SAND2024-11988

SANDIA REPORT
SAND2024-xxxx
Printed September 2024

Sandia National Laboratories

MAPIT User's Guide: v1.4.6-beta

Nathan Shoman

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

ACKNOWLEDGMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

This work is supported by the DOE/NE Office of Materials and Chemical Technologies (NE-43) - Materials Protection, Accounting, and Control Technologies (MPACT) program.

MPACT
Materials Protection Accounting
and Control Technologies

This page intentionally left blank.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

C/S	Containment and Surveillance
CFR	Code of Federal Regulations
DA	Destructive Assay
FAP	False Alarm Probability
ID	Inventory Difference
KMP	Key Measurement Point
LEU	Low Enriched Uranium
LWR	Light Water Reactor
MB	Material Balance
MBA	Material Balance Area
MBP	Material Balance Period
MTHM	Metric Ton Heavy Metal
MTIHM	Metric Ton Initial Heavy Metal
MUF	Material Unaccounted For
NDA	Non-destructive Assay
NMA	Nuclear Material Accountancy
NRC	Nuclear Regulatory Commission
PD	Probability of Detection
SEID	Standard Error of Inventory Difference
Sigma MUF	Standard Error of MUF
SITMUF	Standardized Independent Material Unaccounted For
SQ	Significant Quantity

1. INTRODUCTION

MAPIT (Material Accountancy Performance Indicator Toolkit) is a Python package designed to aid in developing effective material accountancy systems for bulk nuclear facilities. The inherit flexibility is designed to allow safeguards practitioners to ask the ”what if?” questions while providing transparency into commonly employed statistical tests.

MAPIT provides both a graphical user interface (GUI) and an application program interface (API). The API can be used with other Python libraries to extend functionality and integrate with other analytical workflows.

The purpose of this guide is to serve as an introduction to practical usage of MAPIT and it’s underlying principles. This guide is *not* intended to be an comprehensive guide to safeguards or material accountancy. The reader is encouraged to review suggestions for additional reading in the theory guide (Section 2) for further understanding. Guidelines such as INFCIRC/153 [1], DOE Order 0474.2 [2], and 10 CFR part 74 [3] are good places to start for understanding the regulatory aspects of safeguards and accountancy.

The MAPIT authors will endeavour to update this guide on a semi-regular basis as major changes are made to MAPIT. Additionally, the content of this guide will be available in a web format as part of our public repository. The web format will be the always be the most up-to-date version given the challenges with documenting an entire code base as parts of the web format are automatically generated. Further, the web format does a better job of displaying code and allowing users to copy and paste parts of examples.

Finally, please note that MAPIT has a permissive license, so feel free to use MAPIT for your own applications as long as you acknowledge our work. We want MAPIT to improve accessibility of safeguards to as many interested users as possible. If you have any suggestions/bugs, feel free to open an issue, pull request, or send us an email at MAPIT-dev@sandia.gov.

2. THEORY GUIDE

This section contains the mathematical context for the statistical tests used for material accountancy. The goal is to tie the code of MAPIT to the mathematical frameworks that drive modern accountancy practices. While the specifics of accountancy goals may vary depending on the regulatory stakeholder, the objective is similar; to detect unauthorized removal of accountable material. MAPIT contains statistical routines that are used by many stakeholders. Even if a particular test of interest is not included in MAPIT, the API makes it easy to use core MAPIT routines in an arbitrary Python environment to perform extended analyses using outside libraries.

First, the error model for measurements made for accountancy purposes is introduced. Next, key statistical quantities such as MUF/ID and σMUF/SEID are introduced. Finally, several statistical tests and transforms used to detect material loss are described.

2.1. Error model

The error model in MAPIT is implemented by `MAPIT.core.Preprocessing.SimErrors`. However, this function is not intended to be used standalone through direct calls, rather, it is designed to be called through the `MBArea` class of `StatsProcessor`. The `MBArea` class can recalculate errors using the `calcErrors()` method.

2.1.1. Historical context

The multiplicative error model used in MAPIT was chosen based on the widespread and commonplace use within the IAEA. It was recognized that as early as the IAEA's founding in 1957 that there would be a need to account for nuclear material in facilities [4, 5]. This created the subsequent need for statistical methods to estimate uncertainties in measurements. One key component for the selection of an error model is the necessity to propagate error from many measurements for downstream statistical analyses. The multiplicative error model arose from the need to propagate measurements and perform top-down (i.e., empirical) uncertainty quantification using both in-field IAEA measurements and operator declared measurements. The specific values used in the multiplicative error models are determined using a variety of techniques that changes based on the context of the underlying data. We refer the reader to supplementary reading, Section 2.1.4 for more information on the history and determination of the multiplicative error model.

2.1.2. Theory

No measurement, except counting, is completely accurate which results in a non-zero measurement error. This is the reason that robust statistical analysis of the material balance is required; if material in a facility were exactly known, detecting loss of that material would be trivial. Statistical analyses in material accountancy often assumes a multiplicative error model (describe in the following equation)

$$\tilde{x}_t = x_t(1 + r_t + s_t)$$

Where:

- $\tilde{x}_t$ is the observed value (i.e., what is actually measured) at time t
- x_t is the true, but unknowable value at time t
- r_t is relative random error of x
  - Specifically r_t is a random variate of the distribution $\mathcal{N}(0, \delta_r^2)$ where δ_r^2 is the random relative standard deviation.
- s_t is the relative systematic error of x
  - Specifically, s_t is a random variate of the distribution $\mathcal{N}(0, \delta_s^2)$ where δ_s^2 is the systematic relative standard deviation.

Here, random error refers to sources of error that can be reduced through repeated measurements of the same item. Systematic errors refers to short-term biases that are generally irreducible. These systematic biases can arise from a variety of sources such as calibration errors. Regardless of the measurement type (random or systematic), errors are characterized by a mean zero normal distribution with non-zero standard deviation. The distributions characterizing the random and systematic error can vary based on a variety of factors such as measurement type, measurement system, and even the specific isotope measured.

Tip

Systematic errors behave as a bias. Consequently, the systematic variate, S_t, applied to the true value x_t, from the multiplicative model described above is not updated at every time step. This contrasts with the random variate which *is* updated at each time step. The systematic variate is held constant and only updated on a periodic basis that corresponds to a specified calibration period. The details of the calibration period are measurement system specific.

Important

As of version 1.4.6, there is no functionality to specify a calibration period directly. The API can be combined with data manipulation to simulate different calibrations by slicing the data and passing it to `MAPIT.core.Preprocessing.SimErrors`.

By default, MAPIT assumes a single calibration for the length of the dataset that does not vary with time. For example, a time series with 1000 steps will be assumed to have a single systematic variate, drawn from a distribution described by the user supplied error matrix, that is applied to every time step and does not change with time. A new feature is planned for FY25 that will add the capability to specify a calibration period.

2.1.3. *Implementation*

The multiplicative error model described in the introduction assumes that there is a single iteration and location. For example, the model in the introduction might express the simulated error for a single input key measurement point. There might be multiple key measurement points in practice resulting an error model with location l and time t such that:

$$\tilde{x}_{l,t} = x_{l,t}(1 + r_{l,t} + s_{l,t})$$

It is often desireable to consider simulated statistics and calculate error for multiple iterations to help determine performance statistics of a safeguards system even if, in practice, only a single iteration is measured. The multiplicative error model can be further expanded such that an iterative dimension, n, that reflects independent draws of the underlying random and systematic error distributions, is also considered. Note that there is no iteration added to the unobservable true value, $x_{l,t}$ as it is not a random variate.

$$\tilde{x}_{n,l,t} = x_{l,t}(1 + r_{n,l,t} + s_{n,l,t})$$

For simplicity, assume that the systematic error does not have a calibration period and applies for an entire iteration (i.e., the same bias is applied for all time steps of an iteration). One naive implementation of the multiplicative model might then be as follows:

```
for n in range(len(iterations)):
    for p in range(len(locations)):
        sysError = np.random.normal(loc = 0, scale = sysSTD)

        for t in range(len(timesteps)):
            randomError = np.random.normal(loc = 0, scale = randSTD)
            x_observed[n, p, t] = x_true[p, t] * (1 + randomError + sysError)
```

This approach is valid, but scales poorly. MAPIT vectorizes both the iteration and time step dimension on a per location basis. Each location might have a different sample rate, so it is not possible to develop a fully vectorized implementation. The multiplicative model, in a vectorized form, can first be expressed as follows when vectorizing the time dimensions:

$$\widetilde{\boldsymbol{x}}_{\boldsymbol{n,l}} = \boldsymbol{x}_l(1 + \boldsymbol{r}_{n,l} + \boldsymbol{s}_{n,l})$$

Additionally, the iterative dimension can be vectorized resulting in the following:

$$\widetilde{\boldsymbol{X}}_{\boldsymbol{l}} = \boldsymbol{x}_l(1 + \boldsymbol{R}_l + \boldsymbol{S}_l)$$

Note

While the random error component, $R_{n,l,t}$ is sampled at every time step, sensor setup might complicated the specification of the systematic error component. It is assumed here that the systematic error changes with location but *not* time as no calibration time is assumed ($S_{n,lt=0} = S_{n,l,t=50} = S_{n,l,t=t}$).

The implementation of this error model is performed in MAPIT by `MAPIT.core.Preprocessing.SimErrors` starting on line 352:

```
352 def SimErrors(rawData, ErrorMatrix ,iterations, GUIObject=None, doTQDM=True,
     ↪  batchSize=10, dopar=False, bar=None):
```

The function generates `iterations` simulated realizations of measurements (i.e., the iteration dimension n) for each list entry. Each `iteration` represents a possible result of measuring at the specific key measurement point (i.e., location dimension p) represented by the list entry.

First, a list of arrays is initialized to hold the errors calculated by the function (Lines 408-409).

```
408     for i in range(0, len(rawData)):
409         AppliedError.append(np.zeros((iterations,
             ↪  rawData[i].shape[0]),dtype=np.float32))
```

Each entry in the `AppliedError` list is an array with shape (iterations n, time steps t) and *presumably* refers to a different location in a measurement type. For example, the first entry in the list might be a (time steps t, 1) shaped array of data collected at the first input key measurement point. Since each list entry might have a different number of time steps, the arrays are stored in a list rather than being concatenated. The list and constituent arrays are preinitalized.

The main calculation loop occurs between lines 442 and 479:

```
442 for j in range(0,outerloop):
443   startIdx = j*batchSize
444   endIdx = startIdx+batchSize
445   sysRSD = np.random.normal(size=(batchSize,1,1), loc=0,
       ↪  scale=ErrorMatrix[i,1])
446   randRSD = np.random.normal(size=(batchSize,rawData[i].shape[0],1), loc=0,
       ↪  scale=ErrorMatrix[i,0])
447   AppliedError[i][startIdx:endIdx,] = rawData[i][:,0].reshape((1,-1)) *
       ↪  (1+sysRSD+randRSD).reshape((batchSize,-1))
448
449   if GUIObject is not None:
450     totalloops = (outerloop+1)*len(rawData)
451     GUIObject.progress.emit(loopcounter / totalloops*100)
452     loopcounter+=1
453
454   if doTQDM and not dopar:
455     pbar.update(1)
456
```

```
if remruns > 0:
  sysRSD = np.random.normal(size=(remruns,1,1), loc=0,
   ↪  scale=ErrorMatrix[i,1])
  randRSD = np.random.normal(size=(remruns,rawData[i].shape[0],1), loc=0,
   ↪  scale=ErrorMatrix[i,0])
  AppliedError[i][endIdx:,] = rawData[i][:,0].reshape((1,-1)) *
   ↪  (1+sysRSD+randRSD).reshape((remruns,-1))

  if GUIObject is not None:
      totalloops = (outerloop+1)*len(rawData)
      GUIObject.progress.emit(loopcounter / totalloops*100)
      loopcounter+=1

  if doTQDM and not dopar:
    pbar.update(1)

else:
  sysRSD = np.random.normal(size=(iterations,1,1), loc=0,
   ↪  scale=ErrorMatrix[i,1])
  randRSD = np.random.normal(size=(iterations,rawData[i].shape[0],1), loc=0,
   ↪  scale=ErrorMatrix[i,0])
  AppliedError[i] = rawData[i][:,0].reshape((1,-1)) *
   ↪  (1+sysRSD+randRSD).reshape((iterations,-1))

  if GUIObject is not None:
    GUIObject.progress.emit(i/len(rawData)*100)

  if doTQDM and not dopar:
    pbar.update(1)
```

`SimErrors` has a parameter `batchSize` that controls the number of `iterations` that are calculated at once. The most efficient implementation would calculate all `iterations` at once using a single matrix calculation. However, this could consume more memory than available in some scenarios, so the `batchSize` parameter is provided. The code tries to batch `iterations` into `batchSize` chunks. If `iterations` is not equally divisible by `batchSize`, then an extra `remRuns` sized calculation is performed after the all `iterations/batchSize` chunks are calculated.

MAPIT specifically uses vectorized representations to more efficiently calculated the simulated error model. The sections below map the model components to the relevant code expressions.

Model Component	Code Expression
$\boldsymbol{r}_{n=1:\text{batch},p}$	`randRSD = np.random.normal(size=(batchSize, rawData[i].shape[0], 1), loc=0, scale=ErrorMatrix[i,0])`
$\boldsymbol{s}_{n=1:\text{batch},p}$	`sysRSD = np.random.normal(size=(batchSize, 1, 1), loc=0, scale=ErrorMatrix[i,1])`
$\boldsymbol{x}_p$	`rawData`
$\widetilde{\boldsymbol{x}}_{n=1:\text{batch},p}$	`AppliedError[i][startIdx:endIdx,] = rawData[i][:,0].reshape((1,-1)) * (1+sysRSD+randRSD).reshape((batchSize,-1))`

Important

Note that `sysRSD` has a shape of 1 for dim 1 (i.e., `sysRSD.shape[1] = 1`) as a single random variate from the underlying distribution is applied to all time steps for location l. Numpy broadcasting ensures that the shape of is the same for all. This will be configurable in a future update.

Additional code present in this section is used to support GUI changes, such as updating of a progress bar update, and is not important to the core multiplicative error model calculation.

2.1.4. Further reading

- Statistical error model-based and GUM-based analysis of measurement uncertainties in nuclear safeguards - a reconciliation [5]
    - Discussion about the development of the multiplicative error model and how UQ models like GUM relate
- International Target Values for Measurement Uncertainties in Safeguarding Nuclear Materials [6]
    - Reference values for δ_r and δ_s for different measurement systems and materials
- Near Real Time Accountancy (IAEA STR-403)
    - Finalized but not yet released by IAEA at time of writing
- Handbook of Nuclear Engineering: Proliferation Resistance and Safeguards [7]
    - Specifically the section on “Statistics for Accountancy”

2.2. MUF and Sigma MUF

MUF and σMUF calculations are implemented by `MAPIT.core.StatsTests.MUF` and `MAPIT.core.StatsTests.SEMUF` respectively. These functions are not intended to be called directly, rather, the intended usage is by calling the `calcMUF()` and `calcSEMUF()` methods of the `MBArea` class.

2.2.1. Historical context

Fulfilling safeguards regulations and agreements requires demonstrating that nuclear material has not been lost, removed, or otherwise been unaccounted. Both containment and surveillance methods in addition to direct accountancy of nuclear material is used to meet these requirements. Containment and surveillance (C/S), as the name implies, is used to contain (e.g., seal containers) and surveil (e.g., optical cameras) nuclear material which provides continuity of knowledge. C/S is complemented by quantitative material accountancy which seeks to quantify the amount and form of nuclear material in a given area. MAPIT focuses on providing tools and routines used in accountancy of nuclear material. The material balance, discussed in depth here, is the cornerstone of nuclear material accountancy. For a longer overview of the goals and safeguards and C/S or the history of material accountancy, readers are encouraged to check the further reading, Section 2.2.8, as a detailed history of these topics are out of scope for this document.

2.2.2. Theory: MUF

Accountancy of nuclear material is of interest to many regulatory bodies. One principle quantity used to ensure material is accounted for is the material balance calculation. This is sometimes also called Material Unaccounted For (MUF) or Inventory Difference (ID). Material balance calculations are performed over defined physical areas of a nuclear facility, called material balance areas (MBAs), at regular intervals called material balance periods (MBPs). There are a variety of metrics and criteria used to determine both the MBA (in both structure and quantity) for a given facility and the associated material balance period. Discussion of MBA selection criteria will not be discussed here and we refer the reader to the references (Section 2.2.8) for further inquiry. The material balance can be calculated by understanding the inputs, inventories, and outputs for a given material balance area. First, let the material balance period be represented as a non-zero, positive real integer:

$$\text{mbp} \in \mathbb{R}_*^+$$

Next, consider the sequence of material balance period values:

$$\text{MBP} = \{1 * \text{mbp}, 2 * \text{mbp}, ..., \}$$

Then the ith material balance can be calculate as follows:

$$\mathrm{muf}_i = \sum_{l \in l_0} \sum_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{t,l} - \sum_{l \in l_1} \sum_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{t,l} - \sum_{l \in l_2} (C_{i,l} - C_{i-1,l}) \tag{2.1}$$

> **Note**
>
> muf is calculated on a *per material basis*. For example, a material balance for uranium and plutonium would be calculated independently.

Alternatively, if continuous flows are present (i.e., continuous inputs and outputs), then the material balance can be represented as:

$$\mathrm{muf}_i = \sum_{l \in l_0} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{t,l} - \sum_{l \in l_1} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{t,l} - \sum_{l \in l_2} (C_{i,l} - C_{i-1,l}) \tag{2.2}$$

- $I_{t,l}$ is the input to the material balance area at time t and location l
- $O_{t,l}$ is the output of the material balance area at time t and location l
- $C_{i,l}$ is the inventory at time MBP_i and location l
  - C was chosen to denote *container* and avoid overloaded notation between inventory and input terms

> **Note**
>
> Location refers to a location within a material balance area. A balance area with 3 tanks might then have 3 inventory terms, once for each tank location. Location l is analogous to a key measurement point (KMP).

