# A High Performance Julia Implementation of Shapely Effects for Global Sensitivity Analysis

Devang Sehgal* and Anurag Vaidya*

**Abstract.** In this project, we present a high performance Julia implementation of Shapley effects for performing global sensitivity analysis. While many methods exist for sensitivity analysis, they assume independence between the input features of a function, which may be incorrect for many real world scenarios. We are motivated to implement Shapley effects, and make it a part of GlobalSensitivity.jl, because they can handle correlated inputs. In this report, we first describe how Shapley effects are calculated using Monte Carlo simulations. Then we show the correctness of our implementation by testing it on commonly used analytical functions and comparing the results with Sobol indices. We show the utility of Shapley effects in handling correlated inputs by considering a Jackson model of a manufacturing plant. We also demonstrate how our implementation can be used on differential equations by considering a dynamic prey-predator system. Finally, we do a thorough performance analysis of the algorithm, and optimize it to be 20× faster than the Python implementations of Shapley effects. Our Julia code is made public at https://github.com/ajv012/shapley_julia.

**1. Introduction: Global Sensitivity Analysis.** Global Sensitivity Analysis (GSA) is vital for understanding complex systems and model behavior. It measures how the output changes when inputs vary, identifying important inputs and assessing model robustness. This can allow researchers to make more informed use of models in real world scenarios and develop sparser version of models by removing non-essential inputs. Taking derivatives of the output w.r.to the inputs can give a local measure of how much the output changes for a small change in the input. Probabilistic programming provides an alternative approach to sensitivity analysis, by asking how the output of the model changes on average when the input is changed [6]. To motivate the need for GSA, consider a factory receiving multiple orders daily. The output rate of the factory depends on the rate at which six interconnected workstations in the factory work. GSA would tell the factory manager the output rate of the factory is the most sensitive to which workstations, thus helping in managing the operations and logistics. GSA has real world benefits not just in manufacturing, but in studying many systems, like how forest fires spread [7].

GlobalSensitivity.jl is the SciML implementation for GSA methods. Some of the commonly used GSA methods which are already implemented in GlobalSensitivity.jl include Derivative-based Global Sensitivity Measures (DGSM), Morris method, Sobol's method and Fourier Amplitude Sensitivity Resampling (FAST). Of these, DGSM and Morris method rely on the idea of successively linearizing to approximate the function in question, with Morris taking finite differences to approximate derivatives. Sobol's method, considered a gold standard in GSA, decomposes the variance of model into summands of variances of the input parameters in increasing dimensionality [8] by assuming that the inputs are independent. It is one of the

*Department of Health Sciences and Technology, MIT

most widely used methods. GlobalSensitivity.jl provides a simple and intuitive interface for conducting GSA on models of interest, which looks like (where $f$ is the model, method is the technique to use):

```
res = gsa(f, method, param_range; samples, batch=false)
```

Sobol indices are commonly used but assume independence among inputs, which may not hold in real-world scenarios such as physiology-based pharmacokinetic models of the human body's organs. **In contrast, Shapley effects, introduced in cooperative game theory in 1953, do not make any assumptions about input independence and can handle correlated inputs effectively.** The aim of Shapley effects is to determine the contribution of each player fairly in the total payoff achieved by the coalition of players. One notable advantage of Shapley effects is their normalization property, where the sum of effects over all individual variables equals the variance. This normalization property provides a better interpretability in determining the relative importance of variables. Unlike Sobol indices, which cannot be interpreted as percentages of variance, Shapley effects offer this additional insight. Furthermore, while comparing first-order and total order Sobol indices is challenging due to potential differences in their summation values, Shapley effects exhibit a summation property that ensures fairness and avoids assigning excessive importance to a few inputs—a known issue with Sobol indices [7]. The goal of this project is to implement a high performance version of Shapley effects in Julia and add it to GlobalSensitivty.jl. The key motivation for is to improve the accuracy of GSA in the case of correlated inputs. Adding computationally efficient version of Shapley effects, which can be used for a wide range of functions, will significantly improve the accessibility of GlobalSensitivty.jl. Our high performance implementation of Shapley effects, which is inspired by the R code and pseudo-algorithm provided by the authors of [7] and a Python implementation of Shapley effects [5], is made public at https://github.com/ajv012/shapley_julia.

The report achieves the following:

1. Section 2 describes how Shapley effects are calculated, which involves estimating the incremental cost of adding an input feature of a function to the entire feature set. We describe how Monte Carlo sampling is used to calculate the conditional variances and exemplify the process further by providing code snippets.
2. The sensitivity of the method to different hyper-parameters is analyzed thoroughly in Section 4.2.
3. In Section 4.1 the correctness of our implementation of Shapley effects is measured on common analytical functions, like the linear and Ishigami function, routinely used in sensitivity analysis. We also compare the results with Sobol indices and highlight the better interpretability provided by Shapley effects.
4. We demonstrate the real world utility of Shapley effects by analyzing a Jackson model of a factory system in Section 4.3. We show how Shapley effects can handle correlated inputs better than Sobol indices.
5. In Section 4.4, we show that in addition to analytical functions, our implementation of Shapley effects can be applied to differential equations. We use the Lotka–Volterra

equations to model a prey-predator system and show how Shapley effects can vary over time.

6. We perform a thorough performance analysis of our algorithm in Section 5. We show how our implementation of Shapley (optimized on serial and parallel code) is 20× faster than a counterpart Python implementation of Shapley effects.

## 2. Implementation of Shapley effects.

Estimating the attribution of features of a function can be thought of as finding the contributions of players in a cooperative game. Traditionally, measures like Sobol indices assume that all players act independently, however this may not be true as their might be interactions between players that need to be accounted for. Shapley effects is a variance based method that assesses the equitable allocation of a player's contribution in a cooperative game by taking into account all of the interactions a player has with others.

Before we understand how Shapley effects are calculated, we introduce some notation and definitions. Formally, consider we have a function $f$ which has $M$ inputs represented by $\boldsymbol{X_M} = \{X_1, X_2, \dots X_m\}$. An input set $\boldsymbol{X_J}$ represents a subset of inputs, i.e. $\boldsymbol{J} \subset \boldsymbol{M}$. The marginal distribution of each of the $X_i$ is denoted as $D_i$, and the joint distribution of any set of input features $\boldsymbol{X_J}$ is denoted as $C_J$. The response of the function to the input set is denoted as $Y = f(\boldsymbol{X_M})$. The uncertainty in the output $Y$ as a result of $\boldsymbol{X_M}$ is quantified by $\mathrm{Var}[Y]$ with respect to the joint distribution $C_M$. Shapley effects quantify how much of the $\mathrm{Var}[Y]$ can be attributed to each of the $X_i$.

The essence of finding the Shapley effect for a feature $i$ is to calculate what is the incremental cost of adding the feature $i$ to a subset of features $\boldsymbol{J} \subset M$, which is the then averaged over all of the possible sets $\boldsymbol{J} \subset \boldsymbol{K} \setminus i$. In order to systematically consider all the possible subsets of players, we follow these steps:

1. Consider all possible permutations of the players in the game, denoted as $\Pi(M)$. This will have $m!$ permutations.
2. For a permutation $\pi \in \Pi(M)$, define the set $\mathbf{Pi}(\pi)$ as the players that precede player $i$ in $\Pi$.
3. Define the incremental cost of including player $i$ in $\mathbf{Pi}(\pi)$ as $c(\mathrm{Pi}(\Pi) \cup \{i\}) - c(\mathrm{Pi}(\Pi))$.

