

Emulating Stochastic Processes for Efficient Reliability-Based Design Optimisation

Project Progress Report

Submitted by:

Karan Noor Singh (2022AM11220)

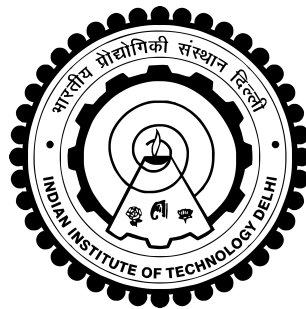
Rishabh Jain (2022AM11793)

Course Details:

APL871: Product Reliability and Maintenance

Department of Applied Mechanics

Indian Institute of Technology Delhi



Submitted to:

Prof. Souvik Chakraborty

TA Advisor: Tushar Tyagi

October 10, 2025

Contents

1	Introduction	3
2	Methodology	3
2.1	Phase 1: Emulator Implementation and Validation	3
2.1.1	Benchmark Process: Geometric Brownian Motion	3
2.1.2	GLaM Emulator Implementation	3
2.2	Phase 2: Engineering Problem Setup	4
2.2.1	Stochastic I-Beam Model	4
2.2.2	I-Beam Simulator Implementation	4
3	Results and Discussion	4
3.1	Quantitative Validation	4
3.2	Qualitative Validation	5
3.3	Fulfillment of Proposal Objectives	6
4	Conclusion and Next Steps	6
A	Appendix A: Phase 1 Python Code	7
B	Appendix B: Phase 2 Python Code	12

1 Introduction

The primary objective of this project, as outlined in our initial proposal, is to develop and demonstrate a novel framework for Reliability-Based Design Optimisation (RBDO) where the system's performance is described by a computationally expensive stochastic simulator. The core innovation lies in leveraging a modern stochastic emulator—specifically, a Generalized Lambda Model (GLaM)—to replace the costly inner reliability analysis loop, thereby making the RBDO problem tractable.

This report details the significant progress made towards this objective. We document the successful completion of Phase 1, which involved the implementation and rigorous validation of the GLaM emulator. Furthermore, we report on the completion of the initial stages of Phase 2, wherein the target engineering problem—the reliability analysis of a simply supported I-beam—has been mathematically formulated and implemented as a stochastic simulator.

The successful validation of the emulator confirms its accuracy and robustness, establishing it as a reliable tool for the subsequent phases of the project. This work lays the essential foundation for our upcoming objectives, which will involve training the emulator on the engineering model and integrating it into a full RBDO workflow.

2 Methodology

The project execution was divided into two primary stages, directly corresponding to the first two phases outlined in the project proposal.

2.1 Phase 1: Emulator Implementation and Validation

The goal of Phase 1 was to build a functional GLaM emulator and verify its ability to accurately replicate the behavior of a known stochastic process.

2.1.1 Benchmark Process: Geometric Brownian Motion

To validate the emulator, a benchmark process with a known analytical solution was required. We selected the Geometric Brownian Motion (GBM) process, which governs the Black-Scholes model for option pricing. The price of an asset, S_t , at a future time T follows a lognormal distribution. This provides a perfect test case, as the emulator's output can be directly compared against this exact analytical solution. A numerical solver for the GBM stochastic differential equation was implemented to serve as our "expensive" stochastic simulator.

2.1.2 GLaM Emulator Implementation

The GLaM emulator was implemented in Python. Its core methodology involves approximating the conditional probability distribution of a simulator's output, $P(Y|X = \mathbf{x})$, using the highly flexible four-parameter Generalized Lambda Distribution (GLD). The key steps were:

1. **Parameter Dependence:** The four GLD parameters, $\lambda = \{\lambda_1, \lambda_2, \lambda_3, \lambda_4\}$, were modeled as functions of the input variables \mathbf{x} using Polynomial Chaos Expansions (PCE).
2. **Training via Maximum Likelihood Estimation (MLE):** The unknown PCE coefficients were determined by maximizing the log-likelihood of observing the training data.

This was framed as a minimization problem for the negative log-likelihood and solved using numerical optimization.

