# **PEIT**H($\Theta$):
**P**erfecting **E**xperiments with **I**nformation **T**heory

23rd June 2017

# Contents

# 1. Introduction

Quantitative descriptions of biological systems are being used ever more frequently, helping us to understand them in detail whilst also providing a framework to make predictions on their future behaviour. Popular approaches to representing these systems include ordinary (ODEs) and stochastic (SDEs) differential equations, where parameters are commonly inferred from experimental data.

Through conducting more and more experiments on a biological system, and thus obtaining more and more data, we hope that parameter inference within mathematical models become increasingly more accurate. This, however, is a luxury that is commonly not entertained as constraints, such as cost and time, limit the available resources to conduct an *infinite* number of experiments. Furthermore, the data collected from each experiment are not equal in *information* content, thus different experiments can lead to better or worse results compared to others. The question quickly arises:

> *How can we optimally pick an experiment to best infer parameters or make predictions on the outcome of biological systems?*

This has been explored in the work of Liepe et al. [1] They approach the question within an information theory framework by calculating the mutual information associated with an experiment and suggest to conduct the experiment for which this is maximised. Further, they validate their approach by simulating data for each experiment and using this to fit the parameters through approximate Bayesian computation (ABC). [2]

Using a Bayesian framework, [1] the parameters of the model are treated as random variables with prior probability distribution, $\Pi(\boldsymbol{\Theta})$ and posterior probability distribution, $\Pi(\boldsymbol{\Theta}|\mathbf{X} = \mathbf{x})$, where we have realisation $\mathbf{x}$ of random variable $\mathbf{X}$. The entropy,

$H(\boldsymbol{\Theta}) = -\mathrm{E}_{\boldsymbol{\Theta}}(\log p(\boldsymbol{\Theta}))$, of a random variable $\boldsymbol{\Theta}$ is defined as its uncertainty. Further, the definition of the posterior uncertainty is $H(\boldsymbol{\Theta}|\mathbf{X}_q = \mathbf{x}) = -\mathrm{E}_{\boldsymbol{\Theta}}(\log p(\boldsymbol{\Theta}|\mathbf{X}_q = \mathbf{x}))$ for a given outcome $\mathbf{x}$ of experiment $q$, where $q \in Q$ and $Q = \{q_1, \ldots, q_{|Q|}\}$ is the set of available experiments. Obviously this is not available prior to any experiment but, given a distribution of $\mathbf{X}_q$, the average posterior uncertainty is $H(\boldsymbol{\Theta}|\mathbf{X}_q) = \mathrm{E}_{\mathbf{X}_q}(H(\boldsymbol{\Theta}|\mathbf{X}_q = \mathbf{x}))$, which can be obtain a priori. Thus the mutual information is obtainable as:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) = H(\boldsymbol{\Theta}) - H(\boldsymbol{\Theta}|\mathbf{X}_q) = \int\int p(\boldsymbol{\theta}, \mathbf{x}_q) \log \frac{p(\boldsymbol{\theta}, \mathbf{x}_q)}{p(\boldsymbol{\theta}) p(\mathbf{x}_q)} d\boldsymbol{\theta} d\mathbf{x}_q \tag{1.1}$$

This gives a measure of, for each experiment $q$, the average reduction in uncertainty for the parameters $\boldsymbol{\Theta}$ thus providing an approach to quantitatively capture those experiments that provide *substantial* and *relevant* information [1].

Liepe et al. [1] also extend this concept to reduce the uncertainty in the estimation of a subset of parameters as well as reducing the uncertainty in an experimental outcome. By denoting $\boldsymbol{\Theta}_c$ a subset of the elements of the vector $\boldsymbol{\Theta}$, then deriving the mutual information for the former case leads to:

$$\mathcal{I}(\boldsymbol{\Theta}_c, \mathbf{X}_q) = H(\boldsymbol{\Theta}_c) - H(\boldsymbol{\Theta}_c|\mathbf{X}_q) = \int\int p(\boldsymbol{\theta}, \mathbf{x}_q) \log \frac{p(\boldsymbol{\theta}_c, \mathbf{x}_q)}{p(\boldsymbol{\theta}_c) p(\mathbf{x}_q)} d\boldsymbol{\theta} d\mathbf{x}_q \tag{1.2}$$

To understand the latter, for predicting experimental outcome, suppose there is an experiment that could not feasibly be carried out. Further, suppose we label the outcome by the random variable $\mathbf{Y}$ and have a set of experiments $Q$. Then we can select which experiment, $q$, to conduct based on:

$$\mathcal{I}(\mathbf{Y}, \mathbf{X}_q) = H(\mathbf{Y}) - H(\mathbf{Y}|\mathbf{X}_q) = \int\int p(\mathbf{y}, \mathbf{x}_q) \log \frac{p(\mathbf{y}, \mathbf{x}_q)}{p(\mathbf{y}) p(\mathbf{x}_q)} d\mathbf{y} d\mathbf{x}_q \tag{1.3}$$

More often the not these integrals are analytically intractable therefore requiring numerical estimation. To this extent the integrals of equations 1.1, 1.2, and 1.3 are approxiamted using Monte Carlo estimates [1] leading to the following 3 equations:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) \approx \frac{1}{N_1} \sum_{i=1}^{N_1} \left[ \log\left( p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(i)}\big) \right) - \log\left( \frac{1}{N_2} \sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(j)}\big) \right) \right] \tag{1.4}$$

where $\boldsymbol{\theta}^{(i)}$ is a sample from the prior distribution, $\Pi(\boldsymbol{\Theta})$, and $\mathbf{x}_q^{(i)}$ is a realisation of experiment $q$ defined by parameter $\boldsymbol{\theta}^{(i)}$.

$$\mathcal{I}(\boldsymbol{\Theta}_c, \mathbf{X}_q) \approx \frac{1}{N_1} \sum_{i=1}^{N_1} \left[ \log\left( \frac{1}{N_3} \sum_{j=1}^{N_3} p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(i,j)}\big) \right) - \log\left( \frac{1}{N_2} \sum_{k=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(k)}\big) \right) \right] \quad (1.5)$$

where $\boldsymbol{\theta}^{(i,j)}$ is a sample from $\Pi(\boldsymbol{\Theta}|\boldsymbol{\Theta}_c = \boldsymbol{\theta}_c^i)$ with $\boldsymbol{\theta}^{(i)}$ and $\mathbf{x}_q^{(i)}$ as before.

$$\begin{aligned} \mathcal{I}(\mathbf{Y}, \mathbf{X}_q) \approx & \frac{1}{N_1} \sum_{i=1}^{N_1} \left[ \log\left( \frac{1}{N_2} \sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{y}^{(i)} | \boldsymbol{\theta}^{(j)}\big) p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(j)}\big) \right) \right. \\ & - \log\left( \frac{1}{N_3} \sum_{k=N_1+N_2+1}^{N_1+N_2+N_3} p\big(\mathbf{y}^{(i)} | \boldsymbol{\theta}^{(k)}\big) \right) \\ & \left. - \log\left( \frac{1}{N_4} \sum_{l=N_1+N_2+N_3}^{N_1+N_2+N_3+N_4} p\big(\mathbf{x}_q^{(i)} | \boldsymbol{\theta}^{(l)}\big) \right) \right] \end{aligned} \quad (1.6)$$

where $\mathbf{y}^{(i)}$ is a realisation of experiment $\mathbf{Y}$ defined by parameter $\boldsymbol{\theta}^{(i)}$ with $\boldsymbol{\theta}^{(i)}$ and $\mathbf{x}_q^{(i)}$ as before.

# 2. Software Package Outline

## 2.1 Overview

Functionally, PEITH(Θ) can be separated in to three major components: Inputs, Algorithms and Output (see Figure 2.1). Using PEITH(Θ), the user will only interact with the Input component. Here the user has the choice between two different input methods, a low level/more involved and a high level/more automated input pipeline. The detailed use of these pipelines is described in chapter 4.

Once the user input has been successfully parsed and processed, the data is passed on to the main algorithms of PEITH(Θ). First, trajectories are simulated based on the model and priors using `CUDA-sim` [3], which are subsequently used to estimate the mutual information between model parameters and an experiment or between a reference and alternative experiment. A detailed description of the code base used for entropy estimation and calculation of the mutual information can be found in chapter 5.

Lastly, the role of the output component is to present the estimated mutual information and additional information about the run to the user. The output options available are described in detail in chapter 4.6.

## 2.2 Package Structure

Further extending from the general structure, we divide PEITH(Θ) into different sections, based on their functionality.

1. *Mains* - where PEITH(Θ) is called from

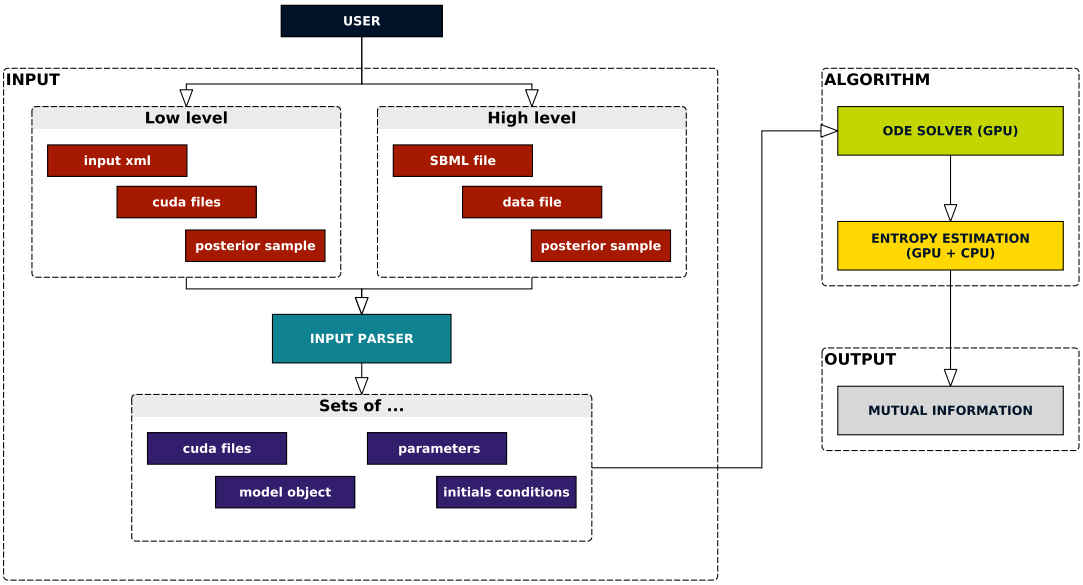



**Figure 2.1: Schematic representation of PEITH(Θ) overall package structure.** PEITH(Θ) is separated into 3 parts: *Input*, *Algorithm* and *Output*. The user(dark blue) inputs data through either the low level or high level pipeline as set of input files(red). These sets are parsed and prepared for simulation(green) by the input parser(light blue). The simulated trajectories are prepared for the entropy estimation and passed on to respective functions to estimate the mutual information(yellow) on GPU and CPU. The output functions(grey) than takes care of the result presentation.

2. *Error Checking & Parsers* - checks the input arguments to PEITH(Θ) and manipulates the input arguments as well as for intermediary steps during the pipeline

3. *Simulations* - prepares input, calls and manipulates the output of CUDA-Sim[3]

4. *Mutual Information & Outputs* - calculates the mutual information for each experiment and outputs results

Next we give further details about each section along with the major functionalities within each.

### Mains

Functions: `main`

`sorting_files`

`template_creator`

`memory_checker`

This is where PEITH(Θ) is called to invoke the pipelines. It comprises of two primary functions. The first, `main`, reads the input arguments from the command line and organises them accordingly. These input arguments are then passed sequentially to `sorting_files` to calculate the mutual information associated to experiments. In the mains we also provide two functions `template_creator` and `memory_checker`. The former creates templates for the input `XML` file and `CUDA` code file (see low level workflow later), whilst the latter estimates the maximum memory requirements when running PEITH(Θ) thus highlighting any potential issues with RAM that may arise in the pipeline of PEITH(Θ).

**Error Checking & Parsing**

Functions: `input_checker`
`printOptions`
`SBML_checker`
`getSpeciesValue`
`generateTemplate`
`getWeightedSample`
`parse_required_single_value`
`parse_required_vector_value`
`process_prior`
`parseint`
`parseint_index`
`parse_fitting_information`
`parse_fitting_information_parameters`
`stringSearch`
`SBML_reactionchange`
`SBML_reactionchanges`
`cudacodecreater`

Class: `algorithm_info`

**Methods**: `__init__`
`print_info`

In our pipeline, error checking and parsing overlap greatly. This is because error checking, in many cases, is conducted simultaneously with parsing. The main functions that are of note here include: `input_checker` - reads in the arguments from the command line making basic checks, `SBML_checker` - in the event `SBML` files are used we make basic checks of the format, `generateTemplate` - creates and input `XML` files from an `SBML` and input data file, `SBML_reactionchanges` - allows the user to generate experiments that are perturbations of the input `SBML` file, `cudacodecreater` - uses `ABC-sysbio` [2] parsers to generate `CUDA` code from `SBML` files and the `algorithm_info` class - the main object used within the pipeline and holds all relevant information to the experiments.

**Simulations**

Packages: `CUDA-Sim` [3] - *modified version*

Functions: `run_cudasim`

Class: `algorithm_info`

**Methods**: `THETAS`

`getAnalysisType`

`getpairingCudaICs`

`sortCUDASimoutput`

`addNoise`

`fitSort`

`scaling`

`scaling_ge3`

`copyTHETAS`

This section of the package is involved with preparation of the inputs and outputs to `CUDA-Sim` [3], as well as conducting the simulations based on samples from the prior distribution. Since the instances of `algorithm_info` are the central objects of our algorithm, we include further methods here. We sample from our priors using the method `THETAS`. The function `run_cudasim` invokes `CUDA-Sim` [3] and also organises the inputs and outputs to it by invoking further methods of `algorithm_info`. It should be noted that certain methods to `algorithm_info` are only required when running experiment

selection for prediction (`scaling_ge3` and `copyTHETAS`).

### Mutual Information

Functions: `odd_num`

`round_down`

`round_up`

`factor_partial`

`optimise_gridsize`

`optimal_blocksize`

`max_active_blocks_per_sm`

`run_mutInfo1`

`run_mutInfo2`

`run_mutInfo3`

`mutInfo1`

`mutInfo2`

`mutInfo3`

`plotbar`

`csv_output_writer`

Here we focus on calculation of the mutual information after having run our simulations on `CUDA-Sim` [3]. This part of our pipeline follows a similar structure for all three objectives. In order to do experimental selection for prediction on all parameters `run_mutInfo1` is called which in turn invokes `mutInfo1` to calculate the mutual information on GPUs. Similar approaches are used when considering experiment selection for a subset of parameters as well as for prediction. For the output, we have the function `plotbar` and `csv_output_writer`, which generates bar graphs for mutual information as well as giving these values and more within a `CSV` file respectively. Additionally, there are many functions (the second to seventh given in the list above) used to determine the optimal GPU launch configuration.

# 3. Installation

## 3.1 Hardware requirements

### CPU

While most of of the computationally expensive calculations are computed on the GPU, we still recommend an up-to-date CPU with 2.0 GHz or more to avoid creating a computational bottleneck. Only one CPU core is used per invocation of our package.

### GPU

PEITH(Θ) requires a CUDA enabled nvidia GPU[1] with Compute Capability 2.0 - 6.2.[2] As it involves computing FP64, we recommend a GPU optimised for double precision calculations (nvidia Tesla and partly nvidia Quadro).

### RAM

We recommend at least 32 GB of RAM (ideally 128 GB or more) to enable computation of large enough particle sizes to obtain meaningful results. Any RAM size that is below two to three times the size of the global memory of the GPU device, might cause PEITH(Θ) to crash (see *Memory Error* in section A.1.5).

Calculations for the experimental design for experiment outcome prediction (approach 3) are particularly memory intensive. Should your available memory be insufficient, we recommend calculating the mutual information for each alternative experiment separately.

[1] https://developer.nvidia.com/cuda-gpus
[2] https://en.wikipedia.org/wiki/CUDA

On Linux, insufficient memory can be compensated for by increasing the size of the SWAP partition/file on the HDD. Caution: Using the SWAP partition/file for increasing available memory comes with major performance penalties.

## 3.2 Software requirements & Installation

### 3.2.1 Dependencies

Before installing `PEITH`(Θ) the user needs ensure that the following dependencies are installed on their system:

- `CUDA`
- `PyCUDA`
- `numpy`
- `matplotlib`
- `libSBML`

Here we will provide a general guide for setting up a working `anaconda` environment on a Linux system. However, the exact installation steps may slightly vary depending on the user's system and configuration.

`numpy`

Using the `anaconda` python distribution the user can simply install `numpy` with the following command:

```
$ conda install numpy
```

`matplotlib`

In the same manner the user can also install `matplotlib` with the following command:

```
$ conda install matplotlib
```

`libSBML`

The simplest approach to build and install `libSBML` within the `anaconda` distribution is to use the recipe provided by SBMLTeam:

```
$ pip install -i https://pypi.anaconda.org/sbmlteam/simple python\
-libsbml
```

### CUDA

Both CUDA driver and toolkit at version 4 or higher are required to run this package. The user can find the latest driver and software for their hardware on the NVIDIA homepage [3].