Now, Shapley effect for a player $i$ ($S_i$) can be defined as:

$$S_i = \sum_{\pi \in \Pi(M)} \frac{1}{m!} c(\mathrm{Pi}(\Pi) \cup \{i\}) - c(\mathrm{Pi}(\Pi)) \tag{2.1}$$

In line with previous literature, we define our cost function for any subset $\boldsymbol{J} \subset \boldsymbol{M}$ as:

$$c(J) = \mathrm{Var}[Y] - \mathrm{E}[\mathrm{Var}[Y|\boldsymbol{X_J}]] \tag{2.2}$$

The cost function in Equation 2.2 can be understood as the expected reduction in $\mathrm{Var}[Y]$ when the values of $\boldsymbol{X_J}$ are fixed. In other words, how much variance is remaining in $Y$ when the values of $\boldsymbol{X_J}$ are known. Because of this cost function, we need to evaluate $2^m - 1$

variance components for $m!$ permutations of the input features, thus the big-O for the Shapley algorithm is $m!$, where $m$ is the number of input features. This version of the Shapley effects algorithm is termed the "exact permutation" version.

The exact permutation algorithm quickly becomes intractable for large number of input features, both in terms of time complexity and the memory allocation requirements. To make the Shapley effects algorithm tractable for functions requiring a large number of input features, we implement the "random permutation" version of the algorithm, introduced by Castro et. al. [1] in 2009. In this algorithm, the essential idea stays the same, but instead of considering all of the possible $m!$ permutations, we only consider a random subset of all permutations. It has been shown that such an approximation of the Shapley effects converges to the actual values in probability [1].

**2.1. Pseudo-code for calculating Shapley effects.** To calculate the Shapley effect for a feature, we need to find the incremental cost of adding the feature to the feature set of a function, and we define the cost in Equation 2.2. In this section, we describe how our implementation of Shapley effects can be broken into three main steps. First, we use Monte Carlo sampling to generate an input sample, $X$. More specifically, we define distributions by incrementally adding input features and sample these to encode the interactions between different features. The input sample $X$ is then passed through the function of interest $f$; then using the definition of cost function and bootstrap sampling, we calculate the Shapley effects for each feature. We now cover the steps that we follow to generate our input sample $X$ and provide code snippets for further clarity. To understand how we calculate Shapley effects, please take a look at our publicly available implementation at https://github.com/ajv012/shapley_julia.

In order to define our input sample $X$—which encodes the interactions between different features—we consider all the possible permutations of the features and all of the subsets of each permutation. Before, we get into the steps we follow to generate $X$, we outline the hyper-parameters of our algorithm:

1. $N_V$: number of samples used to calculate the variance of the output $Y$.
2. $N_O$: number of samples taken to estimate the conditional variance of the output $Y$ conditioned on a subset of features $\boldsymbol{X_J}$.
3. $N_I$: size of each of the $N_O$ samples taken.
4. $n_{perms}$: number of permutations of the input features considered. If all permutation of $m$ features considered, then we have the exact permutation method. In case of randomly sampling permutations, we have the random permutation method.
5. $n_{boot}$: number of bootstrapped samples used to estimate the cost function.

To make it easier to follow the steps, consider the following example. Say, we have 5 features, $[1, 2, 3, 4, 5]$, and $m = 5$. The steps we follow to define $X$ are:

1. Sample $N_V$ from the joint distribution of all features $\boldsymbol{C_5}$. Store this as sample A.
2. Consider a permutation of the features $[1, 3, 5, 4, 2]$, call it $\pi$.
3. Within $\pi$, select the first feature [1] (call its distribution $X^+$), and the remaining features $[3, 5, 4, 2]$ (call their joint distribution $X^-$).

4. Generate $N_O$ samples of size $N_I$ from the distribution of $X^+$ conditioned on $X^-$. Call this set of samples sample B.
5. Repeat steps (2) and (3) by incrementally adding the next feature in $\pi$ to $X^+$ and keeping the remaining features in $X^-$. Keep concatenating the generated samples to sample B.
6. Repeat steps (2)-(5) for all possible permutations of the features (exact permutation) or a given number of permutation (random permutation).
7. Sample A is used to estimate the variance of the output, whereas sample B is used to estimate the mean conditional variance by $n_{boot}$ bootstrapped samples.

Now we present a code snippet from our implementation that generates sample A and B to encode the correlations between the different features. This is the core of the Shapley algorithm. First, from the method object we extract all of the necessary hyper-parameters. Then, we decide if we are iterating over all permutations of features (exact) or sampling from all permutations (random). We sample $N_V$ from the joint distribution of the marginals. We then have three loops:

- First loop iterates over all permutations (step 2 from above),
- Second loop samples $N_O$ from $X^-$,
- Third loop samples $N_O$ samples of size $N_I$ from $X^+$ conditioned on $X^-$ (step 4).

For rest of the algorithm, i.e., how we compute the effects from the input sample, please refer to our public implementation at https://github.com/ajv012/shapley_julia.

```
if (n_perms==-1)
    estimation_method = "exact";
    perms = collect(permutations(range(1,dim), dim));
    n_perms = length(perms);
else
    estimation_method = "random";
    perms = [randperm(dim) for i in range(1, n_perms)]
end

# Creation of the design matrix
sample_A = copy(transpose(rand(input_distribution, N_V)));
sample_B = zeros((n_perms * (dim - 1) * N_O * N_I, dim));

#---> First loop to go over the permutations
for (i_p, perm) in collect(enumerate(perms))
  idx_perm_sorted = sortperm(perm) # Sort the variable ids
  for j in 1:(dim-1)
    # normal set
    idx_plus = perm[1:j];
    # Complementary set
    idx_minus = perm[j+1:end];
```

```
    sample_complement = sample_subset(input_distribution, N_O, idx_minus);
    for l in range(1,size(sample_complement)[1])
      curr_sample = sample_complement[l, :];
      xj = cond_sampling(input_distribution, N_I, idx_plus, ...
           idx_minus, curr_sample);
      xx = reduce(hcat, (xj, repeat(transpose(curr_sample), N_I)));
      ind_inner = (i_p - 1) * (dim - 1) * N_O * N_I + (j-1) * ...
           N_O * N_I + (l-1) * n_inner;
      ind_inner += 1;
      sample_B[ind_inner:ind_inner + N_I - 1, :] = ...
           @view xx[:, idx_perm_sorted];
    end
  end
end
```

**2.2. Salient features of our implementation of Shapley effects.** Our algorithm for calculating Shapley effects has four salient features. First, unlike the R implementation provided by [7], our implementation decouples the sample generation (where different permutations and their subsets are considered to generate conditional samples) and the Shapley effect calculation stages. In the original implementation, since these stages are not decoupled, they cannot be modulated independently. For example, if one wants tighter confidence interval bounds on their Shapley effects, they would need to increase the $n_{boot}$ variable, but this would cause the sample generated to be extremely large as well. This is not necessary as only more samples are needed to make the bounds tighter. This behavior is not seen in our implementation, where tighter bounds on the Shapley effects can be achieved without increasing the computational requirements for the sample generation phase. The second benefit of decoupling these steps is that different high performance techniques can be applied as per the needs of each stage. For example, since the external library https://github.com/lrnv/Copulas.jl is heavily used in generating the samples, we cannot parallelize it and and can only make improvements to the serial code. In the implementation of [7], this would mean that the Monte Carlo simulations cannot be parallelized. However, with our implementation, we can parallelize the simulations and separately optimize the sample generation code. This is discussed further in section 5. The second salient feature of our implementation of Shapley effects is that it can be applied to both functions with known analytical forms (e.g., linear, Ishigami [2], etc.), as well as a system of differential equations (e.g., Lotka–Volterra equations for a prey-predator system [4]). We analyze both of these systems thoroughly in our experiments in section 3. Third, our implementation allows the users to precisely control the input marginal distributions for all of the features and define a copula to encode the interactions and correlations between the different features. Finally, we implement the exact and random permutation versions of the Shapley effects algorithm so that it is tractable to compute Shapley effects for functions with a larger number of input variables.