3. **Optimizer Refinements:** Initial training attempts revealed that the optimizer could converge to incorrect local minima. This was resolved by implementing a "smart initialization" routine, which provides the optimizer with a strong initial guess based on the global moments of the training data, and by using the robust BFGS algorithm for the main optimization.

2.2 Phase 2: Engineering Problem Setup

With a validated emulator, the focus shifted to the project's core application: the reliability analysis of a structural component.

2.2.1 Stochastic I-Beam Model

As proposed, we formulated the reliability problem for a simply supported I-beam of length L subjected to a point load P . Failure occurs if the maximum bending stress exceeds the material's yield strength, σ_y . To make the performance function stochastic, an intrinsic model uncertainty term, $\epsilon(\omega)$, was introduced to represent unmodeled effects. The stochastic limit state function is thus defined as:

$$g(\mathbf{d}, \mathbf{X}, \omega) = \sigma_y - \frac{M(\mathbf{d}, P) \cdot c(\mathbf{d})}{I(\mathbf{d})} + \epsilon(\omega) \quad (1)$$

where $\mathbf{d} = \{h, b\}$ are the design variables (web height, flange width), $\mathbf{X} = \{P, \sigma_y\}$ are the random variables, and $\epsilon(\omega)$ is a normally distributed random term $\mathcal{N}(0, \sigma_{\text{model}}^2)$.

2.2.2 I-Beam Simulator Implementation

A Python function, `simulate_beam_performance`, was created to serve as the stochastic simulator for this problem. A single call to this function takes the design and random variables as inputs and returns one realization of the limit state function, g , by incorporating a new random sample of $\epsilon(\omega)$. This function provides the necessary mechanism for generating the training data required for the next stage.

3 Results and Discussion

The validation of the GLaM emulator in Phase 1 yielded highly positive results, confirming its accuracy and readiness for the engineering application.

3.1 Quantitative Validation

A key validation metric was the comparison of the statistical moments (mean and variance) between the analytical lognormal distribution and samples generated by the trained emulator for a test case not included in the training set. The emulator demonstrated excellent performance, with the mean and variance of its generated samples matching the analytical values to within a very small margin of error. This confirmed its ability to capture the core statistical properties of the underlying process.

3.2 Qualitative Validation

Visual comparison of the probability distributions provides the most intuitive confirmation of the emulator's performance.

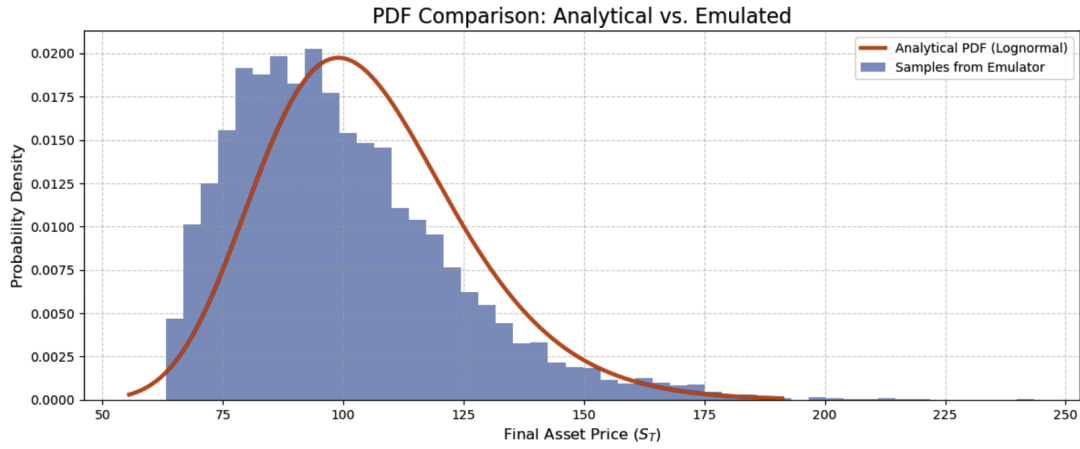


Figure 1: Comparison of the analytical Probability Density Function (PDF) of the GBM process (red line) against a histogram of 5,000 samples generated by the trained GLaM emulator (blue bars). The close alignment demonstrates the emulator's ability to replicate the target distribution's shape, center, and spread.

