How-to: Uncertainty quantification and parameter estimation for systems biology with CIUKF-MCMC

A tutorial for the complete uncertainty quantification and parameter estimation framework for dynamical models in systems biology presented in Linden, Kramer and Rangamani. *Bayesian Parameter Estimation for Dynamical Models in Systems Biology* 2022.

> **Required software**
>
> This tutorial assumes that users have access to the MATLAB and Julia programming languages. We used the following versions of these languages:
>
> - MATLAB: R2021a https://www.mathworks.com/products/new_products/release2021a.html (requires paid license)
> - Julia: v1.6.0 https://julialang.org/downloads/
>
> We specify which programming language is used for each step of the framework. If you do not have a MATLAB license, we list alternative libraries in the Additional Resources section at the end of this tutorial.
>
> Additionally we use the following open-source (free) packages:
>
> - UQLab version 2.0 (MATLAB)
> - SIAN-Julia version 1.0.3 (Julia)

Please note that all scripts associated with this tutorial can be found in the `tutorial/` directory in the associated GitHub repository.

Mitogen-Activated Protein Kinase signaling example model:

We will use the model of the core mitogen-activated protein kinase (MAPK) pathway from Nguyen et al. 2015. This model is also studied in Section 3.2 of Linden et al. 2022.

The core MAPK signaling pathway consists of three main biochemical species RAF ($x_1(t)$), MEK ($x_2(t)$), MAPK ($x_3(t)$), whose dynamics are modeled with the differential equations

$$\begin{aligned}
\frac{\mathrm{d}x_1(t)}{\mathrm{d}t} &= k_1 \cdot (S_{1t} - x_1(t)) \cdot \left[\frac{K_1^{n_1}}{K_1^{n_1} + x_3(t)^{n_1}}\right] - k_2 \cdot x_1(t) \\
\frac{\mathrm{d}x_2(t)}{\mathrm{d}t} &= k_3 \cdot (S_{2t} - x_2(t)) \cdot x_1(t) \cdot \left[1 + \frac{\alpha \cdot x_3(t)^{n_2}}{K_2^{n_2} + x_3(t)^{n_2}}\right] - k_4 \cdot x_2(t) \\
\frac{\mathrm{d}x_3(t)}{\mathrm{d}t} &= k_5 \cdot (S_{3t} - x_3(t)) \cdot x_2(t) - k_6 \cdot x_3(t).
\end{aligned}$$

The 14 model parameters $\boldsymbol{\theta}_f = [k_1, k_2, k_3, k_4, k_5, k_6, K_1, K_2\, S_{1t}, S_{2t}, S_{3t}, \alpha, n_1, n_2]$ and the initial conditions $\mathbf{x}_0 = [x_1(t=0), x_2(t=0), x_3(t=0)]^\top$ control the nature the dynamics. Different combinations of parameters and initial conditions lead to bistable dynamics, limit-cycle oscillations, and mixed multistability (see Figure 5.b of Nguyen et al. 2015). For example, the parameters $\boldsymbol{\theta}_f = [100, 100, 100, 0.1, 0.01, 0.01, 0.01, 0.01, 0.01, 10, 1, 15, 8, 10]^\top$ yield bistability, where the initial condition dictates the if steady state concentration of $x_3(t)$ is at the higher or lower levels. We simulate the model with these parameters and the initial condition $\mathbf{x}_0 = [0.0015, 3.6678, 28.7307]^\top$ to get a trajectory in the lower $x_3(t)$ steady state. The trajectory and noisy samples from the trajectory are shown in the following figure. These noisy samples from the lower steady state will be our *data* moving forward in the tutorial.

