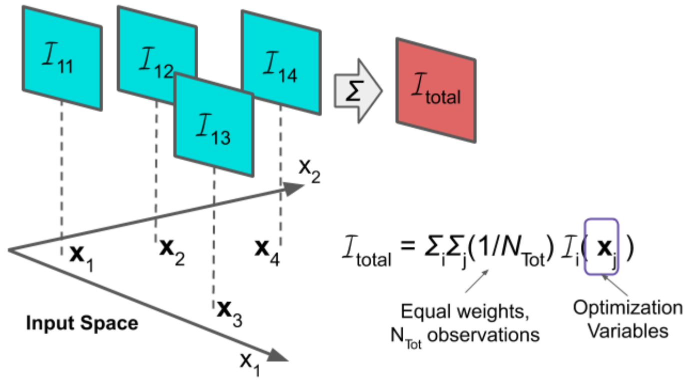


Figure 7.16: A depiction of an NLOED design optimization problem structure with two continuously handled inputs and the `LockedWeights` constraint activated.

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, $\boldsymbol{x}_j$, 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 $\boldsymbol{x}$ 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.

When continuous inputs are used in the default manner, without locked weights, both the weights, $\xi_{i,j}$, and the input points, $\boldsymbol{x}_j$, are treated as optimization variables. This scenario is depicted in Figure 7.17 and would equate with a `continuous_inputs` argument structured as:

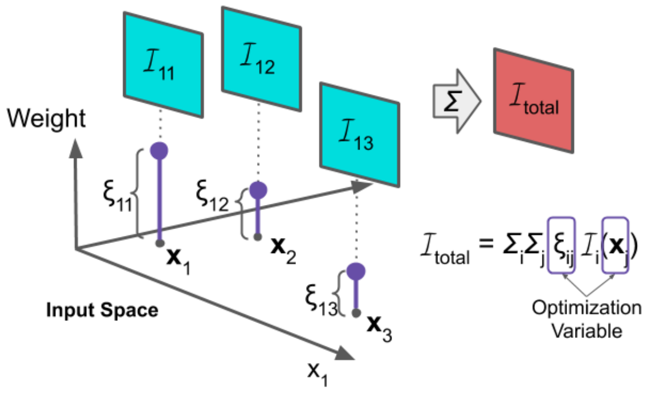


Figure 7.17: A depiction of NLOED design optimization problem structure with two continuously handled inputs, optimized sampling weights, and a single observation variable.

```
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 con-

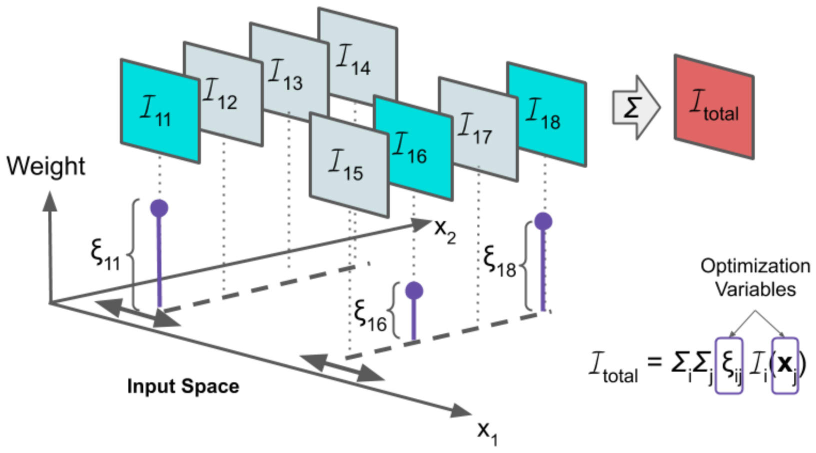


Figure 7.18: A depiction of NLOED's design optimization problem structure with the first input dimension handled continuously and the second input dimension handled discretely, and with a single observation variable.

tinuously 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 7.18. This scenario would correspond to the `discrete_inputs` and `continuous_inputs` arguments structured as;

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

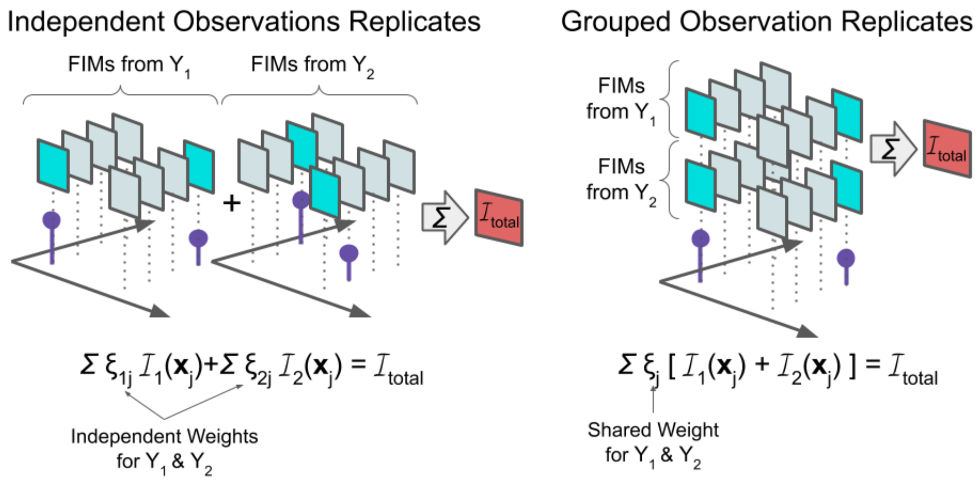

Independent Observations Replicates
FIMs from Y_1
FIMs from Y_2
Σ
$\mathcal{I}_{total}$
$\Sigma\, \xi_{1j}\, \mathcal{I}_1(\mathbf{x}_j) + \Sigma\, \xi_{2j}\, \mathcal{I}_2(\mathbf{x}_j) = \mathcal{I}_{total}$
Independent Weights for Y_1 & Y_2
Grouped Observation Replicates
FIMs from Y_1
FIMs from Y_2
$\Sigma\, \xi_j\, [\, \mathcal{I}_1(\mathbf{x}_j) + \mathcal{I}_2(\mathbf{x}_j)\,] = \mathcal{I}_{total}$
Shared Weight for Y_1 & Y_2

Figure 7.19: 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.

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 7.19 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 7.19 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 `evaluate()` function and simulation tools to assess design modifications and find a reasonable design.

7.4.3 User-callable `Design` Functions

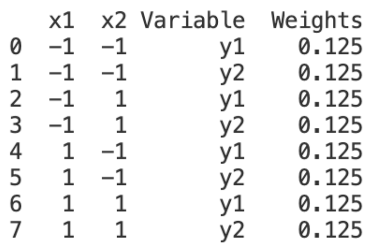
Once the user has instantiated a `Design` object instance, they can use the object to examine the optimal design structure and to generate exact designs that can be implemented in practice or be simulated by the `Model` class. The user can perform these actions by calling functions available within the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

```python
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 7.20. Here



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

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 [15], and thus 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 [15].

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 [60, 61, 62]. 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 [60, 61, 62]. The general call structure for the `round()` function is;

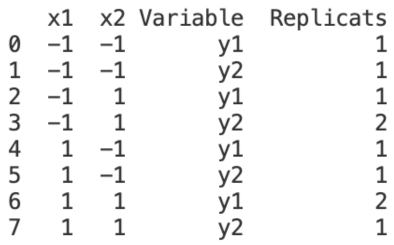
```
round(sample_size, options={})
```

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.

```
1 #set the sample size
2 sample_size = 10
3 #generate the rounded design
4 exact_design = design_object.round(sample_size)
5 #print the resulting exact design
6 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 7.20 is shown in Figure 7.21. The `round()`



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

function here is called with a sample size of 10, however, the 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 [243]. 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.

7.5 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.

7.5.1 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

$$GFP = \alpha_0 + \alpha \frac{Light^n}{K^n + Light^n}. \tag{7.18}$$

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
import numpy as np
import pandas as pd
import casadi as cs
import matplotlib.pyplot as plt
from nloed import Model
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.

```
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.

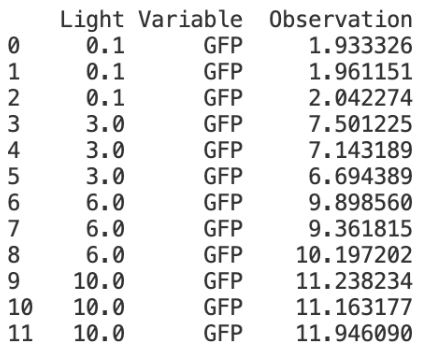
Having created a CasADi symbol for the GFP observation variable, the user can now construct an NLOED `Model` instance. Listing 19 demonstrates this process. In line 24, the `gfp_model` function is tupled with the label `Normal` indicating that it describes the sampling statistics of a normal random variable. 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 7.22. 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 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 $[1.87, 12.20, 1.31, 3.72]$



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

```python
#set options to use a simple initial search
fit_options={'InitParamBounds':[(-1,2),(1,3),(-1,2),(-1,2)],
             'InitSearchNumber':7}
#fit the model to the initial data
fit_info = model_object.fit(init_data, options=fit_options)
#extract the parameter values
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.

```
# enter the initial design information
init_design = pd.DataFrame({'Light':[.1,3,6,10],
                            'Variable':['GFP']*4 ,
                            'replicates':[3]*4})
#request the asymptotic covariance matrix
eval_options={'Method':'Asymptotic',
              'Covariance':True}
# call evaluate() to compute the asymptotic covariance
asymptotic_covariance =
↪  model_object.evaluate(init_design,fit_params,eval_options)
#compute the asymptotic upper and lower 95\% bounds
asymptotic_lower_bound = fit_params -
↪  2*np.sqrt(np.diag(asymptotic_covariance))
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 7.23. 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 7.23: 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.

```
50 #convert the covariance matrix to a Numpy array
51 covariance_matrix = asymptotic_covariance.to_numpy()
52 #select prediction intputs
53 prediction_inputs = pd.DataFrame({'Light':np.linspace(0.1,10,100),
54                                    'Variable':['GFP']*100})
55 #request prediction and observation intervals
56 prediction_options = {'PredictionInterval':True,
57                       'ObservationInterval':True}
58 #call predict()
59 predictions = model_object.predict(prediction_inputs,
60                                     fit_params,
61                                     covariance_matrix = covariance_matrix,
62                                     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 7.24. 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.

Given the analysis of the initial uncertainty in the parameter estimates and the predic-

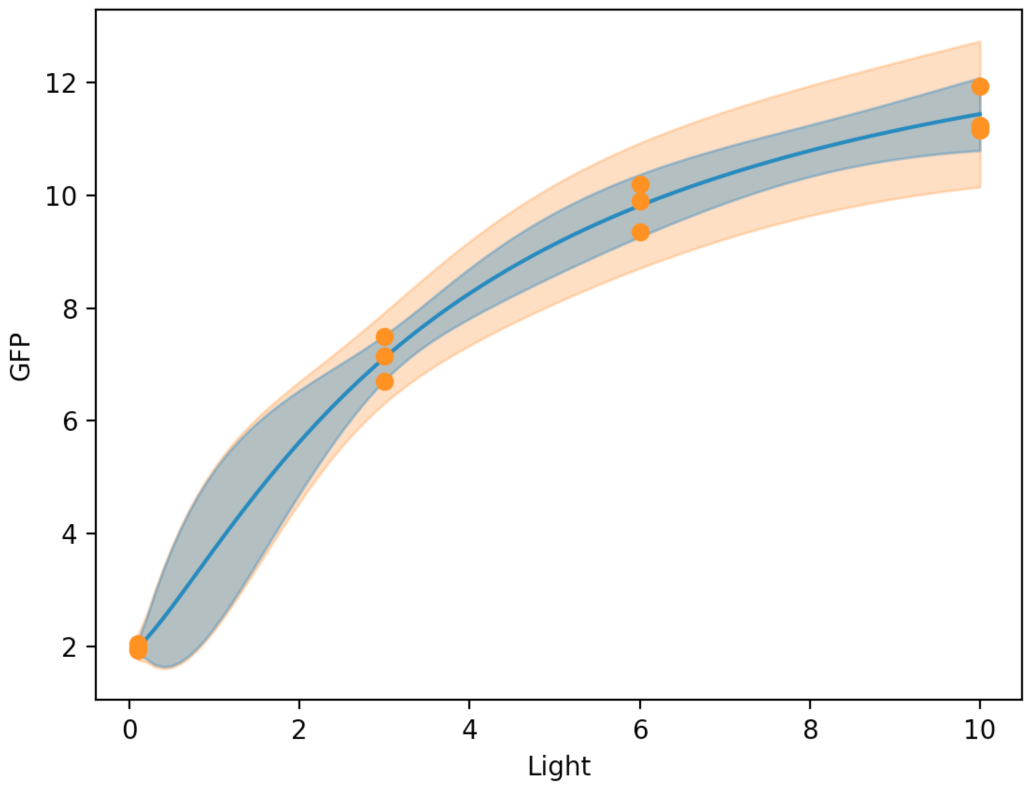


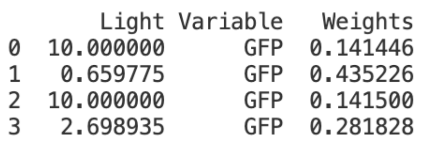
Figure 7.24: Mean GFP response (blue line), 95% prediction intervals (blue region) and 95% observation intervals (orange region) for the optogenetic model after fitting to the initial dataset (orange dots).

tion 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 7.25. 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 7.24. In line 76 of Listing



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

23, we set the sample size to 24 and in line 78 the `round()` function of the `Design` class is used to generate an exact design. The exact design is shown in Figure 7.26.

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

Figure 7.26: 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 7.31. Here we see that the intervals are expect to shrink considerably after adding the optimal data. In addition, we can generate prediction and observation

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

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 7.28

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

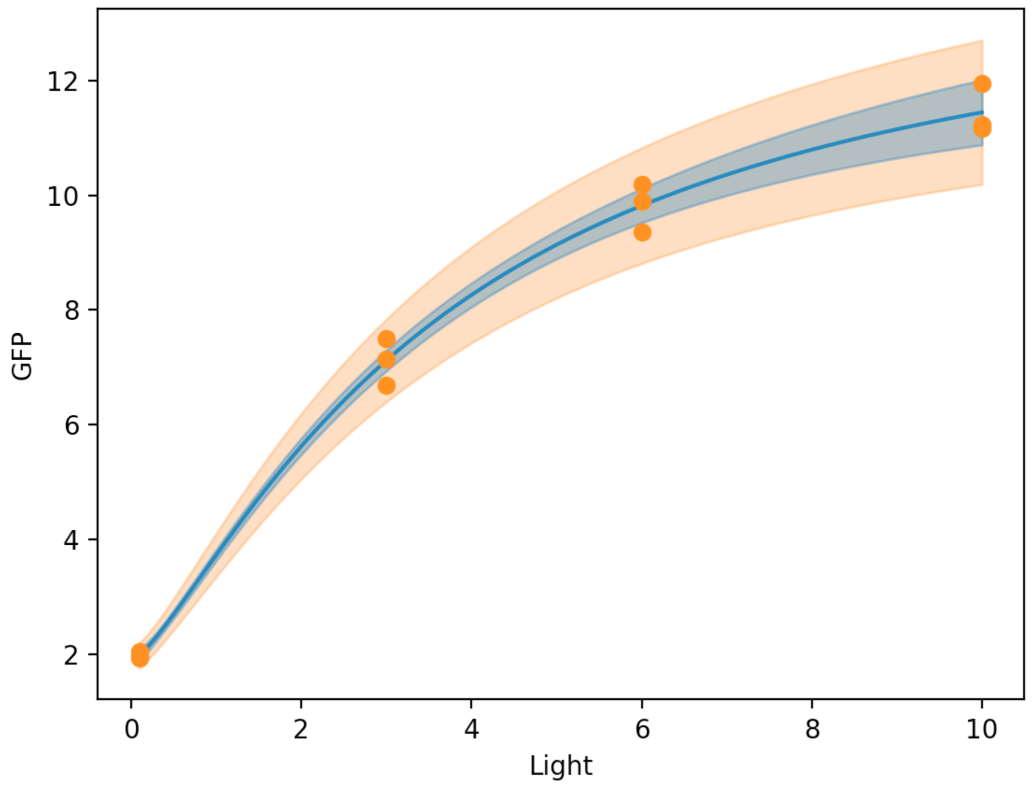


Figure 7.28: A plot of the 95% prediction and observation intervals under the asymptotic covariance matrix from the combined initial and optimal designs, computed with the `predict()` function.

`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 7.29. 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 intervals are also generated during fitting and are given in the returned dataframe. The likelihood intervals for this example are shown in Figure 7.30. Wald-type intervals can also be generated by using the `evaluate()` function as shown in Listing 21. The Wald intervals for this example are shown in Figure 7.30. These intervals differ from those shown in Figure 7.27 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 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.

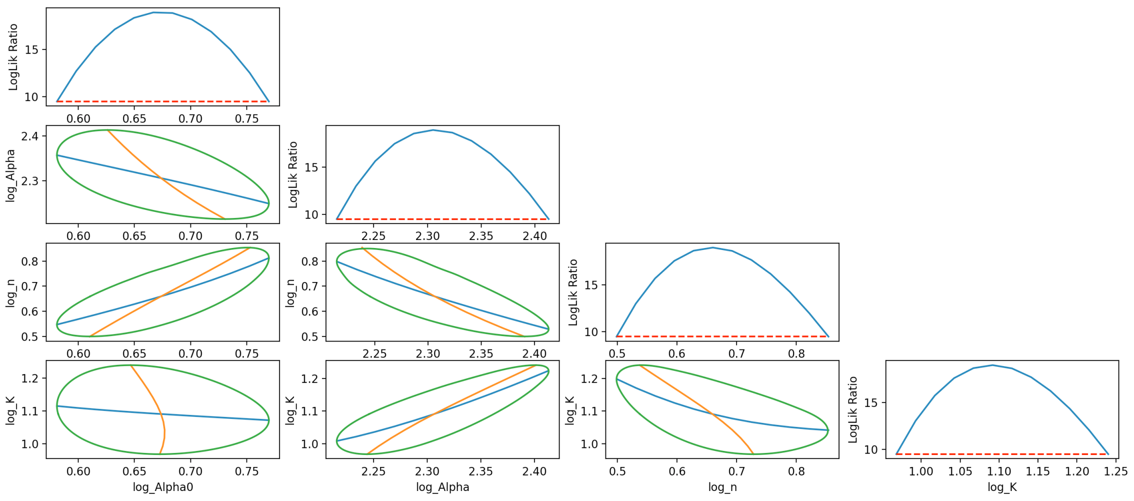


Figure 7.29: Diagnostic plots generated by the `fit()` function for the optogenetic dose-response model, including likelihood hood profiles, trace projections and contour projections.

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

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

7.5.2 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 [196]. 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

$$\begin{aligned} \frac{d[\text{RNA}]}{dt} &= \frac{\alpha}{1+\frac{K}{[\text{Inducer}]}} - \delta[\text{RNA}], \\ \frac{d[\text{Protein}]}{dt} &= \frac{\beta}{1+\frac{L}{[\text{RNA}]}} - \gamma[\text{Protein}]. \end{aligned} \tag{7.19}$$

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.