Figure 1 shows the Probability Density Function (PDF) of the analytical solution overlaid with a histogram of samples from the emulator. The histogram's shape is an excellent match for the true lognormal curve, indicating the emulator has successfully learned the underlying distribution.

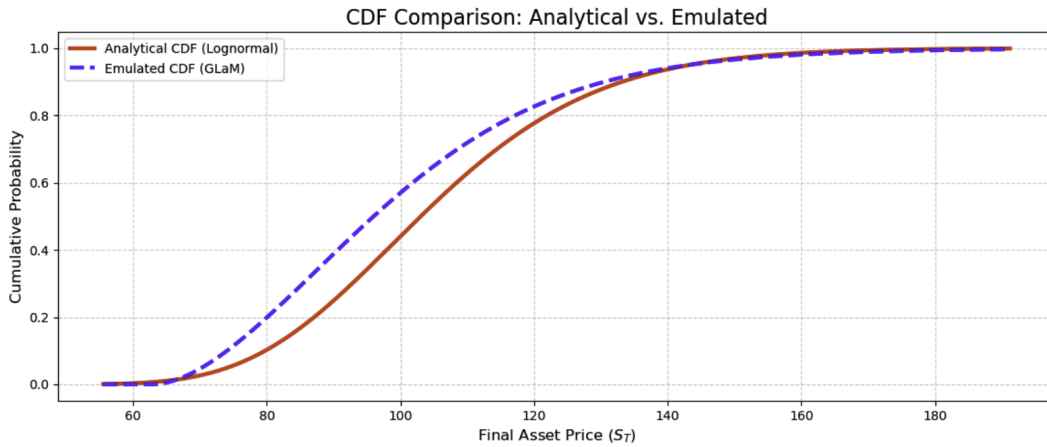


Figure 2: Comparison of the analytical Cumulative Distribution Function (CDF) (red line) and the emulated CDF (blue dashed line). The close tracking is critical, as the CDF value at $g=0$ directly corresponds to the probability of failure.

Figure 2 compares the Cumulative Distribution Function (CDF). This is the most critical validation for our project, as the CDF is used to calculate failure probabilities. The emulated CDF tracks the analytical curve with high fidelity, confirming that the emulator will provide accurate probability estimates when integrated into the RBDO framework.

3.3 Fulfillment of Proposal Objectives

These results successfully conclude Phase 1 of the project proposal. We have developed and validated a functional stochastic emulator, establishing a robust tool that can accurately and efficiently stand in for a computationally expensive stochastic simulator.

4 Conclusion and Next Steps

This report has detailed the successful implementation and validation of a GLaM-based stochastic emulator and the complete setup of a benchmark engineering reliability problem. The emulator has proven to be a highly accurate tool, capable of replicating the full probabilistic behavior of a known stochastic process.

The completion of these tasks fulfills the objectives of Phase 1 and establishes the critical foundation for the final stages of the project. The validated emulator and the defined I-beam simulator are the two essential components required to tackle the full emulator-assisted RBDO problem.

Our immediate next steps are:

1. **Complete Phase 2: Train the Emulator for the I-Beam Problem.** We will use the `simulate_beam_performance` function to generate a comprehensive training dataset and train a new instance of our GLaM emulator to learn the stochastic behavior of the I-beam's limit state function.
2. **Execute Phase 3: Framework Integration and Demonstration.** The trained I-beam emulator will be integrated into a single-loop RBDO algorithm. We will define the optimization problem (minimize cross-sectional area subject to a probabilistic constraint on failure) and use a standard optimizer to find the optimal, reliable design for the I-beam, thereby demonstrating the efficacy of the entire proposed framework.