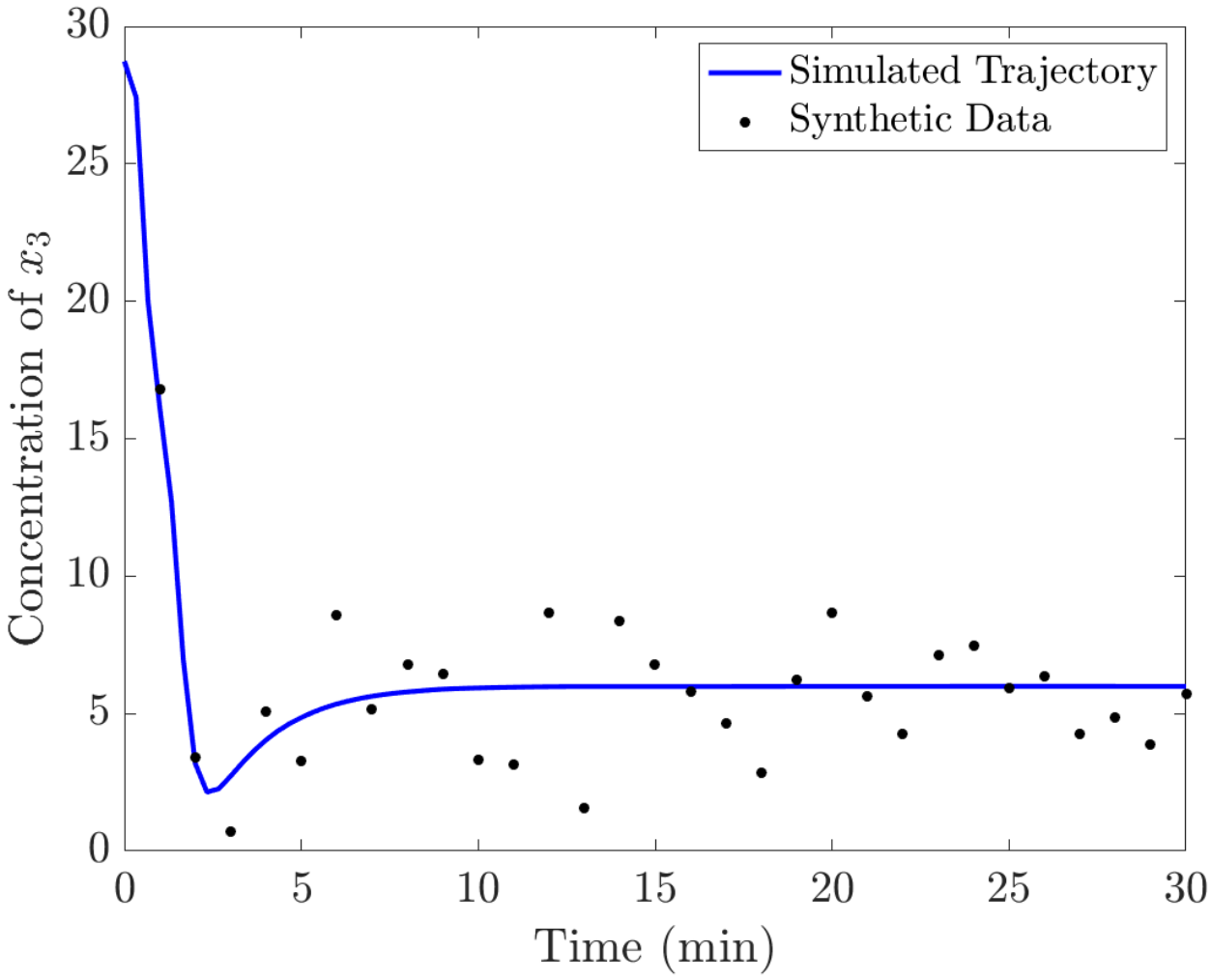

Fig. 1 Example simulation of the MAPK model with parameters and initial conditions that lead to a limit cycle oscillation. The solid blue line indicates the simulation output and the black markers indicate noisy samples of the trajectory.

Tip

This simulation was performed using MATLAB. The corresponding script can be found in `demoModel.m`. Briefly, we used `ode15s()` with the default tolerances to solve the system of ODEs. We define a `solve()` function that wraps `ode15s()` to conveniently solve the system for a specified time interval and set of parameters. Additionally, we provide a function `utils/SBMLtoMATLAB.m` that converts ODE models stored in the SBML format to a MATLAB function.

The `solve()` function is shown below, where `f()` is the function for the ODE, and `jac()` is the function for the Jacobian matrix of the ODE.

```
function xout = solve(x0, f, t, jac, theta)
    % solve the ODE system using ODE15s
    odeOpts = odeset('Jacobian', @(t,x) jac(t,x, theta));
    [~, xout] = ode15s(@(t,x) f(t,x, theta), t, x0, odeOpts);
end
```

For the remainder of this tutorial, our goal is to estimate the model parameters $\boldsymbol{\theta}_f$ from the noisy data and quantify the associated uncertainties. Thus, we want to learn the *posterior* distribution of $\boldsymbol{\theta}_f$.

Specifically, we assume that we have N noisy measurements $\mathcal{Y}_N = [\mathbf{y}_1, \ldots, \mathbf{y}_N]$ of the state vector $\mathbf{x}(t) = [x_1(t),\ x_2(t),\ x_3(t)]^\top$ taken at evenly spaced sample times $t_k \in [t_0, t_{\text{end}}]$. This corresponds to measurements

$$\mathbf{y}_k = \mathbf{x}(t = t_k) + \boldsymbol{\eta}_k, \qquad \boldsymbol{\eta}_k \sim \mathcal{N}(0, \boldsymbol{\Gamma}(\boldsymbol{\theta}_{\boldsymbol{\Gamma}})),$$

where $\boldsymbol{\eta}_k$ is normally distributed measurement noise with diagonal covariance matrix $\boldsymbol{\Gamma}(\boldsymbol{\theta}_{\boldsymbol{\Gamma}}) = \text{diag}(\boldsymbol{\theta}_{\boldsymbol{\Gamma}})$ parameterized by $\boldsymbol{\theta}_{\boldsymbol{\Gamma}}$.

Before performing further analysis, we define the possible ranges (as defined in Nguyen et al. 2015) for each of these parameters in the following table:

Parameter	Nominal value	Lower-bound	Upper-bound
S_{1t}	0.22	0.0	100.0
S_{2t}	10.0	0.0	100.0
S_{3t}	53.0	0.0	100.0
k_1	0.0012	0.0	0.05
k_2	0.006	0.0	0.1
k_3	0.049	0.0	0.05
k_4	0.084	0.0	0.1
k_5	0.043	0.0	0.05
k_6	0.066	0.0	0.1
n_1	5	0	100
K_1	9.5	0.0	10.0
n_2	10	0	100
K_2	15.0	0.0	20.0
α	95.0	0.0	100.0

Moving forward, we assume that the values of S_{1t}, S_{2t}, and S_{3t} are known for the cell-type from which data are collected and fix them to their nominal values. The Hill coefficients n_1 and n_2 are also fixed to nominal values due to limitations of available software in sampling integer parameters. We are now ready to perform *a priori* identifiability and sensitivity analysis to reduce the parameter space down to the identifiable and influential parameters.

Structural Identifiability Analysis: *Which parameters can we estimate?*

Structural identifiability determines which parameters can be uniquely estimated from the available data. Importantly, this analysis does not consider the quality or quantity of the data, but rather the mathematical mapping between the parameters and the measurements. Parameters are characterized as:

- **structurally globally identifiable** - These parameters *can be uniquely determined* from the data, because each unique measurement corresponds to unique parameter values for the same initial condition.
- **structurally locally identifiable** - Several values of these parameters give the same values of a measurement. These parameters *cannot be uniquely determined* across the entire parameter space.
- **nonidentifiable** - These parameters *cannot be uniquely determined* because infinitely many parameter values give the same measurements.