```python
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) [196]. 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.

```
#3 samples per cntrl interval, 3 cntrl intervals
#control intervals are 1+2+3=6 steps long:
#  cntrl_int1    cntrl_int2    cntrl_int3
#|-1---2-----3||-1---2-----3||-1---2-----3|
#set number of control intervals
num_cntrl_intervals = 3
#define a sample pattern to apply in each control interval
sample_pattern= [1,2,3]
#lists to store symbols for each sample point, and times of each sample
sample_list, times = [], []
# set the initial states and initialize the step counter
current_state, step_counter = initial_states, 0
#loop over control invervals
for interval in range(num_cntrl_intervals):
  # loop over sample pattern
  for num_stps in sample_pattern:
    #iterate steps indicated by sample pattern
    for k in range(num_stps):
      #propagate the state variables via integration
      current_state = step_func(current_state, inducer_vector[interval],
       ↪  parameters)
      step_counter+=1
    #save the state symbols and times of each sample
    sample_list.append(current_state)
    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)
```

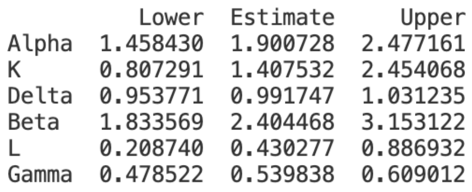
Listing 29: Code showing the assembly of the `observ_list` argument for use in the `Model` class constructor call for the mRNA-protein dynamic model.

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 7.32: 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 7.32. 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 be

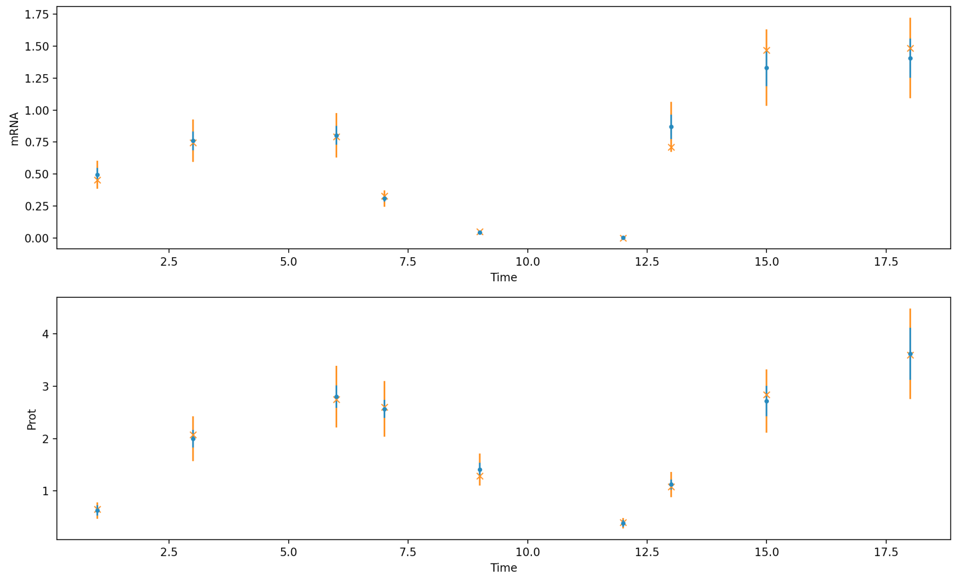


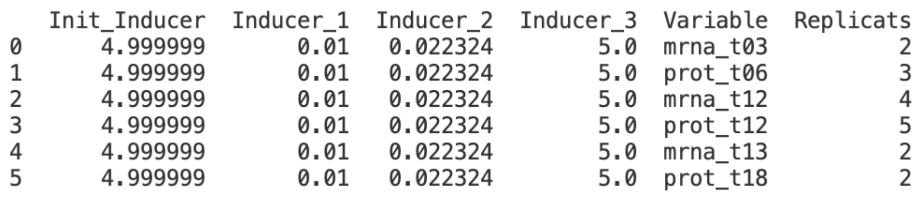
Figure 7.33: Prediction of the mean mRNA and protein levels, along with the 95% prediction and observation intervals generated with the `predict()` function for the mRNA-protein dynamic model.

