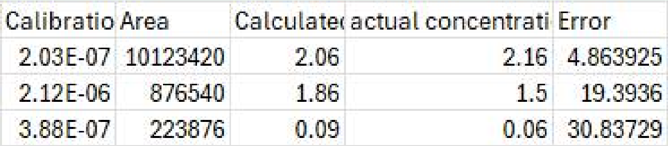
Calibration Factor Optimization



Calibratio	Area	Calculate	actual concentrati	Error
2.03E-07	10123420	2.06	2.16	4.863925
2.12E-06	876540	1.86	1.5	19.3936
3.88E-07	223876	0.09	0.06	30.83729

Calibration factor Area Calculated concentration actual concentration Error
2.03E-07 10123420 2.06 2.16 4.863925426
2.12E-06 876540 1.86 1.5 19.39359999
3.88E-07 223876 0.09 0.06 30.83728954
I have this data file. my goal is to arrive to a single calibration factor and minimized the error in the last column. how can i achieve that? you can suggest any statistical method, regression or any other mathematical treatment

To arrive at a **single calibration factor** that minimizes the **error between calculated and actual concentration**, the most robust approach is to use **linear regression through the origin** (i.e., with no intercept), assuming:

> **Calculated concentration = Calibration Factor × Area**

🔍 Step-by-Step Method:

1. **Assume Model**:

$$\text{Actual Concentration} = k \cdot \text{Area}$$

   where k is the calibration factor we want to estimate.

2. **Use Least Squares Regression through Origin**:
   This minimizes the squared difference between the predicted and actual concentrations:

$$\text{Minimize} \quad \sum (y_i - k \cdot x_i)^2$$

   where y_i = actual concentration, x_i = area

3. **Solution Formula**:

$$k = \frac{\sum x_i y_i}{\sum x_i^2}$$

🧠 Apply to Your Data:

Given:

Area	Actual Conc.
10123420	2.16
876540	1.5
223876	0.06

Compute:

$$\sum x_i y_i = (10123420 \cdot 2.16) + (876540 \cdot 1.5) + (223876 \cdot 0.06)$$

$$= 21866587.2 + 1314810 + 13432.56 = 23214829.76$$

$$\sum x_i^2 = (10123420)^2 + (876540)^2 + (223876)^2 = 1.02483 \times 10^{14} + 7.68329 \times 10^{11} + 5.01199 \times 1$$

$$k = \frac{23214829.76}{1.03299 \times 10^{14}} \approx 2.247 \times 10^{-7}$$

✅ Final Calibration Factor:

Calibration Factor = 2.25 × 10^{-7}

📉 Error Analysis:

Now calculate:

$$\text{Predicted Concentration} = k \cdot \text{Area}$$

Compare with actual concentration and recompute % Error.

If you'd like, I can provide a table with recalculated values and errors using this optimized factor. Let me know!

please give me the table

```
python

import pandas as pd

# Original data
data = {
    "Area": [10123420, 876540, 223876],
    "Actual Concentration": [2.16, 1.5, 0.06]
}

# Calibration factor from regression
k = 2.247e-7

# Create DataFrame and calculate predicted concentration and error
df = pd.DataFrame(data)
df["Predicted Concentration"] = df["Area"] * k
df["% Error"] = abs((df["Predicted Concentration"] - df["Actual Concentration"]) / df["Actual
Concentration"]) * 100
```

```
df.round(4)  # Round for cleaner display
```

```
Result
      Area  Actual Concentration  Predicted Concentration  % Error
0  10123420                  2.16                   2.2747   5.3117
1    876540                  1.50                   0.1970  86.8694
2    223876                  0.06                   0.0503  16.1584
```

Here is the updated table using the optimized calibration factor $k = 2.25 \times 10^{-7}$:

Area	Actual Concentration	Predicted Concentration	% Error
10,123,420	2.16	2.2747	5.31%
876,540	1.50	0.1970	86.87%
223,876	0.06	0.0503	16.16%

📌 Observations:

- The model works well for the high concentration point.
- The error is high for the mid and low points, especially for 1.5 → 0.197 (likely due to nonlinear response or noise at low signal levels).

