Performing Uncertainty Analysis In Three Steps: A Hands-on Guide

This post will walk you through a complete uncertainty analysis case study using Latin Hypercube sampling, Monte Carlo simulation, and...

Shuai Guo

Follow

Towards Data Science androidstudio ~12 min read ·
January 13, 2021 (Updated: December 27, 2021) · Free: No

This post will walk you through a complete uncertainty analysis case study using Latin Hypercube sampling, Monte Carlo simulation, and hypothetical outcome plots.

In generic form, we have a model $f(.)$ with some model parameter θ. This model should simulate some real-life processes. Then, given input x, we can use the model to predict, which leads us to y.

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

More than often, the model parameter is θ uncertain, i.e., we cannot be sure of its exact value. This happens when we calibrate θ from limited, noisy training data.

The result of uncertain θ is that the corresponding predictions y are also uncertain. Since effective decision-making relies heavily on a reliable prediction of y, it is imperative for analysts to report the model prediction uncertainty.

To do that, we will need to do a **forward uncertainty quantification analysis**. This type of analysis is designed for quantifying the model prediction variations given the input data uncertainties. It is called *forward* as the uncertainty information flows from the input, through the model, to the output.

In this article, we will walk through a complete case study to see how forward uncertainty quantification analysis is conducted in practice. Here is an overview of what we are going to do:

1. **Case study** → we consider the modeling of the disease spread in a population, where we treat the *infection* and *recovery rate* as the uncertain parameters, and we investigate the resulting prediction variations of the *highest number of infected cases* and its *occurrence time*.

understandable, easy to implement, and quite popular for uncertainty quantification purposes.

3. **Workflow** → We do the uncertainty analysis in three steps: random sample generation, uncertainty propagation, and uncertainty visualization.

We will dive deeper into the technical details of each step in the following sections.