`PyCUDA`

In order to perform the computationally expensive calculations on the GPU `PEITH`($\Theta$) utilises `PyCUDA`. We recommend to download [4] and install `PyCUDA` in version *2016-1.2*. Please, follow the instruction in the PyCUDA documentation to build the package for your system and CUDA version [5] and ideally assure that `PyCUDA`'s dependecies are satisfied before building it. In short this is achieved with the following commands. First the user needs to navigate into the directory of downloaded `PyCUDA`:

[3] `https://developer.nvidia.com/cuda-downloads`
[4] `https://mathema.tician.de/software/pycuda/`
[5] `https://wiki.tiker.net/PyCuda/Installation/Linux`

```
$ cd /path/to/pycuda
```

If PyCUDA has previously been build, the user should remove any previous configurations and builds prior to installation. In order to achieve this one needs to run following commands:

```
$ rm -rf ./siteconfig.py
$ rm -rf builds
```

Once these files have been deleted, run the configurations script provided in order to prepare PyCUDA to be build. The user most likely wants to change at least the –cuda-root option to define the installation path of the local cuda installation. The `--help` flag provides a list of further option the user might want to define.

```
$ ./configure.py --help
$ ./configure.py --cuda-root=path/to/cuda/installation
```

If the user wants to install PyCUDA to a particular directory, one needs to add the install directory (`--home=/path/to/new/installation/directory`) to the install directive line in the newly created `Makefile`. Furthermore, assure that the lib/python folder in this installation directory is also in `PYTHONPATH`. Now the user just needs to type in order to build PyCUDA:

```
$ make
$ make install
```

### 3.2.2 Installation of PEITH(Θ)

#### Stable version

The latest tested version of PEITH(Θ) is available through the PyPi server and thus provides a straight-forward installation of PEITH(Θ) by using the `pip`:

```
$ pip install peitho
```

### Development version

In case the user wants to install the latest(development) version of PEITH(Θ), one can achieve this by cloning the package from the `git` repository and install the development version by typing the following commands:

```
$ git clone https://github.com/MichaelPHStumpf/Peitho.git
$ cd Peitho
$ pip install .
```

# 4. Using the package

In this section of the manual we describe our pipeline in more detail, expanding on the required arguments supplied by the user, the format of the outputs and explaining the optionality available to the user. Broadly speaking we have developed two pipelines and usage depends on the users computational background or the complexity of what they wish to achieve:

1. *High Level Workflow:* This pipeline takes an `SBML` file with an additional data file highlighting how experiments can be derived from the `SBML` file. This workflow gives computationally inexperienced individuals the ability to utilise our package but limits individuals to those biological systems that can be defined through `SBML` files.

2. *Low Level Workflow:* This pipeline takes only an input `.xml` file, which is linked to a manually written `CUDA` code file (a `.cu` file). This gives computationally experienced users the ability to define more complex biological systems and even supports events and rules during simulation.

## 4.1 Choosing the appropriate design approach

`PEITH`($\Theta$) was built to handle each of the three approaches. As such the pipeline initiated within `PEITH`($\Theta$) depends upon the question we wish to obtain an answer for. If we want to know which experiment, on average, holds the highest information content in relation to all parameters, we can use *experiment selection for inference of all parameters.* If we only want the information content with respect to a subset of parameters, we can use *experiment selection for inference of a subset of parameters.* Finally, should we care to

understand which experiment holds the greatest information content, on average, about the outcome of another experiment, we can conduct *experiment selection for prediction.*

## 4.2 High Level Workflow Input

PEITH($\Theta$) caters to those of little computational experience through the high level workflow. It requires only SBML files, which can be manually written or obtained from the SBML database[1] and an accompanying input data file which are both described next.

### 4.2.1 SBML file

Systems Biology Markup Language (SBML ) files have developed rapidly over the years since their fruition in 2001 and quickly became the standard for biological model description. We explain how an SBML file should be written in order for it to work best within this workflow. For a more in-depth discussion of SBML files we recommend *Stochastic Modelling for Systems Biology* by Darren J. Wilkinson [4]. Please note that there are many levels and versions of SBML. This could result in issues when used within our workflow. Here we exemplify the PEITH($\Theta$) compatible structure of an SBML file written in level 2, version 1 and using the Repressilator model as defined by Elowitz and Leibler. [5]

The preliminary structure of all SBML files is shown and wraps the entirety of the file. It indicates the level and version of the file as well as defining the id and name of the model:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2" level="2" version="1">
    <model id="Rep" name="Repressilator">
    ...
    </model>
</sbml>}
```

We can also define the units which the species of the model take within the SBML file. Here we use the reserved identifier `substance` and units `moles` (since `moles` is the default

[1] SBML database: `https://www.ebi.ac.uk/biomodels-main/`

for `substance` it is sufficient to not specify the `kind` in the following):

```
<listOfUnitDefinitions>
    <unitDefinition id="substance">
        <listOfUnits>
            <unit kind="mole"/>
        </listOfUnits>
    </unitDefinition>
</listOfUnitDefinitions>
```

In every `SBML` file it is required to have at least one `compartment`. By using compartments it is possible to specify where a reaction occurs. We can, for example, distinguish between reactions that occur in the nucleus and those that occur in the cytosol. For the Repressilator model we only require one compartment (which we label as `Cell`) of size 1:

```
<listOfCompartments>
    <compartment id="Cell" size="1"/>
</listOfCompartments>
```

In order for the model to be properly represented we need to define the species. All species, along with their name, are also given a location (i.e. a `compartment`) and an initial condition:

```
<listOfSpecies>
    <species id="m1" initialAmount="0" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
    <species id="p1" initialAmount="1" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
    <species id="m2" initialAmount="0" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
    <species id="p2" initialAmount="0" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
    <species id="m3" initialAmount="0" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
    <species id="p3" initialAmount="0" compartment="Cell"...
        ...hasOnlySubstanceUnits="true"/>
</listOfSpecies>
```

The parameters of the model are defined next. It is important to note that when using the high level workflow it is necessary to define all parameters globally as opposed to

locally for each reaction:

```
<listOfParameters>
    <parameter id="h" value="2"/>
    <parameter id="alpha0" value="10"/>
    <parameter id="alpha" value="1000"/>
    <parameter id="beta" value="5"/>
</listOfParameters>
```

As can be seen above, an SBML can also contain defined values for each parameter of the model. Since we consider the parameters from a Bayesian perspective, they are treated as random variables as opposed to constant values. As such, for the purpose of this package, these defined values from the SBML file are only used if we specify them as unchanged in the data file to be described later.

Finally, we define our reactions and their corresponding mathematical representation. All reactions are described within the following:

```
<listOfReactions>
    ...
</listOfReactions>
```

For brevity we only provide one of these reactions, namely the transcription of $m_1$. The remaining reactions follow in a similar manner for the Repressilator model [1]:

```
<reaction id="m1Transcription" reversible="false">
    <listOfReactants>
        <speciesReference species="EmptySet"/>
    </listOfReactants>
    <listOfProducts>
        <speciesReference species="m1"/>
    </listOfProducts>
    <listOfModifiers>
        <modifierSpeciesReference species="p3"/>
    </listOfModifiers>
    <kineticLaw>
        <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
            <plus/>
            <ci> alpha0 </ci>
            <apply>
                <divide/>
```

```
                    <ci> alpha </ci>
                    <apply>
                        <plus/>
                        <cn type="integer"> 1 </cn>
                        <apply>
                            <power/>
                                <ci> p3 </ci>
                                <ci> h </ci>
                        </apply>
                    </apply>
                </apply>
            </apply>
            </math>
        </kineticLaw>
    </reaction>
```

It is important to note the use of so called *modifier* species. These are species that are not reactants or product of the reaction itself but influence it in some way. The transcription of $m_1$ corresponds to the following equation (with $p_3$ being a modifier specie):

$$\frac{dm_1}{dt} = -m_1 + \frac{\alpha}{1 + p_3^h} + \alpha_0 \tag{4.1}$$

By combining all these constituent parts we provide an example of a well-defined level 2, version 1 `SBML` file with the installation of `PEITH`($\Theta$). We will later use this `SBML` file to exemplify our high level workflow.

### 4.2.2 Data file

The data file enables us to easily define the algorithm parameters, prior distributions, perturbations to the model, as well as changes to other experimental conditions such as the initial conditions.

The file follows a intuitive layout where the start and end of a section is indicated by `section>` and `<section` respectively. In the following paragraphs each section is exemplified in more detail based on the data file used to explore possible experiments for the Respressilator model shown in chapter 6.1.

### Model type

As a first step we define the type of model to be passed into `PEITH`($\Theta$) and to be simulated by `CUDA-sim`[3]. In its current version, `PEITH`($\Theta$) supports ODE models.

*Input format*: ODE

*Input type*: string

*Example*: The Repressilator model is simulated as an ODE model.

```
>type
ODE
<type
```

### Timestep (dt)

Next, we define an internal time step value for the ODE solver (`CUDA-Sim`'s [3] implementation of `LSODA` [6]).For stiff models it is recommended to use a smaller value.

*Input type*: positive float

*Example*: The internal time step is being set to `1` for the Repressilator example.

```
>dt
1
<dt
```

### Particles and samples

In this section we define the overall number of particles which will be simulated as well as samples sizes of $N_1$, $N_2$, $N_3$ and $N_4$ for the calculation of the mutual information(outlined in chapter 1). We provide $N_1$ and $N_2$ when performing *experiment selection for inference of all parameters*, $N_1$, $N_2$ and $N_3$ when performing *experiment selection for inference of a subset of parameters* and $N_1$, $N_2$, $N_3$ and $N_4$ when performing *experiment selection for prediction*. For all three approaches the sum of samples $N_i$ needs to be equal to the total particle number defined.

*Input format*: single value

white space separated list

*Input type*: positive integer

*Example*: We simulate `4,600,000` particles in order to perform "Experiment selection for inference of all model parameters" and we set N1 and N2 to `100,000` and `4,500,000`.

```
>particles
4600000
<particles

>nsample
100000 4500000 0 0
<nsample
```

**Sigma**

Here we provide the variance of the Normal distribution which describes the experimental noise. This noise is added following the simulation of the model trajectories.

*Input format*: positive float

*Example*: The variance of the experimental noise is being set to `5.0`

```
>sigma
5.0
<sigma
```

**Timepoints**

We define the time points at which the concentration of the measured species are obtained during simulation of the models. Time points need to be provided in an ascending order and without repetitions.

*Input format*: white space separated list

*Input type*: positive integer

*Example*: For the Repressilator we propose an experiment in which we are able to measure the species in 2 minutes intervals from 0 to 30.

```
>timepoint
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
<timepoint
```

### Posterior Sample

Next, we define whether a sample from posterior distribution is being provided instead of formal prior distributions. In order to do so, we enter `True` or `False` accordingly. In the case that a sample from a posterior distribution is being provided, we define the names of the files containing the sample of particles from posterior distribution and the associated weights in the second and third row respectively.

*Input format*: True or False
file name

*Input type*: string

*Example*: We are not providing a sample from a posterior distribution as the prior distributions are formally defined.

```
>samplefromposterior
False
data_2.txt
w_1.txt
<samplefromposterior
```

### Prior distribution of model parameter

If no sample from the posterior is provided we need to formally define prior distributions for all model parameters. In the current version, the following priors are supported: constant, uniform, normal, lognormal.

*Input format*: **constant** value **uniform** lower upper

**normal** mean variance **lognormal** mean variance

*Input type*: **string** float (float)

*Example*: Here we define the 4 model parameter to have $\mathcal{U}(1,10)$, $\mathcal{U}(0,20)$, $\mathcal{U}(500,2000)$ and $\mathcal{U}(0,10)$ respectively.

```
>prior
uniform 1 10
uniform 0 20
uniform 500 2000
uniform 0 10
<prior
```

### Prior distribution of initial conditions

Now we need to provide information if the initial conditions of the model are defined by prior distributions and thus become a model parameter them self or not by setting `initprior` to either `True` or `False`.

*Input format*: True or False

*Input type*: string

*Example*: Here the initial condition of the Repressilator model are not defined by prior distributions and thus `initprior` is set to `False`.

```
>initprior
False
<initprior
```

If we set `initprior` to `True`, we define the prior distribution for each initial condition of each species. In the current version the following priors are supported: constant, uniform, normal, lognormal.

*Input format*: **constant** value **uniform** lower upper

**normal** mean variance **lognormal** mean variance

*Input type*: string float (float)

*Example*: Here we define the 6 initial conditions of the model to have $\mathcal{U}$(1,10), $\mathcal{U}$(1,10), $\mathcal{N}$(15,2), $\mathcal{U}$(1,10), $\mathcal{N}$(20,1), and $\mathcal{U}$(1,10) respectively. However, as `initprior` is set to `False` these prior distribution are being ignored by the PEITH($\Theta$).

```
>initials
uniform 1 10
uniform 1 10
normal  15 2
uniform 1 10
normal 20 1
uniform 1 10
<initials
```

### Compartment

We define a constant value or prior distribution for the compartment sizes. In the current version the following priors are supported: constant, uniform, normal, lognormal.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Input format*: | **constant** | value | | **uniform** | lower | upper |
| | **normal** | mean | variance | **lognormal** | mean | variance |
| *Input type*: | string | float | (float) | | | |

*Example*: We define only one compartment with a constant value of `1` in the Repressilator model.

```
>compartment
constant 1
<compartment
```

### Measured species

We need to define the species measured during the experiments. We can define multiple sets of measured species, which can be seen as features of separate experiments (see *Combination*). Simple addition and subtraction of species are supported (see example).

*Input format*: All

speciesX speciesY-speciesZ

speciesX speciesY

*Input type*: string

white space separated list

*Example*: Here we define three different sets of measured species. If all species of the model are being measured simply define `All`. The second set defines the case when only `species1` and `species2` are measurable. The last set defines the scenario when we can only measure species1 and species2 in combination and species 3 by itself.

```
>fit
All
species1 species2
species1+species2 species 3
<fit
```

**Fits of parameter, compartments and initial conditions**

We can define fits for parameters, compartments and initial conditions. We set the fit to either all parameters (`All`), none of the parameters (`None`) or a subset of them. These fits will only be considered if *experiment selection for inference of a subset of model parameters* is being performed, while for the other two approaches the fits are automatically set to `All`, as no subsets of parameters are being considered.

*Input format*: All / None

parameter[index]

initial[index]

compartment[index]

*Input type*: string

whitespace seperated list

*Example*: We set the fit to `parameter1` and `parameter2`, `None` of the intial conditions and to `All` of the compartments.

```
>paramfit
parameter1 parameter2
<paramfit

>initialfit
None
<initialfit

>compfit
All
<compfit
```

### Initial Conditions

We can define multiple sets of initial conditions which are used to define separate experiments (see Combination). The initial conditions for each species will need to be defined by a constant value. A set of initial conditions is framed by `>Initial Conditions X` and `<Initial Conditions X`, where X is the index of the set.

*Input format*: **constant** value

*Input type*: string float

*Example*: Here we have defined two sets of initial conditions for the Repressilator model, framed by `>Initial Conditions 1` and `>Initial Conditions 2`. In the first set all species have an initial value of '0' except for the second species of the model which has a value of '1'. In the second set the first species has an initial value of '2', the second species has value of '3', the third species has value of '4', and so on.

```
>Initial Conditions 1
constant 0
constant 1
constant 0
constant 0
constant 0
constant 0
<Initial Conditions 1

>Initial Conditions 2
constant 2.0
```

```
constant 3.0
constant 4.0
constant 5.0
constant 6.0
constant 7.0
<Initial Conditions 2
```

### Perturbations

PEITH($\Theta$) provide the option of perturbing the original model given in SBML file. For example, we can define an experiment corresponding to a gene knockdown by multiplying a parameter for that reaction by a number $\xi \in (0, 1)$. An example of how this is given in the input data file is shown below:

| | | | |
|---|---|---|---|
| *Input format*: | **parameter** | multiplicative factor | reaction |
| *Input type*: | string | float | integer |

*Example*: Here we have one experiment that represents the original SBML file whilst the other experiment is a type of gene knockdown in which $\alpha_0$ is given a ten fold decrease within reactions 1, 3, and 5.

```
>Parameter - Experiment 1
Unchanged
<Parameter - Experiment 1

>Parameter - Experiment 2
alpha0 0.1 1
alpha0 0.1 3
alpha0 0.1 5
<Parameter - Experiment 2
```

Here notice we also give the option to run the original model defined in SBML by using `Unchanged`. On the other hand `Parameter - Experiment 2` can be understood as given in table 4.1.

| Parameter to change | Factor to change by | Reaction to change in |
|---|---|---|
| `alpha0` | 0.1 | 1 |
| `alpha0` | 0.1 | 3 |
| `alpha0` | 0.1 | 5 |

**Table 4.1: Describing perturbation**. Each row indicates the parameter to change, by what factor to change it by and to which reaction that parameter is associated to.

---

Where the reaction number corresponds to the order the reactions are given in the `SBML` file. More explicitly, suppose that in the original `SBML` file the first reaction is given by the function $r_1(\alpha_0, \boldsymbol{\theta}_{\alpha_0})$, where $\boldsymbol{\theta}_{\alpha_0}$ is the remaining parameters excluding $\alpha_0$. Then by the changes given above this then gives:

$$r_1(\alpha_0, \boldsymbol{\theta}_{\alpha_0}) \mapsto r_1(0.1 \cdot \alpha_0, \boldsymbol{\theta}_{\alpha_0}) \tag{4.2}$$

### Combinations

Lastly, we can set which experiments should be explored by `PEITH`($\Theta$) by defining experiments as combination of a set initial conditions, a parameter perturbation and the species measured.

*Input format*: **initset**[index] **paramexp**[index] **measured**[index]

*Input type*: string string string

*Example*: Here, we define 5 different experiments. While all experiments have the same initial conditions(`initset1 -> Initial Conditions 1`) and the same measurable species(`measured1`), we apply two different parameter perturbations(`paramexp`) for each experiment.

```
>combination
initset1 paramexp1 measured1
initset1 paramexp2 measured1
<combination
```

Alternatively, should we want to explore all possible combinations of the sets of initial conditions, parameter perturbation and measured species, the combinations can be set to "All".

```
>combination
All
<combination
```

## 4.3 Low Level Workflow Input

For this pipeline, also referred to as *using local code*, an input file written in `XML` is provided, which we refer to as the input `XML` file. This file contains the different experiments that will be explored. It makes further reference to a `CUDA` code file that is used by `CUDA-Sim` [3] that is also necessary to utilise `PEITH`($\Theta$). Here we provide a simple example of the input `XML` file and the `CUDA` code file for the Hes1 pathway as described by Liepe et al. [1]

### 4.3.1 input `XML` file

The input `XML` file follows a similar structure to the data file and can, generally speaking, be separated into two parts.

The first part of the input `XML` file defines general runtime arguments for the model simulation by `CUDA-sim` and mutual information estimation and global feature of the model-of-interest.

#### Number of experiments

First, we will need to define the number of experiment described in the input `XML` file.

*Input format*: single value

*Input type*: integer

*Example*: We define a single experiment in this input `XML` file.

```
<experimentnumber> 1 </experimentnumber>
```

### Model type

Next, the type of model being simulated by `CUDA-sim` needs to be defined. In its current version, PEITH($\Theta$) supports ODE models.

*Input format*: ODE

*Input type*: string

*Example*: We simulate the Repressilator model as an ODE model.