**3. Experiments.** To comprehensively test our implementation of Shapley effects, we perform a series of correctness and exploratory experiments and compare the results with Sobol indices. We also show the utility of Shapley effects. In order to investigate the effect of different experimental parameters on estimated Shapley effects—namely $N_V$, $N_O$, and $N_I$—we use the same systems as in the correctness case.

**3.1. Correctness experiments.** To determine the correctness of our implementation of Shapley effects, we implement commonly used test cases for GSA for which we know what the expected relative attributions as given by theory. First, we test on the simple linear case (Equation 3.1), with $A = 7$ and $b = 0.1$, with the assumption that $x$ has a uniform marginal distribution in the interval $-\pi$ to $\pi$. It is expected that feature $x$ would account for 100% of the variance in the output, so the Shapley effect for $x$ should be close to 1.

$$y = Ax + b \tag{3.1}$$

Next, we compare the Shapley effects implementation on the Ishigami function [2]. The Ishigami function (3.2) is a recurrent test case for sensitivity analysis methods and uncertainty. it is non-linear, non-monotonic, and displays strong inter-dependence between its features, specifically the first and third as seen in the last term of Equation 3.2. For our tests, we take all $x_i$ to be uniformly distributed on the interval $-\pi$ to $\pi$, and take $a = 7$ and $b = 0.1$. It is expected that most attribution be given to $x_1$ and $x_2$, and little attribution goes to $x_3$ because of its dependence on $x_1$.

$$f(x_1, x_2, x_3) = \sin(x_1) + a\sin^2(x_2) + bx_3^4 \sin(x_1) \tag{3.2}$$

To test the correctness in both linear and Ishigami function, we compare the output of our Shapley algorithm with that Sobol first and total order indices. We do not expect the exact attribution values to match due to their different meanings, but expect similar trends in relative importance given to different features. Additionally, in both the linear and Ishigami function tests, we also provide the function with an extra feature, which remains unused. The purpose of this test is to ensure that zero attribution is given to this unused feature.

**3.2. Sensitivity to Hyper-parameters.** To determine the effect of hyper-parameters required for Shapley effects ($N_V$, $N_O$, and $N_I$), we compute the Shapley effects for the Ishigami function with various combinations of these hyper-parameters and also keep track of the run time and memory allocations. Trends in these measures will help in determining good trade-offs between speed, memory, and accuracy of calculated Shapley effects.

**3.3. Case study 1: Manufacturing system model.** The first example we consider is a make-to-order manufacturing system, where we model manufacturing of multiple product types using a Jackson network [3]. Queueing network models are extensively employed in industrial engineering and operations research to optimize manufacturing and service systems. Our objective in conducting sensitivity analysis is to identify the specific product type that has the greatest impact on fluctuations in the expected order completion time for all jobs. This valuable information can assist companies in effectively managing system tension and reducing overall fluctuations. To achieve this, we estimate the Shapley effects and compare them with the first-order and total effects, providing a comprehensive evaluation of their respective contributions within this manufacturing system.

Consider the network depicted in Figure 1, which represents a manufacturing line with six workstations labeled A-F, each handling a different job. Throughout a month, the daily arrival rates of six jobs, namely $X_1 \ldots X_6$, remain consistent. The model's output is the monthly expected completion time for the manufacturing line ($\eta$), with the $\boldsymbol{X}$ representing the input containing the six arrival rates. Since fluctuations in this anticipated time can result in costs for the company, we use Shapley effects to identify the job types that have the most significant impact on the variation in expected completion time.

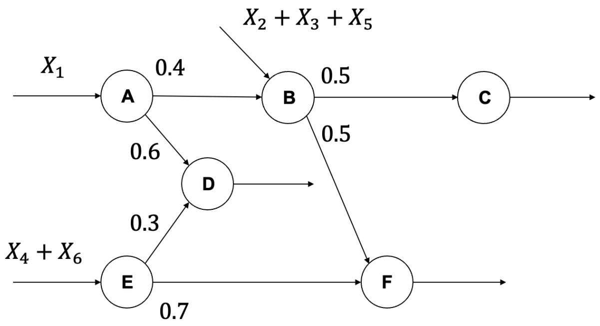


**Figure 1.** *Jackson model of manufacturing line with size workstations (A-F) and six incoming orders ($X_{1-6}$).*

Different jobs arrive at different workstations. Job 1 arrives at workstation A, jobs $2, 3, 5$ arrive at workstation B, and jobs $4, 6$ arrive at workstation E. After each of the job is processed, they are routed to the next workstation with passing probabilities shown on the arrows in Figure 1. Job arrival and processing times are assumed to be independent. The processing rates at each workstation are fixed at: $\mu_A = 1.2, \mu_B = 1.5, \mu_C = 4, \mu_D = 1.8, \mu_E = 3.6, \mu_F = 1.5$ (per day) [7]. Since calculation of Shapley effects requires specification of marginal distribution of each input, we used Distributions.jl to model each arrival rate with a beta distribution, $\mathrm{B}(\alpha = 1.5, \beta = 2.0, \min = 0.5, \max = 0.8)$.

The Jackson model calculates the expected job completion time $\eta$ given the arrival rates vector $\boldsymbol{X}$, as shown in Equation 3.3. First, the daily arrival rates at each workstation $v_{A-F}$ are calculated, which are then combined to get the job completion time of the network $\eta$.

(3.3)
$$\begin{aligned}
v_A &= X_1 \\
v_B &= 0.4X_1 + X_2 + X_3 + X_5 \\
v_C &= 0.3X_1 + 0.15X_4 + 0.15X_6 \\
v_D &= 0.6X_1 + 0.3X_4 + 0.3X_6 \\
v_E &= X_4 + X_6 \\
v_F &= 0.85X_4 + 0.85X_6 + 0.3X_1
\end{aligned}$$

$$\eta(X_1, \ldots X_6) = \{\sum_{i=A}^{F} \frac{v_j}{\mu_j - v_j}\} \times (\frac{24}{\sum_{i=1}^{6} X_i})$$

#### 3.3.1. Numerical experiment: How do correlated inputs affect feature attributions?.

We first calculate the Shapley effects of the six order arrival rates assuming that all orders arrive independently. In line with [7], we use the parameters $N_V = 2,000, N_O = 100, N_I = 2,$ and $n_boot = 1,000$, $n_{perms} = 6!$. In order to make a fair comparison between Sobol indices and Shapley effects, we also implement Sobol first and total order indices so that the same sampling method used to calculate Shapley effects can be used to find Sobol indices.

However, in real world manufacturing and operations research, it is highly unlikely that all order rates are independent of each other. Let's consider a scenario where products of types 1 and 2 exhibit a complementary nature, resulting in a positive correlation between their respective demands. Conversely, products of types 3 and 4 act as substitutes for one another, leading to a negative correlation in their demands. We now incrementally test the effect of adding in correlated inputs on the calculated Shapley effects and Sobol indices. First, we consider "small correlation", where $\mathrm{Corr}(X_1, X_2) = 0.25$ and $\mathrm{Corr}(X_3, X_4) = -0.25$. Next, we consider "large correlation" where $\mathrm{Corr}(X_1, X_2) = 0.75$ and $\mathrm{Corr}(X_3, X_4) = -0.50$. We compare the Shapley effects and Sobol indices for both the scenarios.