The material balance has three primary terms; input, output, and inventory. For each term, there may be multiple different locations so l_0, l_1, l_2 are used to denote the set of different measurement locations for input, output, and inventory, respectively. Correspondingly, the sum $\sum_{l \in l_0}$ indicates that the quantity should be summed over all input locations. The material balance simply sums all inputs over all locations during the material balance period, sums all the outputs over all locations during the material balance period, and takes the difference of inventory terms between the previous and current balance times for all locations. These terms are then used to calculate input - output - change in inventory.

There is a separate expression for material balances with discrete items versus continuous flows. The expression for the latter uses integrals over the time period of interest to denote that these quantities are usually integral rather than summed. For example, discrete canisters of material arriving to a material balance area might have their contents weighed, assayed, and summed for the material balance period. A continuous input might be a flow measured in mass per unit time, which would be integrated over the material balance period instead of summed.

Note

Material balances are calculated at discrete intervals of time and are consequently not continuous. There are a variety of ways to represent the material balance graphically. In MAPIT, we opt to have a continuous representation by holding the value calculated at $t = \text{MBP}_i$ to $t = \text{MBP}_{i+1}$ at which point the value is updated. This representation results in a step-like representation which can be seen in the figure below.

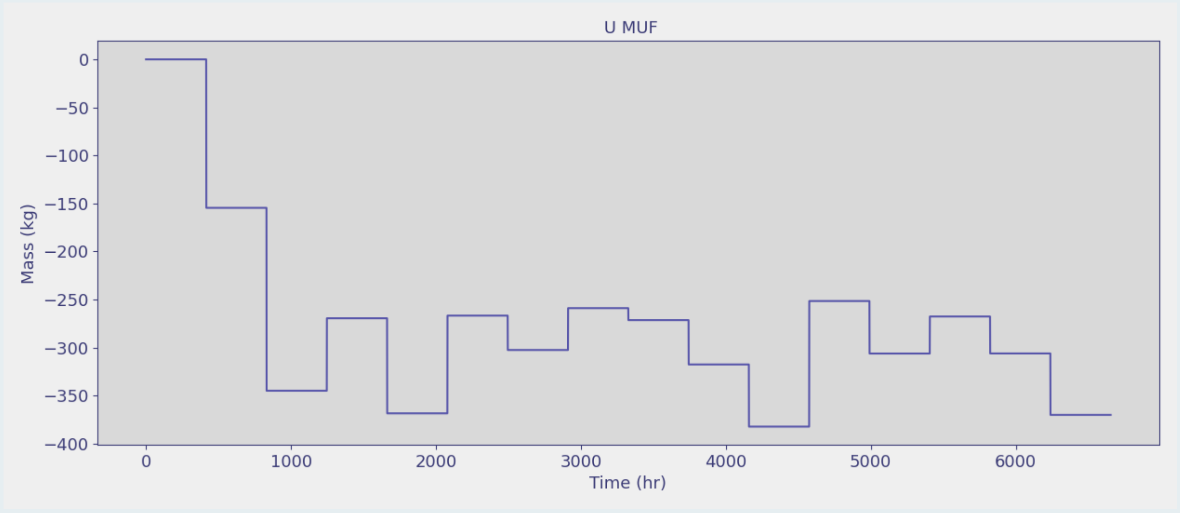


Figure 2-1: Single iteration of a material balance calculation

The material balance is intuitive and does not make assumptions about potential material loss pathways. The material balance should be exactly zero due under normal operating conditions as all material should be accounted for. In contrast, the material balance should be non-zero under anomalous conditions that cause material losses or gains that are not measured. However, the material balance will always be non-zero when bulk items are measured and included in the balance, even under normal operating conditions, due to the presence of measurement error. Additional analyses are then required to detect material loss in the presence of measurement uncertainty.

It is useful to express a series of MUF values as a sequence which facilitates various trend testing and statistical analyses:

$$\mathbf{muf} = \{\text{muf}_0, \text{muf}_1, ...\text{muf}_n\} \tag{2.3}$$

The MUF sequence can be represented as a Gaussian distribution as, over long enough periods of time and enough samples, even measurement biases behave as random errors. The fundamental goal of material accountancy then is to detect a *shift* in the distribution of values in a MUF sequence. Consider the two normal distributions below with different means and a standard deviation of one. This shift in the mean between two distributions represents behavior that would occur during an anomalous operation of a facility. However, the mean shift between the distribution would be difficult given that the shift is small compared to the standard deviation of the distributions.

Tip

You can verify this yourself by calculating an odds ratio using Bayesian principles. That is, for two normal distributions $\mathcal{N}(\mu_1, \sigma_1^2), \mathcal{N}(\mu_2, \sigma_2^2)$, the odds of belonging to $\mathcal{N}(\mu_1, \sigma_1^2)$ over $\mathcal{N}(\mu_2, \sigma_2^2)$ (assuming equal odds of MUF value being normal or off-normal) can be calculated as the ratio of the two PDFs. While this isn't an exactly true statement (the odds of a MUF value having equal odds of being normal or off-normal isn't quite true), this is a simple way to illustrate the relationship between the shift in the mean of a MUF sequence and the standard deviation.

See also The guided exercises, Section 5, included in the documentation contain concrete examples of the impact of measurement error on the MUF distribution and overall detection probability.

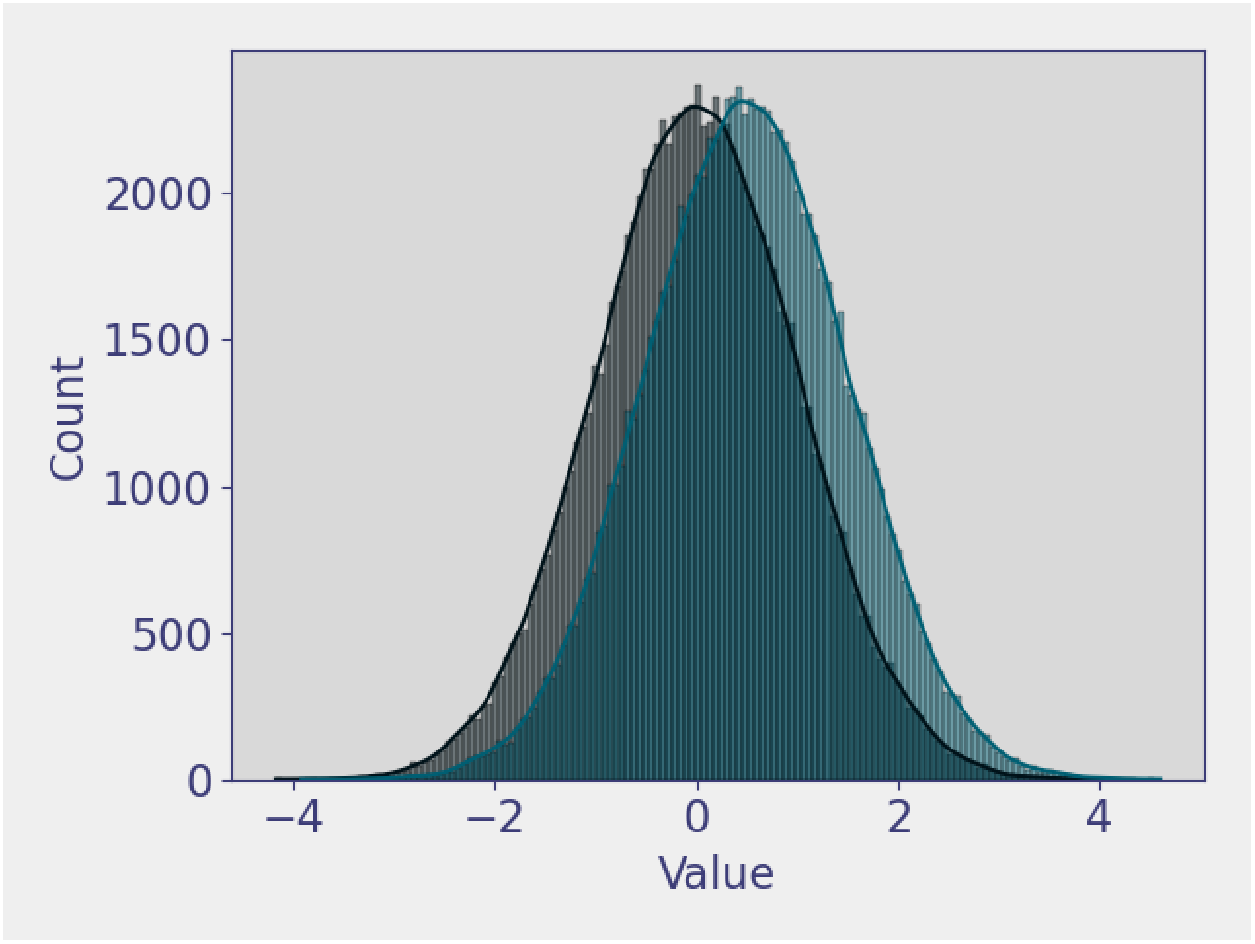


Figure 2-2: Normal distributions with small relative mean shift compared to uncertainty

Now consider the same mean shift, but with a smaller standard deviation for both distributions. The overlap between the distributions has become notable smaller which makes discriminating between the two distributions easier. Any loss pattern, and thus the underlying anomalous MUF distribution, would be difficult to quantify in practice. However, this example should make clear that the uncertainty of the MUF sequence is an important factor in detecting material loss. There are a variety of techniques that can be applied to the MUF sequence to monitor for anomalous behavior, but all techniques are ultimately limited by the uncertainty.

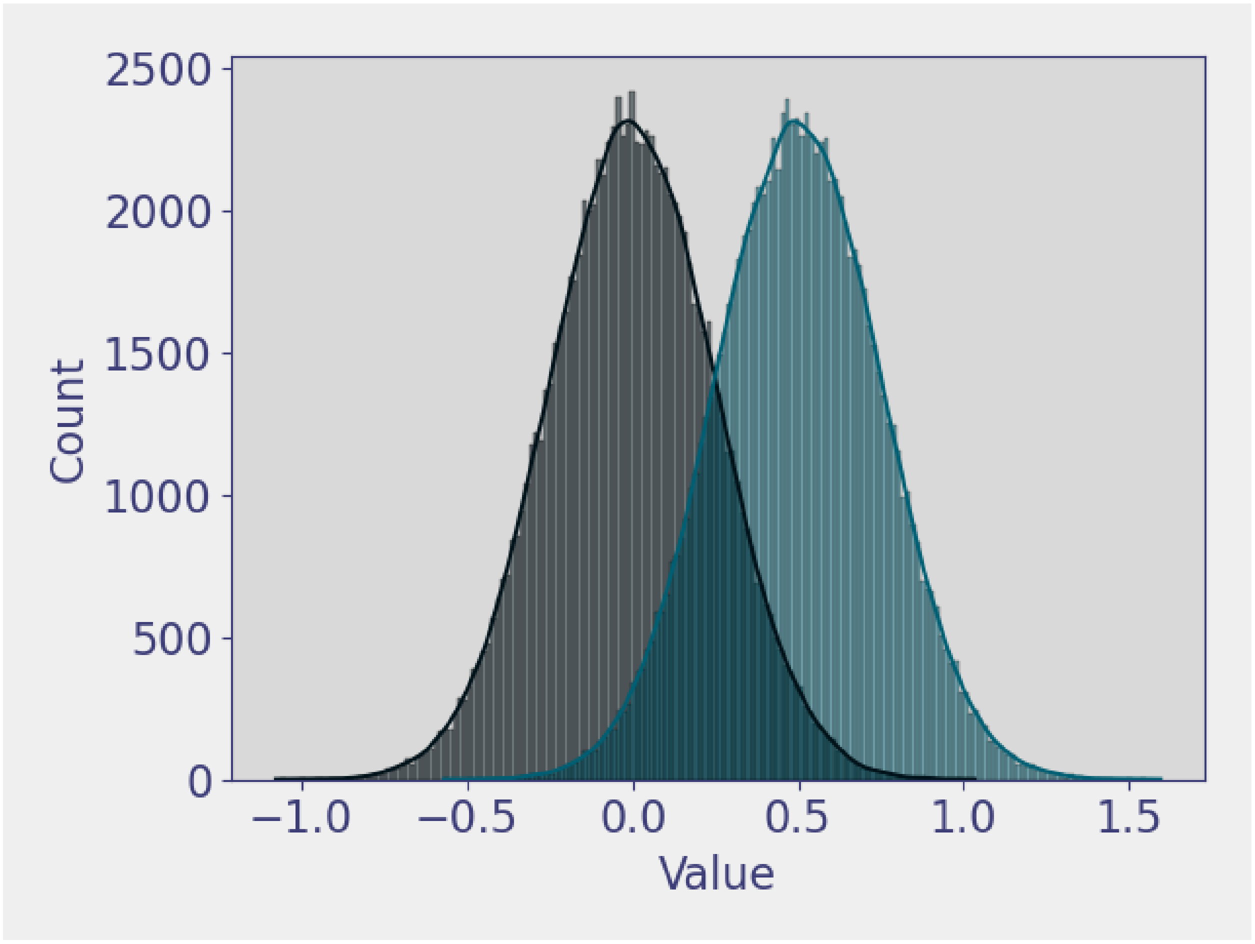


Figure 2-3: Differentiating between distributions becomes easier as the mean shift increases or the uncertainty decreases

2.2.3. Theory: Sigma MUF

σMUF, or the uncertainty in the MUF sequence, is an important metric given it's impact on the probability of detection for anomalous conditions. It is important to quantify σMUF given the connection to probability of detection. Regulatory stakeholders often implement limits on σMUF for facilities. The following derivation below describes the analytical expression for σmuf.

Note

For simplicity, the following derivation shows the calculation of a single entry of the σMUF sequence (i.e., σMUF_i). This calculation would need to be performed at each balance period to form the full σMUF sequence. Note that the σMUF sequence is as follows:
$\sigma\text{MUF} = \{\sigma\text{muf}_0, \sigma\text{muf}_1, \ldots, \sigma\text{muf}_n\}$

$$\sigma \mathrm{muf}_i^2 = \mathrm{var}(\mathrm{muf_i})$$

For simplicity we introduce the notation of $\sum_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{l,t} = I_l^*$ and $\sum_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{l,t} = O_l^*$ for the case of discrete input and outputs to denote the *total* input and output over the i-th material balance. It follows that

$$\sigma \mathrm{muf}_i^2 = \mathrm{var}\left(\sum_{l \in l_0} I_l^* - \sum_{l \in l_1} O_l^* - \sum_{l \in l_2} (C_{i,l} - C_{i-1,l})\right) \tag{2.4}$$

Substituting in the multiplicative error model for each term and starting with the input:

$$\begin{aligned}
\mathrm{var}\left(I_l^*\right) &= \mathrm{var}(I_{\mathrm{true},l}^* * (1 + R_i + S_i)) \\
&= \mathrm{var}(I_{\mathrm{true},l}^* + I_{\mathrm{true},l}^* R_i + I_{\mathrm{true},l}^* S_i) \\
&= \mathrm{cov}(I_{\mathrm{true},l}^* + I_{\mathrm{true},l}^* R_i + I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^* + I_{\mathrm{true},l}^* R_i + I_{\mathrm{true},l}^* S_i) \\
&= \mathrm{cov}(I_{\mathrm{true},l}^*, I_{\mathrm{true},l}^*) + \mathrm{cov}(I_{\mathrm{true},l}^*, I_{\mathrm{true},l}^* R_i) + \mathrm{cov}(I_{\mathrm{true},l}^*, I_{\mathrm{true},l}^* S_i) \\
&\quad + \mathrm{cov}(I_{\mathrm{true},l}^* R_i, I_{\mathrm{true},l}^*) + \mathrm{cov}(I_{\mathrm{true},l}^* R_i, I_{\mathrm{true},l}^* R_i) + \mathrm{cov}(I_{\mathrm{true},l}^* R_i, I_{\mathrm{true},l}^* S_i) \\
&\quad + \mathrm{cov}(I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^*) + \mathrm{cov}(I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^* R_i) + \mathrm{cov}(I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^* S_i)
\end{aligned} \tag{2.5}$$

Note the following:

- $I_{\mathrm{true},l}^*$ behaves as a constant for a specific slice in time
    - The true input won't change at an instant in time regardless of how many times it is measured

Continuing on and zeroing out constant terms it follows that

$$\begin{aligned}
\mathrm{var}\left(I_l^*\right) &= \mathrm{var}(I_{\mathrm{true},l}^* * (1 + R_i + S_i)) \\
&= \mathrm{cov}(I_{\mathrm{true},l}^* R_i, I_{\mathrm{true},l}^* R_i) + \mathrm{cov}(I_{\mathrm{true},l}^* R_i, I_{\mathrm{true},l}^* S_i) \\
&\quad + \mathrm{cov}(I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^* R_i) + \mathrm{cov}(I_{\mathrm{true},l}^* S_i, I_{\mathrm{true},l}^* S_i)
\end{aligned}$$

It's generally assumed that the random and systematic errors are random variables, and consequently uncorrelated with each other, which leads to

$$\begin{aligned}\operatorname{var}\left(I_l^*\right) &= \operatorname{var}(I_{\text{true},l}^* * (1 + R_i + S_i)) \\ &= \operatorname{cov}(I_{\text{true},l}^* R_i, I_{\text{true},l}^* R_i) + \operatorname{cov}(I_{\text{true},l}^* S_i, I_{\text{true},l}^* S_i) \\ &= (I_{\text{true},l}^*)^2 * \operatorname{var}(R_i) + (I_{\text{true},l}^*)^2 * \operatorname{var}(S_i)\end{aligned}$$