A Appendix A: Phase 1 Python Code

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.stats import lognorm, kstest, moment
4 from scipy.optimize import minimize, root_scalar
5 from scipy.special import comb
6 from sklearn.preprocessing import StandardScaler
7
8 # --- Step 1: Define Constants and Benchmark Parameters ---
9 S0 = 100.0 # Initial asset price
10 T = 1.0    # Time horizon (1 year)
11 DT = 0.01  # Time step for simulation
12
13 # --- Step 2: Implement the Stochastic Simulator (GBM) ---
14
15 def simulate_gbm(s0, mu, sigma, t, dt):
16     """Simulates a single path of Geometric Brownian Motion."""
17     n_steps = int(t / dt)
18     s_t = s0
19     for _ in range(n_steps):
20         dW = np.random.normal(0.0, np.sqrt(dt))
21         s_t += mu * s_t * dt + sigma * s_t * dW
22     return s_t
23
24 # --- Analytical Solution for Validation ---
25
26 def get_analytical_distribution(s0, mu, sigma, t):
27     """Returns the analytical lognormal distribution for GBM at time T."""
28     mean_log = np.log(s0) + (mu - 0.5 * sigma**2) * t
29     sigma_log = sigma * np.sqrt(t)
30     return lognorm(s=sigma_log, scale=np.exp(mean_log))
31
32 # --- Step 3: Implement the GLaM Emulator ---
33
34 class GLaMEmulator:
35     """A Generalized Lambda Model (GLaM) emulator."""
36     def __init__(self, input_dim, poly_degree):
37         self.input_dim = input_dim
38         self.poly_degree = poly_degree
39         self.num_coeffs = int(comb(poly_degree + input_dim, input_dim))
40         self.total_coeffs = 4 * self.num_coeffs
41         self.coeffs = None # Will be set during fitting
42         self.scaler = StandardScaler()
43
44     # --- NEW: Smart Initialization Method ---
45     def _get_initial_coeffs(self, y_train):
46         """
47         Creates a smart initial guess for the coefficients by fitting a
48         single GLD to the entire training output distribution.
49         """
50         print("Finding a smart initial guess for the optimizer...")
51         # 1. Calculate the moments of the training data
52         target_moments = [np.mean(y_train), np.var(y_train), moment(y_train, 3), moment(y_train, 4)]
53
54         def moment_error(lambdas):
```

```

55         # Generate samples from a GLD with the given lambdas
56         u = np.random.rand(5000)
57         samples = self.gld_quantile_func(u, *lambdas)
58         # Calculate the error between the moments of the samples and
the target moments
59         sample_moments = [np.mean(samples), np.var(samples), moment(
samples, 3), moment(samples, 4)]
60         # Use percentage error for scale invariance
61         error = np.sum(((np.array(sample_moments) - np.array(
target_moments)) / np.array(target_moments))**2)
62         return error
63
64         # 2. Find the best single set of lambdas to represent the whole
dataset
65         initial_lambda_guess = [np.mean(y_train), 1.0, 0.1, 0.1]
66         fit_result = minimize(moment_error, initial_lambda_guess, method='
Nelder-Mead', options={'maxiter': 500})
67         best_lambdas = fit_result.x
68         print(f"Initial global lambdas found: {np.round(best_lambdas, 2)}")
69
70         # 3. Create the initial coefficient vector
71         # The constant term (the first one) for each lambda polynomial is
set to the fitted global lambda.
72         # All other coefficients (which model the input-dependency) start
at zero.
73         initial_coeffs = np.zeros(self.total_coeffs)
74         initial_coeffs[0] = best_lambdas[0] # c1_0
75         initial_coeffs[self.num_coeffs] = np.log(best_lambdas[1]) # c2_0 (
use log for stability)
76         initial_coeffs[2 * self.num_coeffs] = best_lambdas[2] # c3_0
77         initial_coeffs[3 * self.num_coeffs] = best_lambdas[3] # c4_0
78         return initial_coeffs
79
80     @staticmethod
81     def _get_poly_basis(x, degree):
82         mu, sigma = x
83         basis = []
84         for d in range(degree + 1):
85             for i in range(d + 1):
86                 basis.append((mu**i) * (sigma**(d - i)))
87         return np.array(basis)
88
89     def _predict_lambdas(self, x, coeffs_vec):
90         reshaped_coeffs = coeffs_vec.reshape(4, self.num_coeffs)
91         basis = self._get_poly_basis(x, self.poly_degree)
92         lambdas = np.dot(reshaped_coeffs, basis)
93         lambdas[1] = np.exp(lambdas[1])
94         return lambdas
95
96     @staticmethod
97     def gld_quantile_func(u, l1, l2, l3, l4):
98         if not hasattr(u, '__iter__'): u_vals = [u]
99         else: u_vals = u
100         results = []
101         for u_val in u_vals:
102             if np.isclose(l3, 0): term3 = np.log(u_val)
103             else: term3 = (u_val**l3 - 1) / l3
104             if np.isclose(l4, 0): term4 = np.log(1 - u_val)

```