```
<type> ODE </type>
```

### Particles and samples

We set the overall particle number as well as the individual samples sizes $N_1$, $N_2$, $N_3$ and $N_4$ for the calculation of the mutual information. We provide $N_1$ and $N_2$ when performing *experiment selection for inference of all parameters*, $N_1$, $N_2$ and $N_3$ when performing *experiment selection for inference of a subset of parameters* and $N_1$, $N_2$, $N_3$ and $N_4$ when performing *experiment selection for prediction*. For all three approaches the sum of samples $N_i$ should be equal to the particle size defined.

*Input format*: single value

*Input type*: integer

*Example*: Here we perform *experiment selection for inference of a subset of parameters* and we set $N_1$, $N_2$ and $N_3$ to `1,000` and the total particle number to `3,000` accordingly.

```
<particles> 15000 </particles>
```

```
<nsamples>
<N1> 5000 </N1>
<N2> 5000</N2>
```

```
<N3> 5000 </N3>
<N4> 0 </N4>
 </nsamples>
```

### Timestep (dt)

Next we define an internal time step value for the `CUDA-sim` [3] ODE solver [6]. Consider using a smaller value for stiff model.

*Input type*: positive float

*Example*: Here we set the internal time step to `1.0` for the ODE solver.

```
<dt> 1.0 </dt>
```

### Time points

Furthermore, we define the time points at which the concentration of the measured species are obtained during simulation of the models. The values need to be provided in an ascending order.

*Input format*: white space separated list

*Input type*: positive integer/float

*Example*: In the experiments defined in this input `XML` file we are able to measure the species in 30 minutes intervals from 0 to 240 minutes.

```
<times> 0.0 30.0 60.0 90.0 120.0 150.0 180.0 210.0 240.0  </times>
```

### Sigma

Next we provide the variance of the Normal distribution of the experimental noise added to trajectories following the model simulation.

*Input format*: single value

*Input type*: float

*Example*: The variance of the experimental noise is being set to `0.1`

```
<sigma> 0.1 </sigma>
```

### Number of parameters

Here, we set the number of parameters, which are defined for each experiment in `<parameters> ... </parameters>`. All experiments defined in the same input `XML` are required to have the same number of parameters. The parameter number does not include compartments or initial condition, which are defined by the prior distributions.

*Input format*: single value

*Input type*: integer

*Example*: The Hes1 model used in this example has 4 model parameter.

```
<nparameters_all> 4 </nparameters_all>
```

### Prior distribution of initial conditions

Next, we define the initial conditions of the model, which can be defined by prior distributions, if required. In order to enable this feature we need to set `<initialprior> </initialprior>` to `True`.

*Input format*: True or False

*Input type*: string

*Example*: The initial condition of the Hes1 model are not defined by prior distributions and thus `<initialprior>` is set to `False`.

```
<initialprior> False </initialprior>
```

### Fits of parameter, compartments and initial conditions

Here, we can define fits for the parameters, compartments and initial conditions of the model. We can set the fit to either all parameters (`All`), none of the parameters (`None`) or a subset. These fits will only be considered for *experiment selection for inference of a subset of model parameters*. However, it is good practice to set the fits to `All` while running the two other approaches, as no subsets of parameters are being considered.

*Input format*: All / None
parameter[index]
initial[index]
compartment[index]

*Input type*: string
whitespace seperated list

*Example*: Here we set the fit to `parameter1`, `None` of the initial conditions and to `None` of the compartments.

```
<paramfit> parameter1 </paramfit>

<initfit> None </initfit>

<compfit> None </compfit>
```

### Posterior Sample

Lastly, we define if a sample from posterior is being provided instead of formally defined prior distributions. In order to do so, we enter `True` or `False` value accordingly for `<samplefrompost> </samplefrompost>`. In the case that a sample from a posterior is being provided we need to further define in `<samplefrompost_file> </samplefrompost_file>` and `<samplefrompost_weights> </samplefrompost_weights>` the names of the files containing the sample of particles from posterior and the associated weights respectively. Furthermore, if a posterior sample is provided it is good practice to set the model parameter provided by the sample to `posterior` individual

experiment sections within the input XML file.

*Input format*: True or False

file name

*Input type*: string

*Example*: We are not providing a sample from a posterior distribution as the prior and thus <samplefrompost> </samplefrompost> is set to False and no file names are provided.

```
<samplefrompost> False </samplefrompost>
<samplefrompost_file>  </samplefrompost_file>
<samplefrompost_weights>  </samplefrompost_weights>
```

### Experiments

The second part of the input XML file is being marked by <experiments> ... experiments and contains the individual descriptions of the experiment which will be explored by PEITH($\Theta$). Each experiment is defined within experimentX. ... experimentX borders.

*Example*: In this input XML we define a single experiment.

```
<experiments>
<experiment1>
.....
<experiment1>
</experiments>
```

### Experiment name and associated files

The following experimental features will need to be defined for every single experiment. First, we define the experiment name, this makes it easier to identify specific experiments later on. Next, we declare the associated SBML and CUDA code files. While the deceleration of the SBML is not essential when using the low level workflow input, it is good practise to define the source SBML, if it exists, here.

*Input format*: experiment name

file names

*Input type*: string

*Example*: The experiment is called `hes1`, the associated SBML file is called `hes1.xml` and the associated `CUDA` code file is called `hes1.cu`.

```
<name> hes1 </name>
<source> hes1.xml </source>
<cuda> hes1.cu </cuda>
```

### Measured species

We define which species are being measured during simulating the experiment. As shown for the data file simple addition and subtraction of species are supported (see data file subsection).

*Input format*: All

speciesX speciesY-speciesZ

speciesX speciesY

*Input type*: string

white space separated list

*Example*: For this experiment of the Hes1 model we are able to measure all species.

```
<measuredspecies> All </measuredspecies>
```

### Compartment

For each experiment we define a constant or prior distribution for the compartment sizes. In the current version the following priors are supported: constant, uniform, normal, lognormal. Note: If the compartment size is defined by a prior distribution, the same compartment prior should be defined for all experiments.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Input format*: | **constant** | value | | **uniform** | lower | upper |
| | **normal** | mean | variance | **lognormal** | mean | variance |
| *Input type*: | string | float | (float) | | | |

*Example*: We define only one compartment with a constant value of `1`.

```
<compartments>
<compartment1> constant 1 </compartment1>
</compartments>
```

### Initial conditions

Furthermore, each experiment requires a set of initial conditions. The initial conditions for each species are defined by either a constant value or prior distribution. In the current version the following priors are supported: constant, uniform, normal, lognormal. Note: If the initial conditions are defined by a prior distribution, the same initial condition prior should be defined for all experiments.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Input format*: | **constant** | value | | **uniform** | lower | upper |
| | **normal** | mean | variance | **lognormal** | mean | variance |
| *Input type*: | string | float | (float) | | | |

*Example*: Here we define the initial values of the three species of the Hes1 model as constants with values of `0.5`, `3.0` and `2.0` respectively.

```
<initial>
 <ic1> constant 0.5 </ic1>
 <ic2> constant 3.0 </ic2>
 <ic3> constant 2.0 </ic3>
</initial>
```

### Prior distribution of model parameter

Lastly, each model requires formally defined prior distributions for all model parameters if no sample from a posterior distribution is provided. In the current version the following priors are supported: constant, uniform, normal, lognormal. NOTE: The prior

distribution of the parameters are need to be the same across all experiments within one input `XML` file.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Input format*: | **constant** | value | | **uniform** | lower | upper |
| | **normal** | mean | variance | **lognormal** | mean | variance |
| *Input type*: | string | float | (float) | | | |

*Example*: Here we define the 4 model parameter to have $\mathcal{U}(0,2)$, $\mathcal{U}(1,10)$, $\mathcal{U}(0,0.1)$ and $\mathcal{U}(0,0.1)$ respectively.

```
<parameters>
<parameter1> uniform 0 2 </parameter1>
<parameter2> uniform 1 10 </parameter2>
<parameter3> uniform 0 0.1 </parameter3>
<parameter4> uniform 0 0.1 </parameter4>
</parameters>
```

### 4.3.2 `CUDA` Code

Here we provide a simple example of the `CUDA` code file for the Hes1 pathway as described by Liepe et al. [1] We then use this `CUDA` code file to demonstrate the low level workflow in section 6.2. `CUDA` code is an extension of `C++` and for a more in depth approach to constructing `CUDA` codes files see the manual for `CUDA-Sim` [3].

We first define the number of species, parameters and reactions. This is done by defining the following macros:

```
#define NSPECIES 3
#define NPARAM 4
#define NREACT 1
```

The basic framework for any `CUDA` code file (for ODEs) is a follows:

```
struct myFex{
    __device__ void operator()(int *neq, double *t, double *y,
    double *ydot/*, void *otherData*/)
    {
        int tid = blockDim.x * blockIdx.x + threadIdx.x;
        ...(insert ODEs here)...
```

```
    }
};
```

Inside this function are the corresponding ODEs in which parameters are referred to as `tex2D(param_tex,index,tid)` and species as `y[index]` where `index` is replaced by the integer corresponding to which species or parameter is being referenced:

```
ydot[0] = -0.03*y[0]+1/(1+powf(y[2]/tex2D(param_tex,0,tid),
          tex2D(param_tex,1,tid)));
ydot[1] = -0.03*y[1]+tex2D(param_tex,2,tid)*y[0]-
          (tex2D(param_tex,3,tid))*y[1];
ydot[2] = -0.03*y[2]+tex2D(param_tex,3,tid)*y[1];
```

So for this specific example (the Hes1 pathway) we have the following correspondence between the species and parameters to how they are referenced in the `CUDA` code:

| Parameter | `tex2D(param_tex,index,tid)` |
|---|---|
| $P_0$ | `tex2D(param_tex,0,tid)` |
| $h$ | `tex2D(param_tex,1,tid)` |
| $\nu$ | `tex2D(param_tex,2,tid)` |
| $k_1$ | `tex2D(param_tex,3,tid)` |

| Species | `y[index]` |
|---|---|
| $m$ | `y[0]` |
| $p_1$ | `y[1]` |
| $p_2$ | `y[2]` |

**Table 4.2: Relating parameters and species to `CUDA` code variables**. Here is an example of how to define the variable in the `CUDA` code for a specific parameter/specie in a model.

---

The final part to the `CUDA` code file is:

```
struct myJex{
    __device__ void operator()(int *neq, double *t, double *y,
    int ml, int mu, double *pd, int nrowpd/*, void *otherData*/){
        return;
    }
};
```

## 4.4 Weighted Sample for a Prior Distribution

The previous section allows for a weighted sample from a distribution (potentially a posterior sample from a previous experiment). Here we define the format which the weighted sample should be given to the package. Two separate `.txt` files are required: the first is a sample from the posterior, the second corresponds to the weights of each sample. Clearly, order is important since the first sample should correspond to the first weight given. Furthermore the number of samples must be the same as the number of weights.

The `.txt` for the sample must follow a specific, white-space separated, format: for each sample the first number is the compartment (if compartments are being used), the second is the parameter sample, and third are the initial values:

$$\underbrace{c_1}_{\text{compartments}} \quad \overbrace{p_1 \quad p_2 \quad \dots \quad p_{n-1} \quad p_n}^{\text{parameters}} \quad \underbrace{ic_1 \quad ic_2 \quad \dots \quad ic_{m-1} \quad ic_m}_{\text{initial values}}$$

An example of the sample `.txt` files is given below (the first three lines). Here the first number is the compartment, the next three correspond to the parameters whilst the final numbers are the initial conditions of the model:

```
1.0 1.88315996465 9.05892705551 849.576328268 0.0 1.0 0.0 0.0
1.0 2.13568343441 9.86839806977 1432.14518844 0.0 1.0 0.0 0.0
1.0 1.79521767204 9.45711920306 639.946449346 0.0 1.0 0.0 0.0
...
```

Below we see an example of the `.txt` file corresponding to the weights (the first three lines):

```
0.00163023859637
0.000815462965951
0.00186968463225
...
```

## 4.5 peitho

Now that we have described the input arguments requires, we outline how to invoke PEITH(Θ) through the command line for both workflows. Below we provide examples for each, highlighting the differences between the two.

### High Level Workflow

We give an example of running experimental selection when making prediction on all parameters. This is shown by the flag -a:

```
$ peitho -i1 SBML_file.xml -i2 input_data_file -a 0 -lc 0
         [-if=input_directory] [-of=output_directory]
```

The flags above correspond to the following:

| Flag | Meaning |
|---|---|
| -i1 | SBML files (high level) or input XML files (low level) |
| -i2 | Input data files |
| -a | Type of experimental prediction |
| -lc | Indicates local code (set to 0 if using an SBML file) |
| -if | Directory containing input arguments |
| -of | Directory to store output |

**Table 4.3: Flags for PEITH(Θ).** Outlines some of the flags that PEITH(Θ) can use.

---

More precisely for the flag -a:

In particular for experiment selection for prediction the command line call looks slightly different. To invoke this pipeline at least two SBML files are required, one for the reference model and the other corresponding to at least one experiment. An example of this is (for more details on -lc see section 4.5.1):

| Flag | Meaning |
|---|---|
| `-a 0` | Experiment selection for prediction on all parameters |
| `-a 1` | Experiment selection for prediction on a subset of parameters |
| `-a 2` | Experiment selection for prediction on outcome |

**Table 4.4: Using the `-a` flag.** In depth description of how to use the `-a` flag for the approach the user wishes to use

---

```
$ peitho -i1 SBML_file_reference.xml SBML_file_experiments.xml
         -i2 input_reference_data_file input_experiment_data_file
         -a 2 -lc 00 [-if=input_directory] [-of=output_directory]
```

#### Low Level Workflow

Using the low level pipeline requires one file less, in particular we now only require an input `XML` file. It is not necessary to also give the `CUDA` code file as this should be referenced from within the input `XML` file. An example is:

```
$ peitho -i1 input_xml_file -a 0 -lc 1 [-if=input_directory]
         [-of=output_directory]
```

Again to invoke the pipeline for experiment selection for prediction at least two input `XML` files are required (for more details on `-lc` see section 4.5.1):

```
$ peitho -i1 input_reference_xml_file input_experiment_xml_file
         -a 2 -lc 11 [-if=input_directory] [-of=output_directory]
```

### 4.5.1 Multiple `SBML` files or Local Code

It is also possible to run multiple `SBML` files or even multiple local codes as follows:

```
$ peitho -i1 SBML_file_1.xml SBML_file_2.xml
         -i2 input_data_1_file input_data_2_file
         -a 0 -lc 00 [-if=input_directory] [-of=output_directory]
```

```
$ peitho -i1 input_xml_1_file input_xml_2_file
         -a 0 -lc 11 [-if=input_directory] [-of=output_directory]
```

For both these examples notice that the `-lc` flag now has a digit (either `0` or `1`) for each file referenced within the `-i1` flag. Furthermore it should be emphasised that when using an `SBML` file (i.e. the high level workflow) that there should be an equal number of `SBML` files to input data files.

We have also designed the package to integrate both the high level and low level approach in a single command line call as follows:

```
$ peitho -i1 SBML_file_1.xml input_xml_1_file SBML_file_2.xml
         -i2 input_data_1_file input_data_2_file
         -a 0 -lc 101 [-if=input_directory] [-of=output_directory]
```

Order is important when giving these arguments to our package. The first `SBML` file must correspond to the first input data file. Furthermore, when using both `SBML` files and local code (as seen above) order is still important but clearly the number of input data files using the flag `-i2` will be different to the number of `SBML` and input `XML` files given by `-i1`.

### 4.5.2 Creating templates

In order to simplify writing local code we provide a template creator script. The script will create based on the SBML file provided a general template input `XML` file as well as CUDA code file. Using the `-tc` flag the template creator can be run on a single SBML file as follows:

```
$ peitho -tc -i1 SBML_file.xml [-if=input_directory]
         [-of=output_directory]
```

### 4.5.3 Checking Memory Requirements

The run time of `PEITH`($\Theta$) can be many hours. Thus, in order avoid issues with potential errors due to insufficient memory we include the function `memory_checker`. This

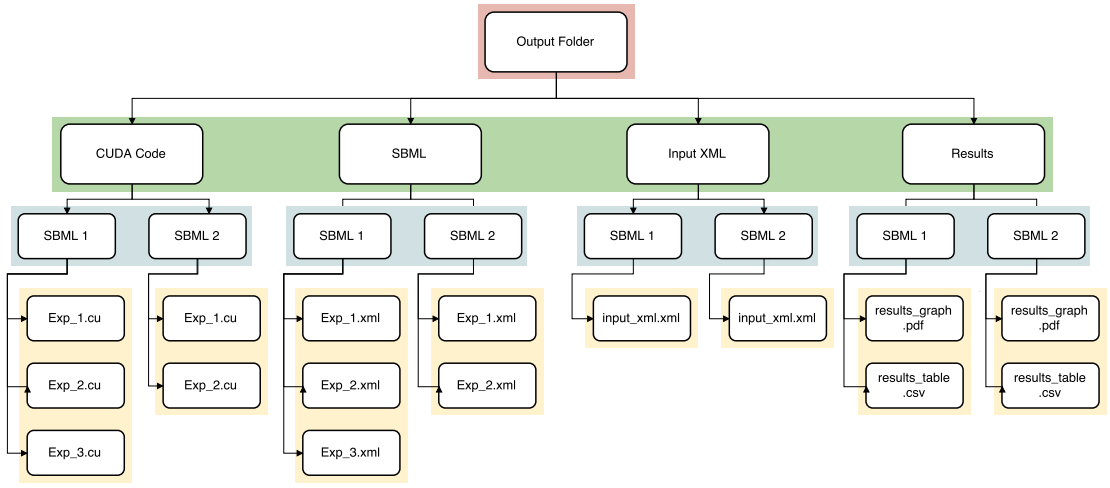



**Figure 4.1: Example of output directory structure when using the high level workflow and passing two `SBML` files.** Here, the top level, highlighted in red, is given the name passed to `PEITH`($\Theta$) using the flag `-of` otherwise it defaults to `Output_Folder`. The second tier, highlighted in green, shows four sub-directories: `CUDA` code (holds `CUDA` code files generated during parsing in `PEITH`($\Theta$)), `SBML` (holds `SBML` files generated during parsing in `PEITH`($\Theta$)), input `XML` (holds input `XML` files generated during parsing in `PEITH`($\Theta$)), and `Results` (holds the results of running `PEITH`($\Theta$) such as mutual information etc). The third tier, highlighted in blue, splits each directory in the green tier into the corresponding `SBML` file. The final yellow tier indicates the files that are in each of the blue tier directories.

---

provides a rough estimate of the RAM requirements needed to run `PEITH`($\Theta$) for each of the three objectives. To invoke `PEITH`($\Theta$)'s `memory_checker` use the flag `-mc` or `--memory_check`, an example is:

```
$ peitho -mc -i1 SBML_file.xml -i2 input_data_file -lc 0 -a 0
         [-if=input_directory] [-of=output_directory]
```

When conducting experiment selection for prediction of output (i.e. for `-a 2`) then an example is:

```
$ peitho -mc -i1 SBML_file_ref.xml SBML_file_exp.xml
         -i2 input_data_file_ref input_data_file_exp -lc 00 -a 0
         [-if=input_directory] [-of=output_directory]
```

An exemplar output is:

```
Estimated RAM requirements: 9.45316547304  GB
```

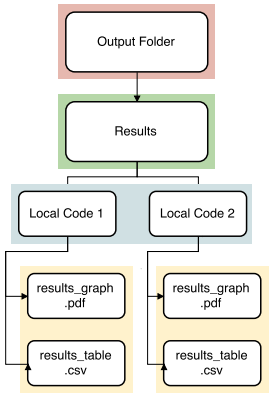



**Figure 4.2: Example of output directory structure when using the low level workflow and passing two input `XML` files.** Here, the top level, highlighted in red, is given the name passed to `PEITH`(Θ) using the flag `-of` otherwise it defaults to `Output_Folder`. The second tier, highlighted in green, shows the `Results`, which holds the results of running `PEITH`(Θ) such as mutual information etc. The third tier, highlighted in blue, splits each directory in the green tier into the corresponding local code. The final yellow tier indicates the files that are in each of the blue tier directories.

---

It should be noted that for experiment selection for inference of all parameters (i.e. `-a 0`) or experiment selection for inference of a subset of parameters (i.e. `-a 1`) only one `SBML` file (high level workflow) or input `XML` (low level workflow) file can be passed at one time (see Troubleshooting).

## 4.6 Output

The structure of the output folder varies between using the high level and low level workflow. Within the high level workflow, alongside the results, there are also sub-directories containing `CUDA` code, `SBML` and input `XML` files as shown in and example in Figure 4.1. On the other hand, for the low level path only the results are returned since the `CUDA` code and input `XML` files are given as arguments, an example of this is given in Figure 4.2. Notice that when using the high level pipeline the naming of files is automatic and done in the order given by the input data file. Next we describe the format of the main output file given by `PEITH`(Θ): the `CSV` file. Alongside this file `PEITH`(Θ) also provides `pdf`s of bar graphs with the mutual information for each experiment.

### 4.6.1 `CSV` file

The `CSV` file is the primary output given by `PEITH`($\Theta$). Alongside the mutual information for each experiment, we also provide the *parameters* (i.e. parameters here is referring to the overall experiment for example the number of particles, time points as opposed to the actually parameters used within the mathematical representation of an experiment). The `CSV` file is semi-colon (`;`) separated with the text delimiter set to quotation marks (`"`). The table below outlines the information returned in the `CSV` file along with an example:

| **Column Name** | **Meaning** | **Example** |
|---|---|---|
| Experiment Name | Name of the experiment given in input `XML` file | Experiment1 |
| Type | Currently set to 0 | 0 |
| Measured Species | Which species, and how they are measured. The species are ordered as given in the `SBML` or input `XML` file | [[0],[1],[2]+[3]] |
| Approach | Indicates whether it is prediction for all parameters (0), a subset of parameters (1) or prediction of output (2) | 0 |
| Fitted Parameters | *Only when conducting approach 1.* Indicates which parameters are to be fitted. The species are ordered as given in the `SBML` or input `XML` file | [0,1] |
| Fitted Initials | *Only when conducting approach 1.* Indicates which initial conditions are to be fitted (when they are treated as parameters). The initial conditions are ordered as given in the `SBML` or input `XML` file | [1,2] |

| | | |
|---|---|---|
| Fitted Compartments | *Only when conducting approach 1.* Indicates which compartments are to be fitted (when they are treated as parameters). The compartments are ordered as given in the `SBML` or input `XML` file | [0] |
| Total Particles | Total number of particles used when calculating the mutual information for the experiment. Note this is the sum of $N_i$ for $i \in \{1, 2, 3, 4\}$. | 4600000 |
| $N_1$ | Size of $N_1$ | 100000 |
| $N_2$ | Size of $N_2$ | 4500000 |
| $N_3$ | Size of $N_3$ (given when using approach 1 or 2) | 0 |
| $N_4$ | Size of $N_4$ (given when using approach 2) | 0 |
| Sigma | Standard deviation of the Gaussian noise | 5 |
| dt | Specifies time step distance for the `LSODA` algorithm implemented in `CUDA-Sim` (see `CUDA-Sim` for more details) [3] | 1 |
| `CUDA` file | Name of the `CUDA` code file | Exp_1.cu |
| `SBML` file | Name of the `SBML` file (if high level workflow used) | Exp_1.xml |
| Data Input file | Name of the input data file (if high level workflow used) | input_data.data |
| Posterior Sample | Indicates whether the prior was given as a weighted sample | False |
| Posterior Sample Particles | Name of the file containing a sample from the given weighted sample (if user gives a weighted sample) | post_sample_particles |

| | | |
|---|---|---|
| Posterior Sample Weights | Name of the file containing the weights from the given weighted sample | post_sample_weights |
| Total `Infs/NaNs` | Total number of `Infs` or `NaNs` encountered during calculation of mutual information | 5 |
| Percentage `Infs/NaNs` | Percentage of `Infs` or `NaNs` encountered during calculation of mutual information | 1 |
| Percentage `Infs/NaNs` | Percentage of `Infs` or `NaNs` encountered during calculation of mutual information | 1 |
| Mutual Information | Approximation of mutual information for the experiment | 123.456 |

**Table 4.5: Description of the `CSV` file.**

# 5. Details

## 5.1 Optimisation of GPU kernel launch configuration

Carrying out computations on a GPU can dramatically reduce computing time compared to normal CPU computations. It should be noted however, that an in-depth understanding of the GPU's highly parallel architecture is usually required in order to exploit the full computational power of the GPU [7, 8] (a brief overview of GPU architecture can be found in appendix B). Even if a program has been fully optimised to achieve minimal runtime on a certain GPU device, it is unlikely that the same configuration would also be optimal for other devices. [9]

To alleviate this problem, `PEITH`($\Theta$) comes with a customised implementation of the CUDA occupancy calculator [10], enabling automated launch configuration in a `PyCUDA`-based code framework.

The launch configurator reads the GPU properties directly from the utilised device (or where unavailable, infers them from the detected compute capability). This allows for optimal launch configuration and minimal runtime irrespective of the employed GPU hardware (conditional a device with compute capability between 2.0 and 6.2).

### 5.1.1 Block size and shape

The launch configurator aims to detect the minimal block size at which maximum occupancy of the GPU is achieved. Considering that our grid on the GPU has a two dimensional square shape, the block shape is set to as close to square as possible to match the grid dimensions (i.e. a block of 96 threads would have the dimensions 8x12 rather than 6x16).

### 5.1.2 Grid size and shape

Based on the maximum grid size (dependent on the available global device memory), the dimensions of the grid are determined to be close to square while remaining a multiple of the block size in both dimensions. In runs with small particle numbers, the grid size might exceed the number of particles in one of the dimensions. In this case the shape of the grid is adapted to accommodate this. Furthermore, the grid dimensions are limited by the device properties, which is considered as well.

## 5.2 Mutual information calculations in detail

Following the parsing of input files, simulation of the trajectories and preprocessing of the data, the mutual information is computed in the `mutInfo` functions. As mentioned in the package outline (section 2.2), there is a separate `mutInfo` function for each of the three objectives.

In order to optimise computational performance and to alleviate numerical issues, the calculation of the mutual information has become rather fragmented in the code throughout development. We therefore concluded that it would be beneficial to outline the structure of the three `mutInfo` functions in some detail in order to facilitate understanding of the workings of our code. We hope this will facilitate further development and adaptation of our implementation to fit novel research problems.

This section is further supported by comments in the code which are meant to provide a more detailed explanation of the code than this high-level structural overview. For further details on the mathematical context of the calculations explained here, please consult section 1 or appendix C of this report as well as the original publication by Liepe et al. [1]

### 5.2.1 Nomenclature and data structure

Here we give a brief overview of the naming conventions and structure of the data in the calculations by using experiment selection for inference of all parameters (`mutInfo1`). We provide Figure 5.1 to support our explanation.

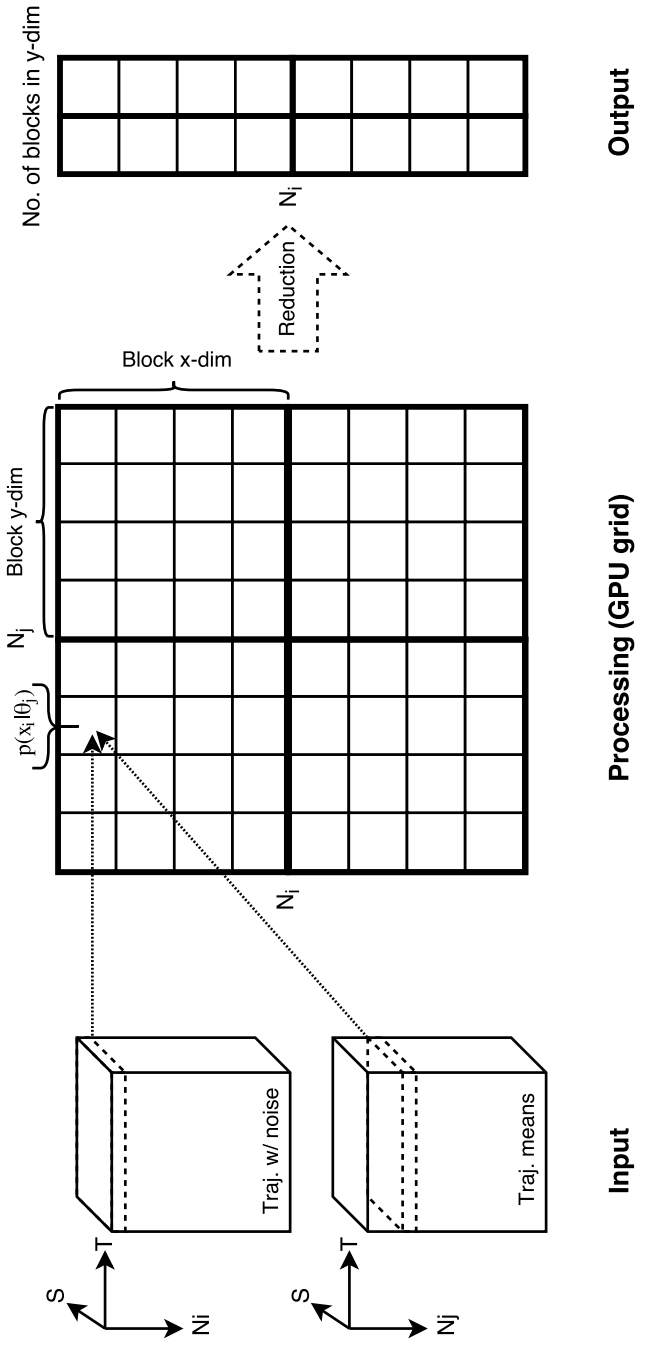



**Figure 5.1: Schematic representation of the data flow on the GPU for a single kernel invocation** with $N_i = N_j = 8$. The input represents two three-dimensional arrays, the first one holding all $x_i$ and the second one holding all $\theta_j$ (for all time points $t \in T$ and all species $s \in S$). The central unit represents a grid with 4 blocks, containing 16 threads each. Each of the threads processes a unique combination of $x_i$ and $\theta_j$.

Calculating the mutual information involves the calculation of the probability density function (pdf) of the normal distribution many times. In an instance of `mutInfo1`, the probability $p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big)$ is evaluated a total $N_1 \cdot N_2 \cdot T \cdot S$ times for each experiment. $N_1$ and $N_2$ are the number of particles specified, $T$ corresponds to the number of time points the system is simulated for and $S$ to the number of species present in the model.

Besides constants, such as the standard deviation $\sigma$, the input for the calculation of the mutual information in `mutInfo1` is comprised of two three-dimensional arrays for each experiment.

The first array is of the dimension $N_1 \times T \times S$, and contains $x_{t,s}^{(i)}, \quad \forall i \in \{0, \ldots, N_1 - 1\}, t \in \{0, \ldots, T-1\}, s \in \{0, \ldots, S-1\}$, where $\mathbf{x}_t^{(i)}, \quad \forall t \in \{0, \ldots, T-1\}$ represents the simulated trajectory of all species for a single particle, $i$, over time, with Gaussian noise added.

The second array is of the dimension $N_2 \times T \times S$, and contains $\boldsymbol{\mu}_{t,s}^{(j)}, \quad \forall j \in \{0, \ldots, N_2 - 1\}, t \in \{0, \ldots, T-1\}, s \in \{0, \ldots, S-1\}$, where $\boldsymbol{\mu}_t^{(j)}, \quad \forall t \in \{0, \ldots, T-1\}$ represents the mean simulated trajectory of all species and a single particle, $j$, over time. As $\boldsymbol{\mu}^{(j)}$ depends directly on $\boldsymbol{\theta}^{(j)}$, they are used interchangeably in the following sections as well as equations 1.4, 1.5, 1.6.

As these two arrays become very large for large particle sizes (i.e. large $N_1$ and $N_2$) their processing can exceed the available grid size on the GPU. This is because the grid shape is determined as $N_1 \times N_2$, i.e. thread $(2, 5)$ in the grid computes the probability $\prod_{t \in T} \prod_{s \in S} p\big(x_{t,s}^{(2)}|\theta_{t,s}^{(5)}\big)$. To alleviate this problem, the input arrays are sliced into $n$ and $m$ smaller arrays of shape $N_i \times T \times S$ and $N_j \times T \times S$ respectively, so that $n \cdot N_i = N_1$ and $m \cdot N_j = N_2$. These arrays are then processed on the GPU in separate kernel calls.

### 5.2.2 Experiment selection for inference of all parameters (`mutInfo1`)

#### Code structure

The repeated calculation of $p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big)$ constitutes the most computationally expensive part of computing the mutual information as given in equation 1.4 and is hence carried out on the GPU for increased performance. As mentioned previously, only a subset the

$\mathbf{x}_q^{(i)}$ and $\boldsymbol{\theta}^{(j)}$ can be loaded and computed on the GPU at any one time due to restriction of the grid size. The mutual information is therefore calculated iteratively for subsets of the input arrays of the dimensions $N_i \times T \times S$ and $N_j \times T \times S$ respectively, where $n \cdot N_i = N_1$ and $m \cdot N_j = N_2$.

As a first step, the optimal GPU block/grid size and shape is determined using the launch configurator described in section 5.1. As the size of the grid corresponds to the maximum number of particles processed per kernel invocation, the number of calls to the GPU kernel can be determined next (noted as $n$ and $m$ above). As $n$ and $m$ are not usually a divisor of $N_1$ and $N_2$ respectively, $N_i$ and $N_j$ take different numbers in the $n$-th and $m$-th iteration of the outer and inner for-loop respectively. For this reason, $N_i$ and $N_j$ are dynamically computed before every kernel invocation.

The GPU kernel itself is invoked in a nested for-loop where the outer loop iterates over $\{0, \ldots, n-1\}$ and the inner loop over $\{0, \ldots, m-1\}$. This ensures that $p\big(x_q^{(i)}|\theta^{(j)}\big)$ is computed for all $\theta^{(j)}$ given a single $x_q^{(i)}$.

Upon completion of a kernel invocation (details on the calculations to be found in the next section), results are returned from the GPU and a second time reduced along one dimension which can be written as $\sum_{j \in N_j} p\big(x_q^{(i)}|\theta^{(j)}\big), \forall i \in \{0, \ldots, N_i - 1\}$. For this and the following summations, `inf` and `nan` values (arising on the GPU for numerical reasons) are masked to allow for the summations to execute. The dimensions of the results at this stage are therefore $N_i \times 1$ for each GPU run. These are then iteratively appended to an array eventually holding the results for all runs and consequently being of the dimensions $N_1 \times m$. A final reduction along the same dimension as before, followed by the evaluation of the natural logarithm of each value, yields a vector containing the results of the following:

$$\log\left(\sum_{j=N_1+1}^{N_1+N_2} \exp\left(\frac{-(\mathbf{x}_i - \boldsymbol{\mu}_j)^2}{2\sigma^2}\right)\right), \quad \forall i \in \{0, \ldots, N_i - 1\} \tag{5.1}$$

The following step then involves summation of all the $N_1$ elements as well as addition of various constants to yield a scalar intermediate result according to the following

expression:

$$\sum_{i=1}^{N_1} -\log\left(\frac{1}{N_2}\sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big)\right) \tag{5.2}$$

In the last step, the previous result is divided by $N_1$ (less any masked `inf` or `nan` values) and the first term of equation 1.4 is added to the expression in the form of the analytic solution to the differential entropy of a normal distribution, $\log\left(\sigma\sqrt{2\,\pi\,e}\right)$ (this replaces the first term of the 1.4). This result is then equal to the expression for the mutual information as initially presented in equation 1.4:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) \approx \frac{1}{N_1}\sum_{i=1}^{N_1}\left[\log\left(p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(i)}\big)\right) - \log\left(\frac{1}{N_2}\sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big)\right)\right]$$

**Computations on the GPU**

The main computation of $p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big)$ takes place on the GPU where the following is calculated for each invocation of the GPU kernel function:

$$\prod_{s=0}^{S-1}\prod_{t=0}^{T-1}\exp\left(\frac{-(x_{s,t}^{(i)}-\mu_{s,t}^{(j)})^2}{2\sigma^2}\right), \quad \forall i \in \{0,\ldots,N_i-1\}, j \in \{0,\ldots,N_j-1\} \tag{5.3}$$

Where $T$ corresponds to the total number of time points for each simulated trajectory and $S$ corresponds to the total number of species present in the model. This is elsewhere in the report presented in its simplified form:

$$d_{i,j} = \exp\left(\frac{-(x_i-\mu_j)^2}{2\sigma^2}\right), \quad \forall i \in \{0,\ldots,N_i-1\}, j \in \{0,\ldots,N_j-1\} \tag{5.4}$$

As equation 5.4 is evaluated for the current grid, the output data $d_{i,j}$ of each thread (subunit of a block) is stored in the associated shared memory of its associated block (group of threads, subunit of the grid).

As a next step, $D$ is reduced along the y-dimension of the block (blockDim.y) as follows:

$$d_{i,0} = \sum_{j=0}^{blockDim.y-1} d_{i,j}, \quad \forall i \in \{0,\ldots,blockDim.x-1\} \tag{5.5}$$

The resulting array from block 1 $\mathbf{d}_0 = (d_{0,0}, \ldots, d_{blockDim.x,0})^\top$ is then transferred to global memory, combined with the remaining $B-1$ $\mathbf{d}_b$ from the other blocks in the grid, and returned to the host as an array of dimensions $N_i \times B$ where $B$ corresponds to the total number of blocks along the y-dimension of the grid.

### 5.2.3 Experiment selection for inference of a subset of model parameters (`mutInfo2`)

When comparing equations 1.4 and 1.5, it becomes apparent that the computations are analogous for the second term but the first term can no longer be approximated by the analytic solution of the differential entropy of a normal distribution. In this section we hence describe the computation of the first term of equation 1.5.

#### Code structure

The `mutInfo2` function is divided into two separate sections. The first section contains the computations of the second term of equation 1.5 and can therefore be understood by reading the outline of the code structure for `mutInfo1` (section 5.2.2). The second section of the code contains the computation of the first term of equation 1.5, the structure of which is outlined here.

A main difference in the structure of the computation for the first term is that unlike before, there is a separate set of $N_3$ mean trajectories (i.e. $\boldsymbol{\theta}^{(i,j)}$) for each of the $N_1$ $\mathbf{x}_q^{(i)}$ (this is also the reason for why simulation of trajectories take disproportionately longer compared to `mutInfo1` so that the actual calculation of mutual information constitutes only a negligible part of the runtime for `mutInfo2`). To account for this structural difference, each of the iterations of the nested for-loop only processes the data for a single one of the $N_1$ trajectories (hence only a one-dimensional grid is launched on the GPU at any one kernel invocation).

Once the size of the one-dimensional block and grid has been determined using the integrated launch configurator, the number of iterations of the inner for-loop, iterating over subsets of the respective set of $N_3$ trajectories per $\mathbf{x}_q^{(i)}$, is determined. As previously,

the calculation of the exact particle number for each run and the slicing of the respective input array is done prior each kernel launch.

The processing of the results from every GPU run is again very similar to the previous section and is hence only discussed briefly here (see section below for detail on the calculations on the GPU). As before, one dimension of the results array returned from the GPU is reduced. As the GPU for this part of the calculation produces a one-dimensional array, said reductions lead to a scalar intermediate result, which after addition of further constants is represented by the following expression:

$$\sum_{i=1}^{N_1}\left[\log\left(\frac{1}{N_3}\sum_{j=1}^{N_3} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(i,j)}\big)\right)\right] \tag{5.6}$$

The result for the second term of equation 1.4 is computed as described in section 5.2.2 and is represented by the following expression:

$$\sum_{i=1}^{N_1}\left[\log\left(\frac{1}{N_2}\sum_{k=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(k)}\big)\right)\right] \tag{5.7}$$

Subtracting 5.7 from 5.6 and dividing by $N_1$ (less any `inf` and `nan` as before) gives the mutual information as given in equation 1.5:

$$\mathcal{I}(\boldsymbol{\Theta}_c, \mathbf{X}_q) \approx \frac{1}{N_1}\sum_{i=1}^{N_1}\left[\log\left(\frac{1}{N_3}\sum_{j=1}^{N_3} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(i,j)}\big)\right) - \log\left(\frac{1}{N_2}\sum_{k=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(k)}\big)\right)\right]$$

**Computations on the GPU**

We only describe the second kernel function here (evaluating the first term of equation 1.5), as the first function is analogous to the one described in section 5.2.2.

The calculations constituting the second GPU function, similarly to before, can be summarised by the following expression:

$$\prod_{s=0}^{S-1}\prod_{t=0}^{T-1}\exp\left(\frac{-(x_{s,t}^{(i)}-\mu_{s,t}^{(i,j)})^2}{2\sigma^2}\right), \quad \forall j \in \{0,\ldots,N_j-1\} \tag{5.8}$$

Note, that here, each invocation of the GPU kernel (i.e. each grid) only contains $x^i_{s,t}$ for a single $i$ since different sets of $\mu^{(i,j)}_{s,t}$ are required for each $x^i_{s,t}$.

The results of above expression are then block-wise reduced in shared memory as described in the previous section with the difference that in this function $i = 1$. This results in a scalar being returned from each block to global device memory rather than a vector as before, which in turn means that the dimensions of the array returned to the host has the dimensions $N_j \times 1$, where $N_j$ is the number of $\mu^{(i,j)}$ in the grid.

### 5.2.4 Experiment selection for prediction (`mutInfo3`)

We can see, when comparing equation 1.4 with equation 1.6, that the calculation of the second and third term of equation 1.6 are analogous with the calculation laid out in section 5.2.2 (the reason the term appears twice in equation 1.6 is that it needs to be calculated for the reference model as well as the proposed alternative model). In this section we hence only describe the computation of the first term of equation 1.6.

#### Code structure

The `mutInfo3` function is divided into three separate sections. The last two sections contain the computations of the second and third term of equation 1.6 and can therefore be understood by reading the outline of the code structure for `mutInfo1` (section 5.2.2). The first term of equation 1.6 is computed in the second section of the code, the structure of which is outlined here.

As previously, the optimal GPU block/grid size and shape is determined using the launch configurator. As the size of the grid corresponds to the maximum number of particles processed per kernel invocation, the number of calls to the GPU kernel is be determined next.

The structure of the nested for-loop is analogous to the one described in section 5.2.2, however, invoking a different calculation on the GPU (see section below for details). Considering the same processing of each of the GPU results as described in section

5.2.2, we obtain results represented by the following expression:

$$\sum_{i=1}^{N_1} \log\Bigg( \sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{y}^{(i)}|\boldsymbol{\theta}^{(j)}\big) p\big(\mathbf{x}^{(i)}|\boldsymbol{\theta}^{(j)}\big) \Bigg) \tag{5.9}$$

Where $\mathbf{y}^i$ corresponds to to the reference model and $\mathbf{x}^{(i)}$ to the alternative model.

The results for the second and third term is computed as described in section 5.2.2 and are represented by the following expressions:

$$\sum_{i=1}^{N_1} \log\Bigg( \sum_{k=N_1+N_2+1}^{N_1+N_2+N_3} p\big(\mathbf{y}^{(i)}|\boldsymbol{\theta}^{(k)}\big) \Bigg) \tag{5.10}$$

$$\sum_{i=1}^{N_1} \log\Bigg( \sum_{l=N_1+N_2+N_3}^{N_1+N_2+N_3+N_4} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(l)}\big) \Bigg) \tag{5.11}$$

Subtracting 5.10, 5.11 and $\log(N_x) \forall x \in \{2,3,4\}$ from 5.9 and dividing by $N_1$ (less any `inf` and `nan` as before) gives the mutual information as given in equation 1.6:

$$\begin{aligned} \mathcal{I}(\mathbf{Y}, \mathbf{X}_q) \approx & \frac{1}{N_1} \sum_{i=1}^{N_1} \Bigg[ \log\Bigg( \frac{1}{N_2} \sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{y}^{(i)}|\boldsymbol{\theta}^{(j)}\big) p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}\big) \Bigg) \\ & - \log\Bigg( \frac{1}{N_3} \sum_{k=N_1+N_2+1}^{N_1+N_2+N_3} p\big(\mathbf{y}^{(i)}|\boldsymbol{\theta}^{(k)}\big) \Bigg) \\ & - \log\Bigg( \frac{1}{N_4} \sum_{l=N_1+N_2+N_3}^{N_1+N_2+N_3+N_4} p\big(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(l)}\big) \Bigg) \Bigg] \end{aligned}$$

**Computations on the GPU**

We only describe the first kernel function here (evaluating the first term of equation 1.6), as the second function is analogous to the one described in section 5.2.2.

The computation in the first GPU function is represented by the following expression:

$$\prod_{s=0}^{S_r-1} \prod_{t=0}^{T_r-1} \exp\left( \frac{-(y_{s,t}^{(i)} - \mu_{s,t}^{(j)})^2}{2\sigma^2} \right) \cdot \prod_{s=0}^{S_a-1} \prod_{t=0}^{T_a-1} \exp\left( \frac{-(x_{s,t}^{(i)} - \mu_{s,t}^{(j)})^2}{2\sigma^2} \right),$$
$$\forall i \in \{0, \ldots, N_i - 1\}, j \in \{0, \ldots, N_j - 1\} \tag{5.12}$$

Where $T_r$ and $T_a$ correspond to the total number of time points for each simulated trajectory of the reference and alternative model respectively and $S_r$ and $S_a$ correspond to the total number of species present in the reference and alternative model respectively.

The results of above expression are then block-wise reduced in shared memory as described in section 5.2.2 and returned to global memory. Next, the result arrays of all blocks in the grid $\mathbf{d}_0 = (d_{0,0}, \ldots, d_{blockDim.x,0})^\top, \forall b \in B$ are transferred to global memory, combined and returned to the host.

# 6. Examples

In this section we exemplify three different usage scenarios of PEITH($\Theta$) based on three different models. The Repressilator is used to demonstrate *Experiment selection for inference of all model parameters*, Hes1 for *Experiment selection for inference of a subset of model parameters*, and p53 for *Experiment selection for prediction.* All examples are available as part of the installation of PEITH($\Theta$). For each experiment we run PEITH($\Theta$) three times.

## 6.1 Repressilator model

To exemplify our pipeline for *Experiment selection for inference of all model parameters*, we use the Repressilator model [5] (as done in the original publication [1]). We start by providing a brief background to model as well as the experiments we consider comparing, followed by the results.

### 6.1.1 Background

The Repressilator is a commonly employed synthetic model of transcriptional regulation. It consist of three proteins (p1, p2, p3) which each inhibit the expression of mRNA (m1, m2, m3) of the next protein in a loop-like structure (see Figure 6.1).

### 6.1.2 Experiment selection for inference of all model parameters

In this model, we would like to infer all four parameters $h, \alpha, \alpha_0$ and $\beta$ and thus we would like to identify the experiment which provides the most information over all parameters . As in the original publication, we determine the mutual information for

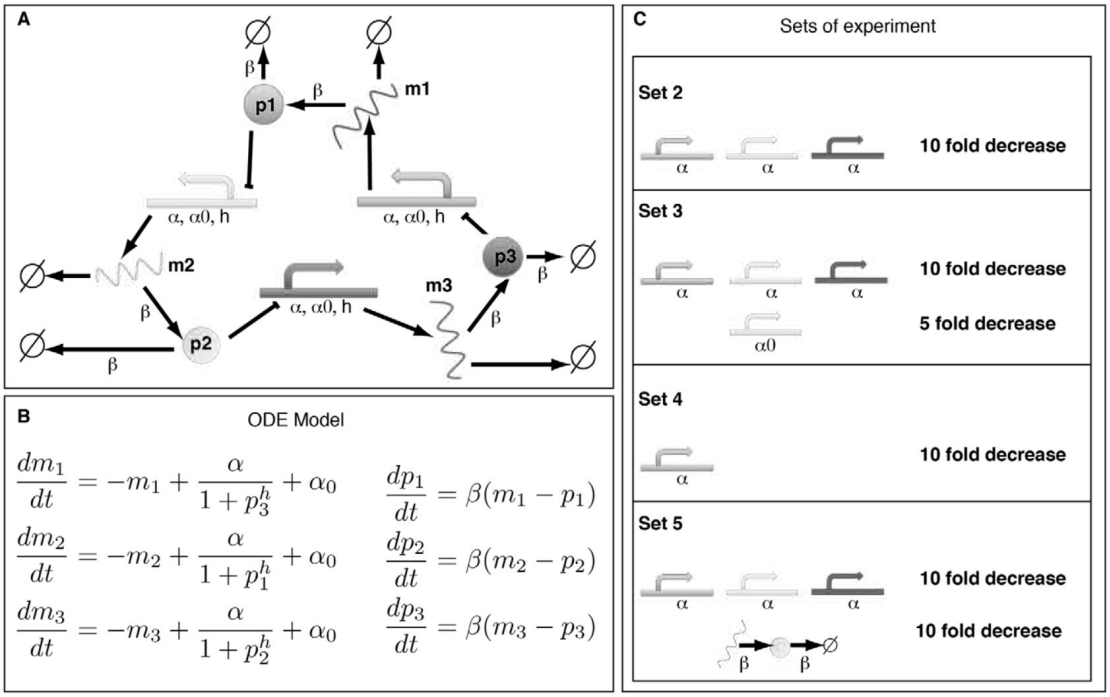



**Figure 6.1: Repressilator model.**(A) Schematic of the Repressilator model, consisting of three proteins (coloured circles labeled *p1, p2, p3*) and their associated mRNA (wavy lines in same colour labeled *m1, m2, m3*. The associated DNA regions are only dispalyed for illustration purposes as they are not included in the model itself. (B) Ordinary differential equations representing the concentration of the modelled species over time. (C) Four proposed perturbations to the Repressilator model to be carried out as experiments alongside the wild-type model. Adapted with permission from the original publication [1]

---

the five different experiments with the parameters as outlined in Figure 6.1 (C) with Model Set 1 representing the wild-type model.

All experiments have a measurement noise with a variance $\sigma = 5$ and initial conditions $(m1, p1, m2, p2, m3, p3) = (0, 1, 0, 0, 0, 0)$. Furthermore, the priors over the parameters are as follows:

$$h \sim \mathcal{U}(1, 10) \quad \alpha_0 \sim \mathcal{U}(0, 20) \quad \alpha \sim \mathcal{U}(500, 2000) \quad \beta \sim \mathcal{U}(0, 10)$$

We use a `SBML` file as the input and invoke `PEITH`($\Theta$) with the following command line call:

```
$ peitho -i1 repressilator.xml -i2 repressilator.data -a 0 -lc 0
         -if=Example_data/Repressilator -of=results/Repressilator
```

In particular the defining sections of `repressilator.data` are:

```
>timepoint
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
<timepoint

>measuredspecies
All
<measuredspecies

>Initial Conditions 1
constant 0
constant 1
constant 0
constant 0
constant 0
constant 0
<Initial Conditions 1

>Parameter - Experiment 1
Unchanged
<Parameter - Experiment 1

>Parameter - Experiment 2
alpha 0.1 1
alpha 0.1 3
alpha 0.1 5
<Parameter - Experiment 2

>Parameter - Experiment 3
alpha 0.1 1
alpha 0.1 3
alpha 0.1 5
alpha0 0.2 3
<Parameter - Experiment 3

>Parameter - Experiment 4
alpha 0.1 1
<Parameter - Experiment 4

>Parameter - Experiment 5
alpha 0.1 1
alpha 0.1 3
alpha 0.1 5
beta 0.1 9
beta 0.1 10
<Parameter - Experiment 5


>combination
```

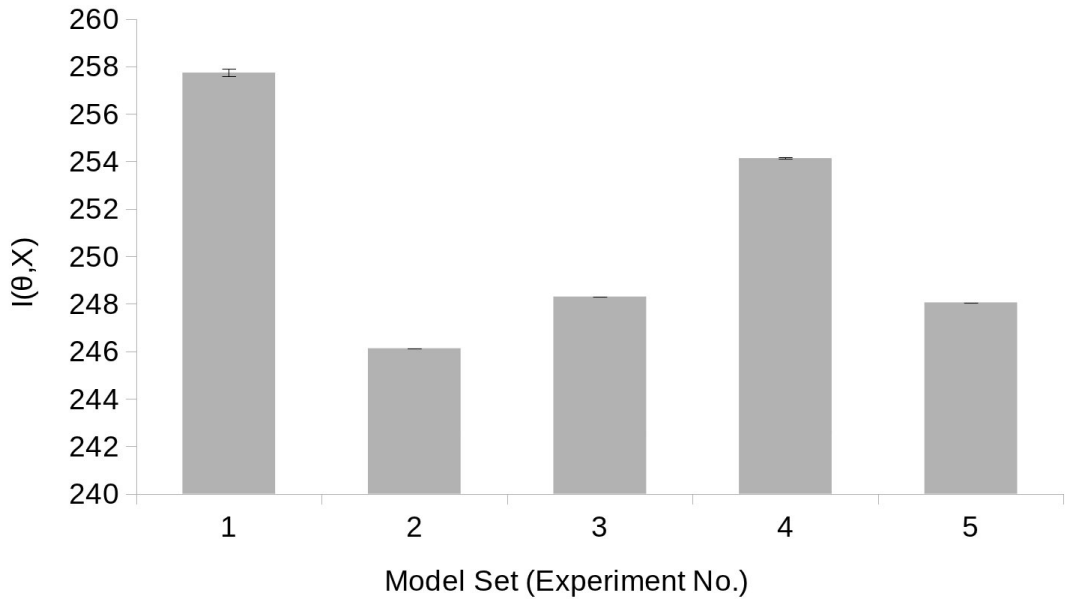



**Figure 6.2: Mutual information for the Repressilator model example.** The model sets are as described in Figure 6.1. The error bars indicate the standard error from three repeats of PEITH(Θ) whilst the bars indicate the mean over those same three repeats. Note: Full y-axis not shown.

---