As the true input, $I_{\text{true},l}^*$, the variance of random variate R_i, and variance of systematic variate S_i cannot be known in practice, the observed inventory, I_l^{*2}, relative random standard deviation for location l, $\delta_{R,i}$, and relative systematic standard deviation for location l, $\delta_{S,i}$, are substituted into the previous expression as an approximation leading to the final expression for the estimated variance of the input.

$$\begin{aligned}\operatorname{var}\left(I_l^*\right) =& \operatorname{var}(I_{\text{true},l}^* * (1 + R_i + S_i)) \\ &\approx (I_l^*)^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right)\end{aligned}$$

Assuming that each input location is uncorrelated, then it is possible to simply sum the variances for each location together such that

$$\operatorname{var}\left(\sum_{l \in l_0} I_l^*\right) \approx \sum_{l \in l_0} \left(\left(I_l^*\right)^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right)\right) \tag{2.6}$$

A similar exercise can be performed on the output term leading to the expression

$$\operatorname{var}\left(\sum_{l \in l_1} O_l^*\right) \approx \sum_{l \in l_1} \left(\left(O_l^*\right)^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right)\right) \tag{2.7}$$

The inventory term differs from the inputs and outputs since there is a temporal correlation between inventory i and $i-1$. Using the same reasoning as before

$$\begin{aligned}\operatorname{var}\left(C_{i,l} - C_{i-1,l}\right) = \operatorname{var}(& \\ &(C_{\text{true},i,l} + C_{\text{true},i,l} R_i + C_{\text{true},i,l} S_i) - \\ &(C_{\text{true},i-1,l} + C_{\text{true},i-1,l} R_i + C_{\text{true},i-1,l} S_i) \\ &)\end{aligned}$$

It is assumed that the constant values for inventories are uncorrelated, random and systematic errors are uncorrelated with each other, and random errors from different times are uncorrelated. *However*, systematic errors at the same location but different time, (i.e., $S_{i,l}$ and $S_{i-1,l}$) are correlated as it is assumed there is no recalibration. This results in the expression below. Note the additional highlighted covariance term that arises correlated systematic errors between successive material balance periods.

$$\begin{aligned}\operatorname{var}\left(C_{i,l}-C_{i-1,l}\right) = {} & (C_{\text{true},i,l})^2 * (R_l^2 + S_l^2) \\ & + (C_{\text{true},i-1,l})^2 * (R_l^2 + S_l^2) \\ & - 2(C_{\text{true},i-1,l})(C_{\text{true},i,l})S_l^2\end{aligned}$$

This is again approximated using the actual measured values as the true values are unobservable and is summed across locations.

$$\begin{aligned}\operatorname{var}\left(\sum_{l\in I_2}(C_{i,l}-C_{i-1,l})\right) \approx {} & \sum_{l\in I_2}(C_{i,l})^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right) \\ & + \sum_{l\in I_2}(C_{i-1,l})^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right) \\ & - \sum_{l\in I_2} 2(C_{i-1,l})(C_{i,l})(\delta_{S,l})^2\end{aligned} \tag{2.8}$$

Substituting expressions for variance of input Equation 2.6, inventory Equation 2.8, and output Equation 2.7 into the definition of σmuf from Equation 2.4 yields the final expression Equation 2.9 below. Note that σmuf is the square root of Equation 2.9 as Equation 2.9 expresses the *variance*.

$$\begin{aligned}\sigma\text{muf}_i^2 \approx {} & \sum_{l\in I_0}\left((I_l^*)^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2)\right) + \sum_{l\in I_2}\left((C_{i,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2)\right) \\ & + \sum_{l\in I_2}\left((C_{i-1,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2)\right) - \sum_{l\in I_2}\left(2 * (C_{i-1,l})(C_{i,l})(\delta_{S,l})^2\right) \\ & + \sum_{l\in I_1}\left((O_l^*)^2 * \left((\delta_{R,l})^2 + (\delta_{S,l})^2\right)\right)\end{aligned} \tag{2.9}$$

Note

This assumes one strata for each measurement. That is, an item or flow is measured once when it has the same attributes. Performing multiple measurements on the same strata will reduce the relative standard deviation terms by approximately $\frac{1}{n}$ and should be accounted for in the σmuf calculation accordingly.

Equation 2.9 can be expanded by replacing the inputs and outputs with their non-integrated quantities. This is done to better illustrate the different components that are calculated in MAPIT.

$$\sigma \text{muf}_i^2 \approx \sum_{l \in l_0} \left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} I_{l,t} \right)^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right) + \sum_{l \in l_2} \left((C_{i,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right)$$
$$+ \sum_{l \in l_2} \left((C_{i-1,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right) - \sum_{l \in l_2} \left(2C_{i-1,l} C_{i,l} (\delta_{S,l})^2 \right)$$
$$+ \sum_{l \in l_1} \left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} O_{l,t} \right)^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right) \tag{2.10}$$

2.2.4. Discussion

Much of the traditional literature and research around material balances and associated testing was developed in the 1980s. The relevant seminal papers are listed in the further reading (Section 2.2.8) portion of this document. There are a few key points from these papers that are important to note.

A natural inclination to improve performance of any testing on the material balance would be to reduce the overall uncertainty through 1) smaller material balance areas which result in smaller inventory terms, 2) more frequent material balance frequency resulting in smaller input and output terms, or 3) some combination of the two. Avenhaus and Jaech[8] considered this question and found that none of those procedures necessarily lead to better performance, and in some instances, might lead to a *lower* detection sensitivity. The work by Avenhaus and Jaech was particularly notable as it lead to several important findings:

- Considering a statistical test with maximum test power (i.e., a test with the highest probability of detection for any loss of material of a particular size) applied to a fix length of time, the **optimal test is one that ignores intermediate balance evaluations**. Put another way, the optimal test for a loss of material over a fixed period of time is one that considers the sum of all intermediate MUF values. This is the same as if no intermediate MUF values had been taken; as if the balance was conducted over the entire period of interest. **This applies to protracted, but not abrupt, losses**. Here, protracted loss is one that occurs over multiple balance periods or areas.
    - Concrete example; there might be a regulatory goal for detecting a loss within 3 months. The optimal statistical test would be a balance over a 3 month period; performing a monthly balance provides no benefit with respect to maximum detection probability of a loss over a three month period.
- Combining intermediate MUF values in some optimal way, perhaps as a weighted average, still results in a lower detection probability than a global MUF that only considers the beginning and ending states.
    - An important exception here is that this statement only applies to the *unknown* loss pattern. Performance improvements can be seen with an optimally ordered MUF sequence if the loss pattern is known, but in practice, the loss pattern is never known

- Even applying statistical tests to each intermediate MUF value and then linearly combining the results (as opposed to a test on the combined MUF values) still results in a lower detection probability than a test on a global MUF.

Further, Avenhaus and Jaech showed that not only do these statements apply to time (i.e., different material balance periods), but *also to space*. Subdividing a material balance area provides no benefit in terms of detection probability, and perhaps even decrease detection probability, compared a test on the larger material balance. These statements also assume a fixed false alarm probability.

Important

Avenhaus and Jaech's work applied *only* to the probability of detection. There are other benefits to subdividing material balances into smaller units of time or space; principally to localize a potential material loss in space or time, but this comes at a cost to detection probability, specifically for protracted losses that are split over multiple balances or areas. Again, this statement applies only to *protracted* losses, not abrupt. Avenhaus and Jaech's work only considered random errors, but Burr and Hamada [4] later went on to show that the inclusion of systematic error does not change the limitations of subdividing material balance areas (i.e., probability of detection does not increase for more frequent balances and smaller balance areas).

2.2.5. Material balance iterations

In practice, only one material balance sequence, $\mathbf{muf} = \{\mathrm{muf}_0, \mathrm{muf}_1, ...\mathrm{muf}_n\}$, can be observed. Following discussion from the error models, it is often advantageous to estimate the performance of a safeguards system by performing statistical analyses and determining probabilities of detection. It would be very difficult to estimate probability of detection for a facility using experimental data alone. However, simulation tools can help provide these estimates. MUF and σMUF can subsequently be represented as 2D matrix such that one dimension represents the MUF sequence in time and the other dimension represents different draws of random variates from the error model. Later sections will have additional discussion on the topic of this matrix representation, but for now, it is important to note this concept as this is how MAPIT calculates MUF and σMUF.

$$\begin{aligned} \mathbf{MUF} &= \{\mathbf{muf}_0, \mathbf{muf}_1, ...\mathbf{muf}_n\} \\ \boldsymbol{\sigma}\mathbf{MUF} &= \{\boldsymbol{\sigma}\mathbf{muf}_0, \boldsymbol{\sigma}\mathbf{muf}_1, ...\boldsymbol{\sigma}\mathbf{muf}_n\} \end{aligned} \tag{2.11}$$

2.2.6. MUF implementation

The MAPIT implementation of MUF is the first function in `MAPIT.core.StatsTests`. The MUF calculation requires a few key variables:

- Input, inventory, and output measurements
- Input, inventory, and output times

    - Together the measurements and times should from a time series. The time entries should represent the time at which the measurement is taken and should monotonically increase. So a measurement taken at the start is t=0 and a measurement taken one day later should be t=24 (hours) or t=1440 (minutes), etc. There are no unit requirements, but the time series should all use the same units of time.

- Material balance period
    - This should have the same units as the input, inventory, and output time

Since each measurement provided to MAPIT could potentially have a different length and/or number of time steps, we first determine the maximum time step:

```
A1 = np.max(np.asarray(list(chain.from_iterable(processedInputTimes))))
 ↪  #unroll list
A2 = np.max(np.asarray(list(chain.from_iterable(processedInventoryTimes))))
A3 = np.max(np.asarray(list(chain.from_iterable(processedOutputTimes))))


timeSteps = np.round(np.max(np.array([A1, A2, A3])))
```

MAPIT calculates the entire MUF and σMUF sequence (i.e., Equation 2.11) which results in a 2D matrix that has a shape (iterations, MBPs) where iterations is the number of iterations/draws from the error model over the total length of the data and MPBs is the total number of balance periods for the dataset.

Example

Consider the following example:

- mbp = 100
    - MBP = $\{100, 200, 300, 400, 500, 600\}$
- Largest time in dataset: 670
- Iterations: 25

MAPIT will calculate a MUF sequence that is 6 balance periods long (0:100, 100:200, 200:300, 300:400, 400:500, 500:600). Since iterations are specified to be 25, MAPIT will perform the calculation of Equation 2.3 25 times, each time, drawing a different set of random variates from the multiplicative error model with relative standard deviations that were specified by the user. The MUF value returned by MAPIT will have a shape of [25, 600].

Important

MAPIT represents periodic statistical quantities as continuous, so although there are only 6 balance periods, this is represented as 600 timesteps (once per unit time). Each material balance iteration (i.e., slice [n, 0:600]) will only have 6 unique values that are held constant between material balance updates.

MAPIT calculates the distribution of MUF values (i.e., $\mathbf{MUF} = \{\mathrm{MUF}_0, \mathrm{MUF}_1, \ldots \mathrm{MUF}_n\}$) using the equation for individual muf_i values (Equation 2.1 or 2.2). There are three dimensions that require iteration; time (the number of balance periods), locations (multiple input/inventory/output terms) and iterations (unique draws of the multiplicative error model). The only component that can be effectively vectorized is the iterations component as the potential for non-uniform sampling rates as a function of time and/or locations require consideration of each component individually. The MUF calculation still remains fairly quick in spite of a largely non-vectorized solution as the underlying computations are fairly trivial.

> At a high level, MAPIT vectorizes the error model iterations, but uses a for-loop to iterate over the location and time components.

Of the two loops (balance periods and locations), MAPIT uses the time component as the outer for-loop and locations as the inner for-loop.

> **Note**
>
> The outer time loop is indexed from `i` to `MBPs` rather than starting at zero. This facilitated more understandable indexing, particularly when slicing some of the time indices, but runs counter to the standard python notation that indices start at 0.

The outer for-loop is defined as follows:

```
for i in range(1, int(MBPs)):  #each MBP
```

The outer loop calculates the muf_i (assuming that the input and outputs are flows):

$$\mathrm{muf}_i = \sum_{l \in l_0} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{t,l} - \sum_{l \in l_1} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{t,l} - \sum_{l \in l_2} (C_{i,l} - C_{i-1,l})$$

The **summation component** represents the inner for-loop and is split across three separate loops, each representing the measurement type:

```
for j in range(0, len(inputAppliedError)):
```

```
for j in range(0, len(outputAppliedError)):
```

```
for j in range(0, len(inventoryAppliedError)):
```

The **integral component** assumed to be flows in units of mass/time, need to be integrated before they can used in the balance calculation. This is done using a customized trapezoidal integration routine, `MAPIT.core.AuxFunctions.trapSum`, which is explained in more depth in the computational considerations (Section 2.6) part of this guide.

The relevant segment of time corresponding to $[\mathrm{MBP}_{t-1}, \mathrm{MBP}_t]$ must be determined before the integration is performed. This is represented by the `logicalInterval` variable (note that the interval for the input terms is shown, but it is also calculated for the output terms):

```python
logicalInterval = np.logical_and(
    processedInputTimes[j] >= MBP * (i - 1),
    processedInputTimes[j] <= MBP * i).reshape((-1,))
```

The relevant times for the current balance period must also be identified for the inventory, but the procedure is easier as only the start and end points of the time series need to be identified, not all values in the interval. This is because the inventories, already assumed to be in the correct mass units, do not need to be integrated and can be subtracted (i.e., $C_{i,l} - C_{i-1,l}$).

Finally, all of the components are combined and MAPIT iterates over material balance periods and locations to calculate **MUF**.

2.2.7. *Sigma MUF implementation*

Important

The final expression for σMUF derived in Equation 2.10 includes a covariance term that accounts for the shared systematic bias across two successive balance periods (assuming no recalibration). However, **the MAPIT implementation of σMUF does not include the shared covariance term**. In practice, σMUF is calculated as a sum of squared errors and neglects covariance. MAPIT consequently uses this more conservative estimate to be better aligned with the state of practice in material accountancy.

The implementation of σMUF follows much of the logic used in the material balance calculation;

- The entire sequence from Equation eq‘mufdist‘ is calculated
- Input, inventory, and output terms are considered separately
- Time components are vectorized whereas locations and balance periods are expressed in explicit floor loops

The structure of the calculation is identical to that of MUF. The key difference is the quantity calculated, here $\sigma\mathrm{muf}_i$ is calculated, instead of muf_i. Recall that $\sigma\mathrm{muf}_i$ is calculated as follows (noting that we are ignoring the covariance term):

$$\sigma \text{muf}_i^2 \approx \sum_{l\in l_0}\left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} I_{l,t}\right)^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right) + \sum_{l\in l_2}\left((C_{i,l})^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right)$$
$$+ \sum_{l\in l_2}\left((C_{i-1,l})^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right)$$
$$+ \sum_{l\in l_1}\left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} O_{l,t}\right)^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right)$$

Following from the muf calculation, integral terms are integrated using `MAPIT.core.AuxFunctions.trapSum` with locations and balance periods iterated over using a for loop. It is often desireable to track the contribution to σMUF by component, so MAPIT rearranges each term slightly to better track each component. For example, the inventory term is expressed as follows:

$$\sum_{l\in l_2}\left(((C_{i,l})^2+(C_{i-1,l})^2) * (\delta_{R,l})^2 + ((C_{i,l})^2+(C_{i-1,l})^2) * (\delta_{S,l})^2\right)$$

Here, the random and systematic contributions can be tracked separately. The input and output terms are calculated in a similar way, but only the input term will be discussed for brevity.

First, the input term is integrated. Then the integral quantity and user supplied relative standard deviations are squared and summed. These are added together and "tiled". Variables `VR` and `VS` are a vector with length equal to iterations and must be "tiled" for all time steps in the given balance. The contribution components are then stored in a separate array. This process repeats for all locations and balance periods.

```python
AFTS = AuxFunctions.trapSum(logicalInterval, processedInputTimes[j],
 ↪  inputAppliedError[j])
VR = AFTS**2 * ErrorMatrix[j, 0]**2
VS = AFTS**2 * ErrorMatrix[j, 1]**2
#variance is stored as a function of time, but contributions are
#stored per MBP which makes it easier to put in a table later
#especially considering the time might be variable

InpVar[:,i * MBP:(i + 1) * MBP] += ((VR + VS) * np.ones((MBP,iterations))).T
SEMUFContribR[:, j, i] = VR
SEMUFContribS[:, j, i] = VS
SEMUFContribI[:, j, i] = AFTS
```

The inventory calculation proceeds in a similar manner but differs in that the first and subsequent balance period calculations differ. The previous inventory during the first balance is assumed to be zero, and rather than trying to account for that in the array by prepending a series of zeros, MAPIT simply drops that term. This is implemented by checking if the first balance period is being calculated (`i==1`), and if so, not including `inventoryAppliedError[j][:, startIdx]`.

```
if i == 1:
  VR = inventoryAppliedError[j][:, endIdx]**2 * ErrorMatrix[locMatrixRow,
   ↪  0]**2
  VS = inventoryAppliedError[j][:, endIdx]**2 * ErrorMatrix[locMatrixRow,
   ↪  1]**2

  SEMUFContribI[:, j + len(inputAppliedError), i] =
   ↪  inventoryAppliedError[j][:, endIdx]

else:

  VR = (inventoryAppliedError[j][:, startIdx]**2 +
   ↪  inventoryAppliedError[j][:, endIdx]**2) * ErrorMatrix[locMatrixRow,
   ↪  0]**2
  VS = (inventoryAppliedError[j][:, startIdx]**2 +
   ↪  inventoryAppliedError[j][:, endIdx]**2) * ErrorMatrix[locMatrixRow,
   ↪  1]**2

  SEMUFContribI[:, j + len(inputAppliedError), i] =
   ↪  inventoryAppliedError[j][:, endIdx]
```

2.2.8. *Further reading*

- Speed and Culpin [9]
- Avenhaus and Jaech [8]
- Handbook of Nuclear Engineering: Proliferation Resistance and Safeguards [7]
- Picard [10]
- Revisiting statistical aspects of NMA [4]
- Fundamentals of material accounting for nuclear safeguards [11]
- Page's test performance (Jones) [12]
    - Pages 402-408
- Page's test performance (Jones) [13]
    - Pages 19-22

2.3. CuMUF

The CuMUF test is implemented by `MAPIT.core.StatsTests.CUMUF`. However, this function is not intended to be called directly, rather, the intended usage is through the `calcCUMUF()` method of the `MBArea` class.