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 7.33 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).

```
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.

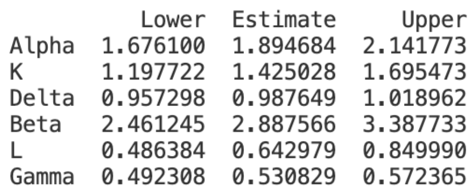
To generate an optimal experiment we need to instantiate a `Design` instance; Listing 31 shows this process. In lines 93-95 we specify that all inputs will be treated as continuous, bounded between 0.1 and 5 and that a single inducer profile will be optimized. In line 97, we create a `fixed_design` dictionary 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 7.34: Output of the optimal exact design generated for the mRNA-protein dynamic model.

The optimal design generated from this process is shown in Figure 7.34. 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



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

and RNA measurements are taken at different 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, 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 7.35. Here we see that interval sizes have shrunk. Figure 7.36 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.

7.6 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.

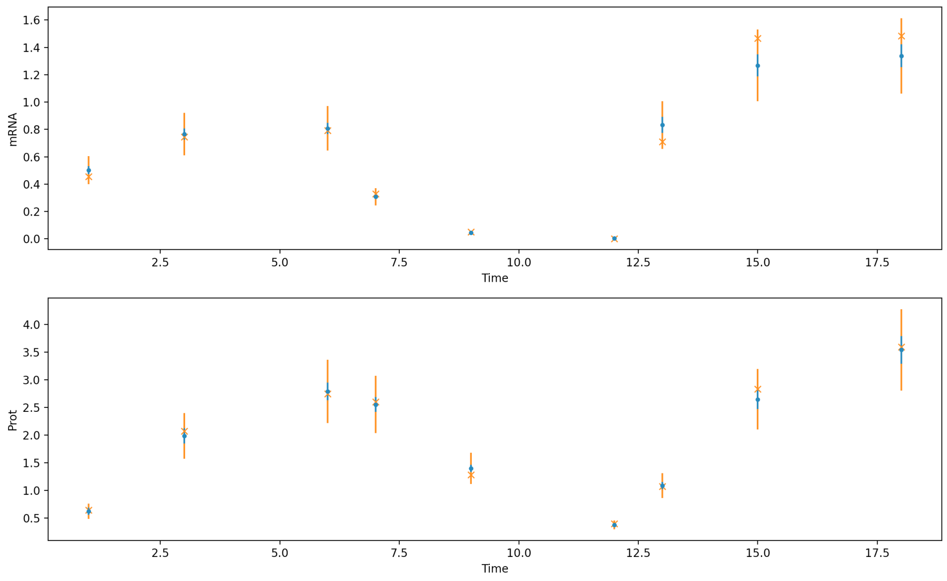


Figure 7.36: Updated 95% prediction and observation intervals for the mRNA-protein model, generated using the `predict()` function under the covariance found using `evaluate()` applied to the combined intitial and optimal designs.

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, $\bar{\boldsymbol{\theta}}$, will also be supplemented with a pseudo-Bayesian approach which will allow for design optimization over a weighted average of several candidate parameter points [244]. 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 [15]. 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 [40, 15]. 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 [20]. 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.

Chapter 8

Conclusion

"If your experiment needs a statistician, you need a better experiment."
-Ernest Rutherford

The main goal in this thesis has been investigating the use of optimal experimental design for calibrating models of microbial gene expression. I have specifically studied models that have yielded unique challenges or opportunities for the application of OED techniques. In applying OED methods to these various case-studies, I have been able to carefully study how models can be used to guide experimentation. I believe this careful consideration of the role of experimentation in model building is important to the wider goals of systems biology. As models and systems-level thinking are increasingly used to encode our understanding of biological systems, it is important to consider how this scientific paradigm should inform experimentation and data collection. This thesis has been an attempt to contribute to this broader theme, and therein lie the main contributions of this work.

8.1 Summary

The results of this thesis have been largely theoretical. However they may have some implications for practical experiments in future works. In Chapter 4 we applied optimal design to a novel physiologically-aware model of gene expression. This work pointed towards potential future work in characterizing genetic parts using automated design of experiments. This work also formulated the optimal design problem – for novel models of this type – as an optimal control problem in the CasADi toolbox, which will provide some guidance for researchers seeking to implement optimal design for similar systems in the future. In Chapter 5, we specifically discussed sampling schedules, and illustrated an efficient optimization approach for generating optimized schedules. While not groundbreaking, this work is timely as many researchers have increasingly begun to use automated culturing devices to perform time-course experiments on both natural and synthetic systems [154]. As noted in Chapter 5, dynamic experiments are complex and in order to achieve an effective experiment, it is important to consider interactions between the applied perturbations and the observation schedule. Using more efficient methods for time-point selection within the overall experimental design procedure can help to ensure important dynamic behaviour is not missed in studying the dynamics of biological systems. Chapter 6 focused on steady state experiments with more complex observation distributions. Many gene regulatory networks, such as oscillators, switches and other multi-stable bifurcating systems can have complicated steady state observation distributions for single-celled data. Little work has been done in designing experiments for recovering data from this type of experiment, most likely due to the complexity in formulating the likelihood and the required asymptotic measures of a design. However, steady state experiments can be easier to implement in practice and can be more repeatable than dynamic experiments. Work in Chapter 6 helped introduce novel experimental design approaches for this scenario, and will hopefully point the way to better use of steady state experiments, and further research on experimental design for valuable but numerically difficult scenarios. Finally in Chapter 7 I presented my recent work on the NLOED software package for experimental design. This package made use of automatic differentiation tools in developing a modelling and optimal design framework tailored to systems biology. The package also includes a number of other tools important for model building and diagnostics. Despite the primarily theoretical focus of this thesis, the NLOED software package will serve to make optimal design more accessible to practicing experimentalists. In line with this thinking, we are also currently working on some laboratory studies using the NLOED package. These ongoing works will serve to stress-test the software package in practical use and, in turn, NLOED will ideally assist in the study of real biological systems.

8.2 Some Critique

The work in this thesis has also revealed some shortcomings in the optimal design methodology in a system biology context. Under ideal conditions, optimal designs can significantly improve parameter estimation performance. However I have found that a large increase in the overall sample size, or the use of novel observation variables can sometimes do far more for model calibration than improving design alone. By this I mean that adding a few measurements of previously non-measured species, or just replicating a non-optimal but reasonably good experiment, can often improve parameter estimates considerably with little or no design work required. Mathematically this also makes sense because the confidence intervals of the parameter estimates generally shrinks with the inverse square root of the sample size. Therefore while each new sample has diminishing returns, if observations are cheap, enough measurements can always be taken to reach any desired confidence level. The value of additional measurements is significantly amplified when these measurements are made for novel species. In this case the new measurement types can provide novel information via the individual FIM matrices they contribute to the experiment's total sum of information. Optimal designs ensure that each new observation is used as best as possible but – when observations are cheap and for a reasonably good starting design – an experimenter may gain more value for their time through replication or learning a new assay, than through implementing optimal design.

I have also often found that intuitive designs are not as mathematically sub-optimal as one might expect. Experimenters naturally spread their observations out in the dimensions they know to be relevant to the system behaviour. This is almost never optimal but it is a form of thorough hedging against unexpected behaviour and it often ensures sufficient information for a reasonable parameter estimate. This suggests that, at least for low dimensional models with simple response patterns, intuition may often be sufficient as well as being more robust than optimal design. In fact, when Bayesian priors are used to incorporate more uncertainty in optimal design algorithms, past works (i.e. see examples in[15]) have shown that such prior information results in a similar spreading of input points selected by the optimal design algorithm. These results suggest that in some sense our experimental intuition is performing some very approximate Bayesian reasoning. In light of this, future work should perhaps focus on creating optimal design methods that better imitate experimental intuition. Allowing for more uncertainty to be specified in the design optimization problem will result in designs that more thoroughly explore the input space, and come closer to matching our expectations for a good overall design. Some work like this is already taking place; examples include methods that allow Bayesian priors over various modelling uncertainties [236, 157], or that directly including more complex sources of variability in the model specification (i.e. hierarchical models) [230]. The NLOED package did make some contribution here by allowing more flexible specification of the observation distribution than has been considered in the past, however more work is certainly needed. Future work should especially focus on the type of uncertainty and variability most relevant to systems biologists. While accommodating uncertainty directly in the design process has not been emphasized as much in the optimal design literature, work in this direction will be a return to an experimental philosophy emphasized within the wider DOE tradition, see [245].

My work on optimal experimental design and model calibration has also led me to believe there has been some overvaluation of large multi-state dynamic models with many parameters in systems biology. These models often contain dozens of parameters and their dynamics are often so complex they can only be studied through simulation. However, these models are easier to build than the real system is to experiment on, and the simulation algorithms for these systems are much more tractable than the fitting and diagnostic methods. Optimal design for these systems, like fitting and diagnostics, is limited by computational costs. This means that it is much easier to study these models in theory than to connect them to the real world in a meaningful way. While this is a shortcoming of optimal design algorithms, I believe it may also suggest a shift in emphasis. The complexity of the model being studied should be dictated by the fitting, diagnostics and the experimental burden, not just by the available listing of biological interactions. By focusing on overly complicated models, modellers are forced to use weaker diagnostic tools, such as Wald-type confidence intervals and least-squares fitting. These methods can be crude compared to Bayesian, likelihood, and simulation-based methods available from statistics – tools available even for non-linear models [50, 242]. The number of observations required to parameterize more complex models also tends to go up super-linearly with the number of free parameters. For models with many states, the more layers of integration between a parameter and the observed state, the more sensitivity information is blunted, yielding insensitive and uninformative observations. I believe these difficulties emphasize the benefits of experimenting on limiting regimes of larger systems. These limiting regimes could involve applying optimal design to a model's steady states, or specific subsystems of the overall system in a piece-wise manner. This approach would be complimentary to general techniques of model reduction but perhaps with more explicit emphasis on using experimental design considerations in selecting an appropriate reduced model. Perhaps a model reduction's utility can in some sense be measured by its identifiability under optimal design. Conversely, sometimes the experimental design itself determines the amount of reduction feasible before the reduced model can no longer adequately capture the experimental observations; a very simple experiment can accommodate a much simpler model. Thus experimental design, model reduction and the limiting experimental scenarios which will be targeted by the researcher are all intimately linked. Regardless, by considering more reduced models in more limiting scenarios, more attention can be paid to experimental design and fitting diagnostics, as well as to model assumptions like the observation distributions and the experimental protocols. I believe this will result in better fits, more confidence in model predictions and more reproducible results. While this comes at the expense of focusing on some limiting behaviour of a system rather than its full dynamics, I believe this may be a useful trade-off in many situations.

Work on my thesis, especially Chapter 4, has also led me to believe that optimal design, and modelling in general, is most valuable when applied to systems with more *operationally* meaningful parameters. Systems biology models are often mechanistic or at least pseudo-mechanistic and so parameters may have an interpretation within the physical process of the system. For example some parameter may describe the rate of interaction between two proteins. However, while the parameter value may tell you the specific rate, it is often not possible with available knowledge to link this rate to the exact amino acid residues responsible for the parameter value observed (at least without considerably more experimental work). This means that even though a parameter may be mechanistically interpretable, it lacks meaning in a operational sense; you do not know what implementable changes to the system will cause changes to the parameter value. However, in some systems, especially gene expression, there is sufficient existing knowledge so that parameters can be linked to specific operational perturbations of the system. This means that we know how to, at least hypothetically, perturb the parameter value via an achievable intervention. In Chapter 4, the physiological model was of particular interest because the parameters provided meaningful characterization of the genetic sequence and the host. The parameters in this model were not all mechanistic (i.e. the phenomenological model of the proteome), however they all had interpretations and many could be operationally linked to possible experimental perturbations. For example parameters governing transcription could be perturbed via changes to the promoter sequence[246], and the parameters for the host's translation capacity could be perturbed via mutation or antibiotic treatments [148]. For systems where parameters can be imbued with this extra operational meaning, accurate parameter estimation becomes more valuable and there are many more uses for a fit model. In this context, parameter estimates now serve to characterize a specific configuration of the systems, and if we make some of the prescribed perturbations we can use fitting to quantify their magnitude. For example if we mutate the promoter sequence we would, as a first estimate, expect only the promoter-related parameters to shift upon refitting the model in Chapter 4. Models with this type of parameterization can also be more easily validated. If we have an independent estimate for the direction or magnitude of a given perturbation on a parameter value (i.e. estimates of mRNA-ribosome binding energy under sequences alterations [190]), then upon perturbing the system, we would expect to see that same direction or magnitude, reflected in the model's parameter estimates after fitting to the new data. If this is not so, than either our model or our biological understanding of the perturbation could be in error, suggesting further refinements. This approach provides a means for stress-testing our models and system understanding. In physics and chemistry physical parameters can often be estimated somewhat directly because the experimental conditions can be reduced to a platonic simplicity. Unfortunately, living biological systems are inherently irreducible past a certain point, making direct measurements of *in vivo* parameter values very difficult. However, perturbing parameters and then quantifying the perturbations with fitting provides at least some independent verification of our systems-level models. This also takes advantage of the lower experimental burden in microbiology compared to other disciplines in which experiments may be very costly or impossible (i.e. economics, ecology, social sciences etc.). Conditions for this type of iterative perturbation and fitting are still somewhat rare in systems biology but careful consideration by theorists about the meaning of their parameter values and the availability of experimental tools to perturb them can perhaps make this multi-faceted approach more common. I also believe careful consideration of experimental design could play a useful role in this approach in the future, especially as it relies on refitting a similar model in multiple scenarios with an explicit emphasis on parameter estimation accuracy.

8.3 Future Prospects

By considering the above critiques carefully I think it is possible to derive some general suggestions for which biological systems are most likely to benefit from optimal design methods in the future. To summarize, these are:

- System in which additional observations are costly or destructive and in which experimentation is inherently iterative rather than occurring in large, cheap batches.
- Systems in which our intuition is likely to fail. This includes systems with many experimental input or observation dimensions, especially when they are linked by a complicated relationship.
- System with non-normal or complex error distributions. This includes multi-stable or quasi-stable systems, especially focusing on single-cell observations. Functional data, where the observation is a real-time function rather than a scalar, are also of interest.
- Systems which are computationally tractable for optimal design and related fitting diagnostics, especially future Bayesian methods. This includes steady state models or dynamic models with few hidden states.

- Systems where the link between parameter values and the system response are non-intuitive. For example an increase in a parameter value can have an effect on the mean or variance of the response that is directionally conditional on other parameters or that is possibly non-monotonic. This may include multi-stable systems, and deterministic approximations to stochastic models where parameters can have complex effects on both the observation mean and variance.

- Systems where the model structure is well established for the type of system but individual instances of the system require accurate parameterization. This would include characterizing design variants of synthetic systems.

- Systems with operationally meaningful parameters; meaningful in having both an operational and mechanistic/functional interpretation. This is especially common in well studied areas like gene expression.

While the optimal design methods used in this thesis are not a panacea, I believe even the local methods used here can be of practical use in many other scenarios. I have often gained considerable insight into the implications of a given modelling decision by considering experimental design as part of the model building process. I have gained these insights even in cases where I have run an optimal design algorithm and found a woefully inadequate design is produced. On investigation it was rarely the optimal design method that failed and more often I realize something unexpected about what the model structure or the assumptions implied about the system behaviour. I think even these qualitative insights make experimental design worth pursuing as a form of good modelling hygiene, to shake assumptions loose and get theorists thinking about the lab.

References

[1] Ferenc Mechler, John Hartigan, and Pamela Hartigan. MATLAB code for Hartigan's dip statistic. http://www.nicprice.net/diptest/, March 2019.

[2] H Bremer and PP Dennis. Modulation of chemical composition and other parameters of the cell at different exponential growth rates. *EcoSal Plus*, 3(1), 2008.

[3] Mathew Stracy, Christian Lesterlin, Federico Garza De Leon, Stephan Uphoff, Pawel Zawadzki, and Achillefs N Kapanidis. Live-cell superresolution microscopy reveals the organization of RNA polymerase in the bacterial nucleoid. *Proceedings of the National Academy of Sciences*, 112(32):E4390–E4399, 2015.

[4] Ulrike Endesfelder, Kieran Finan, Seamus J Holden, Peter R Cook, Achillefs N Kapanidis, and Mike Heilemann. Multiscale spatial organization of RNA polymerase in *Escherichia coli*. *Biophysical journal*, 105(1):172–181, 2013.

[5] Somenath Bakshi, Renée M Dalrymple, Wenting Li, Heejun Choi, and James C Weisshaar. Partitioning of RNA polymerase activity in live *Escherichia coli* from analysis of single-molecule diffusive trajectories. *Biophysical Journal*, 105(12):2676–2686, 2013.

[6] RA Fisher. The arrangement of field experiments. *Journal of the Ministry of Agriculture*, 33:503–515, 1926.

[7] Ronald Aylmer Fisher. Design of experiments. *Br Med J*, 1(3923):554–554, 1936.

[8] David R Hagen, Jacob K White, and Bruce Tidor. Convergence in parameters and predictions using computational experimental design. *Interface Focus*, 3(4):20130008, 2013.

[9] 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.

[10] 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.

[11] Jennifer AN Brophy and Christopher A Voigt. Principles of genetic circuit design. *Nature Methods*, 11(5):508, 2014.

[12] Alec AK Nielsen, Bryan S Der, Jonghyeon Shin, Prashant Vaidyanathan, Vanya Paralanov, Elizabeth A Strychalski, David Ross, Douglas Densmore, and Christopher A Voigt. Genetic circuit design automation. *Science*, 352(6281):aac7341, 2016.

[13] Michael B Elowitz and Stanislas Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767):335–338, 2000.

[14] Timothy S Gardner, Charles R Cantor, and James J Collins. Construction of a genetic toggle switch in *Escherichia coli*. *Nature*, 403(6767):339–342, 2000.

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

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

[17] David Roxbee Cox and Nancy Reid. *The theory of the design of experiments*. CRC Press, 2000.

[18] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[19] Raymond H Myers, Douglas C Montgomery, and Christine M Anderson-Cook. *Response surface methodology: process and product optimization using designed experiments*. John Wiley & Sons, 2016.

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

[21] 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.

[22] 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.

[23] Stephen M Stigler. *The history of statistics: The measurement of uncertainty before 1900*. Harvard University Press, 1986.

[24] Kirstine Smith. On the standard deviations of adjusted and interpolated values of an observed polynomial function and its constants and the guidance they give towards a proper choice of the distribution of observations. *Biometrika*, 12(1/2):1–85, 1918.

[25] Stephen M Stigler. Gergonne's 1815 paper on the design and analysis of polynomial regression experiments. *Historia Mathematica*, 1(4):431–439, 1974.

[26] C Radhakrishna Rao et al. Ra fisher: The founder of modern statistics. *Statistical Science*, 7(1):34–48, 1992.

[27] Stephen M Stigler et al. The epic story of maximum likelihood. *Statistical Science*, 22(4):598–620, 2007.

[28] Joan Fisher Box. Ra fisher and the design of experiments, 1922–1926. *The American Statistician*, 34(1):1–7, 1980.

[29] Do Jo Finney. The fractional replication of factorial arrangements. *Annals of Eugenics*, 12(1):291–301, 1943.

[30] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, 1946.

[31] C Radhakrishna Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Supplement to the Journal of the Royal Statistical Society*, 9(1):128–139, 1947.

[32] George EP Box. *An Accidental Statistician: The Life and Memories of George EP Box*. John Wiley & Sons, 2013.

[33] George EP Box and Kenneth B Wilson. On the experimental attainment of optimum conditions. *Journal of the royal statistical society: Series b (Methodological)*, 13(1):1–38, 1951.

[34] Abraham Wald. On the efficient design of statistical investigations. *The annals of mathematical statistics*, 14(2):134–140, 1943.

[35] Herman Chernoff. Locally optimal designs for estimating parameters. *The Annals of Mathematical Statistics*, pages 586–602, 1953.

[36] Gustav Elfving et al. Optimum allocation in linear regression theory. *The Annals of Mathematical Statistics*, 23(2):255–262, 1952.

[37] J Fellman. Gustav Elfving's contribution to the emergence of the optimal experimental design theory. *Statistical Science*, pages 197–200, 1999.

[38] Henry P Wynn. Jack Kiefer's contributions to experimental design. *The Annals of Statistics*, 12(2):416–423, 1984.

[39] Jack Kiefer. Optimum experimental designs. *Journal of the Royal Statistical Society: Series B (Methodological)*, 21(2):272–304, 1959.

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

[41] Peter Goos and Bradley Jones. *Optimal design of experiments: a case study approach.* John Wiley & Sons, 2011.

[42] Lynda V White. An extension of the general equivalence theorem to nonlinear models. *Biometrika*, 60(2):345–348, 1973.

[43] Kenneth G Russell. *Design of Experiments for Generalized Linear Models.* CRC Press, 2018.

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

[45] Eric A Strömberg, Joakim Nyberg, and Andrew C Hooker. The effect of fisher information matrix approximation methods in population optimal design calculations. *Journal of pharmacokinetics and pharmacodynamics*, 43(6):609–619, 2016.

[46] Joshua F Apgar, Jared E Toettcher, Drew Endy, Forest M White, and Bruce Tidor. Stimulus design for model selection and validation in cell signaling. *PLoS Computational Biology*, 4(2):e30, 2008.

[47] Bence Mélykúti, Elias August, Antonis Papachristodoulou, and Hana El-Samad. Discriminating between rival biochemical network models: three approaches to optimal experiment design. *BMC Systems Biology*, 4(1):38, 2010.

[48] Juliane Liepe, Paul Kirk, Sarah Filippi, Tina Toni, Chris P Barnes, and Michael PH Stumpf. A framework for parameter estimation and model selection from experimental data in systems biology using approximate bayesian computation. *Nature protocols*, 9(2):439, 2014.

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

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

[51] Chien-Fu Wu. Asymptotic theory of nonlinear least squares estimation. *The Annals of Statistics*, pages 501–513, 1981.

[52] Ludwig Fahrmeir and Heinz Kaufmann. Consistency and asymptotic normality of the maximum likelihood estimator in generalized linear models. *The Annals of Statistics*, pages 342–368, 1985.

[53] Bruce Hoadley. Asymptotic properties of maximum likelihood estimators for the independent not identically distributed case. *The Annals of mathematical statistics*, pages 1977–1991, 1971.

[54] Ernesto San Martín and Fernando Quintana. Consistency and identifiability revisited. *Brazilian Journal of Probability and Statistics*, pages 99–106, 2002.

[55] Erich L Lehmann and George Casella. *Theory of point estimation.* Springer Science & Business Media, 2006.

[56] MJ Box. Bias in nonlinear estimation. *Journal of the Royal Statistical Society: Series B (Methodological)*, 33(2):171–190, 1971.

[57] 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.

[58] Erich Leo Lehmann. *Elements of large-sample theory.* Springer Science & Business Media, 2004.

[59] Valerii Fedorov. Optimal experimental design. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(5):581–589, 2010.

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

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

[62] 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.

[63] Sebastian M Castillo-Hair, Oleg A Igoshin, and Jeffrey J Tabor. How to train your microbe: methods for dynamically characterizing gene networks. *Current Opinion in Microbiology*, 24:113–123, 2015.

[64] Nicole A Repina, Alyssa Rosenbloom, Abhirup Mukherjee, David V Schaffer, and Ravi S Kane. At light speed: advances in optogenetic systems for regulating cell signaling and behavior. *Annual Review of Chemical and Biomolecular Engineering*, 8:13–39, 2017.

[65] Giorgia Guglielmi, Henning Johannes Falk, and Stefano De Renzis. Optogenetic control of protein function: from intracellular processes to tissue morphogenesis. *Trends in Cell Biology*, 26(11):864–874, 2016.

[66] Evan J Olson and Jeffrey J Tabor. Optogenetic characterization methods overcome key challenges in synthetic and systems biology. *Nature Chemical Biology*, 10(7):502–511, 2014.

[67] Jared E Toettcher, Delquin Gong, Wendell A Lim, and Orion D Weiner. Light-based feedback for controlling intracellular signaling dynamics. *Nature Methods*, 8(10):837, 2011.

[68] Andreas Milias-Argeitis, Sean Summers, Jacob Stewart-Ornstein, Ignacio Zuleta, David Pincus, Hana El-Samad, Mustafa Khammash, and John Lygeros. *In silico* feedback for *in vivo* regulation of a gene expression circuit. *Nature Biotechnology*, 29(12):1114–1116, 2011.

[69] Andreas Milias-Argeitis, Marc Rullan, Stephanie K Aoki, Peter Buchmann, and Mustafa Khammash. Automated optogenetic feedback control for precise and robust

regulation of gene expression and cell growth. *Nature Communications*, 7:12546, 2016.

[70] Justin Melendez, Michael Patel, Benjamin L Oakes, Ping Xu, Patrick Morton, and Megan N McClean. Real-time optogenetic control of intracellular protein concentration in microbial cell cultures. *Integrative Biology*, 6(3):366–372, 2014.

[71] Jeffrey J Tabor, Anselm Levskaya, and Christopher A Voigt. Multichromatic control of gene expression in *Escherichia coli*. *Journal of Molecular Biology*, 405(2):315–324, 2011.

[72] Evan J Olson, Lucas A Hartsough, Brian P Landry, Raghav Shroff, and Jeffrey J Tabor. Characterizing bacterial gene circuit dynamics with optically programmed gene expression signals. *Nature Methods*, 11(4):449–455, 2014.

[73] Eric A Davidson, Amar S Basu, and Travis S Bayer. Programming microbes using pulse width modulation of optical signals. *Journal of Molecular Biology*, 425(22):4161–4166, 2013.

[74] Remy Chait, Jakob Ruess, Tobias Bergmiller, Gašper Tkačik, and Călin C Guet. Shaping bacterial population behavior through computer-interfaced control of individual cells. *Nature Communications*, 8(1):1535, 2017.

[75] Jacob Stewart-Ornstein, Susan Chen, Rajat Bhatnagar, Jonathan S Weissman, and Hana El-Samad. Model-guided optogenetic study of PKA signaling in budding yeast. *Molecular Biology of the Cell*, 28(1):221–227, 2017.

[76] Jared E Toettcher, Orion D Weiner, and Wendell A Lim. Using optogenetics to interrogate the dynamic control of signal transmission by the Ras/Erk module. *Cell*, 155(6):1422–1434, 2013.

[77] Jannis Uhlendorf, Agnès Miermont, Thierry Delaveau, Gilles Charvin, François Fages, Samuel Bottani, Gregory Batt, and Pascal Hersen. Long-term model predictive control of gene expression at the population and single-cell levels. *Proceedings of the National Academy of Sciences USA*, 109(35):14271–14276, 2012.

[78] Filippo Menolascina, Gianfranco Fiore, Emanuele Orabona, Luca De Stefano, Mike Ferry, Jeff Hasty, Mario di Bernardo, and Diego di Bernardo. In-vivo real-time control of protein expression from endogenous and synthetic gene networks. *PLoS Computational Biology*, 10(5):e1003625, 2014.

[79] Gianfranco Fiore, Giansimone Perrino, Mario di Bernardo, and Diego di Bernardo. *In vivo* real-time control of gene expression: A comparative analysis of feedback control strategies in yeast. *ACS Synthetic Biology*, 5(2):154–162, 2015.

[80] Jean-Baptiste Lugagne, Sebastián Sosa Carrillo, Melanie Kirch, Agnes Köhler, Gregory Batt, and Pascal Hersen. Balancing a genetic toggle switch by real-time feedback control and periodic forcing. *Nature Communications*, 8(1):1671, 2017.

[81] Nan Hao and Erin K O'Shea. Signal-dependent dynamics of transcription factor translocation controls gene expression. *Nature Structural & Molecular Biology*, 19(1):31–39, 2012.

[82] Anders S Hansen and Erin K O'Shea. Promoter decoding of transcription factor dynamics involves a trade-off between noise and control of gene expression. *Molecular Systems Biology*, 9(1):704, 2013.

[83] Corentin Briat, Ankit Gupta, and Mustafa Khammash. Antithetic integral feedback ensures robust perfect adaptation in noisy biomolecular networks. *Cell Systems*, 2(1):15–26, 2016.

[84] Tanya Tschirhart, Eunkyoung Kim, Ryan McKay, Hana Ueda, Hsuan-Chen Wu, Alex Eli Pottash, Amin Zargar, Alejandro Negrete, Joseph Shiloach, Gregory F Payne, et al. Electronic control of gene expression and cell behaviour in *Escherichia coli* through redox signalling. *Nature Communications*, 8:14030, 2017.

[85] Dan I Piraner, Arash Farhadi, Hunter C Davis, Di Wu, David Maresca, Jerzy O Szablowski, and Mikhail G Shapiro. Going deeper: Biomolecular tools for acoustic and magnetic imaging and control of cellular function. *Biochemistry*, 56(39):5202–5209, 2017.

[86] Erdal Toprak, Adrian Veres, Sadik Yildiz, Juan M Pedraza, Remy Chait, Johan Paulsson, and Roy Kishony. Building a morbidostat: an automated continuous-culture device for studying bacterial drug resistance under dynamically sustained drug inhibition. *Nature Protocols*, 8(3):555, 2013.

[87] Chris N Takahashi, Aaron W Miller, Felix Ekness, Maitreya J Dunham, and Eric Klavins. A low cost, customizable turbidostat for use in synthetic circuit characterization. *ACS Synthetic Biology*, 4(1):32–38, 2014.

[88] Dominick Matteau, Vincent Baby, Stéphane Pelletier, and Sébastien Rodrigue. A small-volume, low-cost, and versatile continuous culture device. *PLoS One*, 10(7):e0133384, 2015.

[89] Karl P Gerhardt, Evan J Olson, Sebastian M Castillo-Hair, Lucas A Hartsough, Brian P Landry, Felix Ekness, Rayka Yokoo, Eric J Gomez, Prabha Ramakrishnan, Junghae Suh, et al. An open-hardware platform for optogenetics and photobiology. *Scientific Reports*, 6:35363, 2016.

[90] Hsinkai Wang and Ya-Tang Yang. Mini photobioreactors for *in vivo*, real time characterization and evolutionary tuning of bacterial optogenetic circuit. *ACS Synthetic Biology*, 2017.

[91] A Estevez-Torres, A Yamada, and L Wang. An inexpensive and durable epoxy mould for PDMS. `http://blogs.rsc.org/chipsandtips/2009/04/22/an-inexpensive-and-durable-epoxy-mould-for-pdms/`, 2009. [Accessed: February 8, 2018].

[92] Todd A Duncombe, Augusto M Tentori, and Amy E Herr. Microfluidics: reframing biological enquiry. *Nature Reviews Molecular Cell Biology*, 16(9):554–567, 2015.

[93] Nicolas Szita, Karen Polizzi, Nicolas Jaccard, and Frank Baganz. Microfluidic approaches for systems and synthetic biology. *Current Opinion in Biotechnology*, 21(4):517–523, 2010.

[94] David S Kong, Todd A Thorsen, Jonathan Babb, Scott T Wick, Jeremy J Gam, Ron Weiss, and Peter A Carr. Open-source, community-driven microfluidics with Metafluidics. *Nature Biotechnology*, 35(6):523–529, 2017.

[95] Arthur D Edelstein, Mark A Tsuchida, Nenad Amodaj, Henry Pinkard, Ronald D Vale, and Nico Stuurman. Advanced methods of microscope control using μmanager software. *Journal of Biological Methods*, 1(2), 2014.

[96] Christian Carsten Sachs, Alexander Grünberger, Stefan Helfrich, Christopher Probst, Wolfgang Wiechert, Dietrich Kohlheyer, and Katharina Nöh. Image-based single cell profiling: High-throughput processing of mother machine experiments. *PLoS One*, 11(9):e0163453, 2016.

[97] Ouriel Caen, Heng Lu, Philippe Nizard, and Valerie Taly. Microfluidics as a strategic player to decipher single-cell omics? *Trends in Biotechnology*, 35(8):713–727, 2017.

[98] Michael J Lawson, Daniel Camsund, Jimmy Larsson, Özden Baltekin, David Fange, and Johan Elf. *In situ* genotyping of a pooled strain library after characterizing complex phenotypes. *Molecular Systems Biology*, 13(10):947, 2017.

[99] Jack Kiefer. On the nonrandomized optimality and randomized nonoptimality of symmetrical designs. *The Annals of Mathematical Statistics*, pages 675–699, 1958.

[100] Friedrich Pukelsheim. *Optimal Design of Experiments*. SIAM, 2006.

[101] Ryan N Gutenkunst, Joshua J Waterfall, Fergal P Casey, Kevin S Brown, Christopher R Myers, and James P Sethna. Universally sloppy parameter sensitivities in systems biology models. *PLoS Computational Biology*, 3(10):e189, 2007.

[102] Kamil Erguler and Michael PH Stumpf. Practical limits for reverse engineering of dynamical systems: a statistical analysis of sensitivity and parameter inferability in systems biology models. *Molecular BioSystems*, 7(5):1593–1602, 2011.

[103] Joshua F Apgar, David K Witmer, Forest M White, and Bruce Tidor. Sloppy models, parameter uncertainty, and the role of experimental design. *Molecular BioSystems*, 6(10):1890–1900, 2010.

[104] Oana-Teodora Chis, Alejandro F Villaverde, Julio R Banga, and Eva Balsa-Canto. On the relationship between sloppiness and identifiability. *Mathematical Biosciences*, 282:147–161, 2016.

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

[106] Elizabeth G Ryan, Christopher C Drovandi, James M McGree, and Anthony N Pettitt. A review of modern computational algorithms for Bayesian optimal design. *International Statistical Review*, 84(1):128–154, 2016.

[107] Eva Balsa-Canto, Antonio A Alonso, and Julio R Banga. Computational procedures for optimal experimental design in biological systems. *IET Systems Biology*, 2(4):163–172, 2008.

[108] SP Asprey and S Macchietto. Designing robust optimal dynamic experiments. *Journal of Process Control*, 12(4):545–556, 2002.

[109] Daniel Faller, Ursula Klingmüller, and Jens Timmer. Simulation methods for optimal experimental design in systems biology. *Simulation*, 79(12):717–725, 2003.

[110] Eva Balsa-Canto, Antonio A Alonso, and Julio R Banga. An iterative identification procedure for dynamic modeling of biochemical networks. *BMC Systems Biology*, 4(1):11, 2010.

[111] Yunfei Chu and Juergen Hahn. Integrating parameter selection with experimental design under uncertainty for nonlinear dynamic systems. *AIChE Journal*, 54(9):2310–2320, 2008.

[112] Jakob Ruess, Andreas Milias-Argeitis, and John Lygeros. Designing experiments to understand the variability in biochemical reaction networks. *Journal of the Royal Society Interface*, 10(88):20130588, 2013.

[113] Juliane Liepe, Sarah Filippi, Michał Komorowski, and Michael PH Stumpf. Maximizing the information content of experiments in systems biology. *PLoS Computational Biology*, 9(1):e1002888, 2013.

[114] Thembi Mdluli, Gregery T Buzzard, and Ann E Rundell. Efficient optimization of stimuli for model-based design of experiments to resolve dynamical uncertainty. *PLoS Computational Biology*, 11(9):e1004488, 2015.

[115] Dominik Skanda and Dirk Lebiedz. An optimal experimental design approach to model discrimination in dynamic biochemical systems. *Bioinformatics*, 26(7):939–945, 2010.

[116] Johannes Stegmaier, Dominik Skanda, and Dirk Lebiedz. Robust optimal design of experiments for model discrimination using an interactive software tool. *PLoS One*, 8(2):e55723, 2013.

[117] Robert J Flassig and Kai Sundmacher. Optimal design of stimulus experiments for robust discrimination of biochemical reaction networks. *Bioinformatics*, 28(23):3089–3096, 2012.

[118] Federico Galvanin, Sandro Macchietto, and Fabrizio Bezzo. Model-based design of parallel experiments. *Industrial & Engineering Chemistry Research*, 46(3):871–882, 2007.

[119] Jason N Bazil, Gregory T Buzzard, and Ann E Rundell. A global parallel model based design of experiments method to minimize model output uncertainty. *Bulletin of Mathematical Biology*, 74(3):688–716, 2012.

[120] Federico Galvanin, Massimiliano Barolo, Gabriele Pannocchia, and Fabrizio Bezzo. Online model-based redesign of experiments with erratic models: A disturbance estimation approach. *Computers & Chemical Engineering*, 42:138–151, 2012.

[121] Wei Pan, Filippo Menolascina, and Guy-Bart Stan. Online model selection for synthetic gene networks. In *Proceedings of the 2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 776–782. IEEE, 2016.

[122] Eva Van DerlCinden, Kristel Bernaerts, and JF Van Impe. Accurate estimation of cardinal growth temperatures of *Escherichia coli* from optimal dynamic experiments. *International Journal of Food Microbiology*, 128(1):89–100, 2008.

[123] Gaia Franceschini and Sandro Macchietto. Novel anticorrelation criteria for design of experiments: Algorithm and application. *AIChE Journal*, 54(12):3221–3238, 2008.

[124] Michel Brik Ternbach, Christian Bollman, Christian Wandrey, and Ralf Takors. Application of model discriminating experimental design for modeling and development of a fermentative fed-batch L-valine production process. *Biotechnology and Bioengineering*, 91(3):356–368, 2005.

[125] Jacob Beal. Bridging the gap: a roadmap to breaking the biological design barrier. *Frontiers in Bioengineering and Biotechnology*, 2, 2014.

[126] Jacob Beal, Traci Haddock-Angelli, Markus Gershater, Kim De Mora, Meagan Lizarazo, Jim Hollenhorst, Randy Rettberg, et al. Reproducibility of fluorescent expression from engineered biological constructs in E. coli. *PLoS One*, 11(3):e0150182, 2016.

[127] Sebastian M Castillo-Hair, John T Sexton, Brian P Landry, Evan J Olson, Oleg A Igoshin, and Jeffrey J Tabor. Flowcal: A user-friendly, open source software tool for automatically converting flow cytometry data from arbitrary to calibrated units. *ACS Synthetic Biology*, 5(7):774–780, 2016.

[128] Patrick Weber, Andrei Kramer, Clemens Dingler, and Nicole Radde. Trajectory-oriented Bayesian experiment design versus Fisher A-optimal design: an in depth comparison study. *Bioinformatics*, 28(18):i535–i541, 2012.

[129] Dries Telen, Filip Logist, Eva Van Derlinden, Ignace Tack, and Jan Van Impe. Optimal experiment design for dynamic bioprocesses: a multi-objective approach. *Chemical Engineering Science*, 78:82–97, 2012.

[130] Daniel Silk, Paul DW Kirk, Chris P Barnes, Tina Toni, and Michael PH Stumpf. Model selection in systems biology depends on experimental design. *PLoS Computational Biology*, 10(6):e1003650, 2014.

[131] Andrew White, Malachi Tolman, Howard D Thames, Hubert Rodney Withers, Kathy A Mason, and Mark K Transtrum. The limitations of model-based experimental design and parameter estimation in sloppy systems. *PLoS Computational Biology*, 12(12):e1005227, 2016.

[132] Nicholas Roehner, Eric M Young, Christopher A Voigt, D Benjamin Gordon, and Douglas Densmore. Double dutch: A tool for designing combinatorial libraries of biological systems. *ACS Synthetic Biology*, 5(6):507–517, 2016.

[133] Evan Appleton, Douglas Densmore, Curtis Madsen, and Nicholas Roehner. Needs and opportunities in bio-design automation: four areas for focus. *Current Opinion in Chemical Biology*, 40:111–118, 2017.

[134] Jacob Beal. Bridging the gap: a roadmap to breaking the biological design barrier. *Frontiers in Bioengineering and Biotechnology*, 2:87, 2015.

[135] Sarah Guiziou, Federico Ulliana, Violaine Moreau, Michel Leclere, and Jerome Bonnet. An automated design framework for multicellular recombinase logic. *ACS Synthetic Biology*, 7(5):1406–1412, 2018.

[136] Irene Otero-Muras, David Henriques, and Julio R Banga. Synbadm: a tool for optimization-based automated design of synthetic gene circuits. *Bioinformatics*, 32(21):3360–3362, 2016.

[137] Morgan Madec, Francois Pecheux, Yves Gendrault, Elise Rosati, Christophe Lallement, and Jacques Haiech. Geneda: An open-source workflow for design automation of gene regulatory networks inspired from microelectronics. *Journal of Computational Biology*, 23(10):841–855, 2016.

[138] Linh Huynh and Ilias Tagkopoulos. Fast and accurate circuit design automation through hierarchical model switching. *ACS Synthetic Biology*, 4(8):890–897, 2015.

[139] Guillermo Rodrigo and Alfonso Jaramillo. Autobiocad: full biodesign automation of genetic circuits. *ACS Synthetic Biology*, 2(5):230–236, 2012.

[140] Fusun Yaman, Swapnil Bhatia, Aaron Adler, Douglas Densmore, and Jacob Beal. Automated selection of synthetic biology parts for genetic regulatory networks. *ACS Synthetic Biology*, 1(8):332–344, 2012.

[141] Jacob Beal, Ron Weiss, Douglas Densmore, Aaron Adler, Evan Appleton, Jonathan Babb, Swapnil Bhatia, Noah Davidsohn, Traci Haddock, Joseph Loyall, et al. An end-to-end workflow for engineering of biological networks from high-level specifications. *ACS Synthetic Biology*, 1(8):317–331, 2012.

[142] Jacob Beal, Ting Lu, and Ron Weiss. Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PloS One*, 6(8):e22490, 2011.

[143] Barry Canton, Anna Labno, and Drew Endy. Refinement and standardization of synthetic biological parts and devices. *Nature Biotechnology*, 26(7):787, 2008.

[144] Jason R Kelly, Adam J Rubin, Joseph H Davis, Caroline M Ajo-Franklin, John Cumbers, Michael J Czar, Kim de Mora, Aaron L Glieberman, Dileep D Monie, and Drew Endy. Measuring the activity of biobrick promoters using an in vivo reference standard. *Journal of Biological Engineering*, 3(1):4, 2009.

[145] Noah Davidsohn, Jacob Beal, Samira Kiani, Aaron Adler, Fusun Yaman, Yinqing Li, Zhen Xie, and Ron Weiss. Accurate predictions of genetic circuit behavior from part characterization and modular composition. *ACS Synthetic Biology*, 4(6):673–681, 2014.

[146] Stefano Cardinale and Adam Paul Arkin. Contextualizing context for synthetic biology–identifying causes of failure of synthetic biological systems. *Biotechnology Journal*, 7(7):856–866, 2012.

[147] Stefan Klumpp, Zhongge Zhang, and Terence Hwa. Growth rate-dependent global effects on gene expression in bacteria. *Cell*, 139(7):1366–1375, 2009.

[148] Matthew Scott, Carl W Gunderson, Eduard M Mateescu, Zhongge Zhang, and Terence Hwa. Interdependence of cell growth and gene expression: origins and consequences. *Science*, 330(6007):1099–1102, 2010.

[149] Conghui You, Hiroyuki Okano, Sheng Hui, Zhongge Zhang, Minsu Kim, Carl W Gunderson, Yi-Ping Wang, Peter Lenz, Dalai Yan, and Terence Hwa. Coordination of bacterial proteome with metabolism by cyclic amp signalling. *Nature*, 500(7462):301, 2013.

[150] Sheng Hui, Josh M Silverman, Stephen S Chen, David W Erickson, Markus Basan, Jilong Wang, Terence Hwa, and James R Williamson. Quantitative proteomic analysis reveals a simple strategy of global resource allocation in bacteria. *Molecular Systems Biology*, 11(2):784, 2015.

[151] Andrea Y Weiße, Diego A Oyarzún, Vincent Danos, and Peter S Swain. Mechanistic links between cellular trade-offs, gene expression, and growth. *Proceedings of the National Academy of Sciences*, page 201416533, 2015.

[152] Javier Carrera, Guillermo Rodrigo, Vijai Singh, Boris Kirov, and Alfonso Jaramillo. Empirical model and in vivo characterization of the bacterial response to synthetic gene expression show that ribosome allocation limits growth rate. *Biotechnology Journal*, 6(7):773–783, 2011.

[153] Chen Liao, Andrew E Blanchard, and Ting Lu. An integrative circuit–host modelling framework for predicting synthetic gene network behaviours. *Nature Microbiology*, 2(12):1658, 2017.

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

[155] MD Hoang, T Barz, VA Merchan, LT Biegler, and H Arellano-Garcia. Simultaneous solution approach to model-based experimental design. *AIChE Journal*, 59(11):4169–4183, 2013.

[156] Dennis Janka, Stefan Körkel, and Hans Georg Bock. Direct multiple shooting for nonlinear optimum experimental design. In *Multiple Shooting and Time Domain Decomposition Methods*, pages 115–141. Springer, 2015.

[157] Dries Telen, Dominique Vercammen, Filip Logist, and Jan Van Impe. Robustifying optimal experiment design for nonlinear, dynamic (bio) chemical systems. *Computers & Chemical Engineering*, 71:415–425, 2014.

[158] Zoltan Kutalik, Kwang-Hyun Cho, and Olaf Wolkenhauer. Optimal sampling time selection for parameter estimation in dynamic pathway modeling. *Biosystems*, 75(1-3):43–55, 2004.

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

[160] Fangwei Si, Dongyang Li, Sarah E Cox, John T Sauls, Omid Azizi, Cindy Sou, Amy B Schwartz, Michael J Erickstad, Yonggun Jun, Xintian Li, et al. Invariance of initiation mass and predictability of cell size in *Escherichia coli*. *Current Biology*, 27(9):1278–1287, 2017.

[161] Stephen Cooper and Charles E Helmstetter. Chromosome replication and the division cycle of *Escherichia coli* Br. *Journal of Molecular Biology*, 31(3):519–540, 1968.

[162] H Bremer and G Churchward. An examination of the cooper-helmstetter theory of DNA replication in bacteria and its underlying assumptions. *Journal of Theoretical Biology*, 69(4):645–654, 1977.

[163] Herbert E Kubitschek, William W Baldwin, Sally J Schroeter, and Rheinhard Graetzer. Independence of buoyant cell density and growth rate in *Escherichia coli*. *Journal of Bacteriology*, 158(1):296–299, 1984.

[164] Markus Basan, Manlu Zhu, Xiongfeng Dai, Mya Warren, Daniel Sévin, Yi-Ping Wang, and Terence Hwa. Inflating bacterial cells by increased protein synthesis. *Molecular Systems Biology*, 11(10):836, 2015.

[165] Robert D Finn, Elena V Orlova, Brent Gowen, Martin Buck, and Marin van Heel. *Escherichia coli* RNA polymerase core and holoenzyme structures. *The EMBO Journal*, 19(24):6833–6844, 2000.

[166] Stefan Klumpp and Terence Hwa. Growth-rate-dependent partitioning of RNA polymerases in bacteria. *Proceedings of the National Academy of Sciences*, 105(51):20245–20250, 2008.

[167] Michael Patrick, Patrick P Dennis, Mans Ehrenberg, and Hans Bremer. Free RNA polymerase in *Escherichia coli*. *Biochimie*, 119:80–91, 2015.

[168] Lacramioara Bintu, Nicolas E Buchler, Hernan G Garcia, Ulrich Gerland, Terence Hwa, Jané Kondev, and Rob Phillips. Transcriptional regulation by the numbers: models. *Current Opinion in Genetics & Development*, 15(2):116–124, 2005.

[169] Mattias Rydenfelt, Robert Sidney Cox III, Hernan Garcia, and Rob Phillips. Statistical mechanical model of coupled transcription from multiple promoters due to transcription factor titration. *Physical Review E*, 89(1):012702, 2014.

[170] Rob Phillips. Napoleon is in equilibrium. *Annu. Rev. Condens. Matter Phys.*, 6(1):85–111, 2015.

[171] Ewa Heyduk, Konstantin Kuznedelov, Konstantin Severinov, and Tomasz Heyduk. A consensus adenine at position–11 of the nontemplate strand of bacterial promoter is important for nucleation of promoter melting. *Journal of Biological Chemistry*, 281(18):12362–12369, 2006.

[172] Michael Brunner and Hermann Bujard. Promoter recognition and promoter strength in the *Escherichia coli* system. *The EMBO Journal*, 6(10):3139–3144, 1987.

[173] Marko Djordjevic and Ralf Bundschuh. Formation of the open complex by bacterial RNA polymerase—a quantitative model. *Biophysical Journal*, 94(11):4233–4248, 2008.

[174] Marko Djordjevic. Efficient transcription initiation in bacteria: an interplay of protein–dna interaction parameters. *Integrative Biology*, 5(5):796–806, 2013.

[175] Gary D Stormo. *Introduction to protein-DNA interactions: structure, thermodynamics, and bioinformatics.* Cold Spring Harbor Laboratory Press, 2013.

[176] SR Kushner. Messenger RNA decay. *EcoSal Plus*, 2(2), 2007.

[177] Chaitanya Jain and Joel G Belasco. RNase E autoregulates its synthesis by controlling the degradation rate of its own mRNA in *Escherichia coli*: unusual sensitivity of the rne transcript to RNase E activity. *Genes & Development*, 9(1):84–96, 1995.

[178] Elisabeth A Mudd and Christopher F Higgins. *Escherichia coli* endoribonuclease rnase e: autoregulation of expression and site-specific cleavage of mRNA. *Molecular Microbiology*, 9(3):557–568, 1993.

[179] Chaitanya Jain, Atilio Deana, and Joel G Belasco. Consequences of RNase E scarcity in *Escherichia coli*. *Molecular Microbiology*, 43(4):1053–1064, 2002.

[180] Maria C Ow, Qi Liu, Bijoy K Mohanty, Margaret E Andrew, Valerie F Maples, and Sidney R Kushner. Rnase e levels in *Escherichia coli* are controlled by a complex regulatory system that involves transcription of the rne gene from three promoters. *Molecular Microbiology*, 43(1):159–171, 2002.

[181] Huiyi Chen, Katsuyuki Shiroguchi, Hao Ge, and Xiaoliang Sunney Xie. Genome-wide study of mRNA degradation and transcript elongation in *Escherichia coli*. *Molecular Systems Biology*, 11(1):781, 2015.

[182] Steen Pedersen, Solvejg Reeh, and James D Friesen. Functional mRNA half lives in *E. coli*. *Molecular and General Genetics MGG*, 166(3):329–336, 1978.

[183] Douglas W Selinger, Rini Mukherjee Saxena, Kevin J Cheung, George M Church, and Carsten Rosenow. Global RNA half-life analysis in *Escherichia coli* reveals positional patterns of transcript degradation. *Genome Research*, 13(2):216–223, 2003.

[184] George A Mackie. Rnase e: at the interface of bacterial RNA processing and decay. *Nature Reviews Microbiology*, 11(1):45, 2013.

[185] Jeremy M Berg, John L Tymoczko, and Lubert Stryer. *Biochemistry*. WH Freeman, 5 edition, 2002.

[186] Xiongfeng Dai, Manlu Zhu, Mya Warren, Rohan Balakrishnan, Vadim Patsalo, Hiroyuki Okano, James R Williamson, Kurt Fredrick, Yi-Ping Wang, and Terence Hwa. Reduction of translating ribosomes enables *Escherichia coli* to maintain elongation rates during slow growth. *Nature Microbiology*, 2(2):16231, 2017.

[187] Olivier Borkowski, Anne Goelzer, Marc Schaffer, Magali Calabre, Ulrike Mäder, Stéphane Aymerich, Matthieu Jules, and Vincent Fromion. Translation elicits a growth rate-dependent, genome-wide, differential protein production in *Bacillus subtilis*. *Molecular Systems Biology*, 12(5):870, 2016.

[188] David Kennell and Howard Riezman. Transcription and translation initiation frequencies of the *Escherichia coli* lac operon. *Journal of Molecular Biology*, 114(1):1–21, 1977.

[189] Sang Woo Seo, Jae-Seong Yang, Inhae Kim, Jina Yang, Byung Eun Min, Sanguk Kim, and Gyoo Yeol Jung. Predictive design of mRNA translation initiation region to control prokaryotic translation efficiency. *Metabolic Engineering*, 15:67–74, 2013.

[190] Howard M Salis, Ethan A Mirsky, and Christopher A Voigt. Automated design of synthetic ribosome binding sites to control protein expression. *Nature biotechnology*, 27(10):946–950, 2009.

[191] Federico Garza de Leon, Laura Sellars, Mathew Stracy, Stephen JW Busby, and Achillefs N Kapanidis. Tracking low-copy transcription factors in living bacteria: the case of the lac repressor. *Biophysical Journal*, 112(7):1316–1327, 2017.

[192] Sebastian Sager, Hans Georg Bock, and Moritz Diehl. The integer approximation error in mixed-integer optimal control. *Mathematical Programming*, 133(1-2):1–23, 2012.

[193] Sebastian Sager. Sampling decisions in optimum experimental design in the light of pontryagin's maximum principle. *SIAM Journal on Control and Optimization*, 51(4):3181–3207, 2013.

[194] Samuel S Wilks. Certain generalizations in the analysis of variance. *Biometrika*, pages 471–494, 1932.

[195] Michele Vallisneri. Use and abuse of the fisher information matrix in the assessment of gravitational-wave parameter-estimation prospects. *Physical Review D*, 77(4):042001, 2008.

[196] 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.

[197] 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.

[198] Irene Bauer, Hans Georg Bock, Stefan Körkel, and Johannes P Schlöder. Numerical methods for optimum experimental design in dae systems. *Journal of Computational and Applied mathematics*, 120(1-2):1–25, 2000.

[199] Federico Galvanin, Massimiliano Barolo, and Fabrizio Bezzo. Online model-based redesign of experiments for parameter estimation in dynamic systems. *Industrial & Engineering Chemistry Research*, 48(9):4415–4427, 2009.

[200] Lucia Bandiera, Zhaozheng Hou, Varun Kothamachu, Eva Balsa-Canto, Peter Swain, and Filippo Menolascina. On-line optimal input design increases the efficiency and accuracy of the modelling of an inducible synthetic promoter. *Processes*, 6(9):148, 2018.

[201] Thomas E Gorochowski, Amin Espah Borujeni, Yongjin Park, Alec AK Nielsen, Jing Zhang, Bryan S Der, D Benjamin Gordon, and Christopher A Voigt. Genetic circuit characterization and debugging using RNA-seq. *Molecular Systems Biology*, 13(11):952, 2017.

[202] Jacob Beal, Traci Haddock-Angelli, Geoff Baldwin, Markus Gershater, Ari Dwijayanti, Marko Storch, Kim de Mora, Meagan Lizarazo, Randy Rettberg, et al. Quantification of bacterial fluorescence using independent calibrants. *PloS One*, 13(6):e0199432, 2018.

[203] Naveen Venayak, Nikolaos Anesiadis, William R Cluett, and Radhakrishnan Mahadevan. Engineering metabolism through dynamic control. *Current Opinion in Biotechnology*, 34:142–152, 2015.

[204] Di Liu, Ahmad A Mannan, Yichao Han, Diego A Oyarzún, and Fuzhong Zhang. Dynamic metabolic control: towards precision engineering of metabolism. *Journal of Industrial Microbiology & Biotechnology*, pages 1–9, 2018.

[205] Sue Zanne Tan and Kristala LJ Prather. Dynamic pathway regulation: recent advances and methods of construction. *Current Opinion in Chemical Biology*, 41:28–35, 2017.

[206] Stephanie J Doong, Apoorv Gupta, and Kristala LJ Prather. Layered dynamic regulation for improving metabolic pathway productivity in *Escherichia coli*. *Proceedings of the National Academy of Sciences*, pages 2964–2969, 2018.

[207] Apoorv Gupta, Irene M Brockman Reizman, Christopher R Reisch, and Kristala LJ Prather. Dynamic regulation of metabolic flux in engineered bacteria using a pathway-independent quorum-sensing circuit. *Nature Biotechnology*, 35(3):273, 2017.

[208] Yuki Soma and Taizo Hanai. Self-induced metabolic state switching by a tunable cell density sensor for microbial isopropanol production. *Metabolic Engineering*, 30:7–15, 2015.

[209] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[210] Hidde De Jong, Caroline Ranquet, Delphine Ropers, Corinne Pinel, and Johannes Geiselmann. Experimental and computational validation of models of fluorescent and luminescent reporter genes in bacteria. *BMC systems biology*, 4(1):55, 2010.

[211] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, March 2014.

[212] Domitilla Del Vecchio, Aaron J Dy, and Yili Qian. Control theory meets synthetic biology. *Journal of The Royal Society Interface*, 13(120):20160380, 2016.

[213] Anthony C Atkinson and VV Fedorov. Some history leading to design criteria for bayesian prediction. In *Optimum Design 2000*, pages 3–14. Springer, 2001.

[214] Sayuri Katharina Hortsch and Andreas Kremling. Characterization of noise in multistable genetic circuits reveals ways to modulate heterogeneity. *PloS one*, 13(3):e0194779, 2018.

[215] Irene Otero-Muras, Pencho Yordanov, and Joerg Stelling. A method for inverse bifurcation of biochemical switches: inferring parameters from dose response curves. *BMC Systems Biology*, 8(1):114, 2014.

[216] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[217] Nicolaas Godfried Van Kampen. *Stochastic processes in physics and chemistry*. Elsevier, 1992.

[218] Ryota Tomioka, Hidenori Kimura, Tetsuya J Kobayashi, and Kazuyuki Aihara. Multivariate analysis of noise in genetic regulatory networks. *Journal of Theoretical Biology*, 229(4):501–521, 2004.

[219] Nils Berglund. Kramers' law: Validity, derivations and generalisations. *arXiv preprint arXiv:1106.5799*, 2011.

[220] Luc Pronzato and Andrej Pázman. *Design of experiments in nonlinear models*. Springer, 2013.

[221] John A Hartigan, Pamela M Hartigan, et al. The dip test of unimodality. *The Annals of Statistics*, 13(1):70–84, 1985.

[222] Ian Ford, DM Titterington, and Christos P Kitsos. Recent advances in nonlinear experimental design. *Technometrics*, 31(1):49–60x, 1989.

[223] Luc Pronzato. Optimal experimental design and some related control problems. *Automatica*, 44(2):303–325, 2008.

[224] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2019.

[225] 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.

[226] 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.

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

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

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

[230] 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.

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

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

[233] 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.

[234] 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.

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

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

[237] 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.

[238] 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.

[239] 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.

[240] 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.

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

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

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

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

[245] George EP Box and Norman R Draper. Robust designs. *Biometrika*, 62(2):347–352, 1975.

[246] Vivek K Mutalik, Joao C Guimaraes, Guillaume Cambray, Colin Lam, Marc Juul Christoffersen, Quynh-Anh Mai, Andrew B Tran, Morgan Paull, Jay D Keasling, Adam P Arkin, et al. Precise and reliable gene expression via standard transcription and translation initiation elements. *Nature methods*, 10(4):354–360, 2013.

[247] H Bremer, P Dennis, and M Ehrenberg. Free RNA polymerase and modeling global transcription in *Escherichia coli. Biochimie*, 85(6):597–609, 2003.

[248] S-T Liang, M Bipatnath, Y-C Xu, S-L Chen, P Dennis, M Ehrenberg, and H Bremer. Activities of constitutive promoters in *Escherichia coli. Journal of molecular biology*, 292(1):19–37, 1999.

[249] Antti Häkkinen and Andre S Ribeiro. Characterizing rate limiting steps in transcription from RNA production times in live cells. *Bioinformatics*, 32(9):1346–1352, 2015.

[250] Anantha-Barathi Muthukrishnan, Meenakshisundaram Kandhavelu, Jason Lloyd-Price, Fedor Kudasov, Sharif Chowdhury, Olli Yli-Harja, and Andre S Ribeiro. Dynamics of transcription driven by the teta promoter, one event at a time, in live *Escherichia coli* cells. *Nucleic acids research*, 40(17):8472–8483, 2012.

[251] Meenakshisundaram Kandhavelu, Henrik Mannerström, Abhishekh Gupta, Antti Häkkinen, Jason Lloyd-Price, Olli Yli-Harja, and Andre S Ribeiro. In vivo kinetics of transcription initiation of the lar promoter in *Escherichia coli.* evidence for a sequential mechanism with two rate-limiting steps. *BMC systems biology*, 5(1):149, 2011.

[252] S-T Liang, Y-C Xu, P Dennis, and Hans Bremer. mRNA composition and control of bacterial gene expression. *Journal of Bacteriology*, 182(11):3037–3044, 2000.

[253] Grzegorz Kudla, Andrew W Murray, David Tollervey, and Joshua B Plotkin. Coding-sequence determinants of gene expression in *Escherichia coli. science*, 324(5924):255–258, 2009.

[254] Brian Søgaard Laursen, Hans Peter Sørensen, Kim Kusk Mortensen, and Hans Uffe Sperling-Petersen. Initiation of protein synthesis in bacteria. *Microbiology and molecular biology reviews*, 69(1):101–123, 2005.

[255] John T Betts. Survey of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998.

[256] Matthew Kelly. An introduction to trajectory optimization: how to do your own direct collocation. *SIAM Review*, 59(4):849–904, 2017.

[257] H Diedam and S Sager. Global optimal control with the direct multiple shooting method. *Optimal Control Applications and Methods*, 39(2):449–470, 2018.

[258] MultiMedia LLC. Andersseon, joel and gillis, joris and diehl, mortiz, 2018.

[259] Joel Andersseon, Joris Gillis, and Greg Horn. Casadi examples package, 2018.

[260] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[261] Hans Georg Bock, Ekaterina Kostina, and Johannes P Schlöder. Numerical methods for parameter estimation in nonlinear differential algebraic equations. *GAMM-Mitteilungen*, 30(2):376–408, 2007.

[262] Hans Georg Bock, Stefan Körkel, and Johannes P Schlöder. Parameter estimation and optimum experimental design for differential equation models. In *Model based parameter estimation*, pages 1–30. Springer, 2013.

APPENDICES

Appendix A

Supplementary Materials: Component Characterization in a Growth-Dependent Physiological Context: Optimal Experimental Design

Supplementary materials for the publication "Component Characterization in a Growth-Dependent Physiological Context: Optimal Experimental Design" by Nathan Braniff, Matt Scott, and Brian Ingalls in *Processes* 7.1 (2019): 52.

A.1 Derivation of the Physiological Gene Expression Model

A.1.1 Protein fraction of cell mass

The total cell mass M_{Tot} can be partitioned into fractions of protein and other constituents. The protein fraction of the cellular mass, Φ_{pr}, can be fit to data from [2] with a linear function:

$$\Phi_{pr} = \kappa_{pr}\lambda + \Phi_{pr0} \tag{A.1}$$

The fit shown in Fig. A.1 provides estimates of $\kappa_{pr} = -6.47$ min and $\Phi_{pr0} = 0.65$.

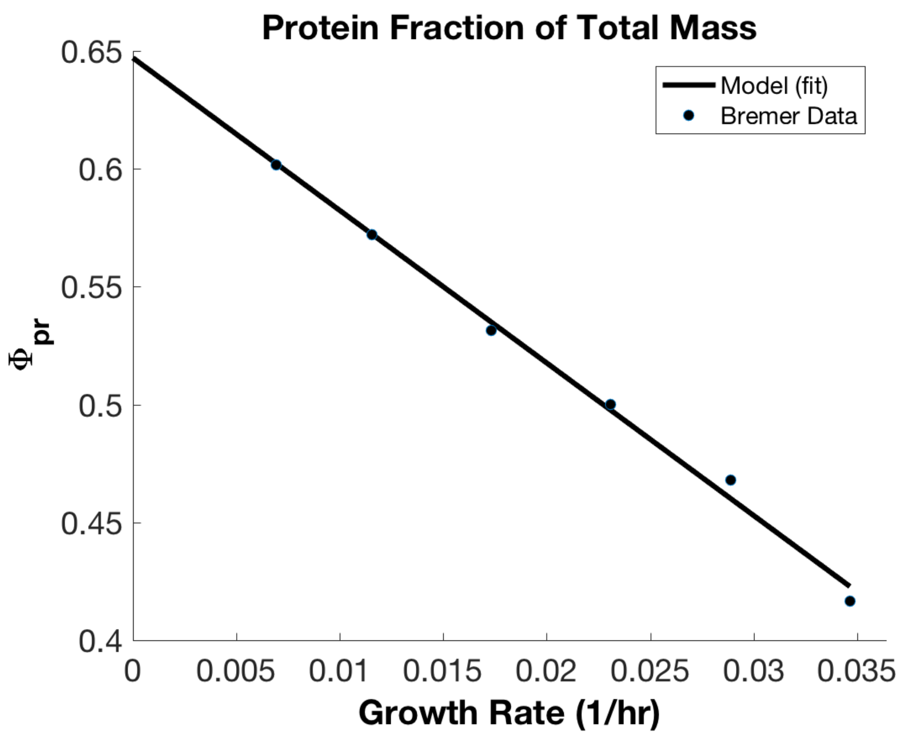


Figure A.1: Fit of protein fraction of the cell mass using data from Table 2 of [2] .

A.1.2 Growth Dependence of the Total RNAP Population

The total RNAP fraction of the overall protein mass, Φ_p, exhibits an approximately linear relationship with growth rate:

$$\Phi_p = \kappa_p \lambda + \Phi_{p0}. \tag{A.2}$$

We fit this to data provided in [2], yielding estimates $\kappa_p = 0.30$ min and $\Phi_{p0} = 0.0074$, shown in Figure A.2.

A.1.3 Available RNAP

The total RNAP population can be partitioned by state: freely diffusing; weakly DNA bound at a non-specific site; actively transcribing other genes; paused and non-functioning

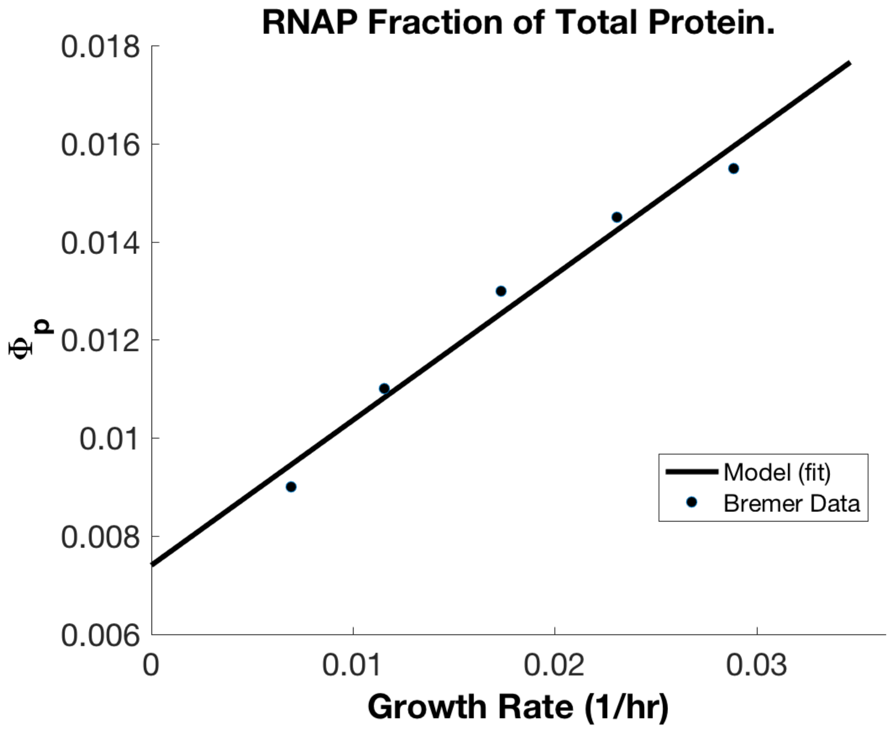


Figure A.2: Fit of RNAP fraction of protein mass using data from Table 3 of [2].

during transcription (paused); or immature [5, 167, 166]. We are interested in those RNAP that can initiate transcription. It has been assumed in past works that transcriptional initiation is proportional to the free fraction [166]. However, our thermodynamic equilibrium model of the promoter accounts for competition between non-specific binding sites and the specific promoter site. We therefore define the combined pool of free and non-specifically bound RNAPs as the *available* RNAP pool, P_a.

We use a course-grained partitioning of total RNAPs, P_{Tot}: dividing them into i) the available RNAPs, P_a, ii) the bound RNAPs, P_b. We will neglect the immature population as this has been measured to be a small fraction ($< 10\%$) of the total [5]. We can therefore write the total as

$$P_{Tot} \approx P_a + P_b \tag{A.3}$$

The available subgroup, P_a, includes those RNAPs freely diffusing in the nucleoid as well as those non-specifically bound to the DNA. These two sub-groups within the available pool, P_a, have been observed to be in rapid equilibrium [5]. We assume that the bound

RNAPs, P_b, include all those bound to the DNA that are actively transcribing or paused in transcription. We write

$$P_a = P_{Tot} - P_b = P_{Tot}(1 - \Phi_b) = P_{Tot}\Phi_a, \tag{A.4}$$

where Φ_b is the fraction of transcription-occupied RNAPs unavailable for initiation of transcription and Φ_a is the faction of those that are available.

The growth-dependent fraction of RNAPs that are available to initiate transcription (non-transcribing) is still poorly understood. Work by Klumpp and Hwa [166] as well as Bremer and colleagues [247, 167] have attempted to describe the partitioning of RNAP into free and occupied fractions across growth rates without consensus. However, all agree the concentration of free RNAP increases with growth rate. Recent spatial imaging experiments of fluorescently-tagged RNAP suggest these previous theories may be partially inaccurate; specifically, this new data suggest much larger fractions of RNAP are busy in transcription, and smaller fractions are non-specifically bound or paused, than previously expected [5, 3].

Current direct measurements of the dependence of Φ_b or Φ_a on growth rate are sparse. Bakshi *et al.* examine only a single growth rate (doubling time of approximately 42 min) at which they estimate the partitioning of RNAP using spatial tracking of tagged molecules [5]. Stracy *et al.* have since compared RNAP partitioning between growth on minimal and rich media in a spatial tracking study [3]. A plot of these three data points is shown in Figure A.3. It should be noted that the studies use somewhat different experimental methodologies, and future work with multiple growth rates in the same strain and conditions is needed to provide a confident description. Growth rates for strains in Stracy *et al.* were previously reported in [4]. From the limited available data (Figure A.3) we hypothesize a linear dependence for Φ_b:

$$\Phi_b = \kappa_b\lambda + \Phi_{b0}. \tag{A.5}$$

Fitting yields the estimates $\kappa_b = 9.3$ min and $\Phi_{b0} = 0.41$. Then $\Phi_{a0} = 1 - \Phi_{b0} = 0.59$ and $\kappa_a = -\kappa_b = -9.3$ min.

We can then re-write this relationship as

$$\Phi_a = \kappa_a\lambda + \Phi_{a0}. \tag{A.6}$$

Using this relation we can construct an expression for the available RNAP by multiplying P_{Tot} by Φ_a;

$$P_a = \frac{\rho V_0}{m_{rnap}} \left(\kappa_a\lambda + \Phi_{a0}\right)\left(\kappa_p\lambda + \Phi_{p0}\right)\left(\kappa_{pr}\lambda + \Phi_{pr0}\right) e^{(C+D)\lambda}. \tag{A.7}$$

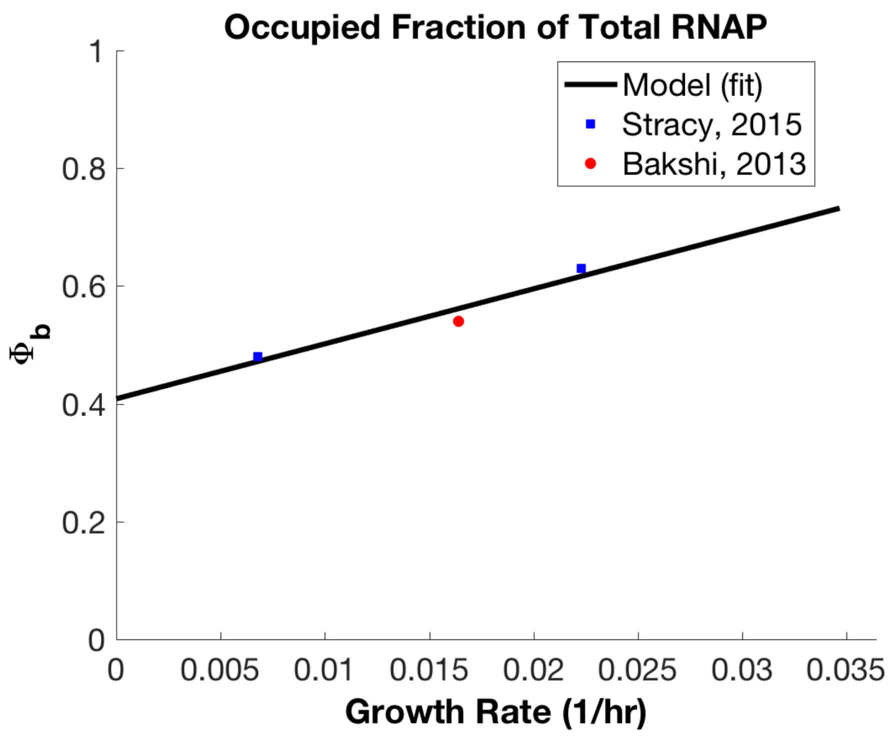


Figure A.3: Fraction of total RNAP occupied in transcription, as it depends on growth rate. Data from [3], [4] and [5].

Dividing the expression for the available RNAP by the expression for cell volume yields the concentration of available RNAP, which our model predicts will decrease with growth rate. Past works have observed an increasing transcription rate per gene (particularly moving from slow to moderate growth rates) [147]. This suggests that the free RNAP concentration must therefore be increasing with the growth rate. However, although our model predicts a decreasing RNAP concentration, the transcription rate is predicted to increase (from slow to moderate growth) as the RNAP density along the genomic DNA increases. This is consistent with the fact that DNA-binding proteins like RNAP and transcription factors (TFs) are mostly confined to the nucleoid DNA [5, 191] where they diffuse along the DNA strand. As a result, in our model, total RNAP concentration is less relevant than the RNAP density along the genomic DNA. We hypothesize this may cause the non-monotonic relation for transcription rate per gene that has previously been observed by Liao *et al.* for constitutive promoters [248].

A.1.4 Transcription Rate

Following [168, 169], our model of transcription involves interactions between RNA polymerases (RNAPs), transcription factors (TFs), promoter copies and non-specific binding sites along the genomic DNA. As described in the main text, at each time point we suppose that there are P_a available RNAP copies and T_a active transcription factor copies diffusing along the genomic DNA, and that the DNA contains N_s non-specific binding sites to which the DNA binding proteins may weakly attach and g copies of the regulated promoter of interest. Further we assume that $N_s \gg P_a, T_a, g$ and that each binding of an RNAP or a TF to a non-specific site or a promoter can be characterized by an associated binding energy: ϵ_{rn} and ϵ_{rg} for RNAP to the non-specific sites and promoters respectively, and ϵ_{tn} and ϵ_{tg} for transcription factor to non-specific sites and promoters respectively (all ϵ are negative [170]).

We use these species and site counts to enumerate the possible arrangements of RNAP and TF across the genome, and we use the binding energies to derive Boltzmann weights for each arrangement [168]. This allows us to construct a partition function. For example if all the P_a RNAPs and u TFs are bound to nonspecific sites, the weighted enumeration for this group of possible micro-states (the partition function) can be written as

$$Z(P_a, u) = \underbrace{\frac{N_s!}{P_a! T_a! (N_s - P_a - T_a)!}}_{\text{\# of micro states}} \underbrace{e^{-P_a \frac{\epsilon_{rn}}{k_B T}} e^{-T_a \frac{\epsilon_{tn}}{k_B T}}}_{\text{energetic favorability}} \approx \frac{N_s^{(P_a+T_a)}}{P_a! T_a!} e^{-P_a \frac{\epsilon_{rn}}{k_B T}} e^{-T_a \frac{\epsilon_{tn}}{k_B T}}, \tag{A.8}$$

where the approximation holds because N_s is large compared with the other quantities [168]. We can then construct the partition function for the total number of arrangements of a single promoter copy ($g = 1$) as

$$\begin{aligned} Z_{g=1}^{Tot}(P_a, T_a) = &\underbrace{Z(P_a, T_a)}_{\text{Empty Promoters}} + \underbrace{Z(P_a - 1, T_a) e^{-\frac{\epsilon_{pg}}{k_B T}}}_{\text{RNAP on Promoter}} \\ &+ \underbrace{Z(P_a, T_a - 1) e^{-\frac{\epsilon_{tg}}{k_B T}}}_{\text{TF on Promoter}} + \underbrace{Z(P_a - 1, T_a - 1) e^{-\frac{\epsilon_{pg}+\epsilon_{tg}+\epsilon_{pt}}{k_B T}}}_{\text{RNAP and TF on Promoter}}. \end{aligned} \tag{A.9}$$

Here ϵ_{pt} is the binding energy between RNA polymerase and transcription factor when both are bound to the same promoter. We can use this expression to write the equilibrium probability of the single promoter being occupied by an RNAP by taking the ratio of the partition functions for the RNAP-bound states to the total partition function;

$$p_{bnd} = \frac{Z_1^{Bnd}(P_a, T_a)}{Z_1^{Tot}(P_a, T_a)}$$

$$p_{bnd} = \frac{\frac{P_a}{N_s} e^{-\frac{\Delta\epsilon_r}{k_B T}} + \frac{P_a T_a}{N_s^2} e^{-\frac{(\Delta\epsilon_r + \Delta\epsilon_t + \epsilon_{pt})}{k_B T}}}{1 + \frac{P_a}{N_s} e^{-\frac{\Delta\epsilon_r}{k_B T}} + \frac{T_a}{N_s} e^{-\frac{\Delta\epsilon_t}{k_B T}} + \frac{P_a T_a}{N_s^2} e^{-\frac{(\Delta\epsilon_r + \Delta\epsilon_t + \epsilon_{pt})}{k_B T}}} \tag{A.10}$$

Here the $\Delta\epsilon$ values are the differences between the energy involved in binding the promoter and the background non-specific binding: $\Delta\epsilon_t = \epsilon_{tg} - \epsilon_{tn}$ and $\Delta\epsilon_r = \epsilon_{rg} - \epsilon_{rn}$. Note, $\epsilon_{tn} > \epsilon_{tg}$ and $\epsilon_{pn} > \epsilon_{pg}$ so that $\Delta\epsilon_t$ and $\Delta\epsilon_r$ are both negative [170]. Denoting the Boltzmann weights as $K_r = e^{-\frac{\Delta\epsilon_r}{k_B T}}$, $K_t = e^{-\frac{\Delta\epsilon_t}{k_B T}}$ and $K_{rt} = e^{-\frac{(\Delta\epsilon_r + \Delta\epsilon_t + \epsilon_{pt})}{k_B T}}$ yields the following simplified form;

$$p_{bound} = \frac{\frac{P_a}{N_s} K_r + \frac{P_a T_a}{N_s^2} K_{rt}}{1 + \frac{P_a}{N_s} K_r + \frac{T_a}{N_s} K_t + \frac{P_a T_a}{N_s^2} K_{rt}} \tag{A.11}$$

Next, with RNAP bound to a certain fraction of the promoters (or on average a certain fraction of the time over the relevant times scale of initiation), we assume open complex and promoter escape occurs at a fixed rate α (NATE cite for open complex and escape rate), giving

$$\text{Initiation Rate (for } g\text{=1)} = \alpha \frac{\frac{P_a}{N_s} K_r + \frac{P_a T_a}{N_s^2} K_{rt}}{1 + \frac{P_a}{N_s} K_r + \frac{T_a}{N_s} K_t + \frac{P_a T_a}{N_s^2} K_{rt}} \tag{A.12}$$

So far, we have described a single promoter. To address the case of multiple promoters we would need to account for the cross-correlation between their occupancy by the transcription factor or RNAP. This has little effect at high TF copy numbers (although at low copy numbers the occupancy of one promoter significantly decreases the odds of another being occupied). We assume that the RNAP and TF populations are considerably larger than g, which allows us to assume the promoters function approximately independently [169]. In that case, we can scale equation (A.12) to arrive at the initiation rate as a function of g.

$$\text{Initiation Rate} = \alpha g \frac{\frac{P_a}{N_s} K_r + \frac{P_a T_a}{N_s^2} K_{rt}}{1 + \frac{P_a}{N_s} K_r + \frac{T_a}{N_s} K_t + \frac{P_a T_a}{N_s^2} K_{rt}} \tag{A.13}$$

We assume that the initiation rate is the limiting step in transcription, as elongation rates are generally faster [249, 250, 251]. We can therefore write the overall transcript production rate as follows;

$$\text{Transcript Production Rate} = \alpha g \frac{\frac{P_a}{N_s}K_r + \frac{P_a T_a}{N_s^2}K_{rt}}{1 + \frac{P_a}{N_s}K_r + \frac{T_a}{N_s}K_t + \frac{P_a T_a}{N_s^2}K_{rt}}$$

Where:

$$\begin{aligned} P_a &= \frac{\rho V_0}{m_{rnap}} \left(\kappa_a \lambda + \Phi_{a0}\right) \left(\kappa_p \lambda + \Phi_{p0}\right) \left(\kappa_{pr} \lambda + \Phi_0^{Prot}\right) e^{(C+D)\lambda}, \\ N_s &= \frac{\eta}{\lambda C}(e^{(C+D)\lambda} - e^{D\lambda}) \\ g &= e^{((C+D)-l_{ori}C)\lambda} \end{aligned} \tag{A.14}$$

A.1.5 Total Ribosome Population

From [148], we have a linear relation for the fraction of protein mass that is composed of ribosomal protein:

$$\Phi_r = \kappa_r \lambda + \Phi_{r0}. \tag{A.15}$$

Fitting the model to data from [2] yields estimates of $\kappa_r = 5.5$ min and $\Phi_{r0} = 0.030$, as shown in Figure A.4.

A.1.6 Translation Rate

Both Klumpp *et al.* and Liang *et al.* note that the translation rate per transcript is roughly constant and speculate that this could be due to a constant concentration of free ribosomes maintained by regulatory feedback [166, 252]. Dai's more recent results suggest that the fraction of inactive (non-translating) ribosomes, Φ_{inact}, is constant at the moderate to fast growth rates we consider here [186]. The total ribosome concentration increases with growth rate, because total ribosome copy number scales faster than the cell volume:

$$\begin{aligned} \frac{R_{Tot}}{V} &= \left(\frac{\rho V_0}{m_{rib}} \left(\kappa_r \lambda + \Phi_{r0}\right) \left(\kappa_{pr} \lambda + \Phi_{pr0}\right) e^{(C+D)\lambda}\right) \left(V_0 e^{(C+D)\lambda}\right)^{-1} \\ &= \frac{\rho}{m_{rib}} \left(\kappa_r \lambda + \Phi_{r0}\right) \left(\kappa_{pr} \lambda + \Phi_{pr0}\right) \end{aligned} \tag{A.16}$$

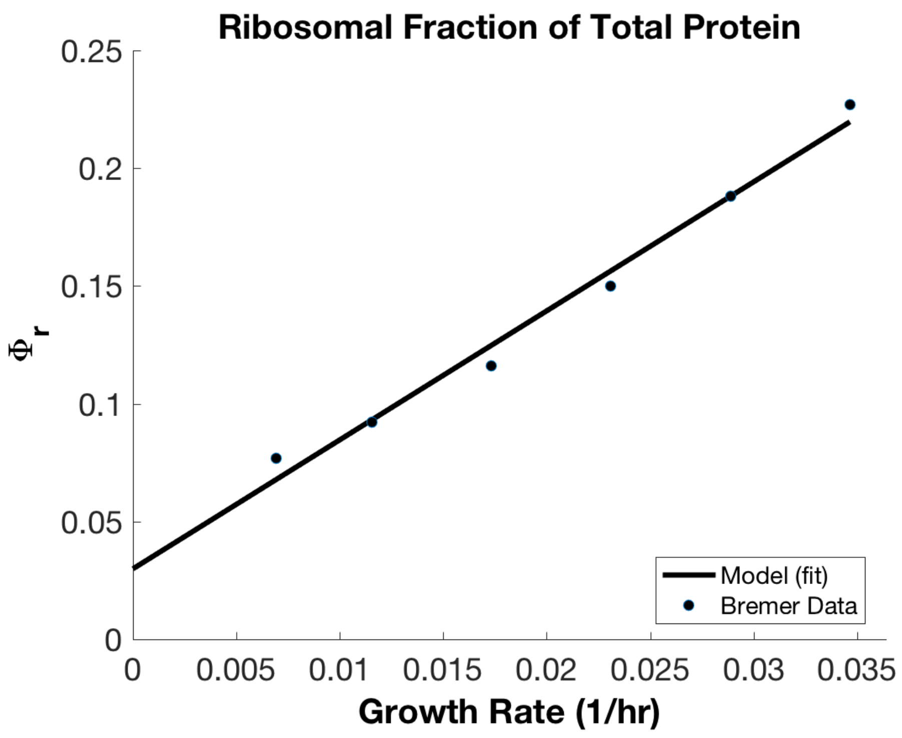


Figure A.4: Ribosomal protein fraction of the total protein mass as a function of growth rate. Data from Table 3 of [2].

Therefore, the inactive ribosome concentration $[R_{inact}]$ also increases, because Φ_{inact} is constant:

$$\frac{R_{inact}}{V} = \Phi_{inact}\frac{R_{Tot}}{V} = \Phi_{inact}\frac{\rho}{m_{rib}}\left(\kappa_r\lambda + \Phi_{r0}\right)\left(\kappa_{pr}\lambda + \Phi_{pr0}\right) \tag{A.17}$$

The inactive ribosomes R_{inact} are either non-functioning (stalled or assembling), R_{nf}, or free, R_f, so

$$R_{inact} = R_{nf} + R_f \tag{A.18}$$

Following [166, 252], we presume that the concentration of free ribosomes, $[R_f]$ is constant. Thus the fraction of inactive ribosomes that are free, Φ_f must scale inversely with the ribosomal fraction of the mass:

$$\Phi_f = \frac{1}{\left(\kappa_r\lambda + \Phi_{r0}\right)\left(\kappa_{pr}\lambda + \Phi_{pr0}\right)} \tag{A.19}$$

and therefore the fraction of inactive ribosomes, Φ_{nf} is

$$\Phi_{nf} = 1 - \frac{1}{(\kappa_r \lambda + \Phi_{r0})(\kappa_{pr}\lambda + \Phi_{pr0})} \tag{A.20}$$

This then yields an expression for the free ribosome concentration that is constant across growth rates:

$$\frac{R_{free}}{V} = \Phi_f \Phi_{inact} \frac{R_{Tot}}{V} \tag{A.21}$$

Using a mass action expression for translation:

$$\text{Translation Rate (in copy \#)} = \beta \frac{R_f}{V} X_{rna} \tag{A.22}$$

This implies the translation efficiency per mRNA, α_p, is constant:

$$\text{Translation Rate (per mRNA)} = \beta \frac{R_f}{V} \tag{A.23}$$

This result conflicts with our assumption of $\Phi_f \approx \Phi_{inact}$ and R_{nf} being negligible. However, in a related study of translation in *Bacillus subtilis*, Borkowski *et al.* observe a decreasing translation efficiency (per mRNA) with increasing growth rates and they infer that this is due to a decreasing free ribosome concentration [187]. The authors use this varying free ribosome concentration to test the mass action (linear) translation model used by Klumpp *et al.* [147]. With a varying free ribosome concentration, R_f/V, the ratio of efficiencies between two different RBS is constant if the mass action (linear) model of Klumpp is used:

$$\text{Ratio of Translation Efficiencies} = \frac{\beta_1 \frac{R_f}{V}}{\beta_2 \frac{R_f}{V}} = \frac{\beta_1}{\beta_2} \tag{A.24}$$

However, Borkowski *et al.* observe that the ratio of translation efficiencies between different transcripts varies across growth rates. This suggests that a different model of translation initiation may be more appropriate, and that constant translation rate may not be caused by constant free ribosome concentrations (applying the same argument to both *B. subtilis* and *E. coli*).

Borkowski *et al.* propose a Michaelis-Menten model of translation initiation in terms of the free ribosome concentration to explain the non-constant translation efficiency ratios [187]:

$$\text{Translation Rate (copy \#)} = \frac{\beta \frac{R_f}{V}}{K_M + \frac{R_f}{V}} X_{rna} \tag{A.25}$$

with translation efficiency then expressed as;

$$\text{Translation Rate (per mRNA)} = \frac{\beta \frac{R_f}{V}}{K_M + \frac{R_f}{V}} \tag{A.26}$$

In this model, the mRNA's RBS is characterized by two constants, β, the maximal translation initiation rate per mRNA, and K_M, a half-saturating constant specific to the given RBS. This model is also justified by the mechanisms of translation initiation in which the mRNA species may be in limiting quantities and become saturated. This Michaelis-Menten formulation agrees with the observation of constant translation efficiency in Klumpp and Liao [147, 252], under the assumption that the RBS of the gene has a low K_M value and is near saturation over the relevant growth rates. We have also assumed that initiation of translation is the limiting step in protein production, and that it is slower than translation elongation [253, 254].

A.2 Details of the Multiple-shooting Algorithm and Optimization

Our OED algorithm can be classified as a direct optimal control approach – following a discretize-then-optimize procedure with respect to the system simulation dynamics [255] – where the system dynamics are implemented numerically, the control variables are discretized and selected, the system response is simulated, and then the controls are adjusted to improve the objective. However, for our dynamic, non-linear system we found this approach performed poorly if implemented in a naive manner. Multiple shooting and collocations methods provide improvements by discretizing the simulation along with the controls [255, 256]. In multiple shooting specifically, the simulation is partitioned into a series of initial value problems [255]. This process increases the dimensionality of the problem but also improves the problem structure, giving the optimization algorithm more information about how each control contributes to the objective function [257]. This process has been referred to as 'lifting', where the problem is lifted into a higher-dimensional, but more easily navigated space [257]. The problem structure can be further improved by including derivative information for the object and constraints, which can be done in a straightforward manner using algorithmic differentiation tools available in CasADi [196].

Below, we provide a brief overview of the multiple shooting algorithm used in this work for optimal experimental design (OED). Further details on the implementation of similar algorithms can be found in [156, 157, 155]. We will describe the algorithm implementation in pseudo-code, outlining the use of CasADi's symbolic interface. CasADi uses symbolics (on the front-end) to create mathematical expressions; details of the back-end implementation can be found in [258]. An example of some (pseudo) syntax for the CasADi MATLAB interface is given in Algorithm 1. In this example we define two symbolic variables 'x'

Algorithm 1 CasADi Example