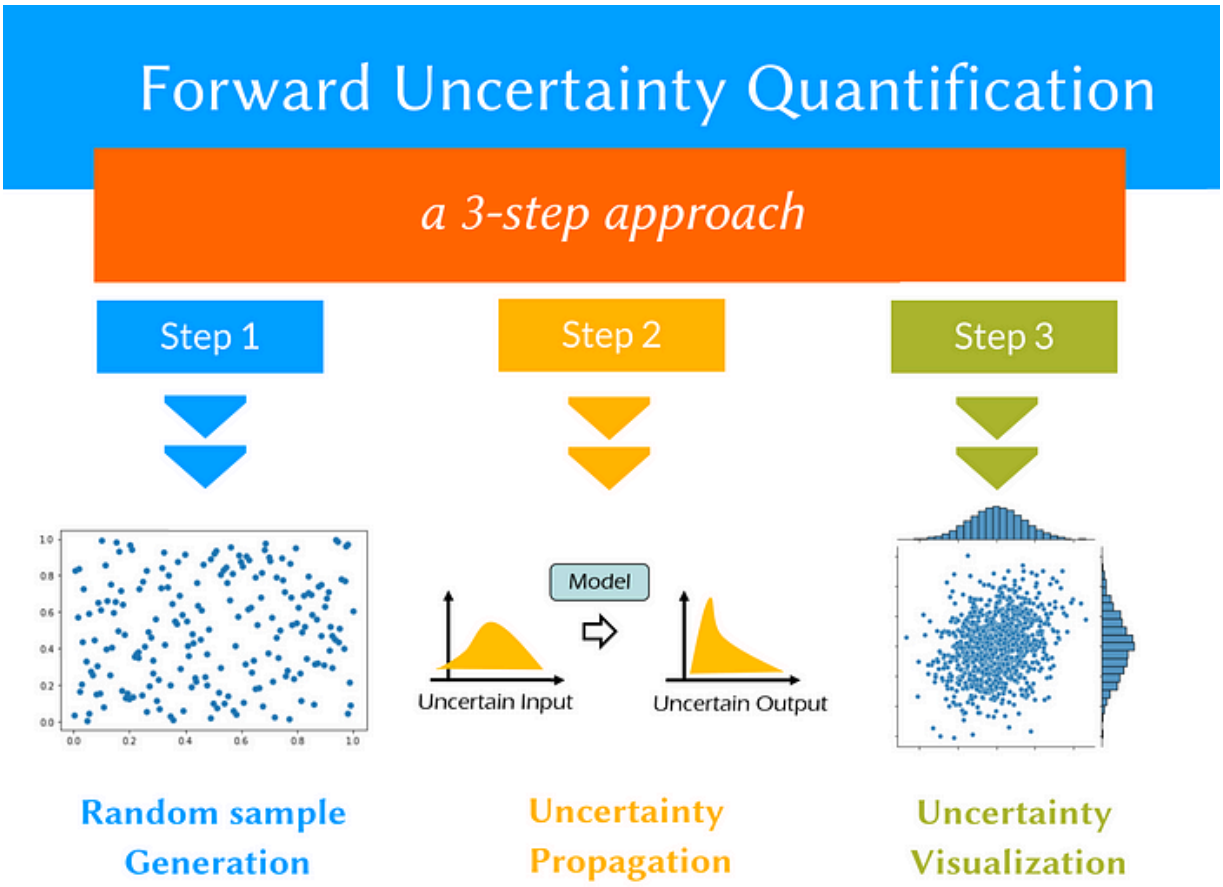

Fig. 1 An overview of the forward uncertainty quantification analysis. (Image by Author)

> You can find the companion **Jupyter Notebook** here, where all the presented analysis and results can be reproduced.

Table of Content

1. Problem Statement

1.1 Background

In this case study, we model the spread of a disease in a population using the **SIR** model. In its basic form, the SIR model divides the total population N into three distinct compartments that vary as functions of time t:

- $S(t)$, the number of individuals who are **S**usceptible but not yet infected with the disease;
- $I(t)$, the number of Infected individuals;
- $R(t)$, the number of individuals who are have **R**ecovered from and immune to the disease.

The SIR model describes the time evolution of the $S(t)$, $I(t)$, and $R(t)$ populations with the following system of ordinary differential equations:

$$\frac{dt}{dt} = \frac{N}{N},$$
$$\frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I,$$
$$\frac{dR}{dt} = \gamma I,$$

where β denotes the infection rate, and γ represents the recovery rate.

1.2 Uncertainty Analysis

We treat β and γ as the two uncertain model parameters. Those model parameters are typically unknown when encountering an outbreak of a new disease (e.g., COVID-19).

In practice, official records concerning the evolution of the number of infected and recovered patients are used to estimate β and γ. Unfortunately, those records may not be accurate, especially at the early stage of the outbreak. Consequently, β and γ estimation from the "noisy" data will be uncertain. Deriving parameter estimation uncertainty is in the domain of **system identification**, and it usually happens before the forward UQ analysis.

Since our current study focuses on the forward uncertainty quantification analysis, we will simply assume that the β–γ estimation has already been done, and the following bivariate normal distribution characterizes their uncertainties:

$$\begin{pmatrix} \beta \\ \gamma \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} 0.22 \\ 0.1 \end{pmatrix}, \begin{pmatrix} 2e & 4e \\ 4e^{-5} & 1e^{-4} \end{pmatrix} \right)$$

Here, (0.22, 0.1) represents the mean values of (β, γ), (2e-4, 1e-4) represents their variance values, and 4e-5 denotes their covariance value.

Running the SIR model also requires other parameters besides β and γ:

- $I(0)$, number of initially infected individuals;
- $R(0)$, number of initially recovered/immune individuals;
- N, the population size.

We simply assume they are constants for the current study, with $I0$=8, $R0$=0, and N=1000.

2. Monte Carlo simulation

Monte Carlo simulation is a simple yet powerful statistical method. It enables us to generate representative samples from the target output distribution without even knowing the distribution form, which is achieved by simply simulating the model outputs under various input scenarios. Later on, we can retrieve the output distribution based on the accumulated samples.

Implementing Monte Carlo simulations is straightforward:

1. Draw a large number of random *samples* of β and γ from their probability distribution (i.e., bivariate normal);

highest number of infected cases and its occurrence time;

3. Based on the ensemble of predictions, we can estimate the joint/marginal probability distributions of the two outputs.

To better understand how Monte Carlo simulations can help quantify the model prediction uncertainty, take a look at my post here:

3. Preparations

Before we get our hands on the uncertainty analysis, let's do some necessary preparations.