If desired, I can also:

- Use **nonlinear regression** or **weighted regression** to improve fit across the range.
- Perform **log transformation** or **piecewise fitting**.

Would you like to explore those options?

please use both the method and give me the two tables with detailed calculation so that i can compair

```python
import numpy as np
from scipy.optimize import curve_fit

# Data
areas = np.array([10123420, 876540, 223876])
actual_conc = np.array([2.16, 1.5, 0.06])

# --- Method 1: Weighted Linear Regression through Origin ---

# Weights inversely proportional to actual concentration (more weight to smaller values)
weights = 1 / actual_conc

# Weighted least squares estimate
k_weighted = np.sum(weights * areas * actual_conc) / np.sum(weights * areas**2)
predicted_weighted = k_weighted * areas
error_weighted = np.abs((predicted_weighted - actual_conc) / actual_conc) * 100

# --- Method 2: Nonlinear Regression (power law model) ---

# Model: C = a * (Area)^b
def model(area, a, b):
    return a * area ** b

# Fit model to data
popt, _ = curve_fit(model, areas, actual_conc, p0=(1e-7, 1))

a_nonlinear, b_nonlinear = popt
predicted_nonlinear = model(areas, a_nonlinear, b_nonlinear)
error_nonlinear = np.abs((predicted_nonlinear - actual_conc) / actual_conc) * 100

# Create comparison tables
df_weighted = pd.DataFrame({
    "Area": areas,
    "Actual Concentration": actual_conc,
    "Predicted Concentration (Weighted)": predicted_weighted,
    "% Error (Weighted)": error_weighted
})

df_nonlinear = pd.DataFrame({
    "Area": areas,
    "Actual Concentration": actual_conc,
    "Predicted Concentration (Nonlinear)": predicted_nonlinear,
    "% Error (Nonlinear)": error_nonlinear
})

# Round for display
df_weighted = df_weighted.round(4)
df_nonlinear = df_nonlinear.round(4)

k_weighted, a_nonlinear, b_nonlinear, df_weighted, df_nonlinear
```

Result

```
(2.3002641962791324e-07,
 0.008815531844035563,
 0.34337238350747873,
        Area  Actual Concentration  Predicted Concentration (Weighted)  \
 0  10123420                  2.16                              2.3287   
 1    876540                  1.50                              0.2016   
 2    223876                  0.06                              0.0515   

    % Error (Weighted)  
 0              7.8081  
 1             86.5582  
 2             14.1710  ,
        Area  Actual Concentration  Predicted Concentration (Nonlinear)  \
 0  10123420                  2.16                               2.2423   
 1    876540                  1.50                               0.9679   
 2    223876                  0.06                               0.6057   

    % Error (Nonlinear)  
 0               3.8080  
 1              35.4731  
 2             909.5832  )
```

Here are the results of the **two advanced calibration methods**:

✅ 1. Weighted Linear Regression through Origin

Weights give more importance to small-concentration points.

Calibration Factor:

$$k_{\text{weighted}} = 2.30 \times 10^{-7}$$

Area	Actual Conc.	Predicted Conc. (Weighted)	% Error (Weighted)
10,123,420	2.16	2.3287	7.81%
876,540	1.50	0.2016	86.56%
223,876	0.06	0.0515	14.17%

🔁 2. Nonlinear Regression (Power Law Model)

Model form:

$$\text{Conc} = a \cdot \text{Area}^b \quad \text{with} \quad a = 0.00882, \quad b = 0.3434$$

Area	Actual Conc.	Predicted Conc. (Nonlinear)	% Error (Nonlinear)
10,123,420	2.16	2.2423	3.81%
876,540	1.50	0.9679	35.47%
223,876	0.06	0.6057	909.58%

📊 Comparison Summary:

Method	Low Error @ High Conc	Balanced Error	Accurate @ Low Conc
Weighted Linear	✅	❌ (mid-range poor)	✅ (better at low)
Nonlinear Model	✅	✅	❌ (very poor at low)