The **S**tructural **I**dentifiability **AN**alyzer (SIAN) software from Hong et al. 2019 (implemented in Julia, e.g., SIAN-Julia) is used for structural identifiability analysis. This software is also available for the Maple programming language, here, and as an web-based tool, here. All of the mathematical details of SIAN can be found in Hong et al. 2020.

> **SIAN installation in Julia**
>
> We use version 1.0.3 of the SIAN-julia package for structural identifiability analysis. This version of SIAN-julia can be installed using Julia's inbuilt package manager by running
>
> ```
> ] add SIAN@1.0.3
> ```

> **Warning**
>
> The SIAN software cannot analyze parameters that are restricted to integer values. In it's original formulation, two of the MAPK model parameters, n_1 and n_2, can only take on integer values. Thus, we exclude them any of the following analysis and fix them to their nominal values.

Following the SIAN-Julia syntax, we use the following Julia script (also in `identifiability.jl`) to test the structural identifiability of all model parameters in the MAPK model:

```
using SIAN

# Run with full state measurements
ode = @ODEmodel(
    x1'(t) = (k1 * (100 - x1(t)) * ((K1^10) / (K1^10 + x3(t)^10))) - (k2 * x1(t)),
    x2'(t) =  (k3 * (100 - x2(t)) * x1(t) * (1 + ((alph * x3(t)^15) / (K2^15 + x3(t)^15)))) - (k4 * x2(t)),
    x3'(t) = (k5 * (100 - x3(t)) * x2(t)) - (k6 * x3(t)),
    y1(t) = x1(t),
    y2(t) = x2(t),
    y3(t) = x3(t)
);

# run with 12 threads
output = identifiability_ode(ode, get_parameters(ode), p_mod=p_mod=2^29 - 3, nthrds=12);
```

The resulting output states that 8 of the 10 analyzed parameters are globally identifiable and the remaining two parameters K_1 and K_2 are locally identifiable. It also determines if the initial conditions are identifiable, but we assume that their values are known *a priori*. Thus, we will fix K_1 and K_2 to avoid any difficulties associated with estimating locally identifiable parameters.

```
=== Summary ===
Globally identifiable parameters:                [k5, k6, x1, k1, x2, k2, alph, k4, x3, k3]
Locally but not globally identifiable parameters: [K1, K2]
Not identifiable parameters:                     []
===============
```

Global Sensitivity Analysis: *Which parameters are important to estimate?*

After identifiability analysis, global sensitivity analysis (GSA) helps us determine which of the identifiable parameters have the strongest influence on the model outputs. We refer the reader to Saltelli et al. 2007 for a detailed textbook on motivation and theory of GSA.

In this work, we use Sobol's variance-based GSA method to rank the input parameters by their contributions to the total variance of the model output quantities. The set of influential parameters is the subset of parameters with sensitivities that are greater than a prescribed cutoff value.

We will use UQLab for global sensitivity analysis in MATLAB. However we also perform equivalent analysis in Julia for the synaptic plasticity model in the manuscript, see, e.g `synaptic_plasticty/sensitivity.ipynb` for an example. We list additional tools for global sensitivity analysis in in Python and Julia in the additional resources section.

> **UQLab Toolbox**
>
> UQLab is a widely used toolkit for uncertainty quantification that is developed and maintained out of ETH Zurich in Switzerland. The toolkit offers software for Markov chain Monte Carlo sampling, sensitivity analysis, polynomial chaos expansion, Gaussian process modeling, and many more uncertainty quantification analyses.
>
> The MATLAB software is available to download from https://www.uqlab.com. We use **version 2.0** for all analysis.

We make several algorithmic and modeling decisions before GSA:

1. **Quantity of interest (QoI)** Standard GSA approaches assess the sensitivity of a scalar output quantity with respect to the model inputs. Such models $y_{\text{output}} = \mathcal{M}(\boldsymbol{\theta})$ map the input vector $\boldsymbol{\theta} \in \mathbb{R}^p$ to a scalar output quantity $y_{\text{output}} \in \mathbb{R}$. However, dynamical systems biology models produce a time-series of biochemical species concentrations. Thus, we need to define scalar outputs that represent meaningful biological features of the time series. Examples of QoIs we use include the steady state concentration of a certain species or the period of a limit cycle oscillation.
2. **Parameter sampling distribution** GSA algorithms sample the input parameters to evaluate the sensitivity measures. We need to specify the marginal distributions for each of the inputs. In this work, we assume that all inputs vary uniformly over their physiological ranges; inputs are assumed to be uniformly distributed random variables.
3. **Sampling design** In addition to a specifying the sampling distributions, we need to choose a sampling algorithm. UQLab implements several sampling schemes including Monte Carlo sampling, Latin hypercube sampling, Halton pseudorandom sampling, and Sobol pseudorandom sampling. We always use the Sobol pseudorandom sampling strategy, because it is equivalent to the standard method implemented in Julia (see here).
4. **Sensitivity index estimator** Lastly, we need to choose an estimator to approximate the Sobol' sensitivity indices from the samples and model evaluations. The sensitivity indices are integral quantities, expectations and variances, so they are approximated numerically with Monte Carlo estimators (see Saltelli et al. 2007). Throughout this work, we use the default estimators in UQLab that implement the Monte Carlo estimators derived in Janon et al. 2014.

> **Warning**
>
> In certain applications, specific parameter combinations may cause failed or erroneous simulations. For example, the simulation will fail if the initial conditions in the MAPK model exceed the the total concentration parameters S_{1t}, S_{2t}, S_{3t}. To avoid these errors, we restrict the bounds of the parameter distributions such that the total concentration is always greater than the initial condition. Accordingly, we find that careful selection of the sampling distributions is essential to avoid these errors and perform accurate GSA.