3.1 Packages

```
# Data analysis
import pandas as pd
import numpy as np

# Simulate SIR model
from scipy.integrate import odeint

```

Besides the basic data analysis and visualization packages, we need to import some additional packages to facilitate the target uncertainty analysis:

- `pyDOE` (line 9): DOE stands for "Design of Experiments." We use this package to generate random samples of β and γ. In particular, we will use **Latin Hypercube Sampling** to achieve this goal. To install `pyDOE`, use `pip install pyDOE`.
- `celluloid` (line 19): we use this package to create animations to enhance uncertainty visualization. To install `celluloid`, use `pip install celluloid`.

3.2 Simulating SIR Model

To keep things organized, it is beneficial to define a function to simulate the SIR model before running the uncertainty analysis.

```
1  def SIR_model(beta, gamma, t, N, I0, R0):
2      """The function solves the SIR model to simulate
3      the disease spreading"""
4
5      S0 = N - I0 - R0
6
7      def deriv(y, t, N, beta, gamma):
```

The `SIR_model` function takes in the infection rate β, recovery rate γ, a grid of time points (in days) t to calculate the epidemic evolution, the population size N, and the initially infected and recovered cases $I0$ and $R0$, respectively.

The system of ordinary differential equations is solved by `scipy.integrate.odeint` function (line 14), which yields $S(t)$, $I(t)$, $R(t)$ values calculated at the specified time grids t.

To get an intuitive understanding of the SIR model prediction results, we could run the above SIR model given the mean values of β and γ, i.e., β=0.22 and γ=0.1.

```
# Grid of time points (in days)
t = np.arange(0, 101, 2)

# Run SIR model to predict epidemic evolution
```

The evolutions of $S(t)$, $I(t)$, and $R(t)$ are shown in the figure below. We can see that as more people get recovered, the susceptible and infected populations drop down significantly.

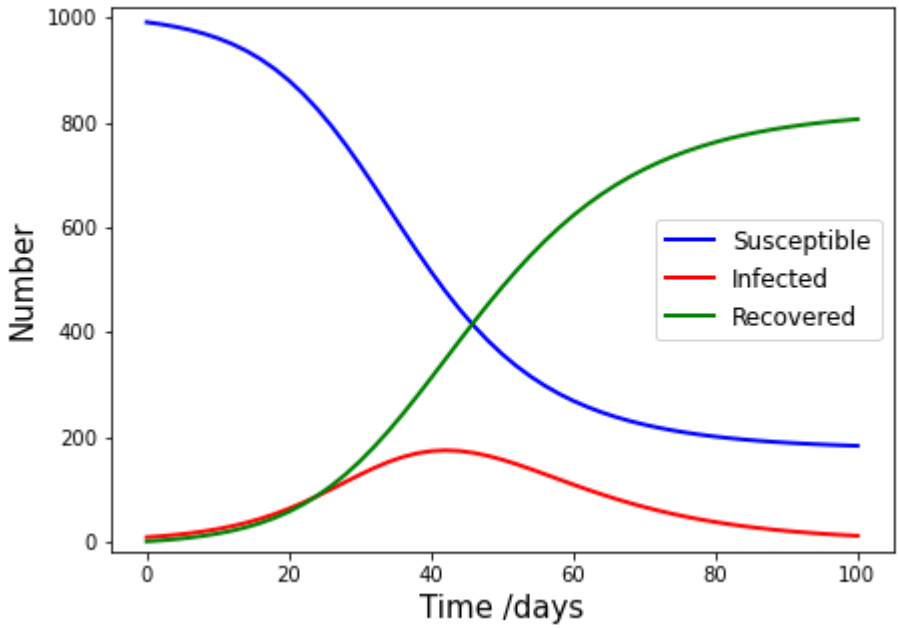

Fig. 2 Predicted epidemic evolution. (Image by Author)

By post-processing the time series of $I(t)$, we can obtain our outputs of interest: the number of infected individuals peaks 40 days after the outbreak, reaching a total of 190 infected people.

Now we've got everything ready. It's time to do some uncertainty analyses! In the following three sections, we will go through each of the three steps in detail.

4. Random Sample Generation

samples will be used in the subsequent Monte Carlo simulations.