```
initset1 paramexp5 measured1
<combination
```

We run this model with a total particle size of 4600000 where: $N_1 = 100000$ and $N_2 = 4500000$. The complete input data files are available with the installation of PEITH(Θ).

### 6.1.3 Results

Running the Repressilator model [5] with PEITH(Θ), we obtain the results as shown in Figure 6.2. They suggest that the top three experiments to conduct are:

1. Wildtype (Model Set 1) - *Mutual Information = 257.736*
2. 10-fold decrease of $\alpha$ in expression of *m1* (Model Set 4) - *Mutual Information = 254.133*
3. 10-fold decrease of $\alpha$ in expression of *m1, m2, m3* and 5-fold decrease of $\alpha 0$ in expression of *m2* (Model Set 3) - *Mutual Information = 248.305*

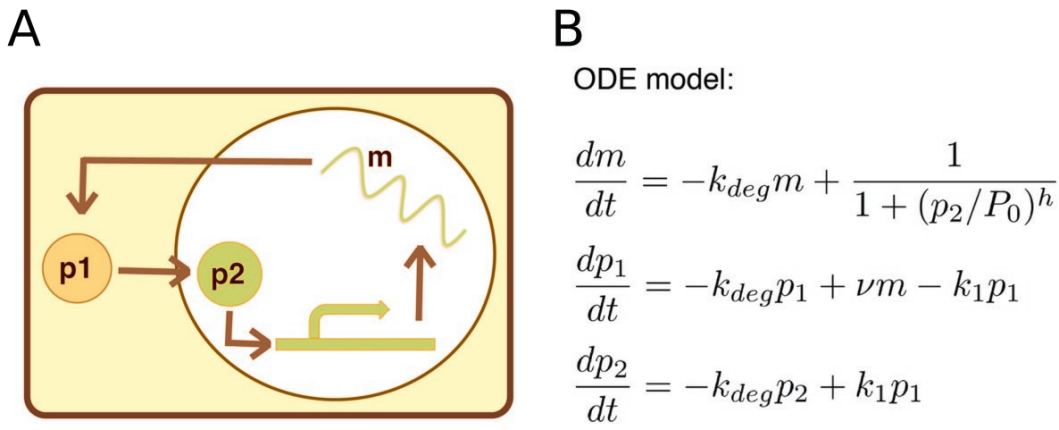



**Figure 6.3: Hes1 oscillator model**[11].(A) Schematic representation of Hes1 model. The green box represents a cell with its nucleus(white circle). `Hes1` mRNA ($m$) is transcribed in the nucleus, shuttled into the cytoplasm and translated into cytoplasmic Hes1 protein ($p1$), which in turn is being shuttled back into the nucleus and making it nuclear Hes1 protein ($p2$). $p2$ in turn regulates the transcription of $m$ (B) set of ordinary differential equations describing the oscillator system with three parameter $k_1$, $P_0$, $\nu$ and $h$. Figure was adapted with permission from [1]

---

We found these results to be in full agreement with the results produced by the proof-of-concept implementation which `PEITH`($\Theta$) was built on.

#### Runtime

We ran the Repressilator example on a system including an Nvidia Tesla K80 GPU, an Intel Xeon E5-2643 v3 (3.40GHz), and 128GB of RAM. With this set up, a single invocation of `PEITH`($\Theta$), calculating mutual information for the five experiments given above, took **3h 45m** to run (averaged over three runs).

## 6.2 Hes1 model

In order to exemplify our pipeline for *Experiment selection for inference of a subset of model parameters* we use the Hes1 oscillator system, which Liepe et al. [1] used to validate the algorithm `PEITH`($\Theta$) is based on.

### 6.2.1 Background

The Hes1 transcription factor is known to be involved in a number of important process, amongst those are cell differentiation and segmentation of vertebrate embryos. The oscillation of the Hes1 system is thought to be linked to the formation of spatial patterns during development [12]. This is simple three-component oscillatory ODE-model [11] summarised in Figure 6.3. The Hes1 model has 4 parameters namely $k_1$, $P_0$, $\nu$ and $h$ and 3 species: *Hes1* mRNA(m1), Hes1 nuclear protein(p1) and Hes1 cytoplasmic protein(p2).

### 6.2.2 Experiment selection for inference of a subset model parameters

Using the "Experiment selection for inference of a subset model parameters" approach we want to identify the measured species which provides the highest information about a single parameter of the model. Thus we specify two experiment, in the first experiment we can only measure mRNA($m$), while in the second experiment we consider the case that we can only measure the total concentration of nuclear and cytoplasmic Hes1 protein ($p1 + p2$). We estimate the mutual information between single parameters and mRNA/protein and calculate the mean and $\sigma^2$ over 3 repeats.

All experiments explored have a measurement noise with a variance $\sigma = 0.1$, initial conditions $(m1, p1, p2) = (0.5, 3.0, 2.0)$ and the priors over the model parameters are defined as follows:

$$k_1 \sim \mathcal{U}(0, 2) \quad P_0 \sim \mathcal{U}(1, 10) \quad \nu \sim \mathcal{U}(0, 0.1) \quad h \sim \mathcal{U}(0, 0.1)$$

For each experiment the total number particle simulated and the sample sizes for the estimation of the mutual information are as follows:

$$particle = 15000 \quad (N_1, N_2, N_3) = (5000, 5000, 5000)$$

As mentioned above we use the low level input pipeline of `PEITH`($\Theta$) to run the Hes1 model. An example command line is given below:

```
$ peitho -a 1 -of=results/Hess1/h1_h_run1 -i1 hess1_h.xml -lc 1
         -if=Example_data/Hess1
```

Please find the input `XML` files used as part of this example in the `Example_data` folder of the `PEITH`($\Theta$) package.

### 6.2.3 Results

In accordance with the data by Liepe et al. [1], our results(shown in Figure 6.4 indicate that *Hes1* mRNA holds a higher information content for all parameter compared to the jointly measured Hes1 proteins. Liepe et al. [1] further substantiated these results through simulation and parameter inference. Furthermore, our results also display the same pattern of information content in-between model parameters and the measured species. In more detail, measurement of mRNA provides more information about the model parameters $P_0$ ($mean = 44.6720$; $\sigma^2 = 3.8786$) and $\nu$ ($mean = 40.0966$; $\sigma^2 = 0.9370$) compared to $h$ ($mean = 33.3035$; $\sigma^2 = 1.7326$) and $k_1$ ($mean = 35.2899$; $\sigma^2 = 0.9345$) and the measurement of nuclear and cytoplasmic protein holds the highest information for model parameter $k_1$ ($mean = 22.2882$; $\sigma = 1.2236$).

#### Runtime

We ran the Hes1 example on a system including an Nvidia Tesla K20m GPU, an Intel Xeon E5-2620 CPU (2.00GHz), and 132GB of RAM. With this set up, calculating the mutual information for each pair of experiments for one parameter took **49 minutes**.

## 6.3 p53 model

To exemplify our pipeline for *Experiment selection for prediction* we use the p53 model. We start by providing a brief background to model as well as the experiments we consider comparing, followed by the results.

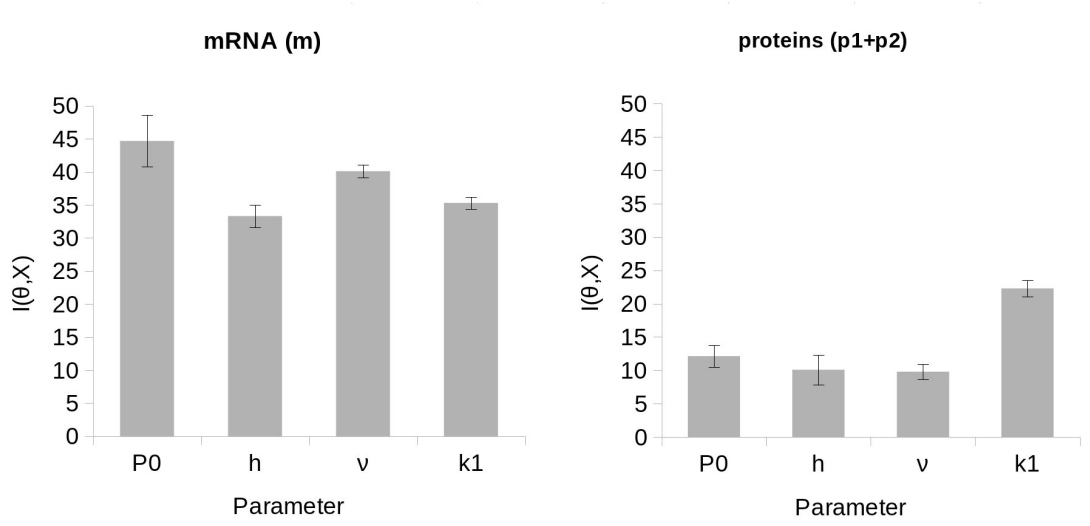


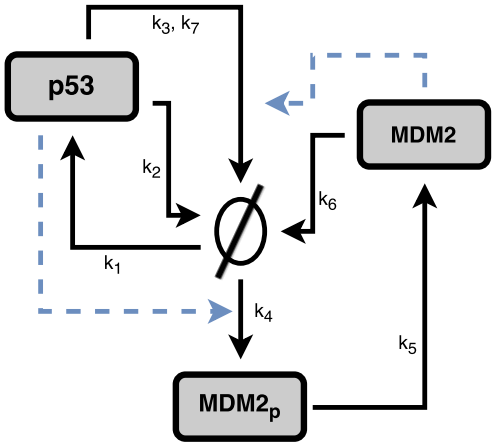

**Figure 6.4: Estimated mutual information of Hes1 model example.** The left panel shows the mutual information between measured mRNA (m) and each single model parameter in an unperturbed system. The right panel shows the mutual information between measured cellular protein (p1+p2) and each single model parameter. Bar graphs represent the mean and standard error over three repeats of estimation of mutual information.

---

### 6.3.1 Background

We use a simplified model of the p53 network. It involves three interacting proteins: p53, MDM2-precursor and MDM2. A diagram of the network can be seen in Figure 6.5 alongside the system of differential equations that we use to model the system. More specifically, the system consists of the following reactions:

| | |
|---|---|
| p53 production | $\emptyset \xrightarrow{k_1} p53$ |
| p53 degradation (independent of MDM2) | $p53 \xrightarrow{k_2} \emptyset$ |
| p53 degradation (dependent on MDM2) | $p53 \xrightarrow{k_3, k_7, MDM2} \emptyset$ |
| $\text{MDM2}_p$ production | $\emptyset \xrightarrow{k_4, p53} MDM2_p$ |
| MDM2 synthesis | $MDM2_p \xrightarrow{k_5} MDM2$ |
| MDM2 degradation | $MDM2 \xrightarrow{k_6} \emptyset$ |




$$x_1 = p53, \quad x_2 = MDM2_p, \quad x_3 = MDM2$$

$$\frac{dx_1}{dt} = k_1 - k_2 x_1 - k_3 \frac{x_1 x_2}{x_1 + k_7}$$
$$\frac{dx_2}{dt} = k_4 x_1 - k_5 x_2$$
$$\frac{dx_3}{dt} = k_5 x_2 - k_6 x_3$$

**Figure 6.5: p53 model.** Left is a schematic of the p53 model. The central image is the empty set with solid black lines indicating reactions along with the associated parameters for that reaction. The dashed blue lines indicate modifier species that influence the reaction to which they point but are not used within the reaction. In the grey boxes are the three proteins involved: p53, $MDM2_p$, and MDM2. Right gives the system of ODEs for which the dynamics of the p53 model follow.

---

### 6.3.2 Experiment Selection for Prediction

Measuring MDM2 in isolation of $MDM2_p$ is often not feasible and only the total amount of MDM2 (i.e. the sum of MDM2 and $MDM2_p$) can be experimentally assessed. As such, we consider measuring only MDM2 from 0 to 25 hours every 6 minutes as our experiment which we wish to make predictions on (this experiment corresponds to $\mathbf{Y}$ in equation 1.3). The experiments that we can carry out (these experiments correspond to $\mathbf{X}_q$ in equation 1.3) are as follows:

1. Wildtype measured from 0 to 10 hours every hour
2. Wildtype measured from 0 to 25 hours every 5 hours
3. Wildtype measured from 0 to 30 hours every 2 hours
4. $4 \times k_5$ measured from 0 to 10 hours every hour
5. $4 \times k_5$ measured from 0 to 25 hours every 5 hours
6. $4 \times k_5$ measured from 0 to 30 hours every 2 hours
7. $4 \times k_1$ measured from 0 to 10 hours every hour

8. $4 \times k_1$ measured from 0 to 25 hours every 5 hours

9. $4 \times k_1$ measured from 0 to 30 hours every 2 hours

All the experiments (both $\mathbf{Y}$ and $\mathbf{X}_q$) have a measurement noise with a variance $\sigma = 15$ and initial conditions $(p53, MDM2_p, MDM2) = (70, 30, 60)$. Furthermore, our priors over the parameters are as follows:

$$k_1 \sim \mathcal{U}(90, 120) \quad k_3 \sim \mathcal{U}(5, 20) \quad k_5 \sim \mathcal{U}(0, 5) \quad k_7 \sim \mathcal{U}(0, 5)$$
$$k_2 \sim \mathcal{U}(1, 5) \quad k_4 \sim \mathcal{U}(10, 20) \quad k_6 \sim \mathcal{U}(0, 5)$$

We also use `SBML` files for both the reference and experiments. Furthermore, since we are using a number of different time points this means we need to invoke `PEITH`($\Theta$) with new command line calls. As such we use the following: (a) corresponds to 0 to 10 hours every hour, (b) corresponds to 0 to 25 hours every 5 hours and (c) corresponds to 0 to 30 hours every 2 hours. An example of one of these command line calls:

```
$ peitho -i1 p53_model_ref.xml p53_model_exp.xml
         -i2 p53_ref.data p53_a.data
         -a 2 -lc 00 -if=Example_data/p53 -of=results/p53/p53_a
```

In particular the defining sections of `p_ref.data` are:

```
>timepoint
0.0 0.1 0.2 ... 24.8 24.9 25.0
<timepoint

>measuredspecies
species3
<measuredspecies

>Initial Conditions 1
Unchanged
<Initial Conditions 1

>Parameter - Experiment 1
Unchanged
<Parameter - Experiment 1

>combination
initset1 paramexp1 measured1
```

```
<combination
```

Here `species3` corresponds to just MDM2, whilst we are just using the wild type as our reference model, with measuring every 6 minutes from 0 to 25 hours. In comparison the defining sections of `p53_a.data`, where `a` corresponds to time point (a):