We are now ready to configure UQLab to perform GSA on our model. The following MATLAB code (also available in the `sensitivity.m` script) defines the inputs for UQLab. Inputs and options in UQLab are always specified via MATLAB structure arrays. Chapter 3 of the UQLab sensitivity analysis documentation outlines these structures for GSA.

First we follow the UQLab documentation about the `input` object to specify uniform distributions for the identifiable parameters.

```
IOopts.marginals = [];
for i = 1:numel(freeParamIndex)
    pidx = freeParamIndex(i);
    IOopts.marginals(i).Name       = paramNames{pidx};
    IOopts.marginals(i).Type       = 'Uniform';
    IOopts.marginals(i).Parameters = paramBounds(pidx,:);
end
input = uq_createInput(IOopts); % create input struct
```

Next we specify the sampling algorithm, estimator, and total number of Monte Carlos samples for UQLab:

```
SobolSensOpts.Type = 'Sensitivity';
SobolSensOpts.Method = 'Sobol';
SobolSensOpts.Input = input;
SobolSensOpts.Sobol.Sampling = 'sobol';
SobolSensOpts.Sobol.SampleSize = 5000;
```

> **Tip**
>
> Large sample sizes are important for accuracy but can significantly increase computation times; finding a balance between accuracy and computation time is key. Starting small and increasing the sample size `SobolSensOpts.Sobol.SampleSize` until the input ranking stabilizes can help in choosing a large enough sample size.

Then, we define a function that simulates the ODE and evaluates the corresponding QoIs:

```
function Y = numericalModel(theta, x0, f, t, jac, qoiFuncs)
    % Function of the form Y = M(theta); maps from the inputs (parameters) to the output QoIs
    % Inputs:
    % - theta:    the model inputs, in the notation of UQLab theta is X
    % - x0:       initial condition for the ODE
    % - f:        ODE function should be @f(t, x, theta)
    % - t:        time (range of vector) to solve ode
    % - jac:      function for jacobian should be @f(t, x, theta)
    % - qoiFuncs: cell array of function handles for the qoi functions
    %               qoi funcs must map from xout --> qoi
    %
    % Outputs:
    % - Y:    output vector of quantities of interest

    f = @(t,x) f(t,x,theta);
    jac = @(t,x) jac(t,x,theta);
    sol = solve(x0, f, t, jac); % solve ODE using ode15s

    Y = zeros(1, numel(qoiFuncs));

    for i = 1:numel(qoiFuncs)
        Y(i) = feval(qoiFuncs{i}, sol);
    end
end
```

Here the `solve()` function solves the ode specified by `f()` with MATLAB's `ode15s()` scheme.

We can then specify the QoI functions to compute the limit cycle amplitude and period, and pass everything into UQLab:

```
% Cell array of QoI functions
qoiFuncs = {@(sol) finalValQOI(sol, 1),@(sol) finalValQOI(sol, 2), @(sol) finalValQOI(sol, 3)};
qoiNames = {'x1-ss', 'x2-ss', 'x3-ss'};

% Function to eval MAPK model
Y = @(X) numericalModel(X, x0, @(t,x,theta) MAPK(x,theta, theta_bistable), t, ...
            @(t,x,theta) Jac(x,theta, theta_bistable), qoiFuncs);

% Pass everything to UQLab
modelOpts.mHandle = Y;
modelOpts.isVectorized = false;
Model = uq_createModel(modelOpts);
```

Here `finalValQOOI(sol, index)` returns the value of the `index` state variable at the final time in the trajectory `sol`. The UQLab function `uq_createModel()` creates the model input for the GSA code.

Lastly, we can run GSA by calling the `uq_createAnalysis()` function:

```
oscAnalysis       = SobolSensOpts;
oscAnalysis.Model = oscModel;
oscSensitivtyResults = uq_createAnalysis(oscAnalysis);
```

The output structure `oscSensitivityResults` is

```
  uq_analysis with properties:

     Options: [1×1 struct]
     Results: [1×1 struct]
    Internal: [1×1 struct]
        Name: 'Analysis 1'
        Type: 'uq_sensitivity'
```

where `Results.Total` contains total-order sensitivity indices and `Results.FirstOrder` contains first-order sensitivity indices.

> **Warning**
>
> Sobol sensitivity analysis may return **negative** first-order sensitivity indices. These erroneous results indicate too few Monte Carlo samples were used or that there were numerical errors in the QoI computations. To resolve this, repeat GSA with larger values of the `SobolSensOpts.Sobol.SampleSize` until the input ranking stabilizes and all sensitivity indices are greater than or equal to zero. If negative indices persist, the QoI may be returning nonphysical outputs for certain inputs or it may have an implementation error. Evaluating the QoI with random input samples and analyzing the outputs can help diagnose any problems directly.

We can then plot the Sobol sensitivity indices for the identifiable parameters:

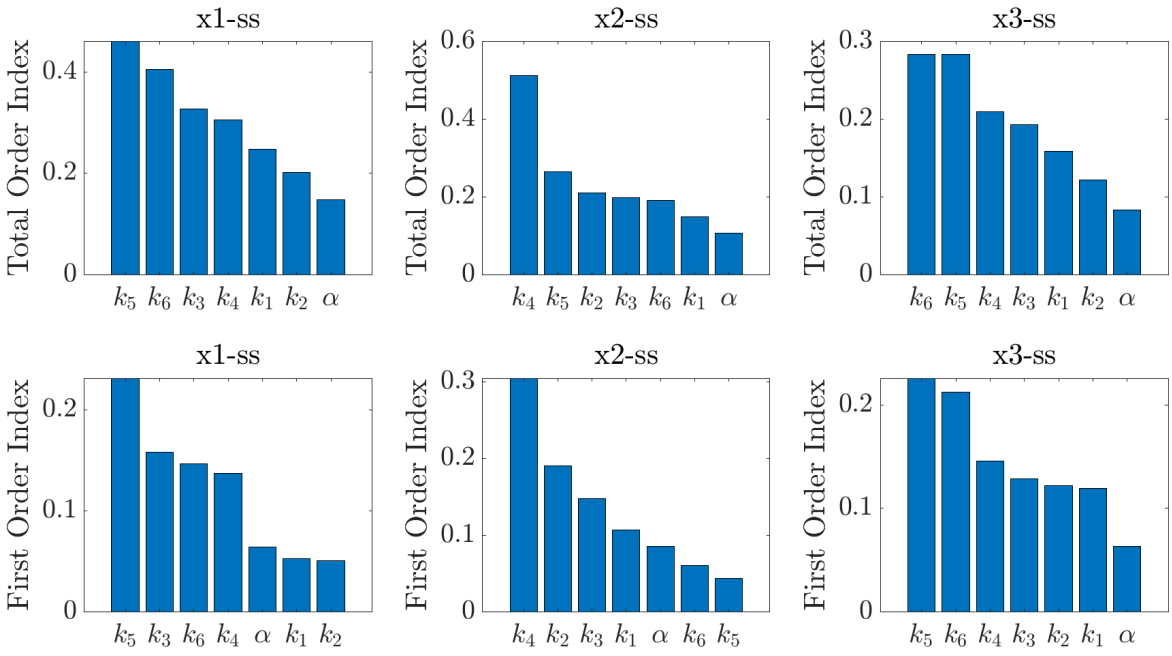

Fig. 2 Sobol sensitivity results for the MAPK steady state concentration with respect to the globally structurally identifiable parameters. The parameters are ordered by the sensitivity index to visualize their relative rankings.

We then select two parameters with the largest sensitivity indices for each of the x_1 and x_2 steady states for estimation and fix the remaining parameters to their nominal values.

> **Parameter subset selection:** ***How do I choose the subset of influential parameters from the GSA rankings?***
>
> Parameter subset selection is a nuanced problem that requires choosing how many sources of variations to discard from the subsequent uncertainty analysis. A recent paper on open problems in mathematical biology (Vittadello and Stumpf, 2022) highlights the need for fully quantitative methods for parameter subset selection. We select of subset of parameters with sensitivity indices above a prescribed cutoff value. The cutoff is chosen to correspond to an obvious drop in the index value or to a logical value such as a first-order index greater than 0.1. A parameter with a first-order index of 0.1 directly contributes to 10% of the variance of the output quantity.

For this example, we choose the four parameters k_2, k_4, k_5, k_6 with the largest sensitivity indices as the set of influential parameters.

Bayesian Estimation with CIUKF-MCMC

We are now ready to estimate the posterior distribution $p(\boldsymbol{\theta}|\mathcal{Y}_n)$ for the four parameters $\boldsymbol{\theta}_f = [k_2, k_3, k_4, k_6]^\top$ and the remaining statistical parameters in $\boldsymbol{\theta}$. Bayes' rule,

$$p(\boldsymbol{\theta} \mid \mathcal{Y}_n) \propto p(\boldsymbol{\theta}) p(\mathcal{Y}_n \mid \boldsymbol{\theta}),$$

states that the posterior distribution is proportional to the product of the prior distribution $p(\boldsymbol{\theta})$ and the likelihood function $p(\mathcal{Y}_n \mid \boldsymbol{\theta})$. Thus, we need to define a likelihood function and prior distribution to apply Bayes' rule.

In this work, we use the CIUKF-MCMC algorithm (based-on UKF-MCMC from Galioto and Gorodetsky, 2020) to compute an approximate likelihood function. We refer the reader to our manuscript for all details of this algorithm and only outline practical usage of the method here. In a nutshell, CIUKF-MCMC constructs a likelihood function $p(\mathcal{X}_n, \mathcal{Y}_n \mid \boldsymbol{\theta})$ that simultaneously considers uncertainty in the data and the states. The constrained interval unscented Kalman filter (CIUKF) approximates a high-dimensional marginalization integral to approximate the marginal likelihood $p(\mathcal{Y}_n \mid \boldsymbol{\theta}_f)$.

The MATLAB functions `ciukflp()` and `ciukflp_quadProg()` evaluate $\log p(\mathcal{Y}_n \mid \boldsymbol{\theta}_f)$ for nonlinear $h(\mathbf{x}(t, \boldsymbol{\theta}))$ and linear $\mathbf{H} \cdot \mathbf{x}(t, \boldsymbol{\theta})$ measurement functions, respectively. These MATLAB functions are located in the `utils/` directory of the GitHub repository. We will use `ciukflp_quadProg()` which has the following function header:

```
function loglikelihood = ciukflp_quadProg(theta, x0, P0, f, H, Q, R, y, alpha,beta,kappa,eps, optOptions)
    ...
end
```

The inputs are:

- `theta` - parameter vector to evaluate the log likelihood. This parameter vector $\boldsymbol{\theta} = [\boldsymbol{\theta}_f, \boldsymbol{\theta}_\Sigma, \boldsymbol{\theta}_\Gamma]$, where $\boldsymbol{\theta}_f$ is the model parameters, $\boldsymbol{\theta}_\Gamma$ is the measurement noise covariance parameters, and $\boldsymbol{\theta}_\Sigma$ is the process noise covariance parameters.
- `x0` - initial condition
- `P0` - initial covariance for the Kalman filter. We usually set this to a diagonal matrix with very small diagonal entries, e.g., `P0 = 1e-16*eye(d)`, where `d` is the dimension of the state.
- `f` - state propagator function (see `utils/propf.m` for details)
- `H` - Measurement operator matrix
- `Q` - function to compute process noise covariance matrix from `theta`, e.g., `Q = @(theta) diag(theta(end-5:end-3))`
- `R` - function to compute process noise covariance matric from `theta`, e.g., `R = @(theta) diag(theta(end-2:end))`
- `y` - data matrix. Each column is one observation of the state
- `alpha` - a parameter for CIUKF, set to `1e-3`
- `beta` - a parameter for CIUKF, set to `1`
- `kappa` - a parameter for CIUKF, set to `0`
- `eps` - small number to ensure symmetric positive definite matrices, set to `1e-10`
- `optOptions` - options for the quadratic programming optimizer for CIUKF, set using `optimoptions` for the `'quadprog()` function, e.g., `optOptions = optimoptions('quadprog', 'Display','off', 'Algorithm', 'trust-region-reflective')`