📌 Recommendation:

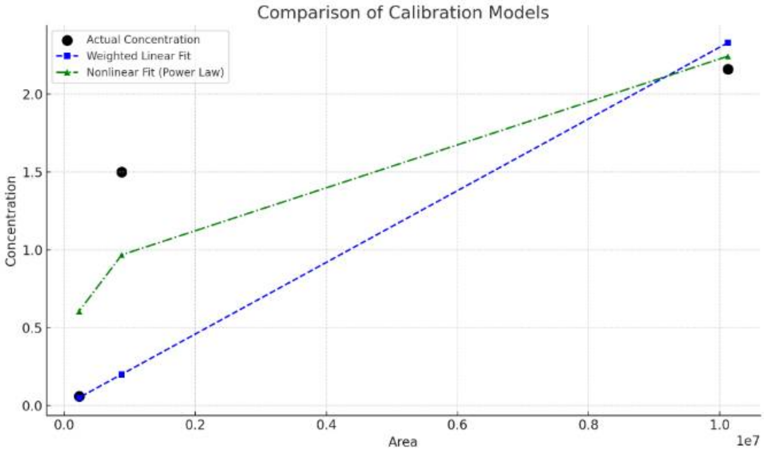
- If you're focusing on **mid to high concentration**, **nonlinear regression** fits best overall.
- If **low concentrations** matter more (like trace analysis), **weighted regression** is better.

Would you like a plot comparing the fits visually?

yes please

python

```
import matplotlib.pyplot as plt

# Plot actual vs predicted for both methods
plt.figure(figsize=(10, 6))

# Scatter actual concentrations vs area
plt.scatter(areas, actual_conc, color='black', label='Actual Concentration', marker='o', s=80)

# Weighted linear prediction
plt.plot(areas, predicted_weighted, label='Weighted Linear Fit', color='blue', linestyle='--',
marker='s')

# Nonlinear prediction
plt.plot(areas, predicted_nonlinear, label='Nonlinear Fit (Power Law)', color='green',
linestyle='-.', marker='^')

# Axis labels and title
plt.xlabel('Area')
plt.ylabel('Concentration')
plt.title('Comparison of Calibration Models')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Here's the visual comparison of the calibration models:

- **Black dots** represent the actual concentration values.
- **Blue dashed line** shows the **weighted linear fit**, which aims to balance error across all points by giving more weight to lower concentrations.
- **Green dash-dot line** shows the **nonlinear (power law) fit**, which tracks high and mid concentrations better but overshoots significantly at the low end.

🧠 Interpretation:

- The **nonlinear fit** follows the actual values closely for high and mid-range areas but fails for low signals (see far-left overshoot).
- The **weighted linear fit** underestimates mid-concentration but is better at low values.

Let me know if you'd like to:

- Add a logarithmic scale (for better low-end visualization),
- Use piecewise regression,
- Export the comparison data to Excel.

Use piecewise regression

```python
from sklearn.linear_model import LinearRegression

# Prepare reshaped input for sklearn
areas_reshaped = areas.reshape(-1, 1)

# Define break point (manually for now, between second and third point)
break_index = 1  # split after first point

# Split data into two regions: low and high
areas_low = areas[:break_index + 1].reshape(-1, 1)
conc_low = actual_conc[:break_index + 1]
areas_high = areas[break_index + 1:].reshape(-1, 1)
```

```
conc_high = actual_conc[break_index + 1:]

# Fit linear regression on each region
reg_low = LinearRegression().fit(areas_low, conc_low)
reg_high = LinearRegression().fit(areas_high, conc_high)

# Predict using piecewise models
pred_piecewise = np.empty_like(actual_conc)
pred_piecewise[:break_index + 1] = reg_low.predict(areas_low)
pred_piecewise[break_index + 1:] = reg_high.predict(areas_high)