4.1 Latin Hypercube Sampling

Latin Hypercube sampling (LHS) is an advanced sampling method aiming at generating "space-filling" samples. Compared with the naive random sampling approach (such as `numpy.random.rand`), Latin Hypercube samples can largely avoid clusters and gaps in the parameter space (see figure below), therefore filling the entire parameter space evenly. This property is desired in the Monte Carlo simulation as it improves the estimation accuracy.

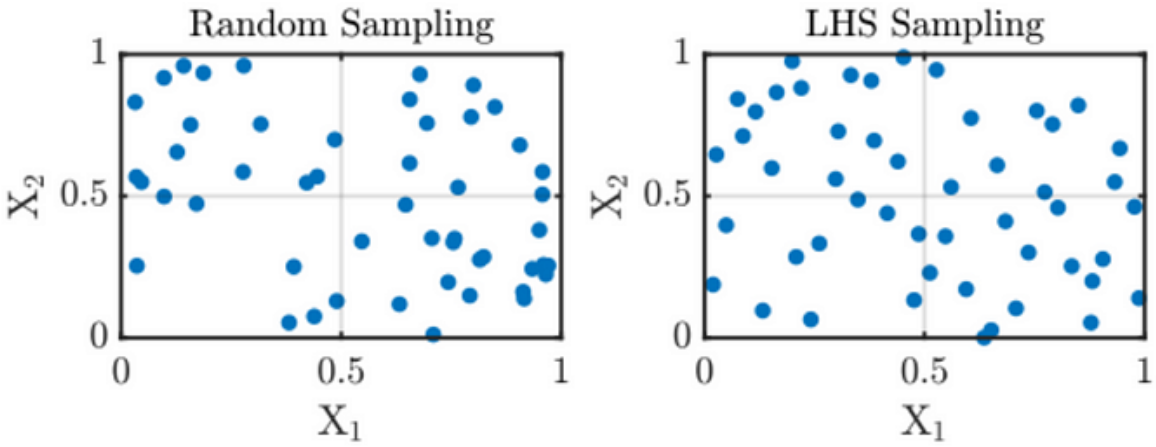

Fig. 3 Latin Hypercube sampling can generate "space-filling" samples. (Image by Author)

In the current study, we adopt the LHS implementation from the `pyDOE` package to generate samples. `pyDOE` is designed to help the scientist, engineer, statistician, etc., to construct appropriate experimental designs. Read more here.

4.2 Sample Transformation

The samples created by `pyDOE.lhs` are distributed uniformly within [0, 1]. As a result, we need to perform **sample transformation** to turn

4.3 The Entire Sampling Process

We do the whole sampling process in three steps:

Step 1: Generate samples from a uniform distribution $U(0,1)$.

```
# Specify random sample number
sample_num = 1000

# Using Latin Hypercube method to generate uniform distribution
uni_samples = lhs(n=2, samples=sample_num, criterion='maximin')
```

UQ_Uniform.py hosted with ❤ by GitHub view raw

For the function `pyDOE.lhs`, it takes in three parameters:

- n: the number of parameters (an integer);
- samples: the number of samples to generate (an integer);
- criterion: determines how `lhs` will sample the points (a string).

So for this case, I instructed `pyDOE.lhs` to generate 1000 samples for 2 parameters, using *maxmin* criterion, which maximizes the minimum distance between sample points. For other available options in *criterion*, check here.

The generated 2D array `uni_samples` has 1000 rows and 2 columns, where each column holds 1000 random realizations drawn from a uniform distribution $U(0,1)$.

Step 2: Turn the uniform distribution into the standard normal distribution.

distribution $N(0,1)$.

To achieve that goal, we apply the **inverse transformation sampling** technique. This transformation process is illustrated in the figure below, where the blue curve is the cumulative distribution function of a standard normal random variable.