Now we must define the prior probability distribution $p(\boldsymbol{\theta})$ for the parameters. Here we assume that all model parameters and noise covariance parameters are independent, so we can specify the marginal priors and compute the joint prior by multiplication.

> **Prior distribution**
>
> The marginal prior probability distribution $p(\theta_i)$ for a parameter θ_i encodes the information that we know about that paramter before incorporating any data. Choosing this distribution effects the inference outcomes, so care must be taken in making this decision. We refer the user to Smith 2013, and Gelman et al. 2021 for in depth introductions to prior distributions.

In this example, we will use weakly informative bounded uniform prior distriubtions for the model parameters and truncated right half normal priors for the noise covariance parameters. Bounded uniform priors allow us to specify the physiological ranges for the model parameters without specifying a most likely value. For example, we set the prior for k_1 as $k_1 \sim \mathcal{U}(0, 0.5)$, where the probability density function $f(x) = \mathcal{U}(a, b)$ is defined

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } z \in [a, b] \\ 0 & \text{else.} \end{cases}$$

The upper and lower bounds of all model parameters are listed in Table parameter-table above. Following Galioto and Gorodetsky 2019 and Gelman 2006, we use right half normal priors with the mean set to 0 for the noise covariance parameters. We found that truncating these distributions at an upper bound equal to the covariance of each observed state variable and setting the variance to $1/3$ of this value improved MCMC convergence.

The MATLAB code for Bayesian inference in this tutorial is in `ciukf_inference.m`, and we include important snippets below.

First, we create a `uq_input` object to store all of the prior information for UQLab. This is

```
priorOptions.Name = 'Prior distribution MAPK';

% Model Parameters
for i = 1:pdyn
    priorOptions.Marginals(i).Name =       paramNames{i};
    priorOptions.Marginals(i).Type =       'Uniform';
    priorOptions.Marginals(i).Parameters = bounds(i,:);
end

% Add noise covariance parameters as priorOptions.Marginals(i + 1)
...

% creat input:
priorDistribution = uq_createInput(priorOptions);
```

Bayes' rule allows us to evaluate the posterior distribution at specific points in parameter space, so we use Markov Chain Monte Carlo (MCMC) to draw samples to help characterize it over the entire parameter space. UQLab provides implementations of several MCMC algorithms including Metropolis Hastings, Adaptive Metropolis, Hamiltonian Monte Carlo, and the Affine Invariant Ensemble Sampler (see UQLab docs for details). We chose to use the Affine Invariant Ensemble Sampler (AIES; see Goodman and Weare, 2010 for an introduction), because this sampler is well suited for sampling in multiscale parameter spaces; we encounter multiscale distributions because the values of parameters in a systems biology model often span several orders of magnitude.

UQLab enables straighforwd application of AIES to our Bayesian inference problems. Following the syntax outlined in the docs, we create the structure arrrays below to specify the AIES algorithm and provide other necessary inputs.

```
% Inverse problem solver
Solver.Type = 'MCMC';
Solver.MCMC.Sampler = 'AIES'; % Affine Invariant Ensemble Algorithm
Solver.MCMC.NChains = 20; % Number of chains for AIES
Solver.MCMC.Steps = 3500; % Steps per chain

% STRUCT for the UQLab
BayesOpts.Type = 'Inversion';
BayesOpts.Name = 'MAPK Example';
BayesOpts.Prior = priorDistribution;
BayesOpts.Data = data;
BayesOpts.LogLikelihood = logLikelihood;
BayesOpts.Solver = Solver;
```

Here, the `Solver` structure specifies settings for AIES, where `Solver.MCMC.NChains` specifies the number of members in the ensemble and `Solver.MCMC.Steps` specifies how many MCMC steps to run AIES. We found that using large ensembles, e.g. `Solver.MCMC.NChains = 150` improves convergence with the CIUKF-MCMC algorithm. However, such large ensembles can slow down computation times, so we set it to `20` in the corresponding tutorial script. Choosing the correct chain length `Solver.MCMC.Steps` is more difficult and typically requires usage of a convergence metric such as the integrated autocorrelation time that we introduce below. MCMC is run by calling:

```
BayesianAnalysis = uq_createAnalysis(BayesOpts);
```

After running the sampler, we can plot the ensemble of Markov chains with the `uq_display()` function in UQLab. Please see the following figures for the traces of the MCMC ensemble for the four model parameters.

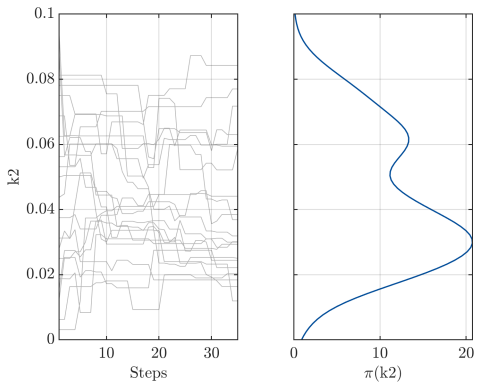

Fig. 3 Ensemble MCMC trajectory (left) and the estimated probability density function (right) for k_2.