```
x=SX.sym('x')
y=SX.sym('y')
z=x+y
f=function('f',{x,y},{z})
f(1,2)
¿¿3
w1=SX.sym('w1')
w2=SX.sym('w2')
f(w1,w2)
¿¿'w1+w2'
```

and 'y' with 'SX.sym()'. We then create the symbolic expression 'z=x+y', where 'z' now symbolically means 'x+y'. To be able to evaluate that expression on new inputs, we define a function 'f' that maps 'x' and 'y' to the output described by 'z'. Below this, we see that we can use the function 'f' to map specific numbers to a numerical output, but we can also use it to create new symbolic expressions (i.e. 'w1+w2'). These in turn could be used to build layers of symbolics as we will do in the OED algorithm. The reader is referred to CasADi manual for further details on the internal functions [258]. We mix use of both the MX and SX symbolic classes, which have different computational properties [258], however the reader can ignore this for general understanding (MX symbolics are created like SX symbols but with 'MX.sym()').

For simplicity we will restate the system dynamics for a single sub-experiment as follows

$$\frac{d\boldsymbol{Y}}{dt} = \boldsymbol{G}(\boldsymbol{Y}, \boldsymbol{\theta}, \lambda, u(t), W(t)) \tag{A.27}$$

Here the state $\boldsymbol{Y}$ contains $\boldsymbol{X}$ (2 components), all $\bar{\boldsymbol{X}}_{\theta_i}$ for each of 6 parameters (12 components), $\hat{w}$ (2 components), and the unique elements of $\mathcal{I}$ (21 components), for a total of 37 components. The details of the dependence of $\boldsymbol{Y}$ on λ, $u(t)$ and $W(t)$ can be found in the main text. Our algorithm begins by defining this right-hand side as a symbolic expression in CasADi, via Algorithm 2. In defining $\boldsymbol{G}(\boldsymbol{Y}, \boldsymbol{\theta}, \lambda, u, w_{rna}, w_{prot})$ algebraically (line 7

Algorithm 2 RHS function definition