2.3.1. Historical context

Cumulative MUF (CuMUF) is the sum of all MUF values over a given period of time. The CuMUF test is noteworthy due to power to detect protracted losses. CuMUF in particular is strong in detecting protracted losses that occur early in the material balance sequence, but performs worse if the loss occurs later in the sequence.

> **Note**
>
> CuMUF is not the same as MUF cusum. MUF cusum is not currently implemented explicitly in MAPIT, but can be constructed using the API.

2.3.2. Theory

The CuMUF metric is calculated as follows:

$$\mathrm{CuMUF}_i = \sum_{t=0}^{i} \mathrm{muf}_t$$

2.3.3. Discussion

The CuMUF statistic is often combined with statistical testing. The most straightforward test is a simple comparison to a critical value. That is, an alarm is triggered if CuMUF rises above a specific value. CuMUF can also be used with other trend methods like control charts or further extended to develop the cusum test.

> **Note**
>
> CuMUF is the optimal statistical for detecting the worst-case loss. The worst-case loss, as originally derived by Avenhaus and Jaech, is the loss for which detection probability is minimized for the optimal statistical test. The worst-case loss was shown to be a loss wherein the per-balance loss is proportional to the row sums of the covariance of the material balance sequence. Note that this loss is the worst-case for any statistical test as it was derived using the assumption that the optimal statistical test for detection of a loss could be known and applied.

2.3.4. Implementation

The CuMUF statistic calculation is relatively straightforward and requires a minimal amount of code. The MUF array is a 2D array with shape (iterations, time steps) where time steps is on a per unit time basis. First, a discrete 1D difference is performed on the MUF array to determine the location of unique MUF value. Those locations are stored to the array `idxs`.

```
z = np.diff(MUF[0,])
idxs = np.concatenate(([0,], np.argwhere(z!=0).squeeze(),
 ↪  [int(MUF.shape[1]-1),])).astype(int)
```

The CuMUF statistic is then calculated for the sequence of MUF values. The resulting statistic is tiled across time steps to again result in a 2D array with shape (iterations, time steps). Without tiling, the shape would be (iterations, number of balance periods). After calculation, the array of statistics is returned.

```
for i in range(1,len(idxs)):
  cumuf[:,idxs[i-1]:idxs[i]] = np.tile(cumuf[:,idxs[i-1]-1] + MUF[:,idxs[i]],
                                       (int(idxs[i] -
                                        ↪  idxs[i-1]),1)).swapaxes(0,1)
```

2.3.5. *Further reading*

- Revisiting statistics for NMA [4]
- Comparison of Page's test on SITMUF to MUF and CUMUF [14]

2.4. SITMUF

The SITMUF transformation is implemented in `MAPIT.core.StatsTests.SITMUF`. This function is not intended to be used standalone through direct calls, rather, it is intended to call `calcSITMUF()` through the `MBArea` class.

2.4.1. *Historical context*

The statistical properties of the MUF sequence has been studied extensively, and as early as the 1980s, it was noted that there was correlation between successive material balance periods. Pike and Woods were the first to develop a concept called ITMUF (Independent Transformed MUF) and later SITMUF (Standardized Independent MUF). The SITMUF sequence is a transformed material balance sequence wherein the mean is approximately zero and the standard deviation is approximately one. There are two key advantages to performing statistical testing on such an independent sequence:

- Alarm thresholds for SITMUF depend only on the sequence length, not the form or properties of the MUF covariance
  - This alleviates the need to determine alarm thresholds by strictly using simulation
- In a near-real time accountancy context, the variance of SITMUF decreases as the approximate covariance of the MUF sequence approaches the true value

- Consequently, the detection probability of SITMUF increases over time, often resulting in a higher detection probability than the MUF sequence alone

The transformation from MUF to SITMUF was quite difficult until Picard showed that the SITMUF transform can be easily expressed using the Cholesky decomposition. A series of papers in the late 1980s onward showed that applying Page's trend test to SITMUF performs well for a wide range of potential material loss scenarios when the pattern is not known. Page's trend test on SITMUF has been frequently used as an effective test in nuclear material accountancy.

2.4.2. Theory

Recall that the **muf** sequence is defined as follows:

$$\mathbf{muf} = \{\mathrm{muf}_0, \mathrm{muf}_1, \ldots \mathrm{muf}_n\}$$

With

$$\mathrm{muf}_i = \sum_{l \in I_0} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{t,l} - \sum_{l \in I_1} \int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{t,l} - \sum_{l \in I_2} (C_{i,l} - C_{i-1,l})$$

It's generally assumed that since the error models are normally distributed, individual muf values (i.e., mbp_i) and the muf sequence (i.e., **muf**) will also be normally distributed. Consequently, the muf sequence can be thought of as a multivariate normal distribution such that:

$$\mathbf{muf} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

The covariance matrix contains the covariance between different material balances in the sequence. For example, consider the entry σ^2_{2n} of the covariance matrix below. This term is the variance between material balance n and 2.

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma^2_{11} & \sigma^2_{12} & \cdots & \sigma^2_{1n} \\ \sigma^2_{21} & \sigma^2_{22} & \cdots & \sigma^2_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma^2_{n1} & \sigma^2_{n2} & \cdots & \sigma^2_{nn} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\Sigma_{i-1}} & \boldsymbol{\sigma_{i-1}} \\ \boldsymbol{\sigma_{i-1}}^T & \sigma_{i,i} \end{pmatrix} \tag{2.12}$$

Note

The covariance matrix $\boldsymbol{\Sigma}$ is symmetric (i.e., $\sigma_{1,2} = \sigma_{2,1}$).

The SITMUF statistic is muf with a mean of zero and standard deviation of one. This can be initially considered through the subtraction of the sequence mean and division by the sequence standard deviation. The independent material balance sequence can be expressed by estimating the sequence mean through the conditional expectation given all previously observed muf values:

$$\text{itmuf}_i = \text{muf}_i - E(\text{muf}_i \mid \text{muf}_{i-1}, \text{muf}_{i-2}, \ldots, \text{muf}_0)$$

Then note that SITMUF is ITMUF divided by the standard deviation:

$$\text{sitmuf}_i = \frac{\text{itmuf}_i}{\sigma_{\text{itmuf}}}$$

Then using the expression for conditional expectation of the multivariate normal distribution with the covariance expression Equation 2.12 from the expression then becomes:

$$\begin{aligned}\text{sitmuf}_i &= \left(\text{muf}_i - \boldsymbol{\sigma_{i-1}}^T \boldsymbol{\Sigma_{i-1}}^{-1} \text{muf}_{i-1}\right)\\ \sigma_{\text{itmuf}} &= \sigma_{i,i} - \boldsymbol{\sigma_{i-1}}^T \boldsymbol{\Sigma_{i-1}}^{-1} \boldsymbol{\sigma_{i-1}}\end{aligned}$$

Picard [10] showed that a convenient way to calculating the above expression is through the use of the Cholesky decomposition. Since Picard fully derives the relationship between the expression above and the Cholesky decomposition, we refrain from showing that work here. The final expression for SITMUF becomes

$$\begin{aligned}\boldsymbol{\Sigma}_i &= \mathbf{C}_i \mathbf{C}_i^{\,T}\\ \text{sitmuf}_i &= \mathbf{C}_i^{-1} \text{muf}_i\end{aligned}$$

Where $\mathbf{C}_i$ is the lower triangular Cholesky factor of the covariance matrix.

The covariance matrix itself is often calculated using relative standard deviations, similar to the calculation for σmuf. In fact, the diagonal terms (i.e., $\Sigma_{1,1}, \Sigma_{2,2}, \ldots$) are the variance of the material balance (the covariance of material balance i with itself is the variance). Recall the expression that was derived from σmuf.

$$\begin{aligned}\sigma_{i,i}^2 \approx &\sum_{l \in l_0} \left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} I_{l,t} \right)^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right) + \sum_{l \in l_2} \left((C_{i,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right)\\ &+ \sum_{l \in l_2} \left((C_{i-1,l})^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right)\\ &+ \sum_{l \in l_1} \left(\left(\int_{t=\text{MBP}_{i-1}}^{\text{MBP}_i} O_{l,t} \right)^2 * ((\delta_{R,l})^2 + (\delta_{S,l})^2) \right)\end{aligned}$$

Note

The covariance term for the variance **will** be included in the covariance matrix calculation.

The off-diagonal is calculated in a similar manner, but has more terms. The off-diagonal is the covariance between material balance i and j.

$$\begin{aligned}\sigma_{i,j}^2 \approx & \sum_{l \in l_2} \left(\left(C_{i,l} C_{j,l} + C_{i-1,l} C_{j-1,l} \right) * (\delta_{S,l})^2 \right) \\ & - \sum_{l \in l_2} \left(\left(C_{i,l} C_{j-1,l} \right) * \left((\delta_{S,l})^2 + P(j-1 == i) * (\delta_{R,l})^2 \right) \right) \\ & - \sum_{l \in l_2} \left(\left(C_{i-1,l} C_{j,l} \right) * \left((\delta_{S,l})^2 + P(i-1 == j) * (\delta_{R,l})^2 \right) \right) \\ & + \sum_{l \in l_0} \left(\left(\int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_j} I_{t,l} \right) \left(\int_{t=\mathrm{MBP}_{j-1}}^{\mathrm{MBP}_j} I_{t,l} \right) (\delta_{S,l})^2 \right) \\ & + \sum_{l \in l_1} \left(\left(\int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_j} O_{t,l} \right) \left(\int_{t=\mathrm{MBP}_{j-1}}^{\mathrm{MBP}_j} O_{t,l} \right) (\delta_{S,l})^2 \right)\end{aligned}$$

Where

$$[P] \equiv \begin{cases} 0 & P == \text{false} \\ 1 & P == \text{true} \end{cases}$$

Note

The goal of the SITMUF transform is to result in an standardized independent sequenceof MUF values.

2.4.3. Discussion

The Cholesky-based approach above has the property that the variance of SITMUF decreases with time as the estimated covariance matrix approaches the true value. This is the implementation used in MAPIT, however, one could also do a yearly SITMUF transform wherein the transform was applied only after the covariance matrix was well approximated from a year's worth of material balances.

2.4.4. Implementation

The SITMUF implementation in MAPIT is particularly computationally intensive as the "NRTA" type calculation is used. In a simulation space, we can calculate the entire covariance matrix at once with all entries from all balance periods. However, a "NRTA" styled calculation performs the SITMUF transform with a reduced covariance matrix that grows as new observations are added. This results in a decrease in variance of SITMUF over time.

The covariance matrix is a NxN matrix at the N-th material balance period. Since MAPIT adopts the "NRTA" style calculation, all NxN entries must be updated at each balance period which simulates the arrival of new information. The MAPIT SITMUF calculation is not well vectorized as the NxN must be resized and calculated at each balance. The calculation starts by looping over balance periods and each entry in the covariance matrix at balance P:

```
for P in range(1,int(MBPs)):
  for j in range(0,P):
```

The variables for the different times have a different meaning than in the expressions that were defined above. This is for legacy purposes and to improve alignment with the papers. The following table describes the mapping between the derived expressions and associated code:

Model Component	Code Expression
Balance i	`I`
Balance j	`IPrime`

Table 2-1: SITMUF formula code mapings

For simplicity, the diagonal and off-diagonal terms are broken into multiple components.

Diagonal Terms

$$\begin{aligned}\sigma^2_{i,i} \approx\ & \sum\nolimits_{l\in l_0}\left(\left(\int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} I_{l,t}\right)^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right) + \sum\nolimits_{l\in l_2}\left((C_{i,l})^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right)\\ & + \sum\nolimits_{l\in l_2}\left((C_{i-1,l})^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right) - \sum\nolimits_{l\in l_2}\left(2C_{i-1,l}C_{i,l}(\delta_{S,l})^2\right)\\ & + \sum\nolimits_{l\in l_1}\left(\left(\int_{t=\mathrm{MBP}_{i-1}}^{\mathrm{MBP}_i} O_{l,t}\right)^2 * ((\delta_{R,l})^2+(\delta_{S,l})^2)\right)\end{aligned}$$

Term 1

```
for k in range(len(inputAppliedError)):
 logicalInterval = np.logical_and(processedInputTimes[k] >= IPrevious_time, processedInputTimes[k] <= I_time).reshape((-1,)) #select the indices for the relevant time
 term1 += AuxFunctions.trapSum(logicalInterval, processedInputTimes[k], inputAppliedError[k]) **2 * (ErrorMatrix[k, 0]**2 + ErrorMatrix[k, 1]**2)
```

Term 2

```
 for k in range(len(inventoryAppliedError)):
  locMatrixRow = k+len(inputAppliedError)

  startIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - IPrevious_time).argmin()
  endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - I_time).argmin()

  term3 += inventoryAppliedError[k][:,endIdx]**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)
```

Term 3

```
 for k in range(len(inventoryAppliedError)):
  locMatrixRow = k+len(inputAppliedError)


  startIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - IPrevious_time).argmin()
  endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - I_time).argmin()

  term3 += inventoryAppliedError[k][:,endIdx]**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)

if j != 0:
  for k in range(len(inventoryAppliedError)):
    locMatrixRow = k + len(inputAppliedError)
    startIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) -IPrevious_time).argmin()
    endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - I_time).argmin()

    term4 += inventoryAppliedError[k][:,startIdx]**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)
    term5 += inventoryAppliedError[k][:,startIdx] * inventoryAppliedError[k][:,endIdx] * ErrorMatrix[locMatrixRow, 1]**2
```

Term 4

Note

The factor of 2 for this term is included later in the code when the terms are added together and assigned to the covariance matrix.

```
 for k in range(len(inventoryAppliedError)):
    locMatrixRow = k+len(inputAppliedError)


    startIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - IPrevious_time).argmin()
    endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - I_time).argmin()

    term3 += inventoryAppliedError[k][:,endIdx]**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)

if j != 0:
    for k in range(len(inventoryAppliedError)):
        locMatrixRow = k + len(inputAppliedError)
        startIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) -IPrevious_time).argmin()
        endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) - I_time).argmin()

        term4 += inventoryAppliedError[k][:,startIdx]**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)
        term5 += inventoryAppliedError[k][:,startIdx] * inventoryAppliedError[k][:,endIdx] * ErrorMatrix[locMatrixRow, 1]**2
```

Term 5

```
for k in range(len(outputAppliedError)):

logicalInterval = np.logical_and(processedOutputTimes[k] >= IPrevious_time,processedOutputTimes[k] <= I_time).reshape((-1,))
locMatrixRow = k + len(inputAppliedError) + len(inventoryAppliedError)

term2 += AuxFunctions.trapSum(logicalInterval, processedOutputTimes[k], outputAppliedError[k])**2 * (ErrorMatrix[locMatrixRow, 0]**2 + ErrorMatrix[locMatrixRow, 1]**2)
```

Term 1

```
for k in range(len(inventoryAppliedError)):
    startIdx =  np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  I_time).argmin() #I
    endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrime_time).argmin()
    startIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrevious_time).argmin()
    endIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrimePrevious_time).argmin()
    locMatrixRow = k + len(inputAppliedError)

    term3a = inventoryAppliedError[k][:, startIdx] *
     ↪  inventoryAppliedError[k][:, endIdx]
    term3b = inventoryAppliedError[k][:, startIdx2] *
     ↪  inventoryAppliedError[k][:, endIdx2]
    term3c = ErrorMatrix[locMatrixRow, 1]**2
    term3 += (term3a+term3b)*term3c
```

Term 2

```
for k in range(len(inventoryAppliedError)):
    startIdx =  np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  I_time).argmin() #I
    endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrime_time).argmin()
    startIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrevious_time).argmin()
    endIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrimePrevious_time).argmin()
    locMatrixRow = k + len(inputAppliedError)

    term3a = inventoryAppliedError[k][:, startIdx] *
     ↪  inventoryAppliedError[k][:, endIdx]
    term3b = inventoryAppliedError[k][:, startIdx2] *
     ↪  inventoryAppliedError[k][:, endIdx2]
    term3c = ErrorMatrix[locMatrixRow, 1]**2
    term3 += (term3a+term3b)*term3c

    term4a = inventoryAppliedError[k][:, startIdx] *
     ↪  inventoryAppliedError[k][:, endIdx2]
    term4b = ErrorMatrix[locMatrixRow, 1]**2
    if IPrime-1 == I:
        term4c = ErrorMatrix[locMatrixRow, 0]**2
    else:
        term4c = np.zeros((iterations,))


    term4 += term4a*(term4b+term4c)
```

Term 3

```python
for k in range(len(inventoryAppliedError)):
    startIdx =  np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  I_time).argmin() #I
    endIdx = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrime_time).argmin()
    startIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrevious_time).argmin()
    endIdx2 = np.abs(processedInventoryTimes[k].reshape((-1,)) -
     ↪  IPrimePrevious_time).argmin()
    locMatrixRow = k + len(inputAppliedError)

    term3a = inventoryAppliedError[k][:, startIdx] *
     ↪  inventoryAppliedError[k][:, endIdx]
    term3b = inventoryAppliedError[k][:, startIdx2] *
     ↪  inventoryAppliedError[k][:, endIdx2]
    term3c = ErrorMatrix[locMatrixRow, 1]**2
    term3 += (term3a+term3b)*term3c

    term4a = inventoryAppliedError[k][:, startIdx] *
     ↪  inventoryAppliedError[k][:, endIdx2]
    term4b = ErrorMatrix[locMatrixRow, 1]**2
    if IPrime-1 == I:
        term4c = ErrorMatrix[locMatrixRow, 0]**2
    else:
        term4c = np.zeros((iterations,))


    term4 += term4a*(term4b+term4c)

    term5a = inventoryAppliedError[k][:, startIdx2] *
     ↪  inventoryAppliedError[k][:, endIdx]
    term5b = ErrorMatrix[locMatrixRow, 1]**2

    if I - 1 == IPrime:
        term5c = ErrorMatrix[locMatrixRow, 0]**2
    else:
        term5c = np.zeros((iterations,))

    term5 += term5a*(term5b+term5c)
```