**3.3.2. Numerical experiment: Are all permutations of input features required?.** One of the limitations of the naive implementation of Shapley effects is that the number of all possible permutations of the features, $n_{perms}$, has factorial time complexity. However, all permutations of features may not be necessary, especially in the scenario with highly correlated inputs. To make the calculation of Shapley effects tractable for large number of input features, we sample from all possible permutations. To validate that sampling permutations leads to approximately similar results, we calculate and compare Shapley effects on the Jackson manufacturing model with different number of permutations ($6! = 720, 700, 600$, and $200$).

**3.4. Case study 2: Prey-predatory system of differential equations.** While the first case study worked with an analytical solution to the Jackson model, in this case study we explore how Shapley effects can be computed for a system of differential equations. We consider the Lotka-Volterra equations, which are a pair of first-order nonlinear differential equations and describe population densities of two inter-connected species, like a prey-predator pair. The populations change through time according to the pair of equations:

$$\frac{dx}{dt} = \alpha x - \beta xy$$

$$\frac{dy}{dt} = \delta xy - \gamma y$$

Where $x$ and $y$ are the population densities of the prey and the predator, respectively, $\frac{dx}{dt}$ and $\frac{dy}{dt}$ model the instantaneous growth rates of populations, $\alpha$ is the maximum per capita growth rate of the prey, and $\beta$ represents the presence of predators on the growth rate. Similarly, $\delta$ and $\gamma$ represent the predator's growth rate and presence of prey, respectively.

The four parameters are given the following initial values, $\alpha_0 = 1.5, \beta_0 = 1.0, \delta_0 = 3.0, \gamma_0 = 1.0$. For the sampling of these parameters, we assume uniform distributions over $[1, 5]$. We solve the problem using the time span of 10 months, and calculate Shapley effects and Sobol indices at five time points (2, 4, 6, 8, and 10 months) using `TsiT5` solver in Julia. Since the goal of this case study is to show that Shapley effects can be applied to a system of equations, we assume all parameters are independent. The hyper-parameters for this study are: $n_{perms} = 4! = 24, N_V = 1000, N_O = 100, N_I = 3, n_{boot} = 1000$.

## 4. Results.

**4.1. Correctness.** To test the correctness of our implementation of Shapley effects, we test it on two commonly used functions in sensitivity analysis—linear and Ishigami functions—and compare the attribution given to the input features with Sobol indices. For the linear system (Figure 2A and B), we see that 99.95% of variance is accounted for by the first feature ($A$ in Equation 3.1) and no attribution is given to the second feature ($B$ in Equation 3.1). This is reasonable because the inputs affect the function output via the feature $A$, and feature $B$ does not account for any of the variance. Similarly, the first order and total order Sobol indices give highest attribution to feature 1. In the Ishigami function, we would expect low attribution for the third feature because it is correlated with the first feature. Shapley effects give 9.8% attribution to the third feature, whereas the remaining attribution is largely given to the first two features (Figure 2C). When interpreting Sobol indices for this example, we see that first order and total order indices demonstrate different stories (Figure 2D). First order indices say that feature 2 is most significant, but total order indices indicate that the first feature is the most significant. Moreover, first order and total order indices cannot be compared against each, so one cannot argue that feature one is most important because total order index assigned to it is the largest out of all Sobol indices (Figure 2D). On the other hand, Shapley effects for different features are comparable because they are percentages of the total variance, hence one can argue that features one and two are more sensitive than feature 3 because they account for a larger percentage of the variation. This highlights the limited interpretation of Sobol indices. Finally, the unused feature, namely the fourth feature, is given the lowest attribution, which is reasonable.

**4.2. Sensitivity to hyper-parameters.** The Shapley effects algorithm has four hyper-parameters $N_V, N_O, N_I, n_{boot}$. We analyze the effect of these hyper-parameters on the accuracy, memory allocations, and computation time of Shapley effects, using the Ishigami function as a test case (Table 1). Since $N_V$ controls the size of the sample used to calculate the output variance, it is important not to make that variable too small, even though larger $N_V$ would directly affect the allocation and computation time. However, between 1000 and 10000, we do not see any significant gains but higher increase in costs (4$\times$ increase in memory allocation and 3$\times$ increase in computation time). It is also important to balance $N_O$ and $N_I$ because the former denotes the number of samples for calculating the conditional variance and the latter controls the size of the sample. If many small samples are taken ($N_I$=2), we can get non-senseical results like negative Shapley effects. Taking very large samples can can cause 7$\times$ increase in computation time. Thus, these two variables need to be balanced. Finally, $n_{boot}$ determines the number of bootstrapped samples of size $N_V$ taken; its effect is directly seen on the size of the confidence intervals. We find that $60,000$ samples leads to tight confidence intervals, and beyond this, there are more costs than gains. Our selected parameters shown in the last line of Table 1 balance the accuracy, speed, and memory allocation for the Ishigami function. One must note that these values are problem-specific and we encourage users to perform such an analysis to find the best set of hyper-parameters for their problem at hand.

**4.3. Case study 1: Manufacturing model.** Regardless of the amount of correlation between the features, Shapley effects find that in the Jackson model for the manufacturing line is the most sensitive to feature 2, and this features accounts for large portions of the output

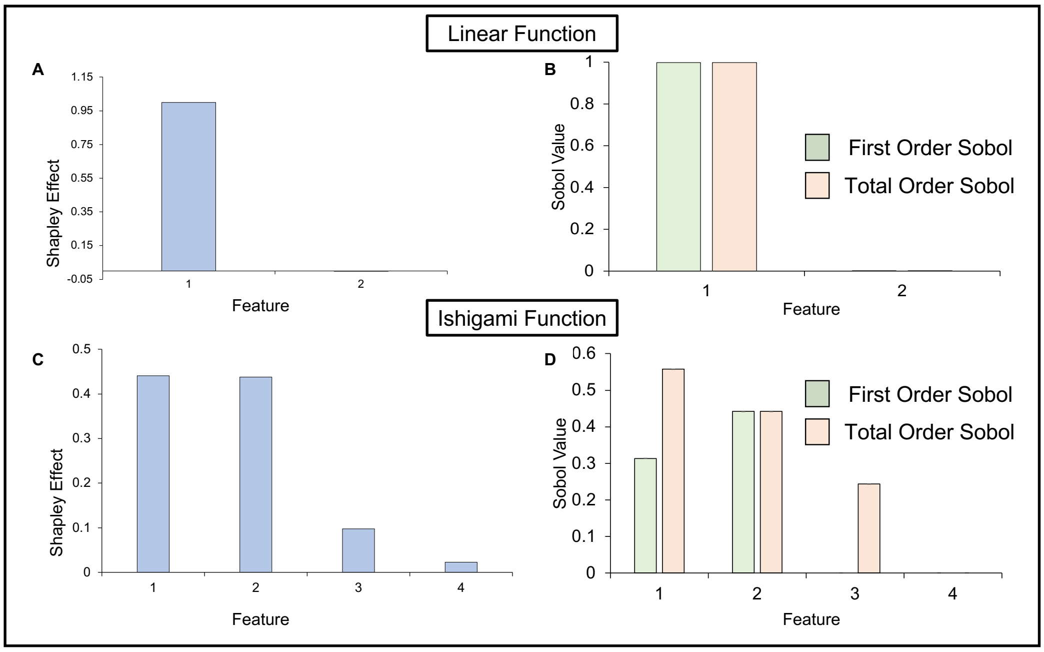


**Figure 2.** *For the linear system described in Equation 3.1 (A) Shapley effects (B) First order and total order Sobol indices. For the Ishigami function described in Equation 3.2 (C) Shapley effects (D) First order and total order Sobol indices. Error bars show 95% confidence interval on $n_{boot}$ samples.*