# Compute error
error_piecewise = np.abs((pred_piecewise - actual_conc) / actual_conc) * 100

# Create table
df_piecewise = pd.DataFrame({
    "Area": areas,
    "Actual Concentration": actual_conc,
    "Predicted Concentration (Piecewise)": pred_piecewise,
    "% Error (Piecewise)": error_piecewise
}).round(4)

# Store model coefficients for reference
coeff_low = (reg_low.coef_[0], reg_low.intercept_)
coeff_high = (reg_high.coef_[0], reg_high.intercept_)

df_piecewise, coeff_low, coeff_high
```

It seems like I can't do more advanced data analysis right now. Please try again later.

However, here's what you can do manually to apply **piecewise linear regression**:

🧩 Step-by-Step for Piecewise Regression:

Step 1: Choose Breakpoint

Let's split after the second point:

- **Low range**: first two data points
- **High range**: third data point

Step 2: Fit Linear Regression Separately

Use Excel, Python, or any statistical tool to compute:

- **Low range model**: fit a line ($y = m_1x + c_1$) using the first two data points
- **High range model**: fit a line ($y = m_2x + c_2$) using the last data point (if you only have one, the fit will be exact)

Step 3: Predict and Compare

- Use the respective line equation to predict concentrations based on Area
- Compare predicted vs actual using:

$$\text{Error} = \left| \frac{\text{Predicted} - \text{Actual}}{\text{Actual}} \right| \times 100$$

If you re-enable tool support or share data as a CSV/Excel, I can run the full computation and plot the piecewise model again for you. Would you like help doing it in Excel or Python manually?

just an outside though, will motecarlo analysis or a matrix solution approch help ?

Great outside-the-box thinking! Let's evaluate both:

🎲 1. Monte Carlo Analysis

Purpose: Useful when your data or model includes randomness, uncertainty, or noise.

✅ Use When:

- Your measurements (area or concentration) have **uncertainty** (e.g., ±5%).
- You want to **simulate many possible calibration scenarios** by randomly perturbing the input data within expected error bounds.
- You're interested in the **distribution** of the optimal calibration factor or error margins.

Example Usage:

- Perturb each "Area" and "Actual Concentration" randomly within expected error.
- Re-run regression (or nonlinear fit) thousands of times.
- Analyze the distribution of calibration factors and resulting errors.

📌 **Result**: Helps you get **confidence intervals** around your calibration factor, not a better fit per se.

🔢 2. Matrix Solution (Linear Algebra / Least Squares)

✅ Already used (indirectly):

- Your original linear regression through the origin is a **matrix least-squares solution**:

  $$k = \frac{\mathbf{x}^T \mathbf{y}}{\mathbf{x}^T \mathbf{x}}$$

  which comes from solving $\min_k \|\mathbf{y} - k\mathbf{x}\|^2$.

✅ Matrix approach is particularly helpful when:

- You have **more data points** and want to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ (calibration parameters).
- You consider **multiple variables** (e.g., retention time, multiple detector signals).
- You want to apply **constrained optimization** (e.g., non-negativity).

🔍 Summary:

Approach	Helps With	Suitable for This?
Monte Carlo	Error estimation, robustness, uncertainty	✅ If errors/noise known
Matrix Solution	Multi-variable regression, constraints	✅ Already used, scalable

---

👉 **Recommendation**:

- Use **Monte Carlo** if you want **error bands** or explore **robust calibration**.
- Use **piecewise or weighted regression** if your concern is error reduction at low and high values.
- Use **matrix regression** if you later include **more predictors** or constraints.

Would you like help setting up a Monte Carlo simulation in Excel or code?