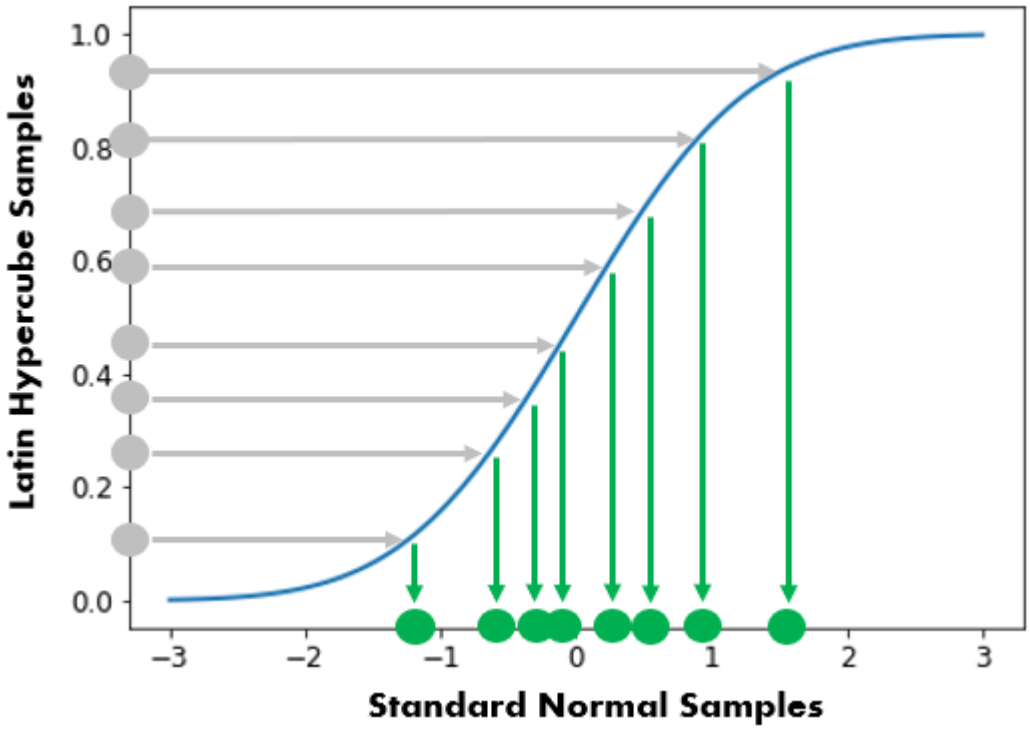

Fig. 4 Inverse transformation sampling technique. (Image by Author)

```
# Transforming to standard normal distribution
std_norm_samples = np.zeros_like(uni_samples)
for i in range(2):
    std_norm_samples[:,i] = norm(loc=0, scale=1).ppf(uni_samples[:,i])
```

UQ_Standard.py hosted with ❤ by GitHub view raw

Step 3: Turn standard normal distribution into the target bivariate normal distribution.

normal distribution $\mathbf{Z}\sim N(0, I)$ can be described as the following:

$$Y = \mu + ZL^T,$$

where L is the lower-triangular matrix obtained by applying the Cholesky decomposition to the covariance matrix $\mathbf{\Sigma}$, i.e.,

$$\mathbf{\Sigma} = LL^T$$

In the above transformation, we assume both Y and Z are row vectors of random variables.

Based on the above theorem, we first apply Cholesky decomposition (`np.linalg.cholesky`) to the covariance matrix of our target normal distribution to get L. Then, we use the above equation to turn the standard, normally distributed samples obtained from the last step into the samples that follow our target bivariate normal distribution.

```
# Specify parameter distribution
mean_val = np.array([0.22,0.1])
cov_val = np.array([[2e-4, 4e-5],[4e-5,1e-4]])

# Transforming to the target distribution
L = np.linalg.cholesky(cov_val)
target_samples = mean_val + std_norm_samples.dot(L.T)
```

We can visualize the transformed samples.

```
4
5   plt.rcParams["axes.labelsize"] = 15
6   sns.jointplot(data=df_samples, x=r"$\beta$", y=r"$\gamma$", height=5);
```