variance (Figure 3). We see that as we increase the correlation between the first two features, the attributions given to them by the Shapley algorithm also scales accordingly, i.e., importance for feature 2 is ”routed” to feature 1, which is what we would expect for highly correlated inputs. On the other hand, for independent and small correlation case, we see that the Sobol indices also assign highest attributions to the first two features. However, we once again see that Sobol first order and total order indices are contradicting each other, and since they are not directly comparable, it is difficult to determine which feature is the most important. Moreover, we see that as we increase the correlation between the features, Sobol indices conclude that the first two features are no longer as important as the others, which is in contrast to the Shapley effects. This helps us validate that when features are correlated, Shapley effects are a better choice for sensitivity analysis because of fewer discrepancies.

Additionally, we use the factory system to determine if all of the permutations of the features are required to accurately calculate the Shapley effects. This is an important question because the Shapley algorithm has factorial complexity with the number of permutations, i.e., the $n_{perms}$ = (number of features)!. So, for the factory system, the total number of permutations to consider are $6! = 720$. For different levels of correlations between inputs, we find that sampling 80% of total permutations is sufficient to get similar Shapley effects (Figure 4). Similar attributions are given to the features as low as 200 permutations (28% of total permutations). Interestingly, we find that the sixth feature gets increased attribution at lower number of permutations (Figure 4) for both the no correlation and high correlation cases.

| $N_V$ | $N_O$ | $N_I$ | $n_{boot}$ | Feature 1 | Feature 2 | Feature 3 | Allocation (GB) | Time (ms) |
|---|---|---|---|---|---|---|---|---|
| **10** | 100 | 3 | 60000 | 0.316 (0.302, 0.329) | 0.497 (0.479, 0.515) | 0.125 (0.116, 0.135) | 1.77 | 479.86 |
| **100** | 100 | 3 | 60000 | 0.484 (0.483, 0.485) | 0.414 (0.413, 0.415) | 0.033 (0.033, 0.034) | 1.86 | 493.34 |
| **1000** | 100 | 3 | 60000 | 0.396 (0.395, 0.396) | 0.383 (0.382, 0.383) | 0.153 (0.153, 0.153) | 2.67 | 631.20 |
| **10000** | 100 | 3 | 60000 | 0.380 (0.379, 0.381) | 0.414 (0.413, 0.414) | 0.140 (0.139, 0.140) | 10.71 | 1772.0 |
| 1000 | **1** | 3 | 60000 | 0.707 (0.706, 0.708) | 0.501 (0.500, 0.501) | -0.253 (-0.254, -0.253) | 1.52 | 333.06 |
| 1000 | **10** | 3 | 60000 | 0.124 (0.123, 0.124) | 0.716 (0.714, 0.719) | 0.067 (0.065, 0.069) | 1.63 | 351.6 |
| 1000 | **100** | 3 | 60000 | 0.319 (0.318, 0.319) | 0.418 (0.417, 0.418) | 0.167 (0.167, 0.169) | 2.67 | 579.71 |
| 1000 | **1000** | 3 | 60000 | 0.431 (0.430, 0.432) | 0.387 (0.387, 0.387) | 0.104 (0.104, 0.104) | 12.97 | 2917.0 |
| 1000 | 100 | **2** | 60000 | 0.459 (0.458, 0.460) | 0.402 (0.402, 0.403) | 0.075 (0.074, 0.075) | 2.66 | 602.27 |
| 1000 | 100 | **5** | 60000 | 0.382 (0.381, 0.382) | 0.394 (0.393, 0.394) | 0.157 (0.157, 0.157) | 2.67 | 714.80 |
| 1000 | 100 | **10** | 60000 | 0.459 (0.458, 0.460) | 0.395 (0.394, 0.395) | 0.083 (0.082, 0.83) | 2.67 | 908.03 |
| 1000 | 100 | **100** | 60000 | 0.396 (0.395, 0.396) | 0.448 (0.448, 0.449) | 0.096 (0.096, 0.097) | 2.69 | 4684.0 |
| 1000 | 100 | 3 | **100** | 0.388 (0.375, 0.401) | 0.413 (0.399, 0.426) | 0.123 (0.115, 0.132) | 0.03 | 7.19 |
| 1000 | 100 | 3 | **1000** | 0.437 (0.432, 0.442) | 0.412 (0.408, 0.416) | 0.078 (0.075, 0.081) | 0.07 | 16.10 |
| 1000 | 100 | 3 | **10000** | 0.436 (0.434, 0.437) | 0.379 (0.378, 0.381) | 0.107 (0.106, 0.108) | 0.48 | 111.98 |
| 1000 | 100 | 3 | **100000** | 0.392 (0.392, 0.393) | 0.440 (0.439, 0.440) | 0.079 (0.079, 0.080) | 4.42 | 1029.0 |
| **1000** | **100** | **3** | **60000** | **0.398 (0.397, 0.398)** | **0.386 (0.385, 0.386)** | **0.147 (0.146, 0.147)** | **2.67** | **605.61** |

**Table 1**

*The effect of different hyper-parameters $N_V, N_O, N_I, n_{boot}$ on the output of the Shapley effects algorithm on the Ishigami function. The values in bold indicate the hyperparameter that is being changed for that set of experiments. The final row shows the set of hyper-parameters used in making Figure 2C. The Shapley effects for the different features are reported with their 95% Confidence Interval in brackets.*

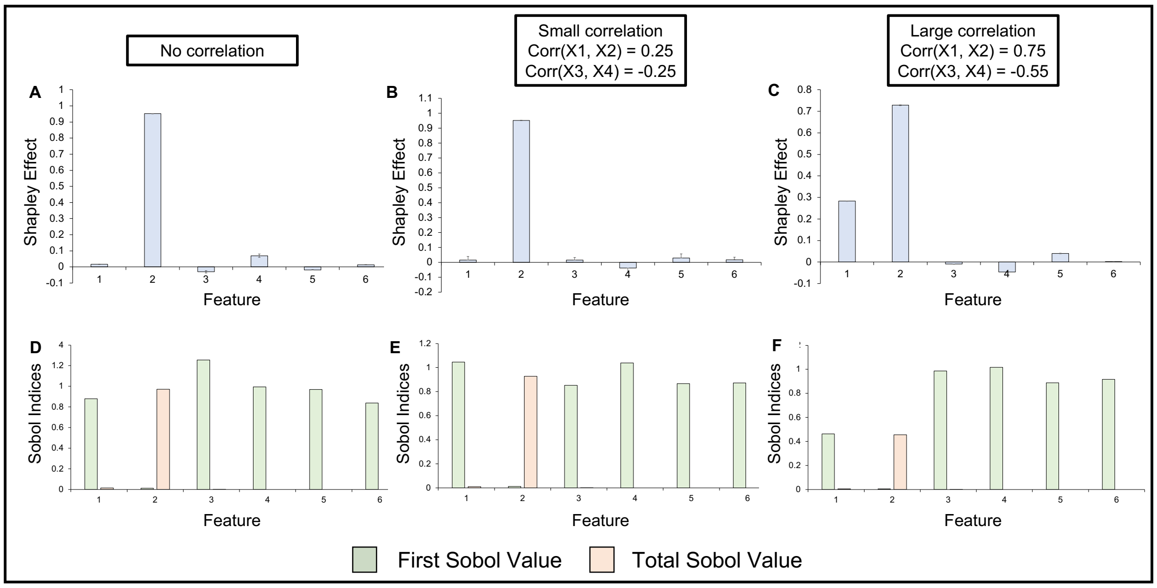


**Figure 3.** *Shapley effects and Sobol indices for the manufacturing line Jackson model explained in Equation 3.3. (A), (B), (C) show Shapley effects for increasing correlations between inputs. (D), (E), (F) show the first order and total order Sobol indices for increasing correlations between inputs. Error bars show 95% confidence interval on $n_{boot}$ samples.*