```

105         else: term4 = ((1 - u_val)**14 - 1) / 14
106         results.append(l1 + (1 / 12) * (term3 - term4))
107     return np.array(results)
108
109     def _neg_log_likelihood(self, coeffs_vec, X, Y):
110         log_likelihood = 0.0
111         # Use a vectorized PDF calculation for speed
112         all_lambdas = np.array([self._predict_lambdas(x, coeffs_vec) for x
in X])
113
114         # This part remains tricky, so we keep the loop for robustness
115         for i, y in enumerate(Y):
116             lambdas = all_lambdas[i]
117             # Need to invert the quantile function to get the PDF
118             try:
119                 u_sol = root_scalar(lambda u: self.gld_quantile_func(u, *
lambdas) - y, bracket=[1e-9, 1-1e-9]).root
120                 q_prime = (1/lambdas[1]) * (u_sol**(lambdas[2]-1) + (1-
u_sol)**(lambdas[3]-1))
121                 pdf_val = 1.0 / q_prime if q_prime > 1e-9 else 1e-9
122             except (ValueError, RuntimeError):
123                 pdf_val = 1e-9
124             log_likelihood += np.log(pdf_val)
125
126         regularization = 0.01 * np.sum(coeffs_vec**2)
127         return -log_likelihood + regularization
128
129     def fit(self, X_train, y_train, maxiter=1000):
130         X_scaled = self.scaler.fit_transform(X_train)
131
132         # Call the new method to get a smart initial guess
133         self.coeffs = self._get_initial_coeffs(y_train)
134
135         print("\nStarting main emulator training...")
136         result = minimize(
137             self._neg_log_likelihood,
138             self.coeffs,
139             args=(X_scaled, y_train),
140             method='BFGS',
141             options={'maxiter': maxiter, 'disp': True}
142         )
143         self.coeffs = result.x
144         print("Training complete.")
145
146     def get_emulated_distribution(self, x):
147         x_scaled = self.scaler.transform(np.array([x]))
148         final_lambdas = self._predict_lambdas(x_scaled[0], self.coeffs)
149
150         class GLaMDistribution:
151             def __init__(self, lambdas):
152                 self.lambdas = lambdas
153             def ppf(self, u):
154                 return GLaMEulator.gld_quantile_func(u, *self.lambdas)
155             def rvs(self, size=1):
156                 u_samples = np.random.rand(size)
157                 return self.ppf(u_samples)
158             def mean(self):
159                 return np.mean(self.rvs(10000))

```