Term 4

```python
for k in range(len(inputAppliedError)):
    logicalInterval = np.logical_and(processedInputTimes[k] >= IPrevious_time, processedInputTimes[k] <= I_time).reshape((-1,)) #select the indices for the relevant time
    logicalInterval2 = np.logical_and(processedInputTimes[k] >= IPrimePrevious_time, processedInputTimes[k] <= IPrime_time).reshape((-1,))  #select the indices for the relevant time

    A = AuxFunctions.trapSum(logicalInterval, processedInputTimes[k],inputAppliedError[k])
    B = AuxFunctions.trapSum(logicalInterval2, processedInputTimes[k],inputAppliedError[k])
    C = ErrorMatrix[k, 1]**2
    term1 += (A*B*C)
```

Term 5

```python
for k in range(len(outputAppliedError)):
    logicalInterval = np.logical_and(processedOutputTimes[k] >= IPrevious_time,processedOutputTimes[k] <= I_time).reshape((-1,)) #select the indices for the relevant time
    logicalInterval2 = np.logical_and(processedOutputTimes[k] >= IPrimePrevious_time,processedOutputTimes[k] <= IPrime_time).reshape((-1,))  #select the indices for the relevant time
    locMatrixRow = k + len(inputAppliedError) + len(inventoryAppliedError)

    A = AuxFunctions.trapSum(logicalInterval, processedOutputTimes[k],outputAppliedError[k])
    B = AuxFunctions.trapSum(logicalInterval2, processedOutputTimes[k],outputAppliedError[k])
    C = ErrorMatrix[locMatrixRow, 1]**2
    term2 += (A*B*C)
```

2.4.5. *Further reading*

- Speed and Culpin [9]
- Avenhaus and Jaech [8]
- Handbook of Nuclear Engineering: Proliferation Resistance and Safeguards [7]
- Picard [10]
- Revisiting statistical aspects of NMA [4]
- Fundamentals of material accounting for nuclear safeguards [11]
- Page's test performance (Jones) [12]

    - Pages 402-408
- Page's test performance (Jones) [13]
    - Pages 19-22

2.5. Page's trend test

Page's trend test on SITMUF is implemented in `MAPIT.core.StatsTests.PageTrendTest()`. This function is not designed to be used through standalone, direct calls, rather, it is designed to be called through the `calcPageTT()` method of the `MBArea` class.

2.5.1. Historical context

The statistical discussed so far (MUF, σMUF, and SITMUF) are often combined with subsequent analyses. Statistical testing must be performed. For example, the CuMUF test (Section 2.3) is a simple one that compares the summed MUF values to a threshold. There are many different statistical tests that have been studied and deployed for use in material accountancy. One such test is Page's trend test. Page has been a popular sequential test given its relatively good performance on a wide range of loss scenarios. Page's trend test can be applied to any sequence, but is most commonly applied to SITMUF. Page's trend test on SITMUF specifically is what is implemented in MAPIT.

2.5.2. Theory

Page's test is similar to CuMUF, but instead of allowing for negative values, the test statistic is constrained to positive values only. Page's statistic can be defined as:

$$P_i(\boldsymbol{y}) = \max(P_{i-1}(\boldsymbol{y}) + y_i - k, 0)$$

Where:

- $P_i(\boldsymbol{y})$ is the ith page statistic for the sequence $\boldsymbol{y}$
- P_{i-1} is the previous page statistic (zero if $i = 0$)
- y_i is the ith element of sequence $\boldsymbol{y}$
- k is a hyper parameter

An alarm occurs if $P_i(\boldsymbol{y}) > h_i$ for some threshold h_i, although h is often taken to be constant. Parameter h is used to tune the false alarm probability while k is designed to give some control over the size loss that the test is designed to detect. Generally, smaller k is better for small protracted losses whereas larger k is better for detecting abrupt losses. A good rule of thumb is to set $k = \sigma/2$ where σ is the magnitude of the loss to detect in terms of material balance standard deviation. In practice, there might be two page's tests calculated on the same sequence, one for abrupt losses and one for protracted losses, each with a different set of (h, k) values.

2.5.3. *Implementation*

> **Note**
>
> As of MAPIT 1.4.6, `K` value is set to 0.5 and is only adjustable through the API, not the GUI.

The implementation of the trend test is straightforward following the equation given in the theory section. The only addition from the page's equation above is the tiling in line 808 which copies the page test statistic for all time steps between material balances.

```
797 for k in range(PageCalcs.shape[0]):
798   for P in range(1,int(MBPs)):
799 
800     if P == 1:
801       RZN = inQty[k,int((P - 1) * MBP)]
802     else:
803       RZN = inQty[k,int((P - 2) * MBP)] + inQty[k,int((P - 1) * MBP)] - K
804 
805     if RZN < 0:
806       RZN = 0
807 
808     PageCalcs[k,int((P - 1) * MBP):int(P * MBP)] = np.ones((MBP,)) * RZN
```

2.5.4. *Further reading*

- Page's test performance (Jones) [12]
  - Pages 402-408
- Page's test performance (Jones) [13]
  - Pages 19-22
- Page's trend test [15]

2.6. Special computational considerations

2.6.1. *Numerical integration*

Inputs and outputs are currently assumed by MAPIT to be flows although a future update will better support discrete items. That is, their units are represented as mass/time. It is assumed that flows will be represented in a continuous space as a non-zero value when the flow is on and zero (or near-zero) when off. These signals must be subsequently integrated to be used as input and/or output terms in the material balance. A routine called `trapSum` performs this task in MAPIT and is described in more detail below.

An important consideration when recording data is the sample frequency. If the sample frequency is very large (i.e., sampled infrequently) then the resulting data stream might not have recorded key flow events. Very small sample frequencies (i.e., sampled frequently) will capture all relevant events, but will result in a large, potentially sparse, dataset. The F3M library has specific blocks that implements an appropriate sample frequency if the F3M framework is being used. Solver step size for simulated models can also have a similar impact; large steps can result in key events being stepped over but small steps can result in a computationally expensive calculation. Keep data sample frequency and model simulation step size in mind when generating data for use within MAPIT.

`MAPIT.core.AuxFunctions.trapSum` is a function within MAPIT that attempts to numerically integrate a segment of data. The function expects that an array of boolean values is supplied which indicates if a region is to be integrated, along with the time step information, effective zero value (i.e., if a signal's off condition is not zero), and finally the data itself.

First, the function attempts to find the left and right indices that are relevant for the requested integration. Next, the data is sliced into the relevant segment.

```
 LI = np.argmin(relevantIndex == False)
RI = len(relevantIndex) - np.argmin(relevantIndex[::-1] == False)
relevantDataVals = data[IDXC, LI:RI:1]
relevantDataValsAbs = np.abs(relevantDataVals)
relevantTimeVals = time[LI:RI:1]
```

If the requested segment has a non-zero signal before and/or after this segment note it (i.e., the first and final time steps are not zero)

```
 if relevantDataValsAbs[0] > baseline_zero:
    partialLeft = True

if relevantDataValsAbs[-1] > baseline_zero:
    partialRight = True
```

The function attempts to find index pairs that represent a pulse of material. For example, if a tank fills and empties within the interval to integrate, it will have a geometric shape. The shape will depend on the flow rate to/from the tank, but it generally be a piecewise discontinuous shape (i.e., a pulse). The function generates a boolean mask for locations where the data is zero and non-zero, the intersection of which can be used to find segments of non-zero data.

Generate the left and right indices of non-zero data segments using the intersection of zero and non-zero values.

```
 Z_mask = np.roll(np.concatenate(
    (np.zeros((1,)), Z_mask, np.zeros((1,)))), 1)[1:-1]

mask = NZ_mask*Z_mask
LeftIndicies = np.where(mask)[0].reshape((-1, 1))

Z_mask = np.roll(np.concatenate(
    (np.zeros((2,)), Z_mask, np.zeros((2,)))), -2)[2:-2]

mask = NZ_mask*Z_mask
RightIndicies = np.where(mask)[0].reshape((-1, 1))
```

Next, if there is a "partial" segment, that is a segment that stretches outside the bounds of the integration window, that case should be handled. This involves injecting some indices manually depending on what segments have been found so far. There's some additional checks to look for errors we have seen in Simulink, and to remedy them if present.

Finally numerical trapezoidal integration is performed on each segment found in the integration window:

```
for Q in range(len(datasegs)):

if datasegs[Q][0, -1] == 0:
    traptot += (np.trapz(datasegs[Q], timesegs[Q]) + 0.5 *
                (timesegs[Q][:, -1]-timesegs[Q][:, -2])*datasegs[Q][:, -2])
else:
    traptot += (np.trapz(datasegs[Q], timesegs[Q]))
```

3. DOWNLOAD AND INSTALLATION

3.1. Objective

This section will show you how to download & install MAPIT. MAPIT is an open-source Python-based program that relies on the work of other open-source libraries to work properly. As such, we cannot distribute a binary executable that contains all necessary code to run. Instead MAPIT must be distributed as a Python *package* and we prefer to use conda, and miniconda in specific, to manage the Python installation. If you're new to Python, we have some tools to help you easily install MAPIT and get started (Section 3.3). If you've used Python before, see Section 3.2.

3.2. Experienced with Python

In most cases, you can simply install directly from our repository. We recommend using an environmental manager (either venv or conda, but we assume conda).

Depending on your conda distribution your install instruction might vary slightly. Here's an example assuming you have an empty conda environment:

```
conda install pip "python<3.12,>3.8"
pip install git+https://github.com/sandialabs/MAPIT
```

Important: We strongly recommend using conda to also install numpy (`conda install numpy`) if on a Apple silicon-based machine as we have observed problems wherein `pip` installs the wrong version (x86_64) which can cause errors, particularly with the MAPIT's SITMUF calculation.

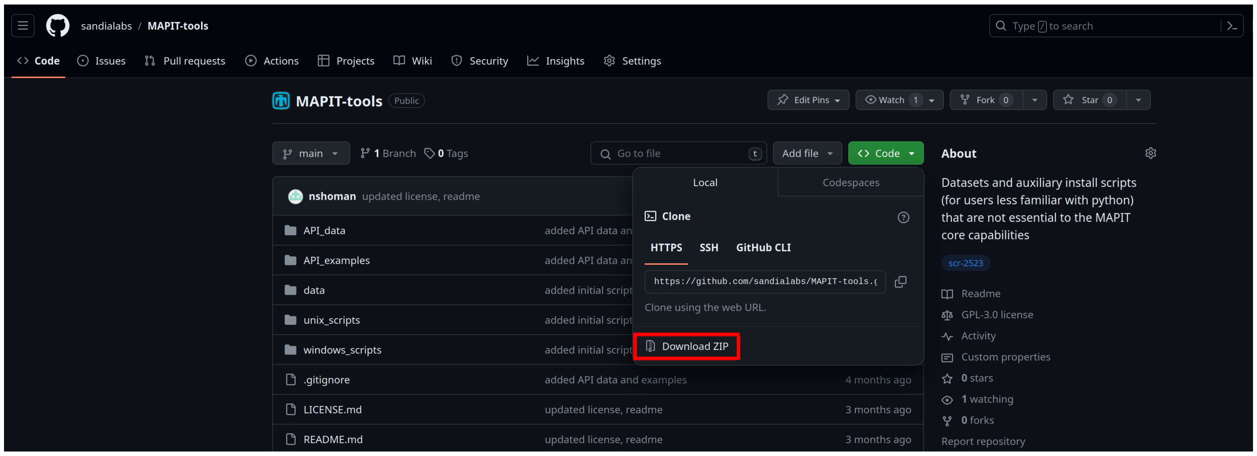
After MAPIT has been installed, you can call the GUI entry point from the command line simply by calling `MAPIT` from your environment. You can also import the MAPIT API and use it as a library.

Additional scripts and tools can be found in the MAPIT-tools repo. The exemplar data is in the folder `data` from the tools repo and can be loaded in MAPIT by going File -> Load exemplar data and pointing the dialog to the `data` folder.

3.3. New to Python

If you're not very familiar with Python, we provide several setup scripts to help you get started.

3.3.1. *Downloading MAPIT tools*



nshoman updated license, readme

API_data	added API data an	
API_examples	added API data an	
data	added initial script	
unix_scripts	added initial script	
windows_scripts	added initial script	
.gitignore	added API data and examples	4 months ago
LICENSE.md	updated license, readme	3 months ago
README.md	updated license, readme	3 months ago

```
https://github.com/sandialabs/MAPIT-tools.g
```

Clone using the web URL.

Datasets and auxiliary install scripts (for users less familiar with python) that are not essential to the MAPIT core capabilities

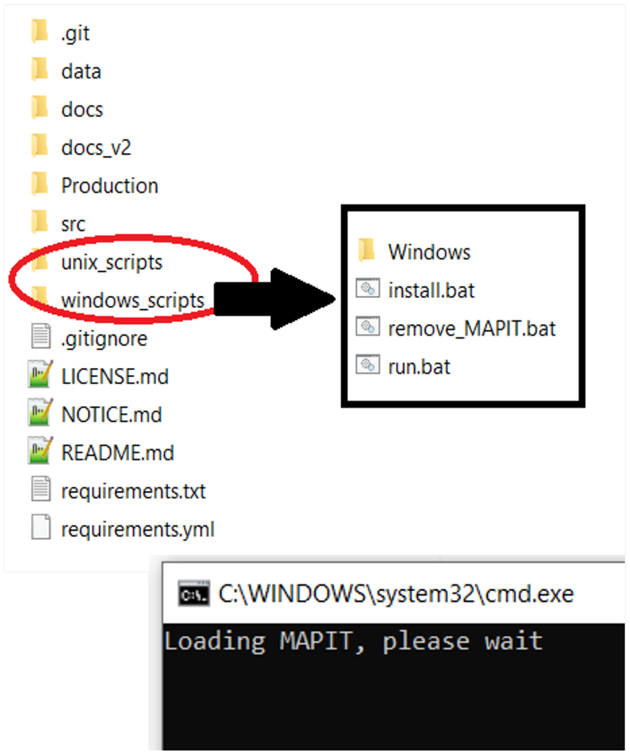
Figure 3-1: MAPIT download location

The MAPIT-tools repo contains additional, non-essential files for running MAPIT. Here you will find the scripts needed to help you install MAPIT if it's your first time using Python. The scripts will download and and install Miniconda, then install MAPIT and it's required dependencies. It will also include shortcuts to launch MAPIT.

- The tools repo can be found at https://www.github.com/sandialabs/MAPIT-tools
    - In the directory click on the green code button in the top right
    - In the pull down menu click the download zip button
    - Once the download is complete, unzip the folder on your computer

3.3.2. *Installing MAPIT*

Once you have downloaded and unzipped the MAPIT-tools folder you will have the following folders on your computer



```
Loading MAPIT, please wait
```

Figure 3-2: MAPIT install script location

- The key folders for the install process are the windows_scripts and linux_scripts
  - If you are using a windows operating system click on the windows scripts_folder
  - If you are using a unix operating system (Mac or Linux) click on the unix scripts_folder
- Inside the respective folders you will see three key files: install, run, and remove_MAPIT
  - Click on the install file and MAPIT will begin the install process
  - The install file will download miniconda3, a minamalist version of Anaconda, and will only download the python packages required to run MAPIT
  - After installing miniconda3, the key python modules are installed
    - This process can take a few minutes, please keep the command prompt or shell open until "MAPIT environment install completed"

4. INTRODUCTORY TUTORIAL

4.1. Fuel fabrication overview

The purpose of this tutorial is to introduce you to the example dataset found in the MAPIT-tools repository. This dataset was based on a fuel fabrication facility described by IAEA STR-150 [16] . There are some unique features of MAPIT that are only present when using the included datasets and might not be available when analyzing an external dataset.

Some key features of the fuel fabrication facility are noted below:

- 300 MT UO_2 throughput
- 3.0% ^{235}U
- Final products are LWR fuel assemblies
- Feed materials:
  - Low enriched UF_6
  - Uranyl nitrate
  - UO_2 powder
  - Material from scrap facility

4.2. Walkthrough

This tutorial describes basic functions of MAPIT and how to get started using the sample dataset that has been included.

4.2.1. Downloading the exemplar dataset

The exemplar dataset for MAPIT is no longer included in the main repository and must be downloaded from the MAPIT-tools repository. See Section 3.3.1 for further details.