**4.4. Case study 2: Prey-predatory system.** The previous case study had an analytical formula that could be analyzed. In this case study, we show that our implementation of Shapley effects can also be applied to a system of differential equations and that the attribution to different features can be analyzed as a function of time. In Figure 5, we see that the different

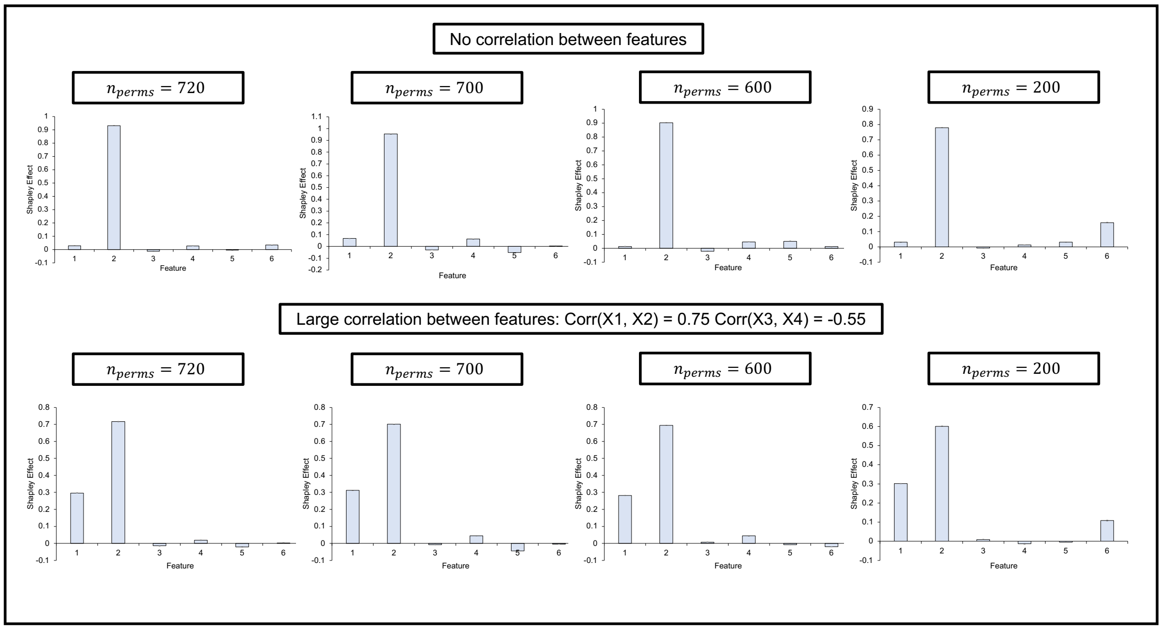


**Figure 4.** *Shapley effects for the manufacturing line Jackson model with different number of randomly selected permutations. We repeat the experiment for when there is no correlation between input features (top row) as well as when there is high correlation between features (bottom row). Error bars show 95% confidence interval on $n_{boot}$ samples.*

parameters of the prey-predator system have different attributions at different timepoints. Here, the interpretability of Shapley effects is again exemplified. First, for the prey, we see that the prey system is most sensitive to the $\delta$ parameter (the effect of the presence of prey on the predator's growth rate), whereas the predator system is generally most sensitive to the $\beta$ parameter (the effect of the presence of predators on the prey growth rate). These are reasonable because the greater growth rate of prey would affect the predatory system (and vice-versa).

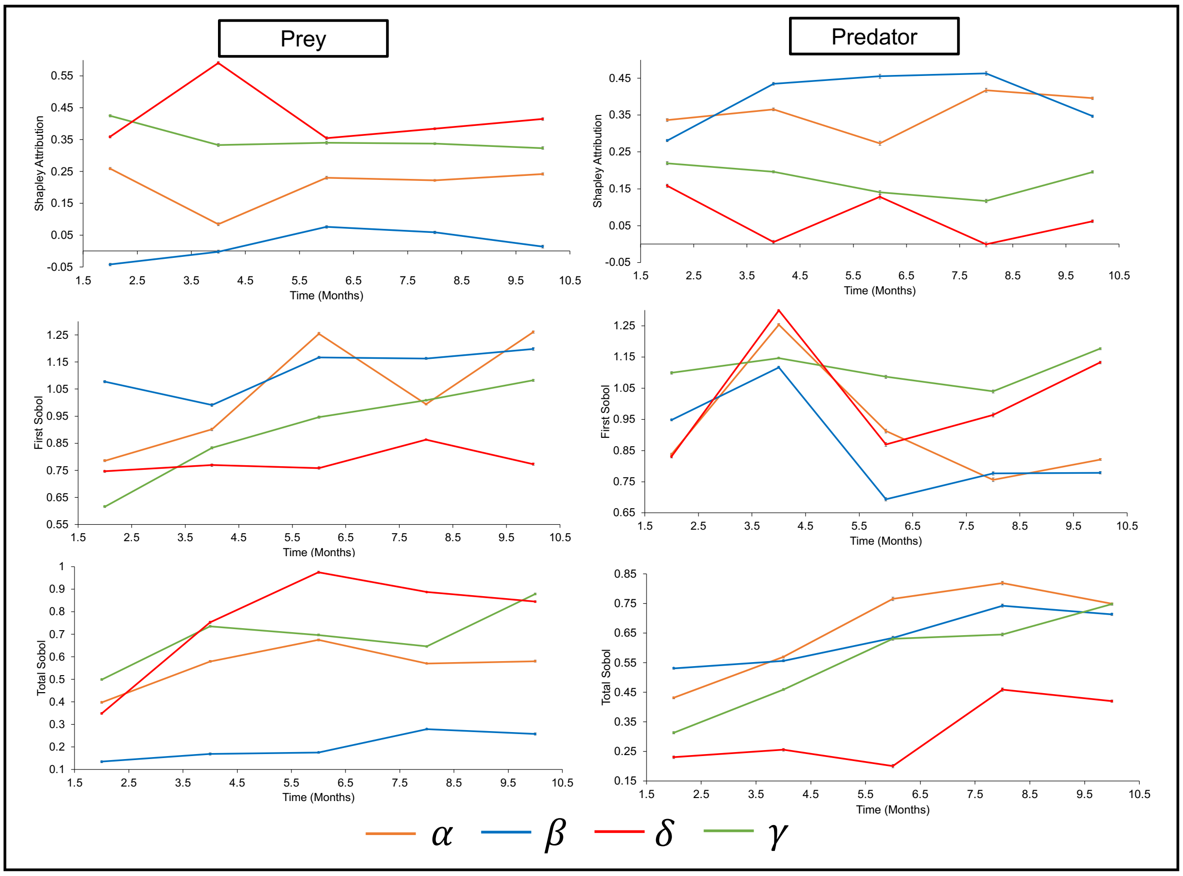


**Figure 5.** *Shapley effects and Sobol indices (first and total order) for the prey-predatory system at different time points.*

## 5. Performance Engineering.

**5.1. Efforts made in performance improvements.** We started with a serial, naive implementation of the program based on the numpy implementation in Shapley effects [5]. To improve performance of our code, we followed the principle of optimising serial performance first, then looking at parallelism. Our efforts in this can be categorised under three main steps:

**5.1.1. Type Stability.** We first went over our code to ensure all functions were type-stable, i.e. the compiler knew what type of output and input to expect for every function in the program. This includes both the core computation GSA function, as well as helper functions we wrote for sampling. This required going through all the function calls and anticipating what the type could be, and mentioning that in the function declaration. This did not yield any noticeable performance improvements over baseline on our test example (Ishigami function). This is not surprising since our baseline code already had some type-stability incorporated and the fact that the program has significant computation complexity and requires a large number of allocations which dwarf the effect of type-stability. Nevertheless, our final implementation of the code is type-stable.