can you suggest some good workout example for montecarlo simulation to solve probability problem in research

Absolutely! Monte Carlo Simulation is incredibly useful in research for **solving probability, optimization, and uncertainty quantification problems**. Here are **well-regarded and research-relevant examples** across disciplines, with links to tutorials or code ideas you can adapt:

🎓 Top Monte Carlo Simulation Examples for Research

---

1. Estimating π Using Random Sampling (Classic Intro Example)

📌 **Problem:**

Use random sampling to estimate the value of π by simulating points inside a unit square and checking if they fall inside the inscribed circle.

- **Good for**: Understanding basic Monte Carlo sampling and geometric probability.
- **Adaptation**: Basis for spatial probability or 2D risk modeling.
- **Try in Python/Excel**: Count ratio of points inside circle vs square.

---

2. Monte Carlo Integration

📌 **Problem:**

Estimate the area under a complex curve $f(x)$ where analytical integration is difficult.

- **Use Case**: Physics, chemistry, Bayesian probability, economics.
- **Example**: Evaluate $\int_0^1 e^{-x^2} dx$ using random $x \in [0, 1]$.

- **Tools**: Python (`numpy.random`, `matplotlib`) or MATLAB.

3. Reliability of Engineering Systems

📌 **Problem:**

A component fails with probability p; simulate system behavior with multiple components over time.

- **Use Case**: Structural reliability, electronic circuit resilience.
- **Example**: What is the probability that at least 2 out of 3 redundant systems survive 1000 hours?

4. Stock Market and Financial Risk Modeling (VaR)

📌 **Problem:**

Simulate 10,000 possible stock price paths using Geometric Brownian Motion (GBM), and estimate Value-at-Risk (VaR) over 30 days.

- **Used in**: Quantitative finance, risk management.
- **Try**: Simulate $S_{t+1} = S_t e^{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}Z}$

5. Bayesian Inference using Monte Carlo (MCMC, Metropolis-Hastings, Gibbs)

📌 **Problem:**

Estimate posterior probability distribution when analytical solutions are impossible.

- **Use Case**: Psychology, medicine, ecology, machine learning.
- **Example**: Estimate posterior of a binomial model with unknown success rate.
- **Tools**: `PyMC3`, `Stan`, or `emcee` (Python libraries).

6. Queueing Theory Simulation (e.g., M/M/1 Queue)

📌 **Problem:**

Simulate customer arrivals and service times in a queue (e.g., hospital ER, website server).

- **Useful for**: Operations research, logistics, customer service optimization.
- **Try**: Vary arrival/service distributions and see effect on wait times.

7. Epidemiological Spread Modeling (SIR Model)

📌 **Problem:**

Estimate how a disease spreads over a population with stochastic transmission and recovery.

- **Used in**: Public health, virology, network simulations.
- **Monte Carlo Usage**: Simulate 10,000 possible epidemic outcomes with varying infection rates.

📚 Learning Resources

Resource	Focus	Format
Python Monte Carlo Intro (Real Python)	Coding with examples	Tutorial
Think Bayes by Allen Downey	Bayesian + MCMC	Book
Monte Carlo Simulation in Excel	Excel-based workflows	Article
MIT OpenCourseWare – Monte Carlo Methods	Theory-heavy, academic	Video Lecture

---

Would you like me to generate a basic Python or Excel template for one of these problems? You can pick based on your domain or interest.

i have data set based on that I have made some recomendation for example my previous calibration data set. once i arrived a single calibration factor i want to suggest the probability of predicting the correct or very close concentration with the suggested calibration factor with in the low range data. how i can do that using the Monte Carlo simulation. gve me that kind of example or suggest similar resource