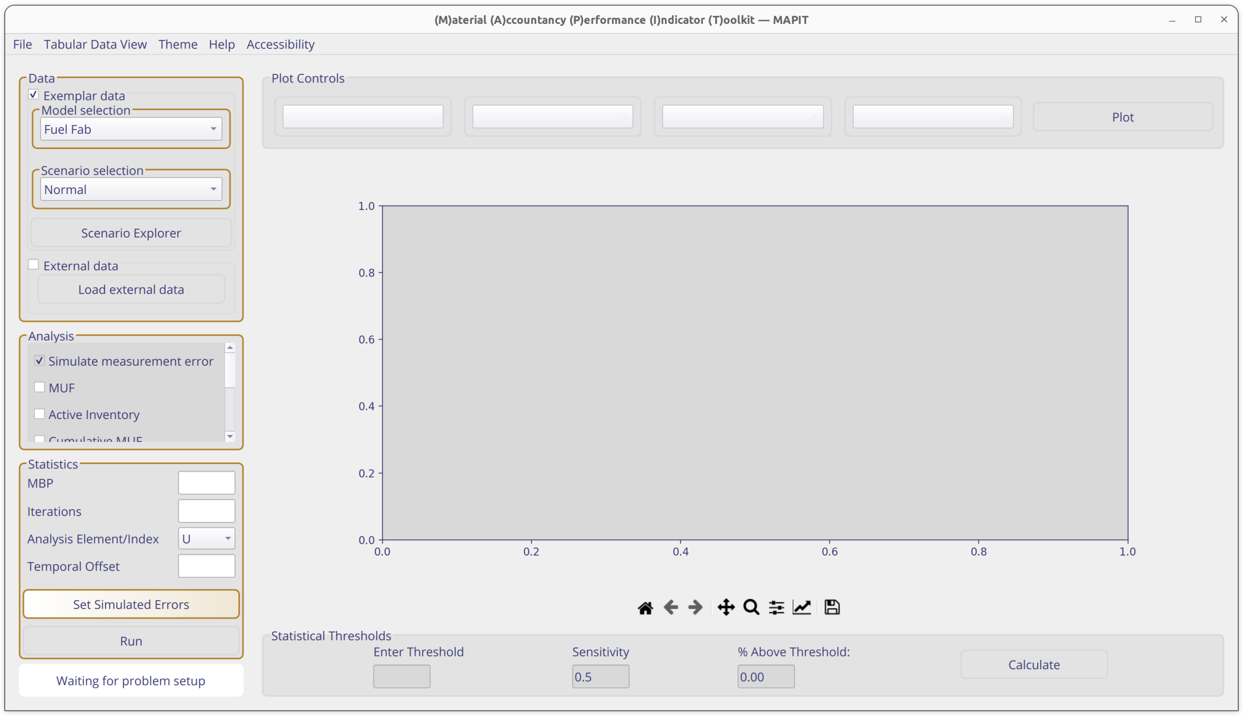
4.2.2. Loading the exemplar dataset

The pre-generated dataset described in the introduction can be imported into MAPIT by specifying the path to the data folder. This usually occurs when launching MAPIT for the first time, but it can also be accessed by selecting *File* > *Load Exemplar Data* > *Select directory*. The path selected here should be the path to the `data` folder downloaded from the MAPIT tools repository. For example, your path might be `/path/to/folder/data`.

Note

Selecting the data path only needs to be performed once and is stored in MAPIT internally. If you move your data directory, you can always redefine the path to the data folder by again going to *File > Load Exemplar Data > Select directory*.

If a valid path to data is selected, then several of the options in the data selection area will populate. Currently, the MAPIT-tools repo includes data from a generic fuel fabrication facility with notional scenarios for nominal behavior, an abrupt material loss, and a protracted material loss. Your MAPIT window should look similar to the image below.



☑ Exemplar data

Model selection: Fuel Fab

Scenario selection: Normal

☑ Simulate measurement error

Analysis Element/Index: U

Sensitivity: 0.5

% Above Threshold: 0.00

Waiting for problem setup

Figure 4-1: MAPIT main interface

4.2.3. *Statistical test configuration*

The boxes with gold borders are the next steps in the MAPIT workflow. The currently available statistical tests are denoted by check boxes, go ahead and select all of them.

The suggest parameters for the statistics parameters are as follows:

- `MBP` (Material Balance Period, units of hours) : **416**
  - Try different `MBP` lengths to see how performance statistics change
- `Iterations` (Number of statistical realizations to run): **50**

    - Increasing the number of `Iterations` can reduce the simulation uncertainty in the probability of detection

- `Analysis Element/Index`: U

    - The exemplar dataset only includes uranium, so multiple options are not available

- `Temporal Offset`: Empty

After setting the required statistics inputs the final step before starting the calculation is setting the errors. As no measurements are perfect it is impossible to know the true value of some quantity of interest (in this case Uranium) at a particular location. Use `Select Errors` to open an interactive dialog to set measurement errors. These can be set individually or as a group. Press `Done` when finished.

Tip

One purpose of MAPIT is to understand how these errors impact common safeguards statistical tests, so feel free to choose any value. The IAEA ITV (International Target Values) [6] provides a good reference for expected performance for different types of measurement systems.

Tip

Entering a customized error table can be tedious. The included example scenario has 31 different measurement locations! MAPIT allows for loading (`Load Error Config`) that reads a .csv table of errors so that manual specification is not required every time MAPIT is run. Similarly, you can use (`Save Error Config`) to save a specified error configuration to disk. The directory containing this configuration file can be found using the `platformdirs` package as follows:

```python
from platformdirs import user_config_dir
print(user_config_dir("MAPIT",None))
```

4.3. Analysis

Once the statistical tests and errors have been configured press `Run` to start the calculation. MAPIT is a lightweight tool that should run fairly quickly for a small number of iterations (100), but varies based on hardware configuration. Progress can be monitored through the dialog and progress bar at the bottom of the tool (see below).

Figure 4-2: Progress bar

Tip

Large-scale jobs can be processed in parallel using `StatsProcessor` from the API.

4.3.1. Plotting

The first step in many analytical workflows is to plot data to gain an intuition for what is happening. MAPIT has multiple plot options (shown below) that dynamically change based on the option selected. Try plotting different quantities of interest (also make sure to note how these change with the selected errors).

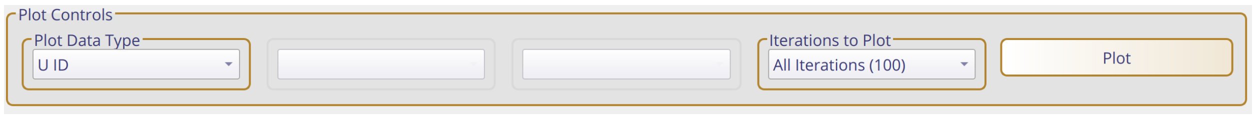


Figure 4-3: Plot type controls

The various options are as follows. Note that some options may not be available depending on what quantity is being plotted.

- **Plot Data Type**
  - Varies depending on selected statistical tests
  - Always includes the "ground truth data" and "observed data"
    - Ground truth is the data before errors are applied by MAPIT
    - Observed data is the ground truth after errors have been applied
  - Requested statistical tests will also be available here
- **Plot Data Location**
  - Only relevant for "ground truth data" and "observed data"
    - Location doesn't apply to the statistical tests at the moment as MAPIT only supports analysis of one material balance area at a time
  - Lists locations based data used
    - Included dataset has locations baked in
    - Attempts to use user provided locations if data was imported
- **Plot Data Nuclide**
  - Only relevant for "ground truth data" and "observed data"
  - Used to plot specific nuclide at a location of interest within the material balance
- **Iterations to Plot**

- Not relevant for "ground truth data"
- Used to control how many iterations are plotted

4.3.2. Thresholds

Statistical tests used in safeguards usually require adjustment of at least one tunable parameter. For example, Page's trend test actually has two (h and k) of which one is made available to users (h).

Important

Page's trend test currently uses k=0 which is the ideal statistic for a one unit shift in SITMUF. See the theory guide in Section 2 for more details.

The statistical threshold area of MAPIT (shown below) allows users to input a value and see how many times that threshold has been crossed. The threshold calculation is generic and can be applied to any of the plot quantities.

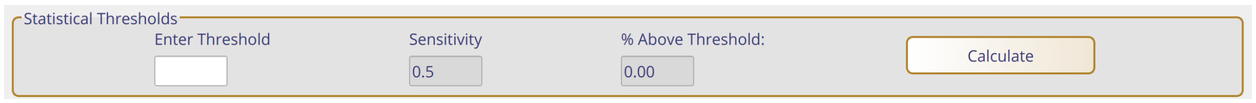


Figure 4-4: Threshold box

Note

The quantity reported by MAPIT, `% Above Threshold`, reflects all of the runs, even if not plotted. For example, if 1000 iterations were requested, then the max quantity of iterations allowed to be plotted at once is 100. However, the threshold will check all 1000 runs and report the quantity that exceeds the threshold.

Note

The threshold tool reports if a particular iteration of a quantity of interest has past the threshold at any time in the dataset. There may be some desire to check a threshold for a limited window of time, however this capability is not yet implemented. In the meantime, please preprocess your data if desired to circumvent this limitation. For example, if you want to know yearly performance, but your dataset is two years long, split the dataset in half before importing into MAPIT.

4.3.3. Error contributions

Understanding the contribution of various facility measurements to the material balance uncertainty is often important. Identifying large sources of error can help prioritize areas for improvement. MAPIT facilitates this analysis by providing tabular data describing the error contribution of various components. This can be accessed through by selecting *Tabular Data View > Error Contribution*.

Additionally, the contribution can be plotted by selecting error contribution in the plot options. These options are only available if SEID/SEMUF has been selected.

4.4. Data export: figures

Figures can be saved by using the save icon at the bottom of the plot (see below) which directly interacts with the Matplotlib backend. Plots can be further customized by using the options on the navigation bar.

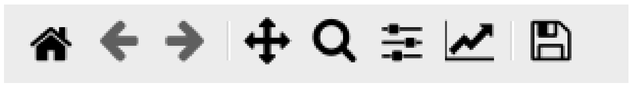


Figure 4-5: Figure navigation bar

4.5. Data export: data

The data can be exported by selecting the `File` menu option in the MAPIT main area and selecting `Save Data`. An option will be presented to save the underlying data used for the safeguards statistical tests (i.e. “observed data”). The default behavior is to save data in .csv format with a shape of [time x iterations]. For example, for a case where 100 iterations were requested and the time was 5000, then the .csv would be of shape $(5000, 100)$.

Caution

Do not expect reliable performance of this capability when using irregularly sampled data. Although MAPIT can handle this type of data, validation efforts are ongoing and have not yet been completed.

Tip

The default output directory can be found by using the `platformdirs` package as follows:

```python
from platformdirs import user_data_dir
print(user_data_dir("MAPIT",None))
```

5. GUIDED EXERCISES

5.1. Exercise 1: General MAPIT familiarity

5.1.1. Objective

The goal of this tutorial is to gain familiarity with basic MAPIT functionality.

Note

Different stakeholders use different terminology for safeguards quantities. In some instances, the terms involve the same mathematical calculation even if the method to obtain the underlying data differs. MAPIT includes both "domestic" and "international" terminology. The following table shows terms that are mathematically equivalent. The operational and policy differences between the terms are not described here.

Generic	International	Domestic
MB	MUF	ID
σMB	σMUF	σID
—	SEMUF	SEID
—	SITMUF	—

Table 5-1: Nomenclature equalities for common accountancy terms

5.1.2. Opening MAPIT

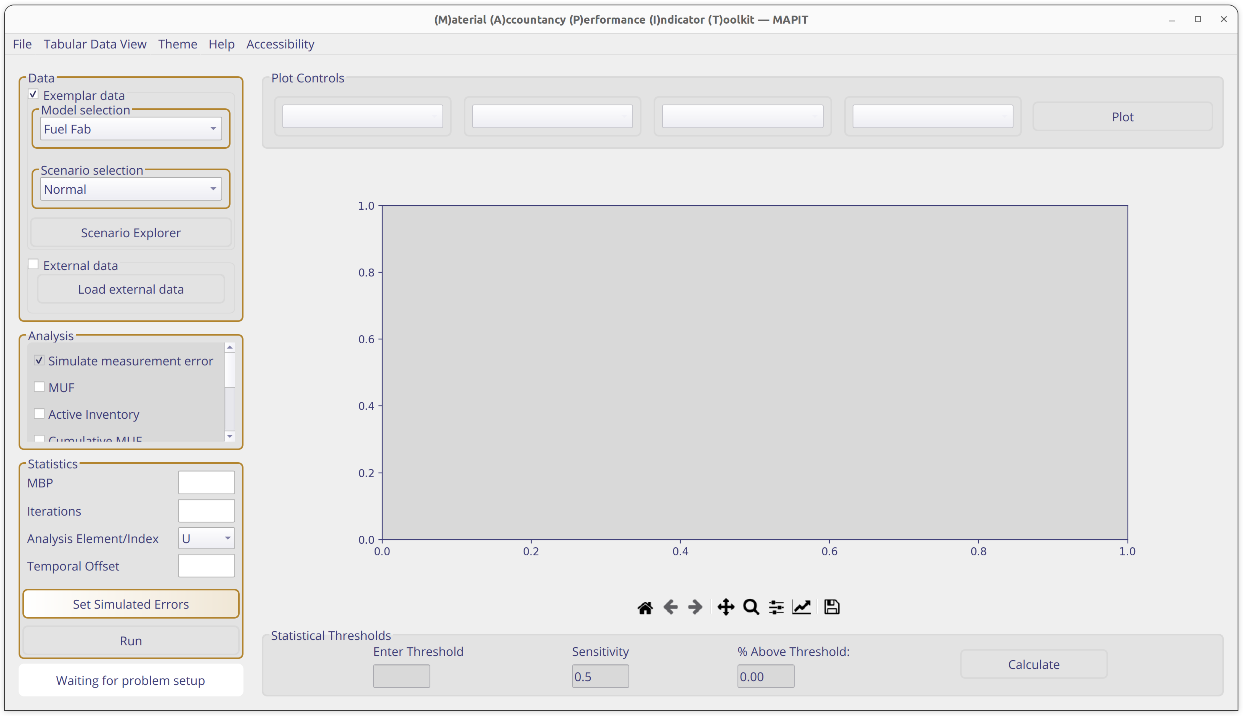
- Start by launching MAPIT
  - If new to Python:
    - Windows: Run `run.bat` located in `\MAPIT\windows_scripts` by double clicking
    - Unix: Run bash `run.sh` in a console, ensuring that the current working directory is located in `\MAPIT\unix_scripts`
  - Otherwise:
    - Run MAPIT from you previously setup environment by running `MAPIT` from the command line

Tip

If you are having trouble viewing MAPIT on your screen, try maximizing or resizing the window.

5.1.3. MAPIT main interface

- The main window of MAPIT should now be shown (see below for light theme example)
  - MAPIT has both light and dark themes available
    - These can be toggled at any time using the `Theme` dropdown menu
  - MAPIT allows users to control the font size
    - The font size can be changed using the `Accessibility` dropdown menu
  - MAPIT preferences regarding style and font size is stored internally and will be retained after closing MAPIT
- If the exemplar data has not been downloaded from the MAPIT-tools repository do so now before proceeding.
  - If there are no options listed in the `Scenario Selection` box of the Data area, make sure the `Exemplar Data` box is checked and the `/path/to/data` is specified correctly in *File > Load Exemplar Data > Select directory*.



☑ Exemplar data
Model selection: Fuel Fab
Scenario selection: Normal
☑ Simulate measurement error
Analysis Element/Index: U
Sensitivity: 0.5
% Above Threshold: 0.00
Waiting for problem setup

Figure 5-1: MAPIT main interface

> **Note**
>
> The status bar in the bottom left lets the user know what the GUI is doing at the moment. Initially when the GUI is opened it states waiting for problem setup. After data is imported, the analysis

boxes are checked, statistical boxes are filled out, the simulated measurement error is specified, and `Run` is pressed, the bar will display the progress through the selected analyses.

Once all of the analyses have been completed, the status bar states execution finished. At this point, the user is able to use the plot controls and statistical thresholds to plot the analyses and the base data. The status bar, GUI animations, and tooltips are all used within MAPIT to help explain the safeguards analysis flow. Initially, the analyses and statistics boxes are highlighted in gold, which indicate those inputs are required for analysis.

Note

Currently, only options for statistical tests on uranium are available as it is the only element tracked in the fuel fabrication facility examples.