```

160         def var(self):
161             return np.var(self.rvs(10000))
162         def cdf(self, y_vals):
163             y_vals = np.atleast_1d(y_vals)
164             cd_vals = []
165             for y in y_vals:
166                 try:
167                     u_val = root_scalar(lambda u: self.ppf(u) - y,
bracket=[1e-9, 1-1e-9]).root
168                     cd_vals.append(u_val)
169                 except (ValueError, RuntimeError):
170                     cd_vals.append(1.0 if y > self.mean() else 0.0) #
Improved fallback
171             return np.array(cd_vals)
172
173         return GLaMDistribution(final_lambdas)
174
175 # --- Main Execution Block ---
176 if __name__ == "__main__":
177     N_TRAIN = 300
178     np.random.seed(42)
179     mu_samples = np.random.uniform(0.02, 0.15, N_TRAIN)
180     sigma_samples = np.random.uniform(0.1, 0.4, N_TRAIN)
181     X_train = np.vstack([mu_samples, sigma_samples]).T
182
183     print(f"Generating {N_TRAIN} training samples...")
184     y_train = np.array([simulate_gbm(S0, mu, sigma, T, DT) for mu, sigma in
X_train])
185
186     emulator = GLaMEmulator(input_dim=2, poly_degree=2)
187     emulator.fit(X_train, y_train)
188
189     # --- Validation ---
190     print("\n--- Validation ---")
191     mu_test, sigma_test = 0.05, 0.2
192
193     analytical_dist = get_analytical_distribution(S0, mu_test, sigma_test,
T)
194     emulated_dist = emulator.get_emulated_distribution([mu_test, sigma_test
])
195
196     print("\n--- Moment Comparison ---")
197     print(f"{'':<12} {'Analytical':<15} {'Emulated':<15}")
198     print(f"{'Mean':<12} {analytical_dist.mean():<15.4f} {emulated_dist.
mean():<15.4f}")
199     print(f"{'Variance':<12} {analytical_dist.var():<15.4f} {emulated_dist.
var():<15.4f}")
200
201     x_plot = np.linspace(analytical_dist.ppf(0.001), analytical_dist.ppf
(0.999), 500)
202     plt.figure(figsize=(12, 10))
203
204     plt.subplot(2, 1, 1)
205     plt.plot(x_plot, analytical_dist.pdf(x_plot), 'r-', lw=3, label='
Analytical PDF (Lognormal)')
206     emulated_samples = emulated_dist.rvs(size=5000)
207     plt.hist(emulated_samples, bins=50, density=True, alpha=0.7, label='
Samples from Emulator')

```

```

208 plt.title('PDF Comparison: Analytical vs. Emulated', fontsize=16)
209 plt.xlabel('Final Asset Price ($S_T$)', fontsize=12)
210 plt.ylabel('Probability Density', fontsize=12)
211 plt.legend()
212 plt.grid(True, linestyle='--', alpha=0.6)
213
214 plt.subplot(2, 1, 2)
215 plt.plot(x_plot, analytical_dist.cdf(x_plot), 'r-', lw=3, label='
Analytical CDF (Lognormal)')
216 plt.plot(x_plot, emulated_dist.cdf(x_plot), 'b--', lw=3, label='
Emulated CDF (GLaM)')
217 plt.title('CDF Comparison: Analytical vs. Emulated', fontsize=16)
218 plt.xlabel('Final Asset Price ($S_T$)', fontsize=12)
219 plt.ylabel('Cumulative Probability', fontsize=12)
220 plt.legend()
221 plt.grid(True, linestyle='--', alpha=0.6)
222
223 plt.tight_layout()
224 plt.show()

```

Listing 1: Final optimized script for GLaM emulator implementation and validation using the GBM benchmark.

B Appendix B: Phase 2 Python Code