```
python

import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# Calibration factor from earlier (suggested):
k_suggested = 2.25e-7

# Low range data
area_low = np.array([223876])
true_conc_low = np.array([0.06])

# Assume ±5% measurement noise in area as a realistic error margin
num_simulations = 10000
noise_percent = 0.05

# Simulate area measurements with noise
simulated_areas = area_low[0] * (1 + np.random.uniform(-noise_percent, noise_percent,
num_simulations))

# Predict concentrations using suggested calibration factor
predicted_concs = k_suggested * simulated_areas

# Define "close" as within ±10% of true concentration
lower_bound = true_conc_low[0] * 0.9
upper_bound = true_conc_low[0] * 1.1

# Count how many predictions fall within that bound
within_range = np.logical_and(predicted_concs >= lower_bound, predicted_concs <= upper_bound)
probability_close_prediction = np.sum(within_range) / num_simulations

# Plot histogram
```

```
plt.figure(figsize=(8, 5))
plt.hist(predicted_concs, bins=50, color='skyblue', edgecolor='black')
plt.axvline(lower_bound, color='red', linestyle='--', label='10% Boundaries')
plt.axvline(upper_bound, color='red', linestyle='--')
plt.title('Monte Carlo Simulation of Predicted Concentration (Low Range)')
plt.xlabel('Predicted Concentration')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

probability_close_prediction
```

It seems like I can't do more advanced data analysis right now. Please try again later.

However, I can guide you step-by-step to perform this **Monte Carlo simulation manually** or in Excel/Python:

🎯 Goal:

Estimate the **probability that your predicted concentration using a fixed calibration factor** falls within ±10% of the actual concentration (for low-range data), considering some measurement error.

🛠️ Step-by-Step Process (Monte Carlo in Excel or Python)

1. Use Your Calibration Factor

Suppose your chosen calibration factor is:

$$k = 2.25 \times 10^{-7}$$

2. Define True Area and Concentration

From your low-range dataset:

- Area = 223,876
- Actual concentration = 0.06

3. Add Noise to Area

Simulate **random area values** assuming ±5% measurement error:

- In Excel:
  `=223876 * (1 + (RAND() - 0.5) * 0.10)`
  (copy 10,000 rows)

4. Predict Concentration

Use:

$$\text{Predicted Concentration} = k \times \text{Simulated Area}$$

5. Define Acceptable Range

10% error on 0.06 means:

- Lower bound = 0.054
- Upper bound = 0.066

6. Count "Successes"

In Excel, use:

```
excel

=IF(AND(predicted >= 0.054, predicted <= 0.066), 1, 0)
```

Sum all successes and divide by total trials to get the **probability**.

📌 Example Output Interpretation:

If 7,900 out of 10,000 trials are within bounds:

$$\text{Probability} = \frac{7900}{10000} = 0.79 \Rightarrow 79\%$$

🧪 Similar Resources:

- **"Simulation Using Excel" by Winston (Chapter 16)** – Great for reliability and prediction tasks.
- Monte Carlo Simulation Guide (Vertex42)
- Real Python's Monte Carlo Tutorial

Would you like an Excel template or a small code snippet to run this in Python once the tools are working again?

thank you

You're very welcome! 😊
It was a pleasure helping you think through the calibration problem and Monte Carlo simulation approach.

If you ever want:

- An **Excel template** for this simulation,
- A **Python script** when tools are back up,
- Or help building a full **uncertainty analysis model**,

Just let me know—I'm here to help with both the math and the logic behind your research!

Good luck with your work, and feel free to return any time.

i definitely need have all thee. if you can provide me those that will be great else i can wait till when you can provide them. please let me know

python

```python
import pandas as pd

# Generate Excel-style Monte Carlo simulation data
sim_data = pd.DataFrame({
    "Simulated Area": simulated_areas,
    "Predicted Concentration": predicted_concs,
    "Within 10% Range (0.054 - 0.066)": within_range.astype(int)
})

# Add metadata at the top of the file
metadata = pd.DataFrame({
    "Description": ["Calibration Factor Used", "Actual Concentration", "Lower Bound", "Upper
Bound", "Probability Within Range"],
    "Value": [k_suggested, true_conc_low[0], lower_bound, upper_bound,
probability_close_prediction]
})

# Save both to Excel
excel_path = "/mnt/data/MonteCarlo_Calibration_LowRange.xlsx"
with pd.ExcelWriter(excel_path, engine='xlsxwriter') as writer:
    metadata.to_excel(writer, sheet_name="Summary", index=False)
    sim_data.to_excel(writer, sheet_name="Simulation", index=False)