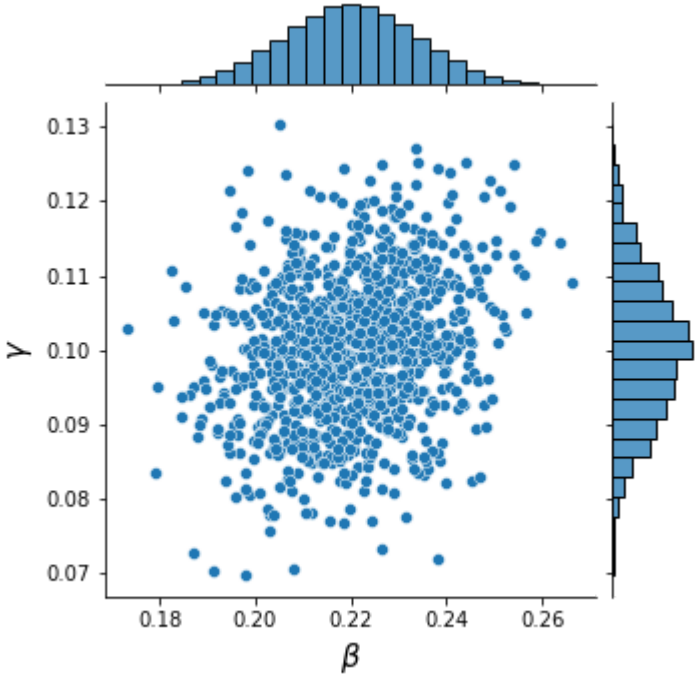

Fig. 5 The samples drawn from the target bivariate normal distribution. (Image by Author)

From the marginal distributions, we can see that both β and γ are indeed following normal distributions, centered at 0.22 and 0.1, respectively. Also, we can see that the β and γ samples are positively correlated from the scatter plot. This matches with the fact that the off-diagonal term of our target covariance matrix is positive ($4e{-}5$, to be exact).

5. Monte Carlo Simulations

Now we are ready to perform Monte Carlo simulations. This step is simple: we just need to create a loop to predict the evolution of S, I, and R for each sample of β and γ.

```
for i in range(sample_num):
    S, I, R= SIR_model(beta_samples[i], gamma_samples[i],
                       t, I0=8, N=1000, R0=0)

```

6. Uncertainty visualization

Now we are ready to visualize the obtained results. To present the output uncertainties, the first thing that comes to our mind would be to use histograms and scatter plots. `Seaborn` package offers `jointplot` function, which allows to simultaneously display the joint distribution of the variables and the marginal distributions of individual variables.

```
# Visualize output uncertainty
outputs = pd.DataFrame({"Max Infection":max_infection,
                        "Occurrence Day":max_time})

plt.rcParams["axes.labelsize"] = 15
h = sns.jointplot(data=outputs, x="Occurrence Day", y="Max Infection",
                  # xlim=(5,165), ylim=(-20,1120),
```

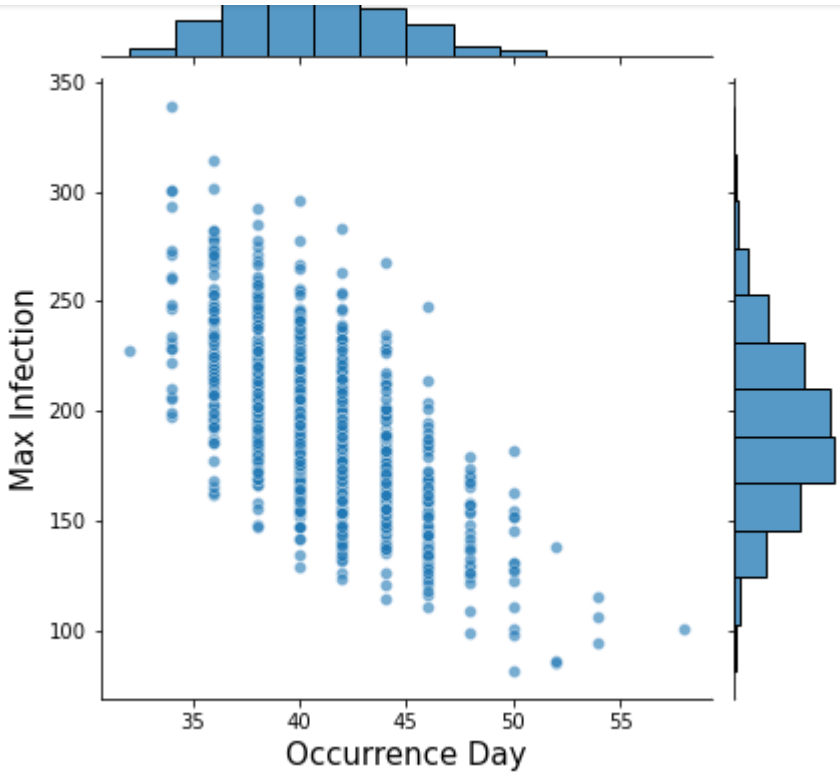

Fig. 6 Output uncertainties presented in histograms and scatter plots. (Image by Author)

From the figure above, we can intuitively understand how the outputs are varying given the uncertain input parameters. There are several things we may notice:

- both the highest number of infected cases and its occurrence day appear to be normally distributed;
- the most likely scenario is that the number of infected cases climbs up to a maximum of 180~200, which occurs 38~40 days after the outbreak;
- our two outputs are negatively correlated. This means that if the number of infected cases reaches its highest early, this number also tends to be high, and vice versa.