```
1: $\boldsymbol{Y}$=SX.sym('$\boldsymbol{Y}$',37)    ▷ Define symbolic variables
2: $\boldsymbol{\theta}$=SX.sym('$\boldsymbol{Y}$',6)
3: $\lambda$=SX.sym('$\lambda$')
4: $u$=SX.sym('$u$')
5: $w$=SX.sym('$w$',2)    ▷ One each for $w_{rna}$, $w_{prot}$
6: RHS=$\boldsymbol{G}(\boldsymbol{Y}, \boldsymbol{\theta}, \lambda, u, w)$    ▷ Implement $G$ algebraically
7: g= function('g',{$\boldsymbol{Y}$,$\lambda$,$u$,$w$},{RHS})    ▷ Create RHS CasADi function
```

above, details omitted), we used CasADi's algorithmic differentiation and matrix algebra abilities. This can be done by first defining the RHS for the state variables $\boldsymbol{X}_{rna}$ and $\boldsymbol{X}_{prot}$ symbolically. Then the sensitivities and FIM RHS functions can be determined from the corresponding RHS expression by using the `jacobian` function, the `jtimes` function and the matrix product operator, among others.

Using the symbolic function for the overall RHS, $\boldsymbol{G}$, we can construct a symbolic function, $\bar{\boldsymbol{G}}_1$, for a single step of an explicit, fixed-step-size numerical integration scheme. We used a fourth-order Runge-Kutta scheme as described in the CasADi examples [259]. External ODE solvers, like the Sundials suite [260], can be used, but defining an explicit integrator in CasADi has the advantage that the integrator itself can be algorithmically differentiated. This is useful for providing first and second order integral information to the NLP solver. In contrast, the use of conditional statements by variable step-size or implicit solvers generally precludes algorithmic differentiation. The single step of the RK4 integrator is given as shown in Algorithm 3, see [259] for further details.

Algorithm 3 Define RK4 Scheme