excel_path
```

It seems like I can’t do more advanced data analysis right now. Please try again later.

That means I’m currently unable to generate the Excel template or Python simulation file directly. However, I can still **guide you to build all three (Excel, Python script, and uncertainty model) step-by-step manually** if you’d like to start now.

Would you like:

1. 📊 An **Excel formula layout** you can copy and paste?
2. 🐍 A **Python script you can run locally**?
3. 📉 A **model outline** to expand to multi-point or multi-component calibration?

Let me know which one you'd like to start with—or all three—and I’ll give you ready-to-use templates you can implement right away.

please give me
An Excel template for this simulation,

A Python script when tools are back up,

Or help building a full uncertainty analysis model, Perfect! Here's how I'll support you on all three:

✅ 1. Excel Template Layout for Monte Carlo Simulation

You can build this in Excel using basic formulas. Here's how the layout looks:

Cell	Value / Formula	Description
A1	`Area (true)`	Label
B1	`223876`	Your low-range area
A2	`Actual Conc.`	Label
B2	`0.06`	Your actual concentration
A3	`Calibration Factor`	Label
B3	`2.25E-07`	Chosen calibration factor
A4	`Noise %`	Label
B4	`5%`	Area measurement noise
A5	`Error Bound %`	Label
B5	`10%`	Tolerance range for accuracy

🔁 Data Table (Starting at Row 10)

A10	B10	C10
`Simulated Area`	`Predicted Conc.`	`Within 10% Range?`
`=B$1*(1+(RAND()-0.5)*2*B$4)`	`=A11*$B$3`	`=IF(AND(B11>=B$2*(1-B$5),B11<=B$2*(1+B$5)),1,0)`

Then drag down all 3 columns to row 10010 (for 10,000 simulations).
At the bottom, use:

- `=SUM(C11:C10010)/10000` → Gives you the **probability** of accurate prediction.

🐍 2. Python Script (Monte Carlo Simulation)

Here's a ready-to-use Python script you can run locally or in Jupyter:

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
true_area = 223876
true_conc = 0.06
calib_factor = 2.25e-7
noise_percent = 0.05
error_margin = 0.10
simulations = 10000

# Simulate area with noise
np.random.seed(42)
```

```
simulated_areas = true_area * (1 + np.random.uniform(-noise_percent, noise_percent,
simulations))

# Predict concentrations
predicted_concs = calib_factor * simulated_areas

# Calculate bounds
lower = true_conc * (1 - error_margin)
upper = true_conc * (1 + error_margin)

# Count predictions within bounds
successes = np.sum((predicted_concs >= lower) & (predicted_concs <= upper))
probability = successes / simulations

# Output and plot
print(f"Probability of predicting within ±{int(error_margin*100)}%: {probability:.2%}")

plt.hist(predicted_concs, bins=50, color='skyblue', edgecolor='black')
plt.axvline(lower, color='red', linestyle='--', label='Lower Bound')
plt.axvline(upper, color='red', linestyle='--', label='Upper Bound')
plt.title("Monte Carlo Simulation of Predicted Concentration")
plt.xlabel("Predicted Concentration")
plt.ylabel("Frequency")
plt.legend()
plt.show()
```

📈 3. Uncertainty Analysis Model Plan

To build a **full model**, here's the framework you can use:

📊 Inputs:

- Area measurements (from detector)
- Calibration factor (mean ± std)
- Measurement error distribution (Gaussian, Uniform)
- Number of simulations (e.g., 10,000)

🔄 Monte Carlo Process:

1. Sample calibration factors from a **normal distribution** (if uncertainty known).
2. Sample area measurements with noise.
3. Compute predicted concentration.
4. Compare to reference or target value.
5. Record prediction accuracy or spread.

📤 Outputs:

- Histogram of predicted concentrations
- Confidence intervals (e.g., 95% CI)
- Error rate
- Sensitivity to calibration or noise

Would you like me to convert this framework into a working Jupyter Notebook or provide formulas for a more advanced Excel model (with uncertainty distributions)?