Based on the obtained results, we may want to ask more questions, like:

- what $\beta-\gamma$ combinations drive the highest number of infected cases exceed a certain threshold?

- how should we reduce the uncertainty of the output predictions?

To answer those questions, we need to further conduct **global sensitivity analysis** and **robust design analysis.** Together with the forward uncertainty quantification, they form the backbone of **uncertainty management in computational science and engineering.** If you want to learn more, please take a look at my post here:

Uncertainty Quantification Explained

A practice for making reliable model-based predictions

towardsdatascience.com

7. Hypothetical Outcome Plots

In addition to static visualizations, we could make the presentation of our results more vivid and intuitive by using a data visualization technique called **hypothetical outcome plots.**

Hypothetical outcome plots are especially good at communicating analysis uncertainty to broader audiences, such as stakeholders, domain experts, etc., who don't necessarily have a strong statistics background. Basically, this method works by creating animations to cycle through a number of different plots, each of which simulates one possible scenario drawn from the outcome distribution. For more details, please check my post here:

Use animations to present the uncertainty vividly.

towardsdatascience.com

In the following, we will use `celluloid` package to create animation. To see a detailed tutorial on how to make animations with `celluloid`, check here:

A Hands-on Tutorial For Creating Matplotlib Animations

A step-by-step guide to sharpening your skills in creating and displaying animations in Jupyter Notebook.

towardsdatascience.com

The animation serves two purposes: first, it simulates the evolution of the susceptible, infected, and recovered cases under various β–γ combinations. Also, it captures the highest number of infected cases and its occurrence day in a scatter plot.

```
# Set up the graph
fig = plt.figure(figsize=(7,5))
ax_show = plt.subplot2grid((6, 1), (0, 0), rowspan=5, colspan=1)
ax_hide = plt.subplot2grid((6, 1), (5, 0), rowspan=1, colspan=1)

ax_show.set(xlim=(-5, 105), ylim=(-20,1020))
ax_show.set_xlabel('Time /days', fontsize=15)
```

8. Takeaway

We've just gone through a complete case study in conducting forward uncertainty quantification analysis. Congratulations!

Here is a summary of what we've achieved:

- Random sample generation, where we used the Latin Hypercube Sampling to generate representative samples of the uncertain input parameters;
- Uncertainty propagation, where we used the Monte Carlo simulations to propagate uncertainty from the inputs to the outputs;
- Uncertainty visualization, where we visualized output uncertainties both in static forms (histograms and scatter plots) and in dynamic forms (hypothetical outcome plots).

One drawback of using naive Monte Carlo simulations is the associated **high computational cost,** induced by running the model many times under various input samples. For our current case, the SIR model can be run fairly quickly. Unfortunately, this is generally not the case for industrial applications.

Usually, high-fidelity physics simulations are employed to make predictions, and one simulation run could easily take up to days, even weeks. For those cases, running hundreds and thousands of iterations of expensive physics simulation models would not be possible.

One way out is by training fast-to-evaluate surrogate models to approximate the physics simulations first. Subsequent Monte Carlo

learn more, check out my posts here:

An introduction to surrogate modeling, Part I: fundamentals

A machine learning approach to accelerate engineering design

towardsdatascience.com

An introduction to Surrogate modeling, Part II: case study

A machine learning approach to accelerate engineering design

towardsdatascience.com

An introduction to surrogate modeling, Part III: beyond basics

A machine learning approach to accelerate engineering design

towardsdatascience.com

About the Author

> *I'm a Ph.D. researcher working on uncertainty quantification and reliability analysis for aerospace applications. Statistics and data science form the core of my daily work. I love sharing what I've learned in the fascinating world of statistics. Check my previous posts to find out more and connect with me on Medium and Linkedin.*