```
8: Define Symbolic $\mathbf{Y}_{input}$    ▷ Define an input $\boldsymbol{Y}$ vector
9: $k_1 = g(\mathbf{Y}_{input}, \boldsymbol{\theta}, \lambda, u, w)$    ▷ Implement the RK4 sub-steps
10: $k_2 = g(\mathbf{Y}_{input} + \Delta t k_1/2, \boldsymbol{\theta}, \lambda, u, w_{rna}, w_{prot})$
11: $k_3 = g(\mathbf{Y}_{input} + \Delta t k_2/2, \boldsymbol{\theta}, \lambda, u, w)$
12: $k_4 = g(\mathbf{Y}_{input} + \Delta t k_3, \boldsymbol{\theta}, \lambda, u, w)$
13: $\mathbf{Y}_{output} = \mathbf{Y}_{input} + \Delta t(k_1 + 2k_2 + 2k_3 + k_4)/6$    ▷ Create a symbolic expression for the
    output
14: $\bar{\boldsymbol{G}}_1$=function('$\bar{\boldsymbol{G}}_1$',{$\mathbf{Y}_{input}$,$\boldsymbol{\theta}$,$\lambda$,$u$,$w$},{$\mathbf{Y}_{output}$})    ▷ Create a function mapping from
    $\mathbf{Y}_{input}$ to $\mathbf{Y}_{output}$
```

Recall that for a single sub-experiment, the input $u(t)$ is piecewise-constant over six 100 min intervals, $W(t)$ (i.e. $w_{rna}(t)$ and $w_{prot}(t)$) is piecewise-constant over forty-eight 12.5 min intervals, and the growth rate λ is constant. We label these discretized controls by their corresponding intervals as follows: growth controls, $\lambda^{(i)}$, where $i \in \{1, , 2, 3\}$ (each sub-experiment); induction controls, $u^{(j,i)}$, where $j \in \{1, \ldots, 6\}$ (each induction interval & sub-experiment); and sampling controls, $w_{rna}^{(k,j,i)}$ and $w_{rna}^{(k,j,i)}$, where $k \in \{1, \ldots, 48\}$ (each sampling interval, induction interval & sub-experiment). Because the 48 sampling intervals are the shortest of the piecewise constant intervals, each has constant controls (in λ, u and $w_{(species)}$) over its duration. We can therefore iterate the single RK4 step function, $\hat{\boldsymbol{G}}_1$, to create an integrator, $\hat{\boldsymbol{G}}$, that maps the state at the beginning of the sampling interval to the end, 12.5 min later, with a constant set of controls (see Algorithm 4).

Algorithm 4 Iterate RK4 over the sampling (smallest) control interval