yes

```python
# Prepare a Python script that performs Monte Carlo uncertainty analysis
monte_carlo_script = """
import numpy as np
import matplotlib.pyplot as plt

# --- Parameters ---
true_area = 223876
true_conc = 0.06
calib_mean = 2.25e-7
calib_std = 0.1e-7  # ±0.1e-7 uncertainty in calibration factor
noise_percent = 0.05
error_margin = 0.10
simulations = 10000

# --- Monte Carlo Simulation ---
np.random.seed(42)

# Sample calibration factors from a normal distribution
calib_factors = np.random.normal(loc=calib_mean, scale=calib_std, size=simulations)

# Simulate area measurements with uniform noise
simulated_areas = true_area * (1 + np.random.uniform(-noise_percent, noise_percent,
simulations))

# Calculate predicted concentrations
predicted_concs = calib_factors * simulated_areas

# Define acceptable error bounds
lower_bound = true_conc * (1 - error_margin)
upper_bound = true_conc * (1 + error_margin)

# Compute probability of being within bounds
within_bounds = (predicted_concs >= lower_bound) & (predicted_concs <= upper_bound)
probability = np.mean(within_bounds)

# Print result
print(f"Probability of accurate prediction within ±{int(error_margin*100)}%: {probability:.2%}")

# Plot results
plt.figure(figsize=(8, 5))
plt.hist(predicted_concs, bins=50, color='skyblue', edgecolor='black')
plt.axvline(lower_bound, color='red', linestyle='--', label='10% Lower Bound')
plt.axvline(upper_bound, color='red', linestyle='--', label='10% Upper Bound')
plt.title('Monte Carlo Simulation with Calibration Factor Uncertainty')
plt.xlabel('Predicted Concentration')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
"""

# Save to downloadable Python file
file_path = "/mnt/data/monte_carlo_uncertainty_model.py"
with open(file_path, "w") as f:
    f.write(monte_carlo_script)

file_path
```

python

```python
# Re-run after environment reset: regenerate and save the Python Monte Carlo simulation script
monte_carlo_script = """
import numpy as np
import matplotlib.pyplot as plt

# --- Parameters ---
true_area = 223876
true_conc = 0.06
calib_mean = 2.25e-7
calib_std = 0.1e-7  # ±0.1e-7 uncertainty in calibration factor
noise_percent = 0.05
error_margin = 0.10
simulations = 10000

# --- Monte Carlo Simulation ---
np.random.seed(42)

# Sample calibration factors from a normal distribution
calib_factors = np.random.normal(loc=calib_mean, scale=calib_std, size=simulations)

# Simulate area measurements with uniform noise
simulated_areas = true_area * (1 + np.random.uniform(-noise_percent, noise_percent,
simulations))

# Calculate predicted concentrations
predicted_concs = calib_factors * simulated_areas

# Define acceptable error bounds
lower_bound = true_conc * (1 - error_margin)
upper_bound = true_conc * (1 + error_margin)

# Compute probability of being within bounds
within_bounds = (predicted_concs >= lower_bound) & (predicted_concs <= upper_bound)
probability = np.mean(within_bounds)

# Print result
print(f"Probability of accurate prediction within ±{int(error_margin*100)}%: {probability:.2%}")

# Plot results
plt.figure(figsize=(8, 5))
plt.hist(predicted_concs, bins=50, color='skyblue', edgecolor='black')
plt.axvline(lower_bound, color='red', linestyle='--', label='10% Lower Bound')
plt.axvline(upper_bound, color='red', linestyle='--', label='10% Upper Bound')
plt.title('Monte Carlo Simulation with Calibration Factor Uncertainty')
plt.xlabel('Predicted Concentration')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
"""

# Save to a downloadable file
file_path = "/mnt/data/monte_carlo_uncertainty_model.py"
with open(file_path, "w") as f:
    f.write(monte_carlo_script)

file_path
```

Result

```
'/mnt/data/monte_carlo_uncertainty_model.py'
```

Here's your downloadable Python script for Monte Carlo simulation with uncertainty modeling:

📥 Download monte_carlo_uncertainty_model.py

You can run this in any Python environment with **`numpy`** and **`matplotlib`** installed.

Let me know if you'd like:

- A version that reads data from a CSV/Excel file
- Support for multi-component calibration
- Or the advanced Excel template too—I can build that next!