```
>timepoint
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
<timepoint

>measuredspecies
species2+species3
<measuredspecies

>Initial Conditions 1
Unchanged
<Initial Conditions 1

>Parameter - Experiment 1
Unchanged
<Parameter - Experiment 1

>Parameter - Experiment 2
k5 4.0 5
<Parameter - Experiment 2

>Parameter - Experiment 3
k1 4.0 1
<Parameter - Experiment 3

>combination
All
<combination
```

Here we are measuring the sum of MDM2 and MDM2$_p$ (i.e `species2+species3`) with time points (a). This input data file corresponds to (1), (4), and (7) as given in the list previously. We run this model with a total particle size of 1450000 where: $N_1 = 100000$ and $N_2 = N_3 = N_4 = 450000$. The complete input data files are available with the installation of `PEITH`($\Theta$).

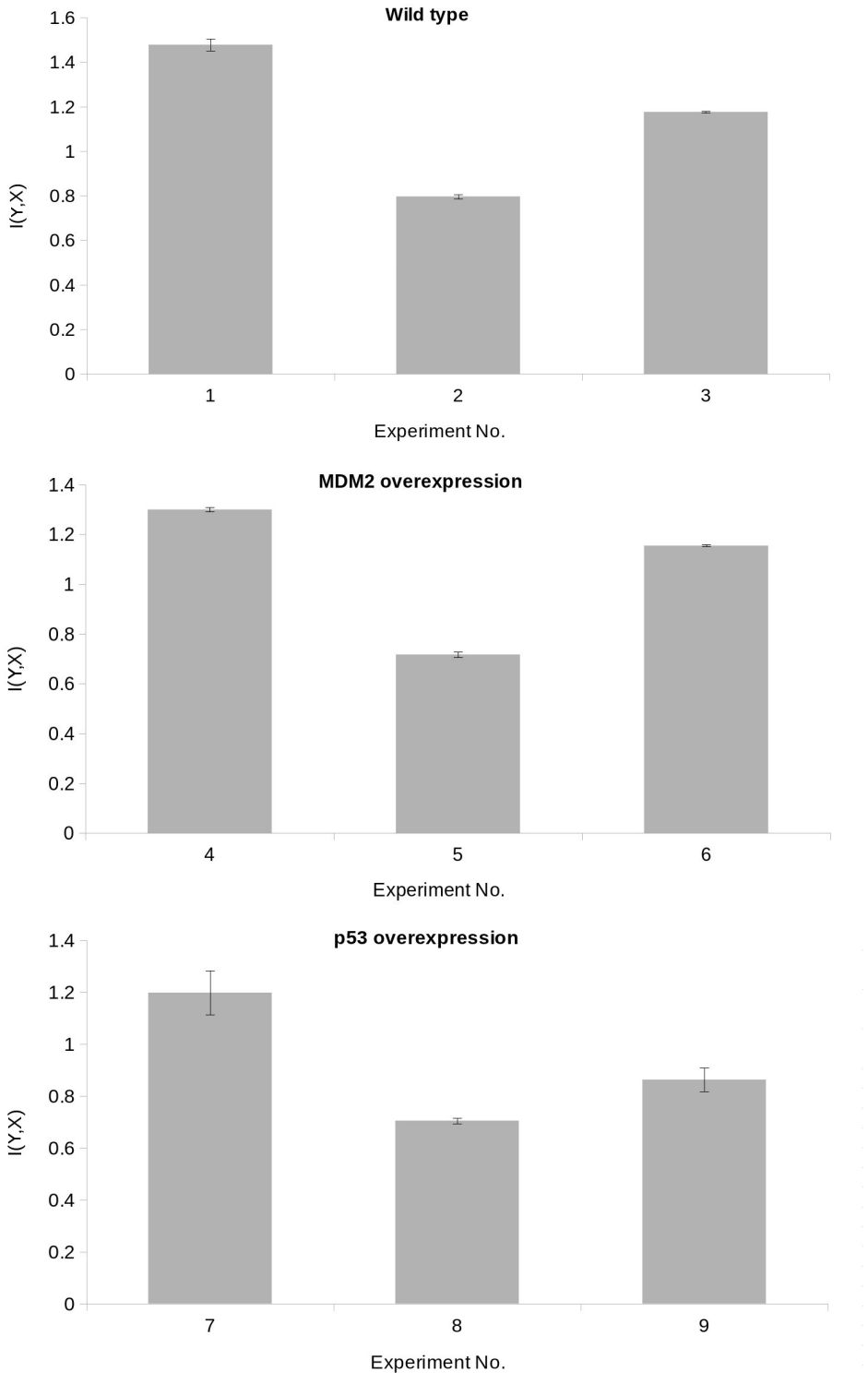



**Figure 6.6: Mutual information for the p53 example.** The numbers along the x-axis indicate the experiment number as shown in the previous section. The top bar graph indicates wild type p53 model. The middle bar graph indicates MDM2 overexpression by multiplying $k_5$ by a factor of 4. The bottom bar graph indicates p53 overexpression by multiplying $k_1$ by 4. The standard errors are also given as error bars over three runs of $\texttt{PEITH}(\Theta)$ for each experiment.

### 6.3.3 Results

Running the p53 model with PEITH($\Theta$) we obtain the results seen in Figure 6.6, which suggest that the top three experiments to conduct are:

1. Wildtype measured from 0 to 10 hours every hour - *Mutual Information = 1.4774*
2. $4 \times k_5$ measured from 0 to 10 hours every hour - *Mutual Information = 1.3000*
3. $4 \times k_1$ measured from 0 to 10 hours every hour - *Mutual Information = 1.1980*

This is generally in line with the results obtained from simulations conducted for each experiment. The results of these can be seen in Figure 6.7. Interestingly using PEITH($\Theta$) captures the importance of the time points used for an experiment. Time point (a) produces a larger value for $H(\boldsymbol{\theta}|\mathbf{X} = \mathbf{x}^*) - H(\boldsymbol{\theta})$ followed by (b) and then (c) as shown in Figure 6.6. Additionally, Figure 6.8 displays simulations of the reference experiment using parameters inferred from data simulated for each experiment defined previously. Qualitatively, these simulations are in line with the predictions made by PEITH($\Theta$).

#### Runtime

We ran the p53 example on a system including an Nvidia Tesla K20m GPU, an Intel Xeon E5-2620 CPU (2.00GHz), and 132GB of RAM. Given this set up we recorded an average runtime of **59 minutes** for the calculation of the mutual information for each experiment.

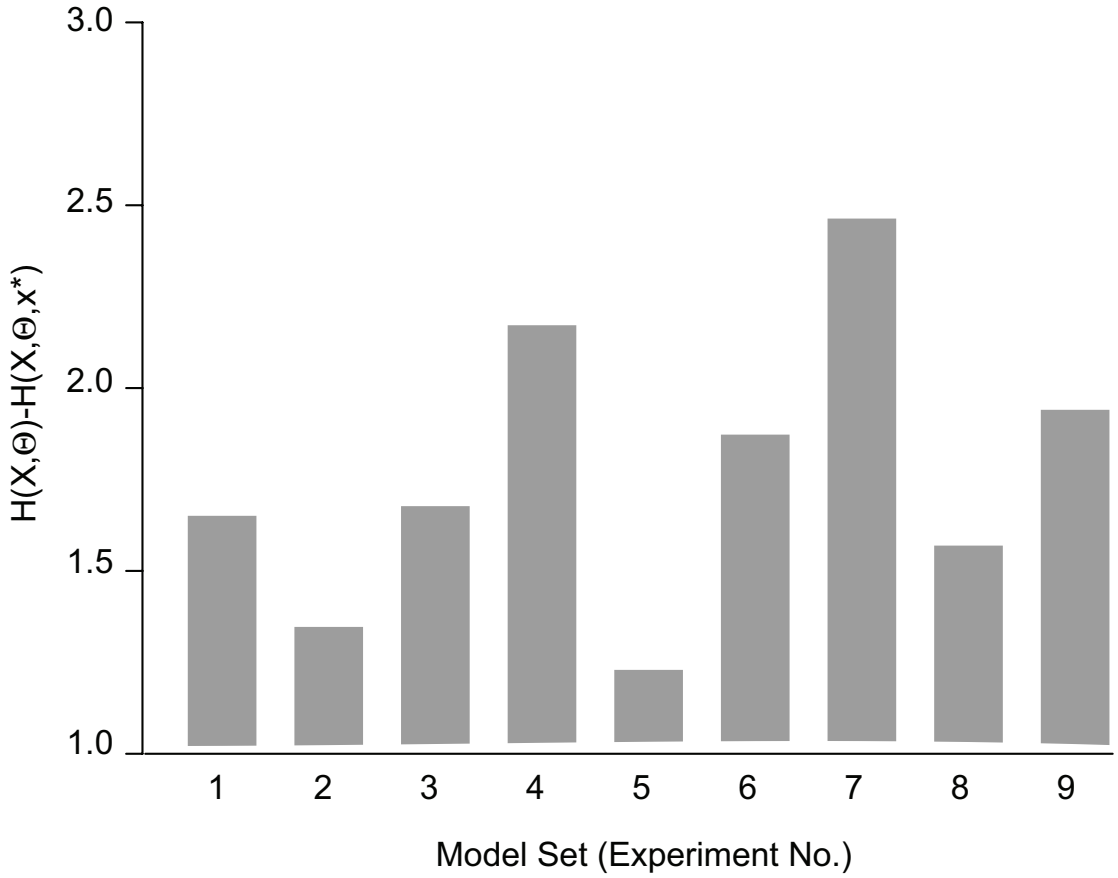



**Figure 6.7: Difference between conditional entropy, $H(\boldsymbol{\theta}|\mathbf{X} = \mathbf{x}^*)$, and entropy, $H(\boldsymbol{\theta})$.** The model sets along the x-axis correspond to the experiment being conducted as seen in the section entitled *Experiment Selection for Prediction*. These use data simulated for each experiment with parameter values $k_1 = 90$, $k_2 = 0.002$, $k_3 = 17$, $k_4 = 1.1$, $k_5 = 0.93$, $k_0.96$, $k_7 = 0.01$ and then using `ABC-sysbio` [1] for parameter inference.

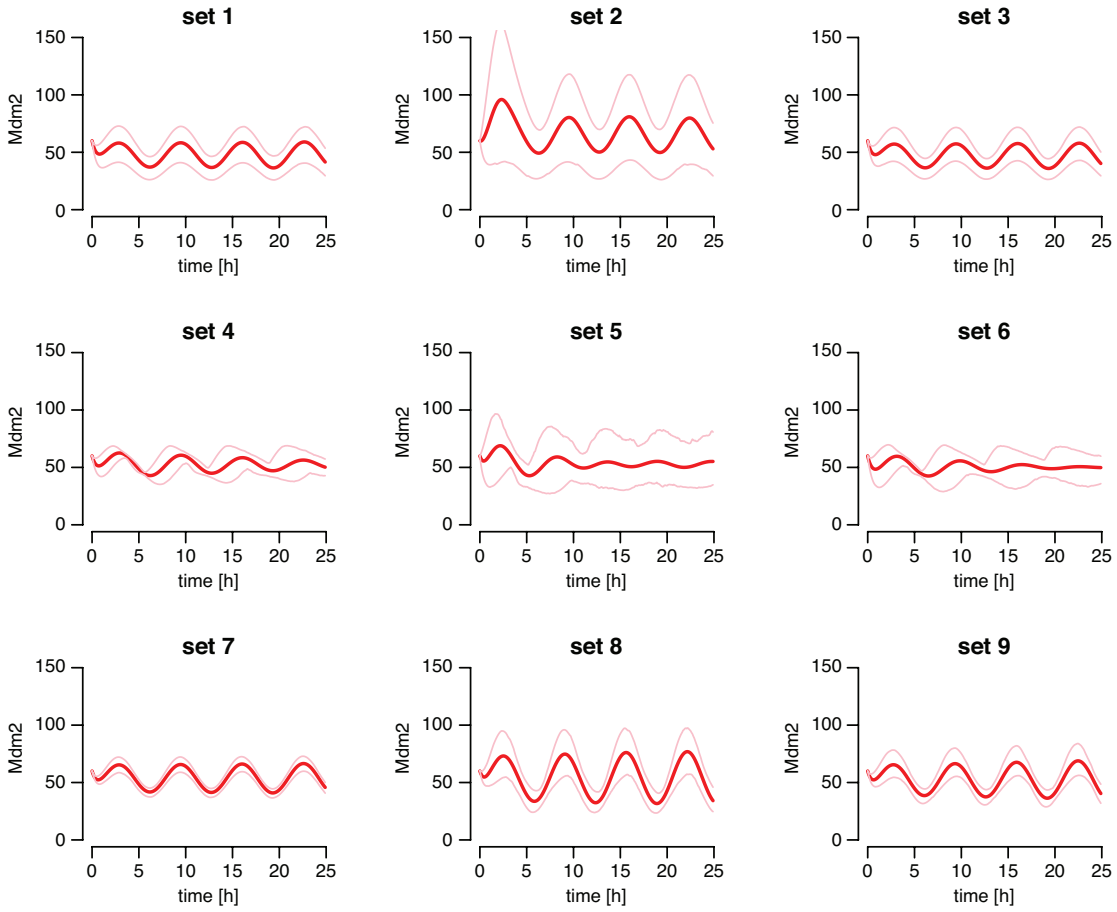



**Figure 6.8: Simulation results for the p53 model** using data simulated for each experiment with parameter values $k_1 = 90$, $k_2 = 0.002$, $k_3 = 17$, $k_4 = 1.1$, $k_5 = 0.93$, $k_0.96$, $k_7 = 0.01$ and then using `ABC-sysbio` [1] for parameter inference. Here the set numbers correspond to the experiments outlined in the list above in the section entitled *Experiment Selection for Prediction.*

# A. Troubleshooting

## A.1 Error messages

### A.1.1 Command Line

```
ERROR: unknown option [UNKNOWN FLAG]
```

If an unknown flag is given to PEITH(Θ) in the command line then PEITH(Θ) prints the unknown flag to the screen whilst also providing a list of valid flag.

### A.1.2 Memory Check

```
ERROR: Can only give an estimation of memory for one SBML or input
XML file at a time for a = 0 or 1 and two files (one for the
reference the other for experiments) when a = 2
```

PEITH(Θ)'s memory checker can only handle one SBML (high level) or input XML (low level) file at a time when conducting experimental selection for inference of all or a subset of parameters. On the other hand, when conducting experimental selection for prediction of output only two *arguments* can be passed: one for the reference model and the other for any potential experiments.

```
WARNING: The estimated RAM requirements are less than the recommended
size of 32 GB
```

In the event that PEITH(Θ)'s memory checker estimates a number less than 32 GB it warns the user of this as in the manual it is recommended that the system used to run PEITH(Θ) has at least 32 GB.

### A.1.3 Input data file & Input XML

#### dt

Data file:

```
ERROR:  Please provide an integer/float value for dt >dt ... <dt in
data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide a float value for <dt> in [NAME OF INPUT XML
FILE]!
```

The dt value for the internal time step of the ODE solver should be provided as a single positive float in the respective sections of the data and input xml file.

#### Particle

Data file:

```
ERROR: Please provide an integer value as number of particles:
>particles ... <particles in input data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide an integer value for <particles> in [NAME OF
INPUT XML FILE]!
```

The number of particles simulated needs to be defined as a single integer in the respective sections of the data and input xml file.

### Model type

Data file:

```
ERROR: The model type is not properly defined: >type ... <type in
data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Type of the model [name] in [NAME OF INPUT XML FILE] is not
supported!
```

In its current version PEITH(Θ) only supports ODE models, which is being defined by the string ODE in the respective sections of the data and input xml file.

### Time points

Data file:

```
ERROR:  Please provide a whitespace seperated list of float value:
>timepoints ... <timepoints in input data file [NAME OF DATA FILE]!
```

```
ERROR: One or more timepoints are defined twice or more: >timepoints
... <timepoints in input data file [NAME OF DATA FILE]!
```

```
ERROR:  Please provide a whitespace seperated list of float value
in ascending order: >timepoints ... <timepoints in input data file
[NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide a white space separated list of values for
<data><times>  in [NAME OF INPUT XML FILE]!
```

The time points at which species are measured during simulation need to be provided as an white space separated list of unique float/integer values in ascending order in the

respective sections of the data and input xml file.

### N samples

Data file:

```
ERROR:  Please provide a whitespace seperated list of integer value
as samples sizes: >nsample ... <nsample in input data file
[NAME OF DATA FILE]!
```

```
ERROR:  N1 and N2 can not be of size 0: >nsample ... <nsample in
input data file [NAME OF DATA FILE]!
```

```
ERROR:  N3 and N4 must be of size 0: >nsample ... <nsample in input
data file [NAME OF DATA FILE]!
```

```
ERROR: Sum of samples N1 and N2 is not equal to number of
particle: >nsample ... <nsample in input data file [NAME OF
DATA FILE]!
```

```
ERROR:  N1, N2 and N3 can not be of size 0: >nsample ... <nsample
in input data file [NAME OF DATA FILE]!
```

```
ERROR: N4 must of be size 0: >nsample ... <nsample in input data
file [NAME OF DATA FILE]!
```

```
ERROR: Sum of samples N1, N2 and N3 is not equal to number of
particle: >nsample ... <nsample in input data file [NAME OF
DATA FILE]!
```

```
ERROR: N1, N2, N3 and N4 can not be of size 0: >nsample ... <nsample
in input data file [NAME OF DATA FILE]!
```

```
ERROR: Sum of samples N1, N2, N3 and N4 is not equal to number of
particle: >nsample ... <nsample in input data file [NAME OF DATA
FILE]!
```

Input xml file:

```
ERROR: Please provide an integer value for <nsamples><N1> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: Please provide an integer value for <nsamples><N2> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: Please provide an integer value for <nsamples><N3> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: Please provide an integer value for <nsamples><N4> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: The sum of N1, N2, N3 and N4 is bigger than given particle
number in [NAME OF INPUT XML FILE]!
```

The user The user has to provide either N1 and N2 or N1, N2 and N3 or N1, N2, N3 and N4 depending type of analysis performed. Regardless, the sum of samples Nx needs to be equal to the specified number of particles. For further details please consult chapter 4.

**Sigma**

Data file:

```
ERROR: Please provide an float value for sigma: >sigma ... <sigma in
input data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide an float value for <data><sigma> in [NAME OF
INPUT XML FILE]!
```

The variance of the experimental noise should be specified as a float value in the respective sections of the data and input xml file.

### Posterior sample

Data file:

```
ERROR: Sample from posterior in in the wrong format:
>samplefromposterior ... <samplefromposterior in input data file
[NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide a boolean value for <samplefrompost> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: Please provide a file name for <samplefrompost_file> in [NAME
OF INPUT XML FILE]!
```

```
ERROR: Please provide a file name for <samplefrompost_weights> in
[NAME OF INPUT XML FILE]!
```

The user can provide a sample from a posterior distribution instead of formally defining prior distribution for the model parameter. If a sample is being provided the user needs to set `samplefromposterior` to `True` in the data file and input xml file. The sample is provided in two files, one containing the particles and one containing the associated weights. The names of these files need to be specified as a `string` in the `samplefrompost_file` and `samplefrompost_weights` sections in the data and input xml file. If no sample is provided the user will only need to specify `samplefromposterior` as `False`.

### Prior distributions and Initial conditions

Data file:

```
ERROR: Input if initial conditions are defined by prior distribu-
tions is not the right format (True or False): >initprior ...
<initprior in input data file [NAME OF DATA FILE]!
```

```
ERROR: Prior distributions of model parameter are not the right
format: >prior ... <prior in input data file [NAME OF DATA FILE]!
```

```
ERROR: Prior distributions of initial conditions are not in the
right format: >initials ... <initials in input data file [NAME
OF DATA FILE]!
```

```
ERROR: Prior distributions/constant of compartments are not in the
right format: >compartment ... <compartment in input data file [NAME
OF DATA FILE]!
```

```
ERROR: Constant values for initial conditions are not in the right
format: >Initial Conditions ... <Initial Conditions in input data
file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Please provide a boolean value for <initialprior> in [NAME OF
INPUT XML FILE]!
```

```
ERROR: Value of the prior for experiment [name] has the wrong format
in [NAME OF INPUT XML FILE]!
```

```
ERROR: Supplied parameter prior [distribution] is unsupported in
[NAME OF INPUT XML FILE]!
```

In its current version four different priors are supported: `constant`, `uniform`, `normal` and `lognormal`. For further detail about the particular format when defining any of these priors, please consult 4.

### Measured species

Data file:

```
ERROR: No measurable species are defined: >measuredspecies ....
<measurespecies in input data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Measurable species are not defined properly with
<measuredspecies> ... </measuredspecies> for experiment [name]
in [NAME OF INPUT XML FILE]!
```

Measured species are specified as white space seperated list of strings in the respective sections in the data file and input xml file. The input follows the general format of `species1 species2 species3`. In the the special case, that all species are being measured, the list can omitted for `All`. For further details about how to specify `measuredspecies`, please consult chapter 4.

### Fitting information

Data file:

```
ERROR: Fitting information for initial condition is not in the right
format: >initialfit ... <initialfit in input data file [NAME OF DATA
FILE]!
```

```
ERROR: Fitting information for model parameter is not in the right
format: >paramfit ... <paramfit in input data file [NAME OF DATA
FILE]!
```

```
ERROR: Fitting information for compartments is not in the right
format: >compfit ... <compfit in input data file [NAME OF DATA FILE]!
```

Input xml file:

```
ERROR: Parameters to be fitted are not defined properly in <paramfit>
... </paramfit> in [NAME OF INPUT XML FILE]!
```

```
ERROR: Initial conditions to be fitted are not defined properly in
<initfit> ... </initfit> in [NAME OF INPUT XML FILE]!
```

```
ERROR: Compartments to be fitted are not defined properly in
<compfit> ... </compfit> in [NAME OF INPUT XML FILE]!
```

The fit to parameters, compartments and initial condition considered by the "Experiment selection for inference of a subset model parameters" approach is specified by `All`, `None` or a white space list of `parameterX`/`initialY`/`compartmentZ` in the respective sections in the data file and input xml file. For further details about how to specify fits, please consult chapter 4.

### Combinations

Data file:

```
ERROR: Combinations of initial conditions, parameter changes and
species fit defining an experiment are not in the right format:
>combination ... <combination in input data file [NAME OF DATA FILE]!
```

A single experiment is specified by a white space separated list of `initsetX paramexpY fitZ` in data input file. For further details on this format please consult chapter 4. In the special case if the user wants to explore all combination of sets of initial condition, parameter perturbations and measured species the white space separated list can be simply omitted for `All`.

**Number of experiments, parameters in input xml file**

```
ERROR: Please provide an integer value for <experimentnumber>!
```

The number of experiment specified in the input xml needs to be provided as integer value at the top of the file.