**5.1.2. Memory Management (Allocations).** Memory management is extremely important as memory requirements can cross 10 GB for even a simple test function like Ishigami on moderate values of the hyperparamters (see Table 1). Our first effort was in pre-allocating large arrays before computation to reduce number of allocation calls inside the loops. This involved use of calls such as

```
shapley_indices = zeros(dim, n_boot, 1);
```

or `similar` for pre-allocation and then updating elements in loop. This was part of our baseline implementation.

The next step was reducing the number of redundant copying of arrays in loops. This involved using `@view` where accessing elements of a matrix was required without modification. In the same vein, it was important to ensure operations where done in-place where possible with use of Julia's dot operator (namely `.=` ). A very valuable resource for understanding the major sources of allocations and finding whether our if our operators were indeed allocating memory as we expected was the use of Julia's track allocations command line feature , namely `--track-allocation=user` flag. This generated a `.mem` file listing the number of allocations per line of code, which was a big help in identifying the key bottlenecks. This flag, along with BenchmarkTools such as `btime` helped us see that most allocations were occurring inside the bootstrap loop (apart from our expected array allocations). Taking a closer look at the boostrap loop, two lines are of particular importance:

```
idx_for_var = rand(1:n_var, n_var);
idx_for_cond_var = rand(1:n_outer, n_outer);
```

These lines are used for 'creating' a sample for cost computation by selecting elements from our generated samples. They involve creating an $N_V$ and $N_O$ element random vector at each iteration of this loop, i.e. resulting in Order $(n_{boot} * N_V + n_{boot} * N_O)$ allocations, a very significant cost considering $n_{boot} = 60,000$ and $N_V = 1000$ are typical and have been used in this report. The allocations can be brought down by making these allocations in-place, i.e. allocating the array once and storing new random elements to it each loop instead of allocating more space. This would look as follows:

```
rand!(idx_for_var, 1:n_var );
rand!(idx_for_cond_var, 1:n_outer);
```

However, we could not simply make all function calls in-place as we wanted to parallelize this loop across multiple threads. There was a trade-off to consider between reducing allocations and reducing computation time by multi-threading. To better understand its effect on performance, we developed both an optimised serial code that reduced allocations as much as possible (by using `rand!()` above), and an optimised parallel code designed to make use of the threading but at the cost of increased allocations.

**5.1.3. Multi Threading/ Parallelism.** Use of reductions: Our implementation relies on generating samples, calculation of the cost function based on the samples and finally calculating the Shapley values. All these steps require construction, storage, and manipulation of large arrays. For efficient construction and concatenation we make use of the `reduce` function in Julia with `hcat` and `repeat` to manipulate our arrays in desired shape. An example is seen as follows:

```
xx = reduce(hcat, (xj, repeat(transpose(curr_sample), n_inner)));
```

Multi threading: Threading is the most obvious use of parallelism in the code, and we also expect it provide the most significant benefit in performance. The most straightforward application is using the `Threads.@threads` call on for loops. We tackle this in two parts, representing the two key segments of our code:

A. Sample generation: Sample generation involves listing the permutations of the dimensions, then generating samples based on conditional distributions and storing them in the input sample array. The sample generation segment has multiple nested loops with function calls to our sampling/distribution helper functions (one of which itself contains loops), so it represents an opportunity for speedup. However, use of threads do not actually improve the performance of our code - at best it simply reproduces the performance of serial code, or only slightly worsens it, when the outermost loop is threaded. Threading the innermost loop significantly worsens the code performance by as much as a factor of 2.

We believe this is down to two main reasons. Firstly, we know generating threads has a significant overhead involved, and memory sharing across threads can lead to slowdowns. This is especially true when the same data structure is being accessed by multiple threads at once, which is true in this case - `sample_B` is being used by all threads for storing results while `input_distribution`, a sklar distribution type, is being passed to functions by all threads. Secondly, the function calls and data types used in this section includes an external library called Copulas. Since we do not have complete control over all aspects in these loops, we are unable to control it's behaviour and modify it to be appropriate for multi-threading (discussed previously in Section 2.2). Therefore, we keep the sample generation part of the code as a serial implementation but optimise it as well as possible in terms of memory allocations.

B. Shapley indices calculations: This segment of the code has two sets of loops. The first is a single for loop over the $n_{boot}$ for computing the cost function, while the second is two nested for loops over $n_{boot}$ and number of permutations that computes and stores the actual Shapley indices. For this part, we use threading across both the outer loops. This gives us a significant benefit and helps optimize the performance of the code. However this benefit must be traded-off against a cost in allocations, discussed below in performance analysis.

Other libraries and tools beyond `Threads.@threads` were tried as well (such as LoopVectorisation.jl) but no noticeable benefit over the `Threads.@threads` was seen. An additional benefit of using `Threads` in this manner was its versatility whereas several other tools had more specific requirements from underlying code.

**5.2. Benchmarks and Performance Analysis.** For analysing the performance of our program, we measure it's performance using the 3 dimensional Ishigami function with $N_V = 1000$, $N_O = 100$, $N_I = 3$, $n_{boot} = 60,000$. We benchmark: (i) the python Shapley effects implementation [5], which served as the inspiration for our code, (ii) our baseline Julia implementation in the structure of the GSA repo, (iii) our optimised serial implementation for the Julia code, and (iv) the optimised parallel implementation for the Julia code. The runtime and memory allocations for the implementations is shown in Table 2. Note that the allocations for python code are not listed as no direct analogue to `btime` or Julia's BenchmarkTools could be found and the python functions for similar functionality may have differences. All benchmarks reported are for a 2020 M1 processor Macbook Air, with 4 threads in case of multi-threading.

| **Implementation** | $N_V$ | $N_O$ | $N_I$ | $n_{boot}$ | $dim$ | **Runtime (s)** | **Allocations (GB)** |
|---|---|---|---|---|---|---|---|
| python-shapley [5] | 1000 | 100 | 3 | 60000 | 3 | 12.009 | n/a |
| julia-baseline | 1000 | 100 | 3 | 60000 | 3 | 1.917 | 4.12 |
| julia-optim. serial | 1000 | 100 | 3 | 60000 | 3 | 1.372 | 1.99 |
| julia-optim. parallel | 1000 | 100 | 3 | 60000 | 3 | 0.582 | 2.67 |

**Table 2**

*Performance benchmarks, namely runtime and allocations, for 4 implementations of Shapley effects for GSA: 'python-shapley' refers to the implementation of Shapley effects in python [5], 'julia-basline' is our implementation of Shapley-effects in the GSA scaffold, 'julia-optim. serial' refers to our optimised serial implementation while 'julia-optim. parallel' is our optimised parallel implementation, running over 4 threads.*

We see that the baseline Julia implementation is itself 6.3× faster than the Shapley effects implementation in python, underscoring the benefits of developing high-performance code in Julia. Our optimised serial implementation is about 1.4× faster than the baseline, while using less than half the memory, 1.99 GB vs 4.12 GB, representing a significant saving. Our optimised parallel code comes in at 3.3× faster than the baseline and 2.4× faster than the optimised serial code. However, its performance comes at a cost - the parallel code uses 35% more memory than the optimised serial code. Overall, our fastest parallel code is over 20x faster than the existing equivalent python implementation, representing a significant markup in speed.