```
 7: $\boldsymbol{Y}_o$=MX.sym('$\boldsymbol{Y}_o$')
 8: $\boldsymbol{Y}_{iter}$=$\boldsymbol{Y}_o$
 9: for 12.5/$\Delta t$ do                                   ▷ Iterate, advancing $\Delta t$ time units each loop
10:     $\boldsymbol{Y}_{iter} = \bar{\boldsymbol{G}}_1(\boldsymbol{Y}_{iter}, \boldsymbol{\theta}, \lambda, u, w_{rna}, w_{prot})$        ▷ Apply $\bar{\boldsymbol{G}}_1$ to the state $\boldsymbol{Y}_{iter}$ each loop
11: end for
12: $\bar{\boldsymbol{G}}$=function('$\bar{\boldsymbol{G}}$',{$\boldsymbol{Y}_o$,$\boldsymbol{\theta}$,$\lambda$,$u$,$w$},{$\boldsymbol{Y}_{iter}$})   ▷ Create function, maps interval start, $\boldsymbol{Y}_o$, to
    end, $\boldsymbol{Y}_{iter}$
```

To determine the D-optimality score we need to integrate the RHS over the total time (0 to 600 min) for each sub-experiment. The final objective value can be computed from the Fisher information entries (the 17th to 37th components) at the final time, $Y(t = t_f)_{17\ldots37}$, in each of the sub-experiments. To apply multiple-shooting, we partitioned each sub-experiment's duration into six shooting intervals. We used intervals of 100 min, corresponding to the six constant-u induction intervals. We treat each of these shooting intervals as a separate initial value problem, with its own initial conditions $\boldsymbol{Y}_o^j$. Algorithm 5 shows how we use $\bar{\boldsymbol{G}}$ to propagate these initial conditions through the series of shooting intervals, linking the initial value problems with constraints to enforce continuity. There are seven initial conditions $\boldsymbol{Y}_o^j$ for each sub-experiment because the final time is treated as a (dummy) initial condition; this increases the sparsity of the problem. Using the initial conditions, $\boldsymbol{Y}_o^j$, as optimization variables, along with the discretized controls, provides a number of benefits. It gives the NLP solver direct access to the system state at regular intervals throughout the simulation time. This improves the problem structure as the NLP solver can alter the states directly. The continuity constraints then propagate this information to the control variables. Moreover, these states increase the sparsity of the NLP problem because the coupling of the controls and the objective across the simulation is partitioned by the additional optimization variables. The system dynamics in each shooting interval only depend on controls in the other intervals via the continuity constraints.

As shown in Algorithm 5, we loop over each sub-experiment, induction/shooting interval and sampling interval, iteratively building up a symbolic expression for the objective function and for the nonlinear constraint functions. At the beginning of the experiment, we create vectors 'CtrlVec' and 'CnstrnVec' which, as we move through the three nested loops, are filled with symbolic terms for each of the NLP optimization variables and the non-linear constraints, respectively. The elements of 'CtrlVec' are individual symbols representing the controls and the shooting initial conditions, which the NLP solver will optimize. The elements of 'CnstrnVec' contain non-linear symbolic expression that evaluate to the constraint functions. The vectors 'lbc' and 'ubc' are the lower and upper bounds for the non-linear constraint functions. If we want two symbolic expressions to be equal, we insert an expression for their difference into 'CnstrnVec', and then set both 'lbc' and 'ubc' to zero to enforce equality. The vectors 'lbw' and 'ubw' likewise constrain the 'CnstrnVec' optimization variable vector to feasible ranges. We start with both 'CtrlVec', 'CnstrnVec', 'lbw', 'ubw', 'lbc' and 'ubc' empty and fill them as we loop over the problem structure.

At line 21 we create a symbol, $\lambda^{(i)}$ for the growth rate control. This is done once for each sub-experiment at the start of the outer loop. We then add it to the control vector, 'CtrlVec', and constrain its range. At line 26 we create a symbol vector for the initial conditions for the sub-experiment. This too is then added to the control vector and constrained to a feasible range. However at line 31, we insert the additional nonlinear constraint that the initial condition must be at steady state (defined by the CasADi function 'SteadyState($\lambda^{(i)}$, $\boldsymbol{\theta}$)', definition not given and must be provided by the user). We enforce equality in the following lines. At line 36 we create an induction control variable $u^{(j,i)}$ and add it to 'CtrlVec', once for each sub-experiment and shooting/induction interval. This also marks the beginning of a shooting interval. In the following lines we constrain the induction to its feasible range. At line 42 we create symbols for the sampling density controls, add them to 'CtrlVec', and then constrain them. At line 47 we use $\bar{\boldsymbol{G}}$ to propagate the symbol for the initial condition, $Y_o^{(j,i)}$, forward, storing it in the same variable. (This does not erase the original contents of $Y_o^{(j,i)}$ from the start of the shooting interval, as those symbols are stored in 'CtrlVec'.) The inner-most loop calls $\bar{\boldsymbol{G}}$ with the corresponding control symbols for the given interval and iteratively advances the state vector. After completion of the inner loop, at line 51, a new shooting initial condition is created and added to 'CtrlVec'. At line 56 we constrain the new initial condition to be equal to the final value of the state vector on the previous shooting interval (the product of the iterated

Algorithm 5 Construct control problem