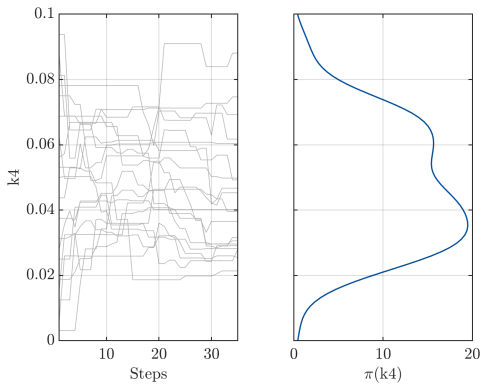

Fig. 4 Ensemble MCMC trajectory (left) and the estimated probability density function (right) for k_4.

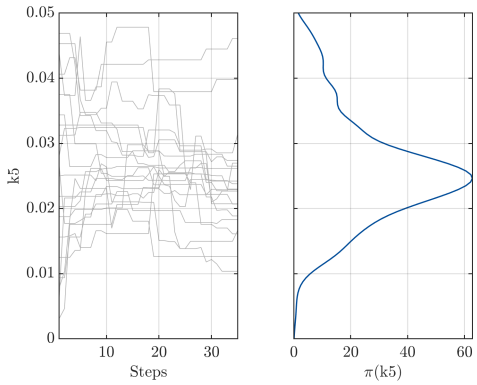

Fig. 5 Ensemble MCMC trajectory (left) and the estimated probability density function (right) for k_5.

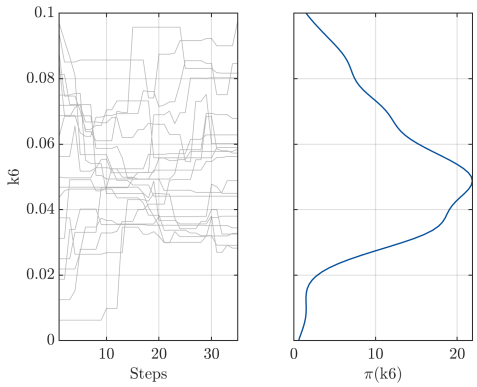

Fig. 6 Ensemble MCMC trajectory (left) and the estimated probability density function (right) for k_6.

> **Warning**
>
> The above example MCMC traces show results with too few, 35, MCMC steps for illustrative purposes only. In practice we need to run MCMC for many more, usually 1,000s to 10,000s of steps.

While this example does not run enough MCMC steps, we can still use it to illustrate two important points in MCMC analysis, burn in and convergence assessment.

First, all four sets of trajectories are still in a transient period, called burn in, during these 35 initial MCMC steps. Burn-in occurs because the distribution being sampled has not yet converged to the target posterior distribution. To ensure that the final samples come from the posterior distrubution, we typically discard these initial samples after MCMC is completed. The question is then: *How many samples do we discard?* One could attempt to eyeball this from the traceplots (plots shown above). However we choose to take a more principled approach by computing the maximum integrated autocorrelation time (IACT) of the ensemble of Markov Chains.

Integrated autocorrelation time (IACT)

The integrated autocorrelation time τ_s of a Markov chain quantifies the time (number of steps) it takes for the samples to completely decorrelate. Given a Markov chain of N samples $\{\boldsymbol{\theta}^i\}_{i=1}^N$, the integrated autocorrelation time is

$$\tau_s = \sum_{T=-\infty}^{\infty} \frac{C_s(T)}{C_s(0)},$$

where $C_s(T) = \lim_{t^* \to \infty} \mathrm{cov}[\boldsymbol{\theta}^{t^*+T},\ \boldsymbol{\theta}^T]$ is the autocovariance function with lag $T \in \mathbb{N}$.

We use the UWerr_fft() function from Wolff. 2004 to compute the IACT. As this function computes τ_s for a single chain of samples, we compute the IACT for every chain of every parameter, take the average for each parameter, and then the maximum of these avergages. This gives us the maximum time it takes for any of the chains in the ensemble to decorrelate. We have written a helper function in `utilities/computeIACT.m` that wraps the `UWerr_fft()` function and performs the averaging and maximization described above.

The IACT τ_s allows us to determine how many samples to discard such that the ensemble *forgets* the initial conditions. To be safe, we typically discard 5-10 times τ_s as burn in and keep the remaining samples. In UQLab, the `uq_postProcessInversionMCMC()` function allows us to remove burn-in by setting the `'burnIn'` parameter to the desired integer number of samples to discard.

Further analysis of the IACT allows us to understand if we have run enough MCMC steps. In practice, we will see τ_s increase with the number of MCMC steps and eventually stabilize. Our goal is run the sampler for enough steps that the IACT has stabilized and that is is significantly smaller than the total number of steps.

In this example, the IACT is `IACT = 4.0465` for the ensemble with 35 steps. Discarding 5 times the IACT samples as burn-in in this toy example would leave us with chains that are only 15 steps long. This is not long enough with respect to the IACT, so in practice, we would want to run MCMC for much longer. One good but potentially infeasible practice is too aim for 100 times IACT number of samples. Once we have run the sampler long enough, discarded the appropriate burn-in and are satisfied with the convergence, we are ready to use the posterior samples for UQ of the model predictions.

Output uncertainty analysis: *How does the parameter uncertainty effect the predicted dynamics?*

After MCMC, we want to understand how uncertainty in the parameters propagates to our model predictions. As most systems biology models are relatively fast to simulate, we choose to take a simulation-based approach for uncertainty propagation. The idea is to draw random posterior samples (parameter sets) and run an ensemble of simulations with the sampled parameters. Analysis of the ensemble of trajectories allows us to estimate the uncertainty in the model predictions.