```
ERROR: Please provide an integer value for <data><nparameters_all>!
```

Within in the same input xml the user is only allowed to specify experiments with the same amount of model parameter. The number of parameters is thus globally specified in the input xml and needs to be provided as integer value.

**Associated files in input xml file**

```
ERROR: Please provide a string value for <name> for
experiment [name] in [NAME OF INPUT XML FILE]!
```

```
ERROR: Please provide an string value for <source> for
experiment [name] in [NAME OF INPUT XML FILE]!
```

```
ERROR: Please provide an string value for <cuda> for experiment
[name] in [NAME OF INPUT XML FILE]!
```

In the input xml file each experiment needs to be assigned a unique name in the `<name>` section. Furthermore, in order to simulate the experiment using `CUDA-sim` an associated `CUDA` code needs to be specified in `<cuda>`. When writing an input xml from scratch a token string for the associated SBML of each experiment needs to specified.

**Other**

```
ERROR: No parameters specified in experiment [name] in [NAME OF
INPUT XML FILE]!
```

The experiment indicated in the input xml file has no parameters defined.

```
ERROR: No initial conditions specified in experiment [name] in
[NAME OF INPUT XML FILE]!
```

The experiment indicated in the input xml has no initial conditions defined.

```
ERROR: No measurable species specified in experiment [name] in [NAME
OF INPUT XML FILE]!
```

The experiment indicated in the input xml has no measurable species defined.

```
ERROR: No experiments specified in [NAME OF INPUT XML FILE]!
```

The input xml file, which had been passed to PEITH($\Theta$), did not specify a single experiment.

```
ERROR: Models don't have the same number of species in [NAME OF
INPUT XML FILE]!
```

```
ERROR: Experimental models don't have the same number of compartments
in [NAME OF INPUT XML FILE]!
```

```
ERROR: Experimental models don't have the same number of parameters
in [NAME OF INPUT XML FILE]!
```

All experiments specified within a single input xml are required to hold the same number of species, compartments and parameters. If the user wants to explore models/experiments with different numbers of species, compartments and parameters one needs write multiple input xml files.

### A.1.4 SBML file

```
ERROR: SBML file - [NAME OF SBML FILE] - has the following errors

line 21: (01009 [Error]) Element tag mismatch or missing tag.
```

Above is an example of an error that arises when there are issues with the format of the SBML file. We use the package libSBML [13] within which we use the method getNumErrors() to obtain any errors associated with the SBML files. For more details on the error messages with relation to SBML files see the libSBML manual [13]. Also consult the section on SBML files given in the manual.

```
ERROR: Parameters not defined properly in input file. Need to be
defined sequentially e.g.

>Parameter - Experiment 1
...
<Parameter - Experiment 1

>Parameter - Experiment 2
...
<Parameter - Experiment 2

>Parameter - Experiment 3
...
<Parameter - Experiment 3

Also if you plan to run an unchanged version of SBML file this must
be Experiment 1 as:
>Parameter - Experiment 1
Unchanged
<Parameter - Experiment 1
```

In the event that perturbations are not well defined in the input data file the above error

message is given, along with details of how it should be written. For further details also see the section on the input data file within the manual.

```
ERROR: Can only take one SBML file at a time when generating
templates
```

When using PEITH(Θ) to generate templates for the input XML and CUDA code file only one SBML file can be parsed at a time.

### A.1.5 Calculation of the mutual information

```
WARNING: GPU not supported. The launch configurator only supports
compute capability 2.0 - 6.2. Configuration for CC 6.2 will be
used. This might cause errors.
```

If the detected CC (compute capability) is found to be outside the supported range (2.0 - 6.2), CC 6.2 will be assumed by the launch configurator. In this case, it is advisable to check if the device properties which are inferred from the compute capability in the launch configurator correspond to the properties of the card used. If necessary, these values can be edited in the `launch.py` script. The required information can be found in appendix G of the CUDA C Programming Guide [14]. The values in question are set to (according to CC 6.2):

```
reg_granul = 256          warp_granul = 4          smem_granul = 256

max_regs_per_sm = 65536   max_blocks_per_sm = 32
```

```
ERROR: Not enought memory (RAM) available to create array for GPU
results. Reduce GPU grid size.
```

```
Memory Error
```

In order to run PEITH(Θ), the available RAM should be at least the size of two to three times the global memory of the GPU device. If this is not the case, the above error might be produced. To alleviate that, it is possible to reduce RAM use through limiting the

GPU memory used by reducing GPU grid size (this might increase runtime of the code). To limit the GPU memory, reduce the pre-factor (currently 0.95) in the first row of the `optimise_gridsize` function in the `launch.py` script (inside the *MutInfo* folder). The memory checker can be used to check how much memory would be required for a given run (see section 4.5.3). Please note that changes in the pre-factor for GPU memory are not considered by the memory checker. To get an accurate prediction after adapting the pre-factor, please also change the pre-factor (default: 0.95) in the `memory_checker` function in the `main.py` script (inside the *Main* folder).

```
ERROR: Too many nan/inf values in output, could not calculate mutual
information. Consider increasing particle size or adapting prior
distributions.
```

Numerical issues in the operations carried out on GPU can give rise to `nan` and `inf` values in the calculations. Particles that give rise to such results are automatically excluded from the calculation of the mutual information. The above error message is displayed if all $N_1$ particles have been removed and hence the mutual information cannot be calculated. A reason for this might be a stiff ODE model.

Potential solutions to this problem includes changing the prior distributions, decreasing the time-step `dt` or increasing the particle size. If that is still unsuccessful, the user could try adapting the `scaling` or `scaling_ge3` function in the `parse_infoEnt_new_2.py` script in order to prevent numbers to become to small that they cannot be represented as a `Float64`[1] any more.

## A.2 Potential problems and output one might encounter running the package

### A.2.1 Negative mutual information

A negative output for the mutual information can be caused by a large proportion of `inf` or `nan` values in the output. For various approaches to alleviate this issue, please consult

[1] `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

the discussion of the error message regarding too many `inf`/`nan` results in section A.1.5 above.

### A.2.2 Erroneous `SBML` file format

Errors may be thrown if the `SBML` file is not given in the required format for the high level workflow. In particular `PEITH`($\Theta$) needs parameters to be defined globally (i.e. outside of the list of reactions) rather than locally. If this is not adhered to then issues can arise when trying to pass the `SBML` file into `CUDA` code and may result in erroneous output from `CUDA-Sim`, which in turn may give the wrong mutual information.

# B. Overview of GPU architecture

In this section we aim to provide a brief overview of the general GPU (graphics processing unit) architecture and what said structure implies for code performance considerations. An in-depth discussion of the GPU architecture can be found in the CUDA C Programming Guide [14].

The main difference between the architecture of a CPU (central processing unit) and a GPU is the highly parallelised nature of the GPU. Instead of a few very powerful processing cores, the GPU has thousands of less powerful cores, which can greatly increase processing speed. This is however only feasible, if the computational task can be split into smaller sub-problems, which can be executed in parallel. On the most basic level, there are three units that code execution on a GPU can be divided into (see figure B.1).

A *thread* is the smallest execution entity on the GPU, with each thread executing a slightly adapted replicate of the current GPU kernel. The difference in the kernel between the threads is only the output of the `threadIdx.x` and `threadIdx.y` functions, which is generally used for indexing. This ensures that every thread processes a different part of the input data (concept of parallelisation) and writes the output to a specific

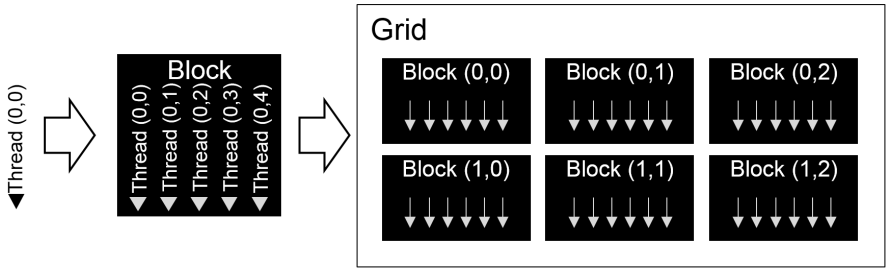



**Figure B.1: Schematic representation of code executing in parallel on a GPU.** Represented here are a single thread, a one-dimensional block of five threads and a two-dimensional grid with six blocks (30 threads).

location. The memory type that is specific for each thread is referred to as *registers*, which is also the memory type with the lowest latency.

A *block* refers to a group of a defined number of threads. It can be one-, two- or three-dimensional. An noteworthy property for performance considerations is that a single block is always executed on a single SM (streaming multiprocessor), the physical processing unit on the GPU. The memory type that is specific to each block is referred to as *shared memory*, which is considered the second fastest memory type. It is shared in the sense that it can be accessed by all threads in the same block but by no other threads in the grid.

The *grid* is the group of blocks that are launched on the GPU upon each kernel invocation. Just as blocks, it can also be one-, two- or three-dimensional. The size of the grid itself (i.e. the number of blocks in the grid) does not have any direct implications on the occupancy of the GPU (see below). It can however affect the performance through dictating the number of memory transfer operations between the host and the device. Generally, choosing the grid size so that nearly all global memory is utilised on every run, can help decrease runtime. Global memory is accessible by all threads in the grid, but has high latency associated with accessing it.

Maximising occupancy is an important strategy to minimising the runtime of computations on the GPU. Occupancy here refers to the number of concurrent threads executing on the GPU at any one time. As all SMs on a single GPU device are identical, limiting occupancy considerations to a single SM is sufficient to achieve maximum occupancy across the whole device.

## B.1 Optimising code performance on the GPU

We have already briefly introduced the concept of *threads, blocks, grids* and *SMs*. Additionally, the concept of *warps* plays a key role in maximising GPU occupancy. A warp is a group of 32 threads, which are launched on a SM at the same time, i.e. the granularity of parallel threads executing on a SM is 32.

While outlining the process of maximising the occupancy on a GPU in detail here exceeds

the scope of this report, we do list the main factors that need to be considered in this process below. In `PEITH`($\Theta$), all of this is handled by the automatic launch configurator implementation.

The following factors need to be considered when maximising the GPU occupancy:

- Properties of the device the kernel is executing on, such as the maximum allowed number of blocks and threads per SM, summarised by the device's compute capability (for a detailed description of compute capabilities and their associated properties, please refer to appendix G of the CUDA C Programming Guide [14])
- Properties of the kernel itself, such as register and shared memory use
- The granularities ('multiples of...'), with which shared memory, registers and warps are allocated on the SM
- If two different block sizes achieve the same occupancy, the smaller block size should be chosen in order to reduce overhead on the SM

A good understanding of the impact of the aforementioned factors on occupancy can be obtained by consulting the CUDA Occupancy Calculator Spreadsheet [10].

# C. Approximation of Mutual Information

We now provide the derivation of the approximation for mutual information as given by Liepe et al. [1] Let $\mathcal{Q}$ be the set of potential experiments, where each experiment $q \in \mathcal{Q}$ is defined by a set of ODEs that have solutions $\boldsymbol{\mu}(\boldsymbol{\theta}, q)$, where $\boldsymbol{\theta}$ is a vector of parameters used to define the system of ODEs. Define $\mathbf{X}_q$ as a vector of random variables representing the outcome from experiment $q$. Then the relationship between $\mathbf{X}_q$ and $\boldsymbol{\mu}$ is as follows:

$$\mathbf{X}_q = \boldsymbol{\mu}(\boldsymbol{\theta}, q) + \boldsymbol{\epsilon} \tag{C.1}$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 I) \tag{C.2}$$

where $I$ is the identity matrix. Additionally suppose there is a prior over $\boldsymbol{\theta}$, $\Pi(\boldsymbol{\theta})$, then this provides the foundation for deriving the approximation to $\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q)$, the mutual information. Starting from the definition:

$$\begin{aligned}
\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) &= \int\int p(\boldsymbol{\theta}, \mathbf{x}_q) \log \frac{p(\boldsymbol{\theta}, \mathbf{x}_q)}{p(\boldsymbol{\theta}) p(\mathbf{x}_q)} d\boldsymbol{\theta} d\mathbf{x}_q \\
&= \int\int p(\boldsymbol{\theta}, \mathbf{x}_q) \log \frac{p(\mathbf{x}_q|\boldsymbol{\theta})}{p(\mathbf{x}_q)} d\boldsymbol{\theta} d\mathbf{x}_q \\
&= \int\int p(\mathbf{x}_q|\boldsymbol{\theta}) p(\boldsymbol{\theta}) \log \frac{p(\mathbf{x}_q|\boldsymbol{\theta})}{p(\mathbf{x}_q)} d\boldsymbol{\theta} d\mathbf{x}_q \\
&= \int p(\boldsymbol{\theta}) \underbrace{\int p(\mathbf{x}_q|\boldsymbol{\theta}) \log \frac{p(\mathbf{x}_q|\boldsymbol{\theta})}{p(\mathbf{x}_q)} d\mathbf{x}_q}_{f(\boldsymbol{\theta})} d\boldsymbol{\theta} \\
&\approx \frac{1}{N_1} \sum_{i=1}^{N_1} f(\boldsymbol{\theta}^{(i)})
\end{aligned} \tag{C.3}$$

The final line is the Monte Carlo approximation where $\boldsymbol{\theta}^{(i)}$s are drawn from the prior, $\Pi(\boldsymbol{\theta})$. Then:

$$\begin{aligned} f(\boldsymbol{\theta}^{(i)}) &= \int p(\mathbf{x}_q|\boldsymbol{\theta}^{(i)}) \log \frac{p(\mathbf{x}_q|\boldsymbol{\theta}^{(i)})}{p(\mathbf{x}_q)} d\mathbf{x}_q \\ &= \frac{1}{N_3} \sum_{j=1}^{N_3} \log \frac{p(\mathbf{x}_q^{(i,j)}|\boldsymbol{\theta}^{(i)})}{p(\mathbf{x}_q^{(i,j)})} \end{aligned} \tag{C.4}$$

where $\mathbf{x}_q^{(i,j)}$ is sampled from $p(\mathbf{x}_q|\boldsymbol{\theta}^{(i)})$. Substituting equation C.4 into equation C.3 gives:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) = \frac{1}{N_1} \frac{1}{N_3} \sum_{i=1}^{N_1} \sum_{j=1}^{N_3} \log \frac{p(\mathbf{x}_q^{(i,j)}|\boldsymbol{\theta}^{(i)})}{p(\mathbf{x}_q^{(i,j)})} \tag{C.5}$$

By assuming that the noise term $\epsilon$ is negligible in comparison to deterministic proportion of $\mathbf{x}_q$, $N_3$ can be set to 1 (it should be noted that this is a crude approximation and can be thought of as a dirac delta function in which the variance, $\sigma^2$, approaches 0). This leads to:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) = \frac{1}{N_1} \sum_{i=1}^{N_1} \log \frac{p(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(i)})}{p(\mathbf{x}_q^{(i)})} \tag{C.6}$$

where $\mathbf{x}_q^{(i)}$ is sampled from $p(\mathbf{x}_q|\boldsymbol{\theta}^{(i)})$. $p(\mathbf{x}_q^{(i)})$ can be decomposed further through another Monte Carlo Approximation:

$$p(\mathbf{x}_q^{(i)}) = \int p(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} = \frac{1}{N_2} \sum_{j=N_1+1}^{N1+N2} p(\mathbf{x}_q^{(i)}|\boldsymbol{\theta}^{(j)}) \tag{C.7}$$

By rearranging equation C.6 and substituting in equation C.7, the following, full, approximation is obtained:

$$\mathcal{I}(\boldsymbol{\Theta}, \mathbf{X}_q) \approx \frac{1}{N_1} \sum_{i=1}^{N_1} \left[ \log\Big( p\big(\mathbf{x}_q^{(i)}|\theta^{(i)}\big) \Big) - \log\Big( \frac{1}{N_2} \sum_{j=N_1+1}^{N_1+N_2} p\big(\mathbf{x}_q^{(i)}|\theta^{(j)}\big) \Big) \right] \tag{C.8}$$

Approximations 1.2 and 1.3 are derived using a similar methodology.

# Bibliography


[1] J. Liepe, S. Filippi, M. Komorowski, and M. P. Stumpf, "Maximizing the information content of experiments in systems biology," *PLoS Comput Biol*, vol. 9, no. 1, p. e1002888, 2013.

[2] T. Toni, D. Welch, N. Strelkowa, A. Ipsen, and M. P. H. Stumpf, "Approximate bayesian computation scheme for parameter inference and model selection in dynamical systems," *Journal of the Royal Society Interface*, vol. 6, pp. 187 – 202, 2009.

[3] Y. Zhou, J. Liepe, X. Sheng, M. P. Stumpf, and C. Barnes, "Gpu accelerated biochemical network simulation," *Bioinformatics*, vol. 27, no. 6, pp. 874–876, 2011.

[4] D. J. Wilkinson, *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC, 2006.

[5] M. B. Elowitz and S. Leibler, "A synthetic oscillatory network of transcriptional regulators," *Nature*, vol. 403, no. 6767, pp. 335–338, 2000.

[6] L. Petzold and A. Hindmarsh, "Lsoda (livermore solver of ordinary differential equations)," *Computing and Mathematics Research Division, Lawrence Livermore National Laboratory, Livermore, CA*, vol. 24, 1997.

[7] J. Luitjens and S. Rennich, "Cuda warps and occupancy," in *GPU Technology Conference*, 2011.

[8] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[9] P. Micikevicius, "Gpu performance analysis and optimization," in *GPU Technology Conference*, 2012.

[10] NVIDIA Corporation, "Cuda occupancy calculator spreadsheet." Available at `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls` (Accessed: 14.03.2017), 2015.

[11] D. Silk, P. D. Kirk, C. P. Barnes, T. Toni, A. Rose, S. Moon, M. J. Dallman, and M. P. Stumpf, "Designing attractive models via automated identification of chaotic and oscillatory dynamical regimes," *Nature Communications*, vol. 2, p. 489, 2011.

[12] H. Hirata, S. Yoshiura, T. Ohtsuka, Y. Bessho, T. Harada, K. Yoshikawa, and R. Kageyama, "Oscillatory expression of the bhlh factor hes1 regulated by a negative feedback loop," *Science*, vol. 298, no. 5594, pp. 840–843, 2002.

[13] B. J. Bornstein, S. M. Keating, A. Jouraku, and M. Hucka, "Libsbml: an api library for sbml," *Bioinformatics*, vol. 24, no. 6, pp. 880–881, 2008.

[14] NVIDIA Corporation, "Cuda toolkit documentation v8.0." Available at `http://docs.nvidia.com/cuda` (Accessed: 07.03.2017), 2016.