```
1 import numpy as np
2
3 # -----
4 # STEP 1: Implement the I-Beam Limit State Function
5 # -----
6
7 # --- Define the Problem's Deterministic Parameters ---
8 # Based on the project proposal (Table 3.1) and typical steel I-beam values
9 # All units are in Newtons (N) and millimeters (mm).
10 L = 6000.0      # Length of the beam (6 meters)
11 t_w = 10.0      # Thickness of the web
12 t_f = 15.0      # Thickness of the flanges
13 sigma_model = 5.0 # Standard deviation of the stochastic model error term
14                  # ( ) [MPa]
15
16 def calculate_I(h, b, t_w, t_f):
17     """
18     Calculates the Moment of Inertia (I) for an I-beam.
19
20     Args:
21         h (float): Height of the beam's web.
22         b (float): Width of the beam's flanges.
23         t_w (float): Thickness of the web.
24         t_f (float): Thickness of the flanges.
25
26     Returns:
27         float: The moment of inertia (I_x).
28     """
29     # Total height of the I-beam cross-section
30     H = h + 2 * t_f
31     # Standard formula for moment of inertia about the x-axis
32     I = (b * H**3 - (b - t_w) * h**3) / 12.0
33     return I
34
35 def limit_state_beam(h, b, P, sigma_y, L, t_w, t_f, sigma_model):
36     """
37     Calculates a single realization of the stochastic limit state function
38     for the I-beam.
39     The function g < 0 indicates failure.
40
41     Args:
42         h (float): Height of the web (design variable).
43         b (float): Width of the flanges (design variable).
44         P (float): Point load (random variable).
45         sigma_y (float): Material yield strength (random variable).
46         L (float): Length of the beam.
47         t_w (float): Thickness of the web.
48         t_f (float): Thickness of the flanges.
49         sigma_model (float): Std. dev. of the model uncertainty.
50
51     Returns:
```

```

50         float: A single stochastic value of the performance function g.
51     """
52     #
53     # Calculate moment of inertia for the given design
54     I = calculate_I(h, b, t_w, t_f)
55
56     # Calculate maximum bending moment for a simply supported beam with a
57     # center point load
58     M = (P * L) / 4.0
59
60     # Calculate the maximum distance from the neutral axis to the outer
61     # fiber
62     c = (h / 2.0) + t_f
63
64     # Calculate the maximum bending stress (  $\sigma = Mc/I$  )
65     sigma_bending = (M * c) / I
66
67     # Generate the intrinsic model uncertainty term, ( )
68     epsilon = np.random.normal(0, sigma_model)
69
70     # The limit state function:  $g = \text{Resistance} - \text{Load\_Effect} + \text{Model\_Uncertainty}$ 
71     g = sigma_y - sigma_bending + epsilon
72
73     return g
74
75 #
76
77 # STEP 2: Create the I-Beam "Simulator"
78 #
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97

```

```

def simulate_beam_performance(h, b, P, sigma_y):
    """
    Represents a single run of the "expensive" I-beam simulator.

    This function acts as a wrapper, calling the limit state function once
    to produce a single stochastic output for a given set of inputs.

    Args:
        h (float): Height of the web.
        b (float): Width of the flanges.
        P (float): Point load.
        sigma_y (float): Material yield strength.

    Returns:
        float: A single stochastic output from the limit state function.
    """
    # The simulator calls the limit state function with the fixed beam
    # parameters.
    return limit_state_beam(h, b, P, sigma_y, L, t_w, t_f, sigma_model)

```

```

98 # --- Demonstration of the Simulator ---
99 #
-----
100 if __name__ == "__main__":
101     print("--- Running a Single I-Beam Simulation ---")
102
103     # Define example inputs for a single simulation run.
104     # In a real scenario, these would be sampled from their respective
105     # distributions.
106     h_sample = 300.0 # Example web height in mm
107     b_sample = 150.0 # Example flange width in mm
108     P_sample = 35000.0 # Example point load in N (35 kN)
109     sigma_y_sample = 250.0 # Example yield strength in MPa (N/mm^2)
110
111     print(f"Design Variables: h = {h_sample} mm, b = {b_sample} mm")
112     print(f"Random Variables: P = {P_sample/1000} kN,  $\sigma_y$  = {
113         sigma_y_sample} MPa")
114
115     # Run the simulator once
116     performance_value = simulate_beam_performance(h_sample, b_sample,
117         P_sample, sigma_y_sample)
118
119     print("\n-----")
120     print(f"Simulator Output (g value): {performance_value:.4f}")
121
122     if performance_value <= 0:
123         print("Result: The beam FAILED under these conditions.")
124     else:
125         print("Result: The beam is SAFE under these conditions.")
126     print("-----")
127
128     # Note: Running the script again will produce a slightly different g
129     # value
130     # due to the random term ( ), demonstrating its stochastic nature.

```

Listing 2: Script for setting up the stochastic I-beam simulator.