The `outputUncertaintyAnalysis.m` script performs this uncertainty analysis and computes the trajectory and uncertainty bound for $x_3(t)$ in the following figure. The `uq_postProcessInversionMCMC()` randomly samples the posterior and stores these in `BayesianAnalysis.Results.PostProc.PostSample`. Simply iterating over these samples and running simulations gives the ensemble of trajectories in the following figure.

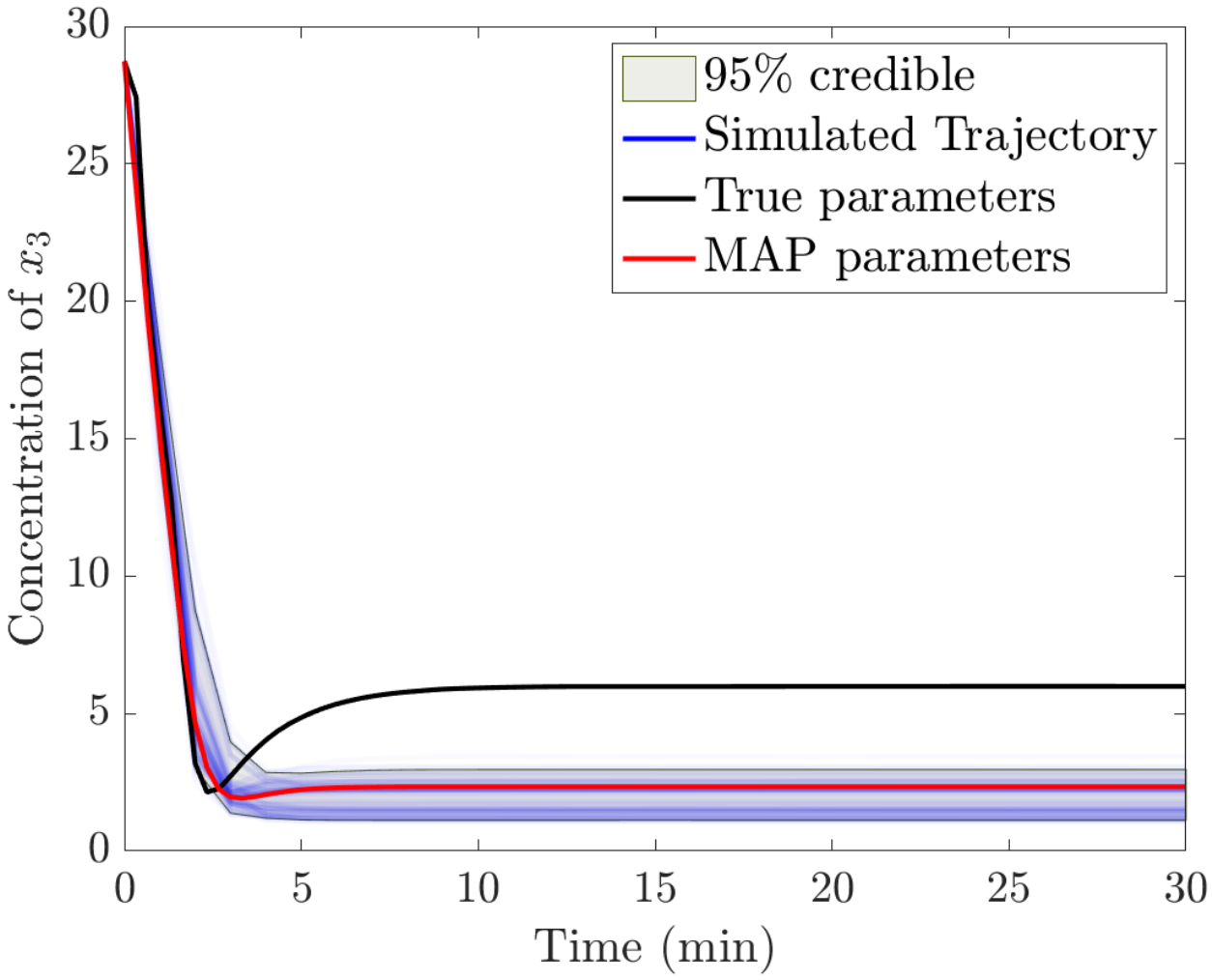

Fig. 7 Quantification of the model's predictive uncertainty by ensemble-based uncertainty propagation. The solid black line shows the *true* trajectory with the nominal parameters, the red line shows the trajectory with MAP (*maximum a posteriori*; most probable) parameters, and the green shaded region is the 95% credible interval. Blue lines are individual trajectories.

Additional statistical analysis can help quantify the models uncertainty in predicting specific biologically relevant features such as the time to steady state or the steady state values of a specific species.

Additional Resources

Identifiability analysis:

Julia

- **StructuralIdentifiability.jl** (https://si.sciml.ai/dev/) Global structrual identifiability analysis in Julia that uses a different algorithm than SIAN-Julia. See Dong et al. 2021 for mathematics and algorithmic details.

Global sensitivity analysis:

Julia

- **GlobalSensitivity.jl** A global sensitivity analysis package that interfaces with **DifferentialEquations.jl** and is a part of the SciML suite of Julia tools.

Python

- **SALib** (https://salib.readthedocs.io/en/latest/) A Python package for sensitivity analysis that offers a wide range of algorithms.

Markov Chain Monte Carlo:

Python

- **emcee** (https://emcee.readthedocs.io/en/stable/) A Python implementation of the Affine Invariant Ensemble Sampler. See Foreman-Mackey et al. 2012 for details.
- **PyMC** (https://www.pymc.io/welcome.html) Probabalistic programming environment that offers state-of-the-art MCMC tools in Python. Widely used in the statistics community.

R or ***Python***

- **STAN** (https://mc-stan.org) Standalone probabilisitc programming language that has interfaces in R and Python. Widely used in the statistics community.

By Nathaniel Linden, Boris Kramer and Padmini Rangamani