```
13: CtrlVec={}                                  ▷ Empty vector for OED control symbols
14: lbw=[]                                      ▷ Empty vector lower bound of OED control symbols
15: ubw=[]                                      ▷ Empty vector upper bound of OED control symbols
16: CnstrnVec={}                                ▷ Empty vector for nonlinear constraint symbols
17: lbc=[]                                      ▷ Empty vector lower bound of nonlinear constraints
18: ubc=[]                                      ▷ Empty vector upper bound of nonlinear constraints
19: FIM= $\bar{0}$
20: for $i = 1:3$ do                            ▷ Loop over sub-experiments
21:     $\lambda_i$ = MX.sym('$\lambda^{(i)}$')  ▷ Create $\lambda$ control, one for each loop
22:     CtrlVec={CtrlVec, $\lambda^{(i)}$}       ▷ Add $\lambda^{(i)}$ to control vector
23:     lbw = [lbw; $\lambda_{min}$];            ▷ Restrict growth rates to feasible range
24:     ubw = [ubw; $\lambda_{max}$];
25:
26:     $\boldsymbol{Y}_o^{(0,i)}$ = MX.sym('$Y_o^{(0,i)}$', 37);   ▷ Create initial condition state for each sub-experiment
27:     CtrlVec={CtrlVec, $\boldsymbol{Y}_o^{(0,i)}$ }             ▷ Add it to the control vector
28:     lbw = [lbw; $\bar{0}$];
29:     ubw = [ubw; $\bar{\mathrm{Inf}}$];
30:
31:     CnstrnVec={CnstrnVec, $\boldsymbol{Y}_o^{(0,i)}$−SteadyState($\lambda^{(i)}$,$\boldsymbol{\theta}$) }   ▷ Constrain IC to be at steady state
32:     lbc = [lbc; $\bar{0}$];                 ▷ Bounds for constraint are 0, implying equality
33:     ubc = [ubc; $\bar{0}$];
34:
35:     for $i = 1:6$ do                        ▷ Loop over shooting/induction interval
36:         $u^{(j,i)}$ = MX.sym('$u^{(j,i)}$')  ▷ Create $u$ control, one for each sub-exp. & induction intrvl.
37:         CtrlVec={CtrlVec, $u^{(j,i)}$}       ▷ Add $u^{(j,i)}$ to control vector
38:         lbw = [lbw; $u_{min}$];              ▷ Restrict $u$ to feasible range
39:         ubw = [ubw; $u_{max}$];
40:
```

```
41:         for $j = 1 : 48$ do
42:             $w^{(k,j,i)}$ = MX.sym('$w^{(k,j,i)}$',2)        ▷ Create $w$ for each sub-exp., induction & samp. intrvl
43:             CtrlVec={CtrlVec, $w^{(k,j,i)}$}        ▷ Add $w^{(k,j,i)}$ to control vector
44:             lbw = [lbw; $\bar{0}$];        ▷ Restrict $w$ to feasible range
45:             ubw = [ubw; $w_{max}$];
46:
47:             $\boldsymbol{Y}_o^{(j-1,i)} = \bar{\boldsymbol{G}}(\boldsymbol{Y}_o^{(j-1,i)}, \boldsymbol{\theta}, \lambda^{(i)}, u^{(j,i)}, w^{(k,j,i)})$        ▷ Advance (symbolic) state vector
48:         end for
49:
50:
51:         $Y_o^{(j,i)}$ = MX.sym('$Y_o^{(j,i)}$', 37);        ▷ Create new shooting interval IC
52:         CtrlVec={CtrlVec, $Y_o^{(j,i)}$ }        ▷ Add it to the control vector
53:         lbw = [lbw; $\bar{0}$];
54:         ubw = [ubw; $\bar{\text{Inf}}$];
55:
56:         CnstrnVec={CnstrnVec, $Y_o^{(j,i)} - Y_o^{(j-1,i)}$ }        ▷ Constrain $Y_o^{(j,i)}$ for continuity with $Y_o^{(j-1,i)}$
57:         lbc = [lbc; $\bar{0}$];        ▷ Bounds for constraints are 0, implying equality
58:         ubc = [ubc; $\bar{0}$];
59:     end for
60:     CnstrnVec={CnstrnVec, $c_{max} - Y_o^{(6,i)}(15..16)$ }        ▷ Constrain integral of samp. density to leq. 12
61:     lbc = [lbc; $\bar{0}$];        ▷ Bounds for constraint is 0, implying equality
62:     ubc = [ubc; $\bar{0}$];
63:
64:     FIM=FIM+$Y_o^{(6,i)}(17..37)$        ▷ Sum FIM terms for each sub-exp.
65: end for
66: Objective= $-\log(\det(\text{sym}(FIM)))$        ▷ Define the overall objective
```

$\bar{\boldsymbol{G}}$), adding it to 'CnstrnVec'. At the end of the sub-experiment loop, line 60, we enforce the integral constraints on the sampling density. At line 64 we add the FIM entries in the final sub-experiment state vector to the running totals across the sub-experiments. Finally, we form the objective expression 'Objective', which contains a (very large) symbolic expression for the objective of the entire experiment, in line 66.

To improved numerical stability, we compute the determinant of the Fisher information matrix using QR factorization: $\mathcal{I}_{Tot} = QR$. The entries in 'FIM' are the unique values of $\mathcal{I}_{Tot}$. The function 'sym()' in Algorithm 5 reforms the complete $\mathcal{I}_{Tot}$ from the vector 'FIM'. Because $\mathcal{I}_{Tot}$ is positive semi-definite $\det(\mathcal{I}_{Tot}) \geq 0$. Further

$$|\det(\mathcal{I}_{Tot})| = |\det(Q)||\det(R)| \tag{A.28}$$

The factorization is such that $|\det(Q)| = 1$. Because R is an upper triangular matrix, the determinant is the product of its diagonal entries:

$$\det(\mathcal{I}_{Tot}) = \prod_m R_{(m,m)} \tag{A.29}$$

and so

$$-\ln(\Theta_D(\mathcal{I}_{Tot})) = -\ln(\det(\mathcal{I}_{Tot})) = -\sum_m \ln(R_{(m,m)}) \tag{A.30}$$

The expression in 'Objective' is a mathematical function of the symbols listed in 'CtrlVec'. Likewise, the vector of expressions in 'CnstrnVec' are also mathematical functions of the symbols in 'CtrlVec'. Because these functions are symbolic, they can be differentiated with respect to any (or all) of the entries in 'CtrlVec' (or the parameter vector $\boldsymbol{\theta}$). This property allows CasADi to automatically generate Jacobians and Hessians for the objective and the constraints when passing the problem to IPOPT. Although generation of the derivatives is automated once the symbolic expression is constructed, choosing a problem structure that achieves maximal sparsity is critical. The degree of sparsity in the Hessian and Jacobians make a significant difference in the computation time. Once the the symbolic expressions for the OED problem have been created, passing them to IPOPT is straightforward using CasADi's interface. Algorithm 6 shows the creation of the solver and its call in CasADi's MATLAB interface. Starting the solver requires an initial guess for the control vector. We generate this by simulating one of the null experiments and storing the state variables and controls at the appropriate times to construct 'CtrlVec'.

Parameter Estimation: We also implemented our weighted least-squares parameter estimation algorithm in CasADi. This was also implemented as a multiple-shooting algorithm and was structured in a similar manner to the OED algorithm above, where the

Algorithm 6 Calling IPOPT

```
67: prob = struct( Objective, CtrlVec, CnstrnVec)    ▷ Package symbol vectors for passage to IPOPT
68: solver = nlpsol( 'ipopt', prob)    ▷ Create a solver instance
69: solver( CtrlVec$_o$, lbw, ubw, lbc, ubc)    ▷ Call solver with initial guess; CtrlVec$_o$, pass upper/lower bounds
```

parameters are treated as time-constant controls in an optimal control problem [261, 262]. The weights in our fitting algorithm were taken as the inverse of the sampling variances, $\sigma^2_{rna} = (0.05)X_{rna}$ and $\sigma^2_{prot} = (0.05)X_{prot}$. In our numerical fitting experiments, for each experimental design (null, null variants, optimal and perturbed optimal), we initialized the parameter estimation algorithm to a random parameter vector drawn uniformly from the feasible parameter range. For each experimental design, a small subset of the 30 fittings either did not converge (max number of iterations or other stopping condition was reached) or converged to clearly erroneous estimates (relative error exceeding several orders of magnitude). We removed these outliers before computing covariances. The number of outliers in each design were as follows: null experiment, 3; growth variant, 2; sampling variant, 1; induction variant, 3; true optimal, 0; perturbed optimal, 0 (for all six).

Timing: Our OED algorithm normally took between 70 and 400 iterations to converge in IPOPT (depending on the number and range of the parameters). The wall-clock time was on the order of an hour. Our parameter estimating algorithm normally took between 10 and 70 iterations to converge. The wall-clock timing was on the order of 10s of minutes. All experiments were done on a Mac mini machine with a 2.6 GHz Intel Core i5 and 16 GB of RAM.