5.1.4. *Performing a basic analysis*

- Start by selecting Normal for Scenario selection if not already selected
    - Continue by selecting the checkboxes for the following statistical tests
        - MUF
        - Cumulative MUF
        - Sigma MUF
        - SITMUF
        - Page's test on SITMUF
- Next, configure required parameters in the statistical box:
    - `MBP`: **416**
    - MBP is the material balance period
    - For this exercise, the MBP least common multiple of the facility's input and output stream period
    - `Iterations`: **50**
    - The number of realizations to run
    - Note that in practice, only a single iterations would be observable
    - If running on a lower performance device, try running with `Iterations`: **20**
    - `Analysis Element/ Index`: **U**
    - MAPIT can perform statistical tests on general datasets (i.e. not just uranium and plutonium)

    - In such scenarios, information must be provided about the element in the dataset that tests should be performed on
    - MAPIT can only process one element at a time
    - `Temporal Offset`: ***Empty***
        - In some cases, it may be desirable to ignore a startup period
        - The offset rebases the calculations to a new zero
- Setup errors by clicking the `Select Errors` button in the statistics box.
    - This opens the error selection pane
    - All errors are in percents
    - Options are provided to adjust measurement errors for various KMPs
    - Users can manually enter values in the boxes
    - Alternatively, the drop down boxes can change all values for that measurement type automatically
        - For example, all random input errors can be changed at once, or all systematic errors
    - Some error configurations can be tedious to input. Functionality is provided to save and load configurations using the `Load Error Config` and `Save Error Config` buttons.
- For now, select 1% errors for all values using the drop down options.
- Press the `Run` button to run MAPIT

Tip

The default config directory can be found by using the `platformdirs` package as follows:

```
from platformdirs import user_config_dir
print(user_config_dir("MAPIT",None))
```

5.1.5. Summary

In this exercise you learned the about the basic functionality of MAPIT:

- Opening MAPIT
- How to load included datasets
- How to run MAPIT
- How to input required parameters to run MAPIT

5.2. Exercise 2: Impacts of measurement error

5.2.1. *Objective*

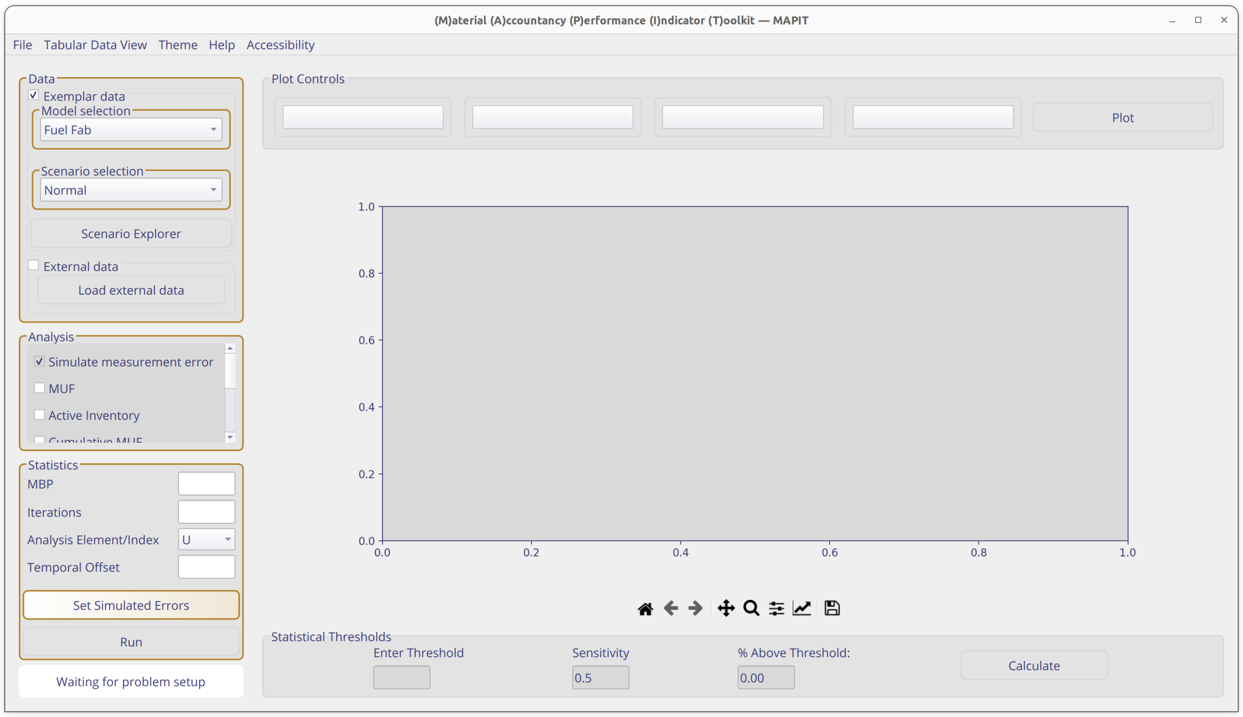
Explore the impact of measurement error on safeguards metrics. Recall that measurement error negatively impacts the ability to detect anomalies such as material loss.

Caution

This exercise assumes that you are familiar with MAPIT and can perform tasks discussed in the previous exercise in Section 5.1 (i.e. launching MAPIT, loading the included scenarios, setting up MAPIT to perform analyses, etc).

5.2.2. *Problem setup*

- Start this exercise by launching MAPIT, selecting the Fuel Fab model under `Model` option, and selecting the `Normal` dataset
  - This tutorial starts by assuming you are at the main MAPIT interface (similar to the image below)



☑ Exemplar data
Model selection: Fuel Fab
Scenario selection: Normal
☐ External data
☑ Simulate measurement error
☐ MUF
☐ Active Inventory
☐ Cumulative MUF
MBP:
Iterations:
Analysis Element/Index: U
Temporal Offset:
Waiting for problem setup
Enter Threshold:
Sensitivity: 0.5
% Above Threshold: 0.00

Figure 5-2: MAPIT main interface

- Select the following statistical tests
  - MUF

  - Cumulative MUF
  - Sigma MUF
  - SITMUF

- Next, configure the required parameters in the statistical box using the same parameters from exercise 1:
  - `MBP`: **416**
  - `Iterations`: **50**
  - `Analysis Element/ Index`: **U**
  - `Temporal Offset`: ***Empty***

- Set the measurement errors by pressing the `Select Errors` button
  - Choose **3% for all random and systematic errors**

- Run MAPIT by pressing the `Run` button

5.2.3. *Data exploration*

- After running, several plot options should be available under the plot controls option (similar to image below)
  - Plot Data Type
  - Plot Data Location (to investigate the behavior at all key measurement points)
  - Contribution Type
  - Iterations to Plot

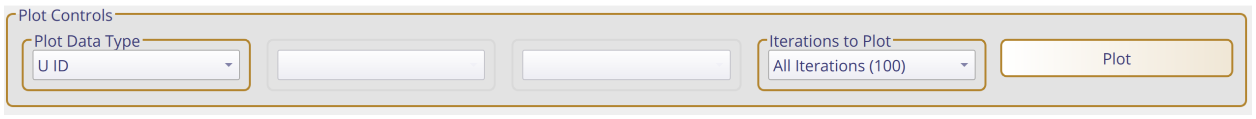


Figure 5-3: Plot data type controls

Note

MAPIT's plotting options are dependent on the data type selected. `Ground Truth` and `Observed Data` are the only data types that have access to the `Plot Data Locations`. The `Sigma MUF Contribution` plots are the only plots that have access to the Contribution option, which will become selectable when the `Sigma MUF Contribution` data is selected. The `Analysis` boxes selected and the `Observed Data` have additional plotting options under the to plot toolbar. The user is able to plot 1 random iteration, the average of all iterations, and all iterations. In this exercise some of the potential plots available are shown.

Note

MAPIT can export any figures for later use. Simply click the floppy disc icon below the plotting window.

- Start by observing the calculated MUF values
    - Plotting options dynamically change based on the data type selected and number of iterations considered
    - Since `Iterations` were set to ≤ 50 (a relatively small number), start by plotting them all
    - The plot should generally look like the images below, but will vary due to the inherent randomness of the calculation

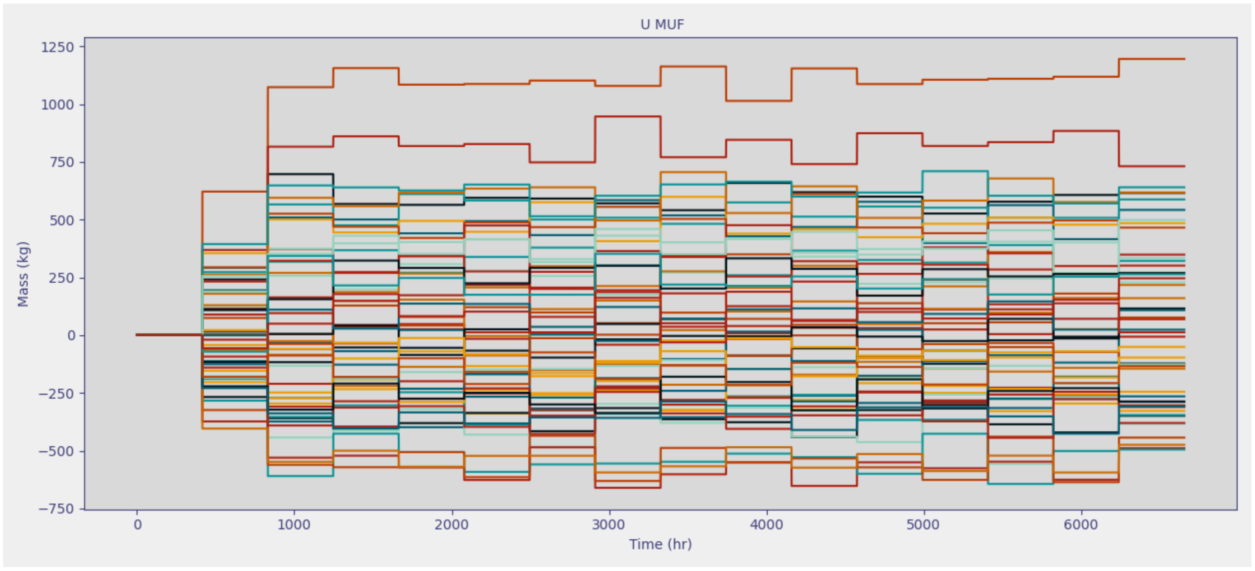


Figure 5-4: Several MUF iterations

- Next, plot the U Sigma MUF (i.e. σMUF) and notice that it tends to remain around 600 kg
    - The first balance period has a smaller Sigma MUF due to startup conditions

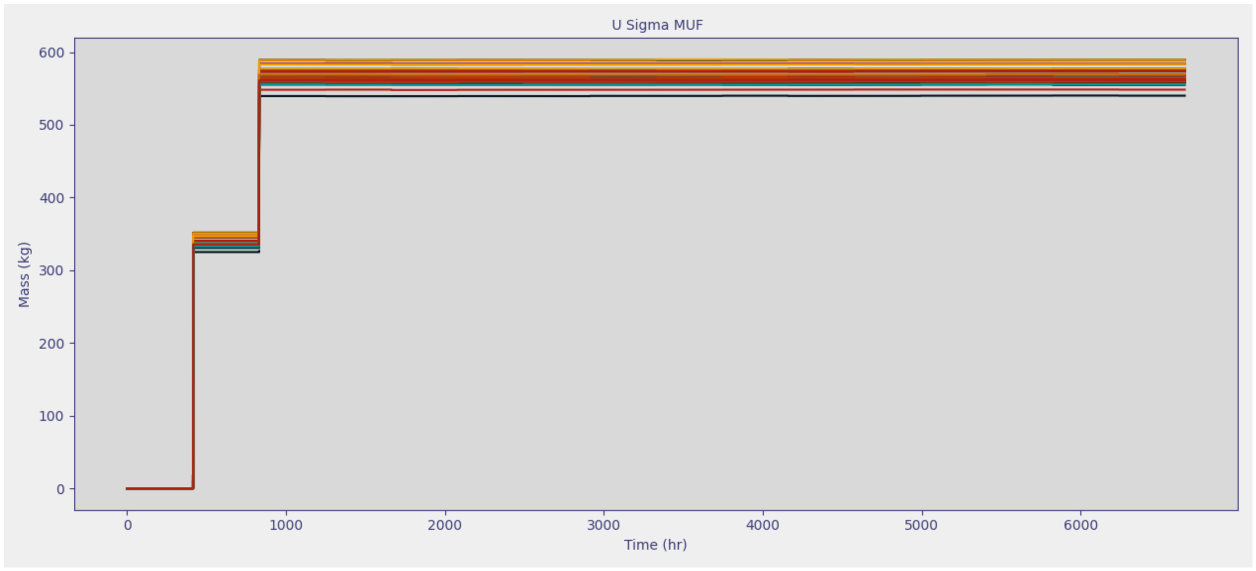


Figure 5-5: Several SigmaMUF iterations

- Try plotting the U SITMUF data
  - Your plot should look similar to the one below
  - Notice that U SITMUF tends to decrease overtime and then reaches a steady state value
  - Also, SITMUF has (approximately) a mean of zero and standard deviation of one once the covariance matrix is well approximated

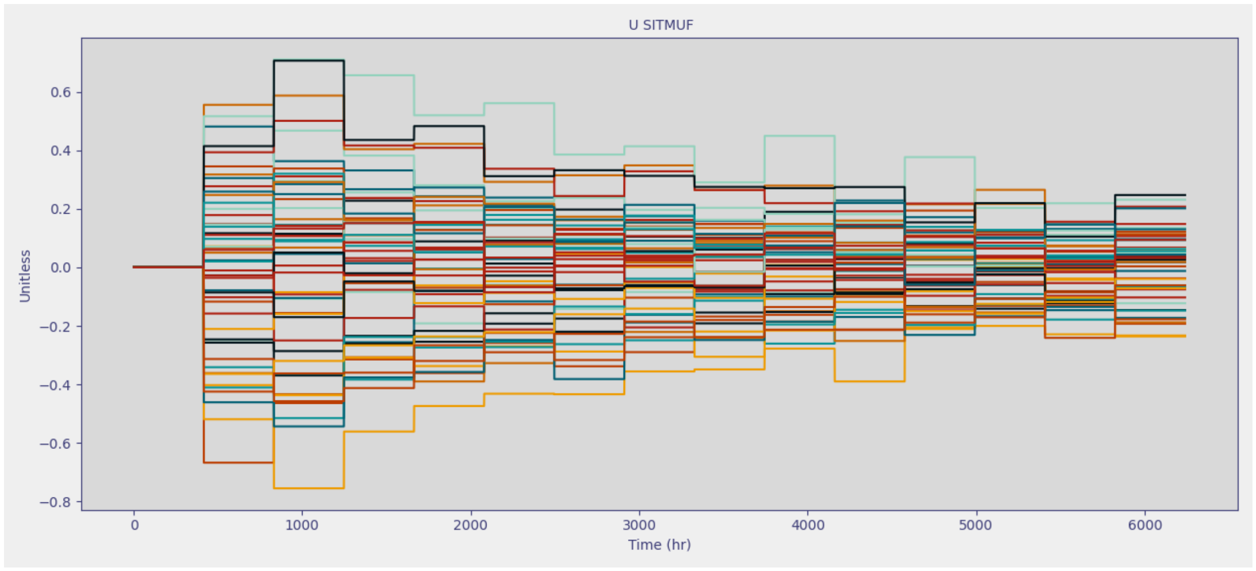


Figure 5-6: Several SITMUF iterations

Note

The results seen when plotting SITMUF match the description from earlier lessons. That is, that

SITMUF is the independent MUF sequence. However, notice that the SITMUF values start larger than their final, steady state values. Recall that the transformation from MUF to SITMUF uses an estimate of the covariance matrix (shown below).

$$\Sigma = \begin{pmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \cdots & \sigma_{1n}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \cdots & \sigma_{2n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1}^2 & \sigma_{n2}^2 & \cdots & \sigma_{nn}^2 \end{pmatrix}$$

Note

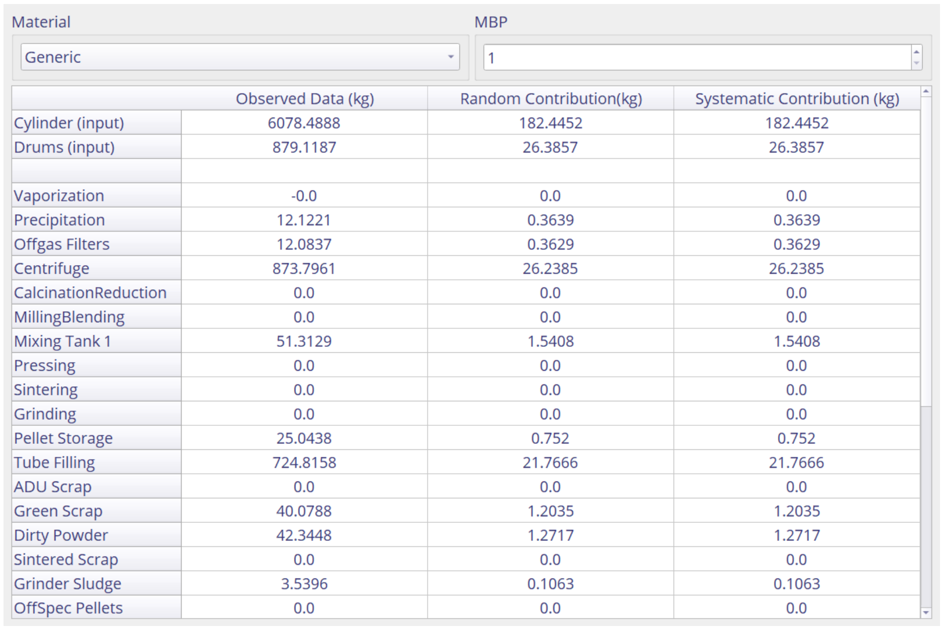
The covariance matrix grows as repeated material balance calculations are made and observed, which results in a better approximation of the true covariance matrix. In fact, the approximation will converge on the true value as the number of measurements approaches infinity. **Consequently, the variance and mean of the transformed sequence, SITMUF, converge to 1 and 0 respectively as the covariance estimate improves**.

- Finally, try plotting the Page U SITMUF. This is Page's trend test on SITMUF which is used to detect subtle trends in SITMUF that could indicate a material loss
    - Page's test will be discussed further in the next exercise
    - Feel free to try entering numbers into the `Enter Threshold` box and pressing `Calculate` MAPIT will return the number of runs that exceed the user specified threshold, which is useful for analyzing performance of a safeguards system.
        - MAPIT returns the % of all runs over the threshold, even if not all are plotted. For example, even if only one of the 50 runs are plotted, it will still return the same value.

5.2.4. *Understanding error contribution*

- After examining the different quantities calculated by MAPIT, open the error contribution table to better understand the contributions of different KMPs to the material balance uncertainty. Do this by selecting the *Tabular Data View* > *Error Contribution* from the top menu bar of MAPIT.
- The error contribution table should display all the locations in addition to their random and systematic contributions to Sigma MUF. Your table should look similar to the image below.
    - The `inventory` column refers to the actual mass at the selected material balance period
        - For flows (e.g. inputs and outputs) this is the time integrated flow over the material balance period
        - For inventories this is the instantaneous inventory value

- The random and systematic contribution are the contributions to Sigma MUF


Material: Generic

MBP: 1

	Observed Data (kg)	Random Contribution(kg)	Systematic Contribution (kg)
Cylinder (input)	6078.4888	182.4452	182.4452
Drums (input)	879.1187	26.3857	26.3857
Vaporization	-0.0	0.0	0.0
Precipitation	12.1221	0.3639	0.3639
Offgas Filters	12.0837	0.3629	0.3629
Centrifuge	873.7961	26.2385	26.2385
CalcinationReduction	0.0	0.0	0.0
MillingBlending	0.0	0.0	0.0
Mixing Tank 1	51.3129	1.5408	1.5408
Pressing	0.0	0.0	0.0
Sintering	0.0	0.0	0.0
Grinding	0.0	0.0	0.0
Pellet Storage	25.0438	0.752	0.752
Tube Filling	724.8158	21.7666	21.7666
ADU Scrap	0.0	0.0	0.0
Green Scrap	40.0788	1.2035	1.2035
Dirty Powder	42.3448	1.2717	1.2717
Sintered Scrap	0.0	0.0	0.0
Grinder Sludge	3.5396	0.1063	0.1063
OffSpec Pellets	0.0	0.0	0.0

Figure 5-7: Error contribution table

- Along with the error contribution table, MAPIT has a set of plots that visualize the error contribution of each key measurement point. The `Contribution` (both absolute and relative) plot the impact individual measurement points have on the Sigma MUF.