As mentioned, the serial code is significantly more memory efficient, 1.99 GB vs 2.67 GB for the parallel implementation. This is down to two factors - an increase in allocations and overhead due to multi-threading, and the fact that we can use in-place operations in the bootstrap loop. Depending on the problem and system used, either of time or memory may be more important to optimise over. In this case, since a sizeable time benefit is seen on just 4 threads, parallelization may be preferred. This is likely to be the case for most systems since HPC setups typically have many more threads. Moreover, while the allocations are increased, these are in temporary variables so memory is flushed at end of loop; the risk of going out of bounds in memory are limited. However, there may be certain cases where memory is more critical and for this reason we publish both our optimised serial and parallel code on the project's GitHub.

For analysing performance of the program, we make use of Table 1, computed using our optimised parallel code on the Ishigami system for different values of $N_V, N_O, N_I, n_{boot}$. In addition, we benchmark our program for different values of dimensions and $n_{boot}$ on Ishigami, also recording the breakup of the time spent on sample generation vs Shapley indices calculation, as shown in Table 3.

From Table 1, we see the most crucial variable influencing performance of the program is $n_{boot}$. Computation time and memory scale roughly linearly (though not exactly, with variation at both highest and lowest values) with $n_{boot}$. This follows our expectation since most of the computation effort and allocations are in the loop going over $n_{boot}$. $n_{boot}$ sets the size of the confidence interval, therefore the user must make this key tradeoff between computation cost and precision. This, and other considerations relating to setting the values of $N_V, N_O, N_I$ were discussed in Section 4.2.

Table 3 shows that sample generation time is independent of bootstrap runs. It depends primarily on the dimension of the problem since that defines the number of permutations over which the samples are to be generated. This is a key feature of our implementation (discussed in Section 2.2), distinct from the Song Nelson Staum paper [7] that inspired this work. Table 3 also shows how increasing dimension drastically increases memory and time requirements in both sections of the code. This underscores the importance of the random permutation implementation - generating all samples using the exact permutation method for even 10+ dimensions may prove to be very costly. Fundamentally, the use of random permutations method makes the problem of Shapley effects calculation tractable.

| $N_V$ | $N_O$ | $N_I$ | $n_{boot}$ | *dim* | **Total runtime (s)** | **Total allocations (GB)** | **Sample time (s)** | **Shapley time(s)** |
|---|---|---|---|---|---|---|---|---|
| 1000 | 100 | 3 | 1000 | **3** | 0.014 | 0.073 | 0.006 | 0.008 |
| 1000 | 100 | 3 | 1000 | **4** | 0.075 | 0.281 | 0.038 | 0.038 |
| 1000 | 100 | 3 | 1000 | **5** | 0.531 | 2.02 | 0.370 | 0.161 |
| 1000 | 100 | 3 | 1000 | **6** | 3.590 | 13.71 | 2.162 | 1.428 |
| 1000 | 100 | 3 | 1000 | **7** | 45.083 | 128.35 | 22.903 | 22.180 |
| 1000 | 100 | 3 | 100000 | **3** | 1.054 | 4.42 | 0.006 | 1.048 |
| 1000 | 100 | 3 | 100000 | **4** | 3.387 | 15.45 | 0.081 | 3.305 |
| 1000 | 100 | 3 | 100000 | **5** | 20.629 | 88.19 | 0.292 | 20.337 |

**Table 3**

*Runtime and allocations as a function of number of dimensions and number of bootstraps. The total runtime is broken down into the time taken for generating the sample (sample time) and the time for calculating Shapley indices (shapley time).*

## 6. Conclusion.

In this report, we discussed the importance of global sensitivity analysis, a crucial technique for analysing the behaviour of a system over entire range of its input parameters. We discussed the motivation for developing an implementation for Shapley effects, a GSA technique which is interpretable and can account for dependencies in input parameters of a system. We described our implementation of Shapley effects in Julia. This involved estimating the incremental cost of adding an input feature of a function to the entire feature set, and using Monte Carlo sampling to calculate the conditional variances. Our implementation decouples the generation of samples from the computation of Shapley indices, allowing for independent modulation of these two segments. It follows the SciML principles and can be applied to functions and differential equations alike. Another key feature is that it allows using a random subset of permutations for generating samples, making estimating Shapley effects tractable in higher dimensional systems as well. We demonstrated the correctness of our system by testing its result on the Ishigami and Linear functions and comparing with Sobol. We demonstrated the benefit of our implementation of the Jackson model of a factory system, showing how the sobol estimates can differ widely from Shapley in the case of dependent inputs. We also applied our program to a prey-predator system modelled by Lotka–Volterra ODEs to underscore its versatility. Finally, we optimised our code for performance by focusing on type-stability, reducing allocations and parallelism. We showed our optimised parallel code running on 4 threads is 20× faster than the equivalent Shapley effects implementation in python, while our optimised serial code in julia is about 9x faster than the python implementation. Our optimised code is available on the project repository.

**Acknowledgments.** We would like to thank Vaibhav Dixit from the Julia hub team for providing valuable feedback in implementation and our approach.

## REFERENCES

[1] J. Castro, D. Gómez, and J. Tejada, *Polynomial calculation of the shapley value based on sampling*, Computers & Operations Research, 36 (2009), pp. 1726–1730.
[2] T. Ishigami and T. Homma, *An importance quantification technique in uncertainty analysis for computer models*, in [1990] Proceedings. First international symposium on uncertainty modeling and analysis, IEEE, 1990, pp. 398–403.
[3] J. R. Jackson, *Jobshop-like queueing systems/comments on" jobshop-like queueing systems"*, Management science, 50 (2004), p. 1796.
[4] Y. Li and Y. Kuang, *Periodic solutions of periodic delay lotka–volterra equations and systems*, Journal of Mathematical Analysis and Applications, 255 (2001), pp. 260–280.
[5] B. Nazih, *Cemracs17 / shapley-effects*, 2017, https://gitlab.com/CEMRACS17/shapley-effects/-/tree/master/.
[6] C. Rackauckas, *Sciml/scimlbook: v1.1*, Nov. 2022, https://doi.org/10.5281/zenodo.7347643.
[7] E. Song, B. L. Nelson, and J. Staum, *Shapley effects for global sensitivity analysis: Theory and computation*, SIAM/ASA Journal on Uncertainty Quantification, 4 (2016), pp. 1060–1083, https://doi.org/10.1137/15M1048070.
[8] X. Zhang, M. Trame, L. Lesko, and S. Schmidt, *Sobol sensitivity analysis: A tool to guide the development and evaluation of systems pharmacology models*, CPT: Pharmacometrics & Systems Pharmacology, 4 (2015), p. 69–79, https://doi.org/10.1002/psp4.6.