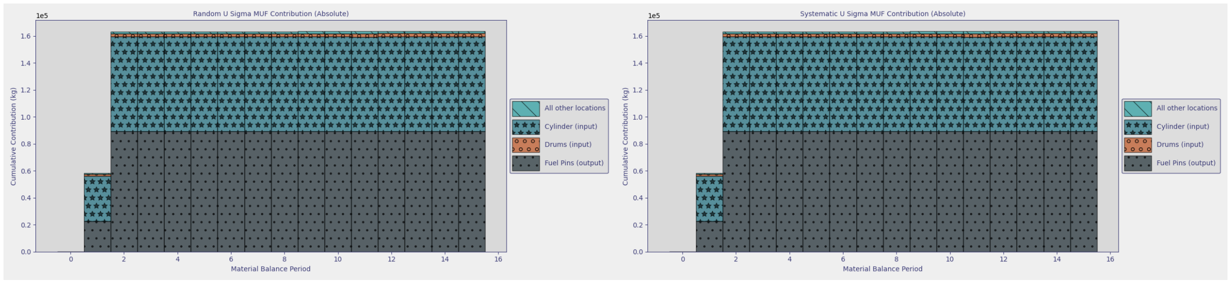


Figure 5-8: Comparison of random and systematic contributions.Note that contributions are identical as random and systematic are set to the same value of 3%.

- Try looking at different material balance periods.
    - Both the plots and the table shows that Cylinder (input) and Fuel Pins (output) have the largest inventory terms and consequently the largest contribution
- Change the measurement uncertainty of the Cylinder (input) and Fuel Pins (output) error terms to 1%.
    - MAPIT does not need to be restarted to perform another calculation on a new dataset, however, note that current results will be lost

- Press the `Select Errors` button and edit the corresponding boxes to reduce the error for the two components.

- Press `Run` to again calculate the statistical quantities.

- Check the newly calculated values by plotting key quantities such as Sigma MUF.

  - Note that Sigma MUF has decreased dramatically from approximately 600 to 210.

  - Examine the `Error Contribution` and note that the contribution from the Cylinder (input) and Fuel Pins (output) has similarly fallen in magnitude.

  - The next exercise will consider the impact of measurement error for safeguards more concretely by considering a hypothetical material loss.

  - The total contribution plot now shows decreased measurement uncertainty on the Cylinder (input) and Fuel Pins (output), other key measurement points now have comparable uncertainty.

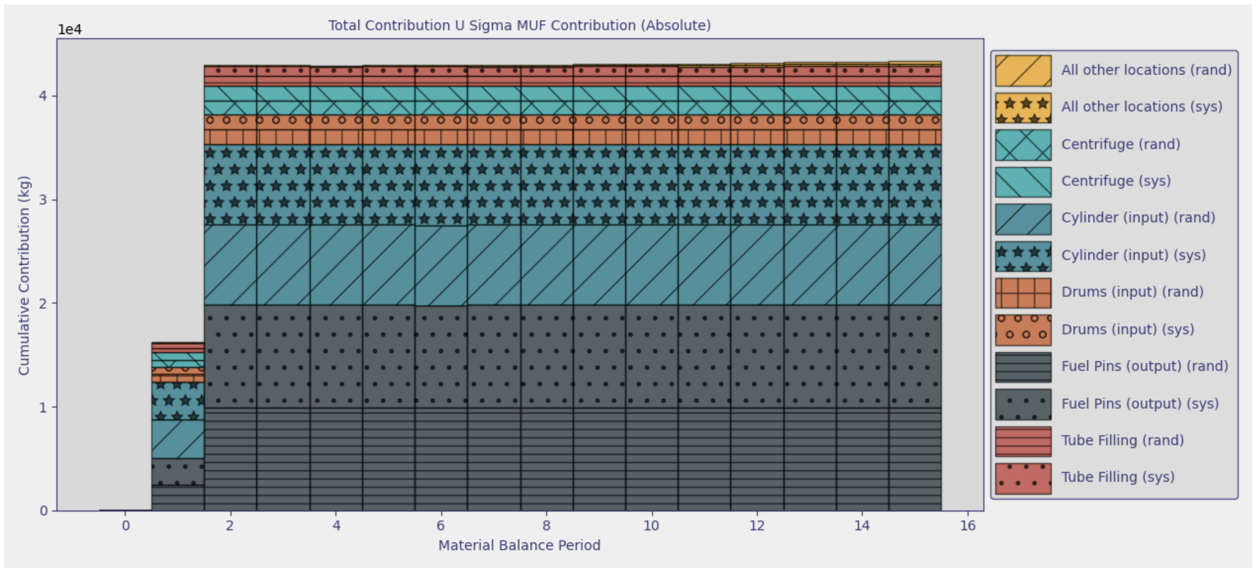


Figure 5-9: Total error contribution by location

Note

The material balance period can similarly impact key quantities like Sigma MUF. Longer balances lead to higher Sigma MUFs whereas shorter balances lead to smaller Sigma MUFs. However, there is a limitation to the gains of shorter material balances. While not explored in this exercise, further details can be found in work by Avenhaus and Jaech [8].

5.2.5. *Summary*

In this exercise, you learned about how to change simulated measurement errors in MAPIT and their impact on calculated statistical quantities. Further, the capability of MAPIT to show individual error components was also introduced.

- Higher measurement error leads to larger Sigma MUF values
- Uncertainty contributions rely on both measurement error and inventory size

5.3. Exercise 3: Material loss

5.3.1. *Objective*

Caution Gain familiarity with the notional material losses. This exercise will prepare you for using MAPIT to evaluate safeguards systems.

This exercise assumes that you are familiar with MAPIT and can perform tasks discussed in the previous exercises in Sections 5.1 and 5.2.

5.3.2. *Problem setup*

- Start this exercise by launching MAPIT, selecting the `SNL curated dataset` option, and loading the `Abrupt` dataset.
    - This tutorial starts by assuming you are at the main MAPIT interface
- Select all the checkboxes for available statistical tests
    - A total of five (5) checkboxes should be checked
- Next, configure the required parameters in the statistical box using the same parameters from exercise 1 and 2:
    - `MBP`: **416**
    - `Iterations`: **50**
        - If running on a lower performance device, try running with `Iterations`:**20**
    - `Analysis Element/ Index`: **U**
    - `Temporal Offset`: ***Empty***
- Set the measurement errors by pressing the `Select Errors` button. Choose **3% for all random and systematic errors.**
- Run MAPIT by pressing the `Run` button.

5.3.3. *Baseline data exploration*

- Start by plotting the various statistical quantities (i.e. MUF, CUMUF, SEID, SITMUF, and Page's trend test)
    - Note that the results look similar to the `Normal` dataset.
    - MUF plot should look similar to the image below.

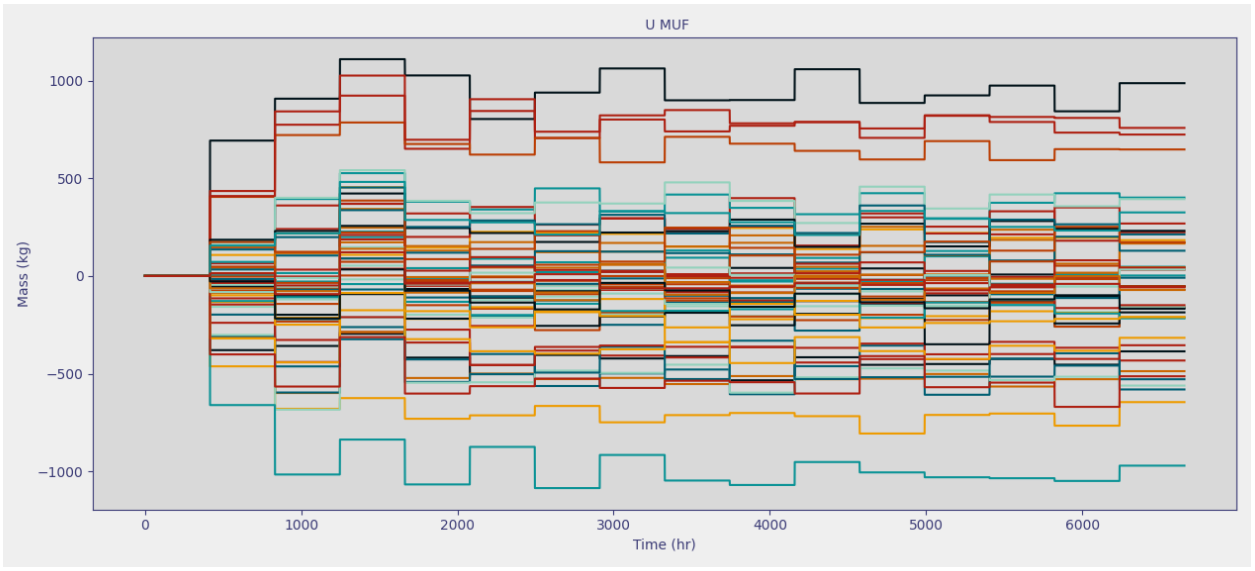


Figure 5-10: Several MUF iterations

- Continue by examining the fuel pins. Do so by selecting Ground Truth Data for the data type and Fuel Pins (output) for the location.
    - Here, Ground Truth Data refers to the true value of the fuel pins (output), which can never be observed in practice.
    - Note: There are many fuel pins that are generated at a fuel fabrication facility. Plotting Fuel Pins (output) could take several seconds depending on your hardware.
    - Your plot should look similar to the image below.

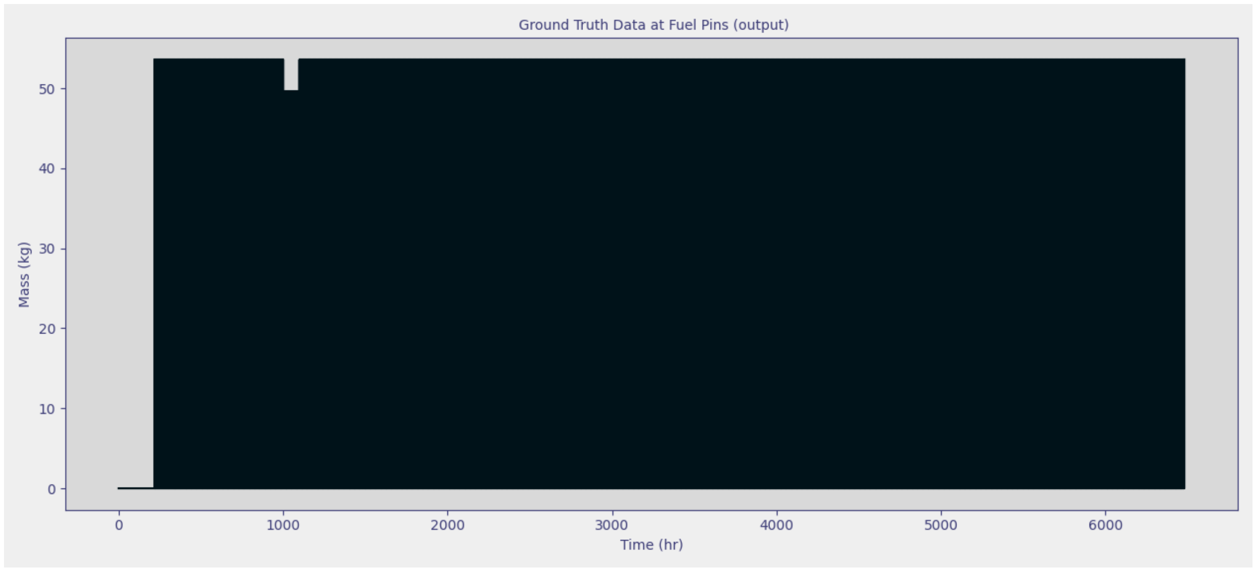


Figure 5-11: Ground truth observation of fuel pin output flow

5.3.4. *Explore impact of lower uncertainty*

- Perform the calculations again using a value of **0.5% for all random and systematic errors**.
    - MAPIT does not need to be restarted. Simply press `Select Errors` and use the dropdown menus to select the new error values. Then press `Run`.
- After MAPIT has run with the updated error values, try plotting the statistical quantities again. You should notice some observable changes have occurred.
    - MUF shows a distinct change during the material balance period in which the material loss occurs (see image below).
    - CUMUF, SEID, SITMUF, and Page's trend test on SITMUF should all exhibit changes due to the presence of the material loss.

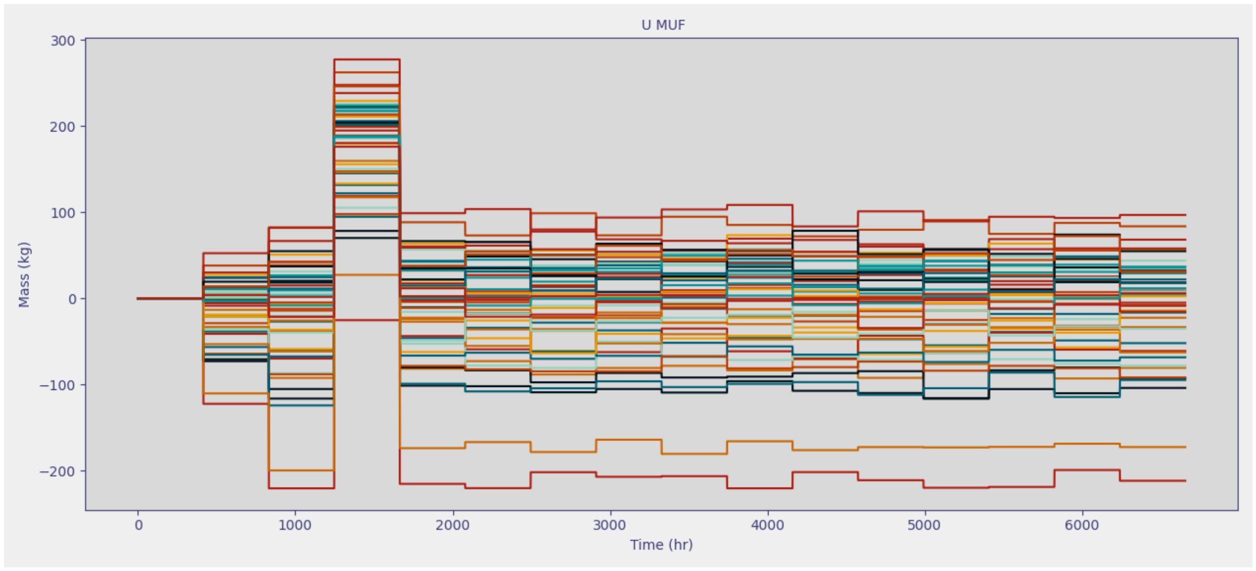


Figure 5-12: Several MUF iterations under loss conditions

Note

This example demonstrates the importance of precise measurement systems. This particular loss was too small to be reliably detected at the 3% uncertainty level.

5.3.5. *Summary*

This exercise introduced concepts related to notional material loss. The abrupt material loss was originally not visible at the 3% measurement uncertainty level. However, after lowering the uncertainty, the change in MUF was clearly visible. This is an important phenomena in safeguards that must be considered when selecting measurement systems for key measurement points. In this exercise you:

- Explored MUF
- Used MAPIT to look at the ground truth data
- Explored the impact of measurement uncertainty

5.4. Exercise 4: Quantifying probability of detection

5.4.1. *Objective*

Understand how MAPIT can be used to evaluate probability of detection. So far, exercises have focused on exploring qualitative changes that are induced by changes in measurement uncertainty or material loss. This section will focus on developing quantitative metrics for safeguards performance.

Caution

This exercise assumes that you are familiar with MAPIT and can perform tasks discussed in the previous exercises; Sections 5.1, 5.2, and 5.3.

5.4.2. Problem setup

- Start this exercise by launching MAPIT, selecting the Normal dataset.
  - This tutorial starts by assuming you are at the main MAPIT interface
- Select all the checkboxes for available statistical tests
  - A total of five (5) checkboxes should be checked
    - MUF
    - Cumulative MUF
    - Sigma MUF
    - SITMUF
    - Page's test on SITMUF
- Next, configure the required parameters in the statistical box using the same parameters from exercise 1 and 2:
  - `MBP`: **416**
  - `Iterations`: **50**
    - If running on a lower performance device, try running with `Iterations`:**20**
  - `Analysis Element/ Index`: **U**
  - `Temporal Offset`: ***Empty***
- Set the measurement errors by pressing the Select Errors button. Choose **0.5% for all random and systematic errors.**
- Run MAPIT by pressing the `Run` button.

Note

Safeguards statistical tests are commonly tuned to have an average false alarm probability of 5% per year. The next steps will use MAPIT functionality to determine this threshold for Page's trend test on SITMUF.

Caution

The data provided in these examples are 6480 hours long, which translate to about 270 days (a reasonable estimate of an operational year). Therefore, one iteration is a single year of simulated operation. When using your own data, use caution in determining statistical thresholds if your datasets have different lengths of time.

5.4.3. *Determining statistical thresholds*

- Plot Page's trend test on SITMUF
  - This is labeled as U Page's test on SITMUF in MAPIT
  - Plotting a quantity is necessary to use the threshold functionality

Note

Plots in this excercise are using 300 iterations to have better statistics, so if less iterations are being run the results may differ slightly from those shown here. In large iteration datasets, MAPIT only plots 15 iterations to ensure the plotting window is not overloaded.

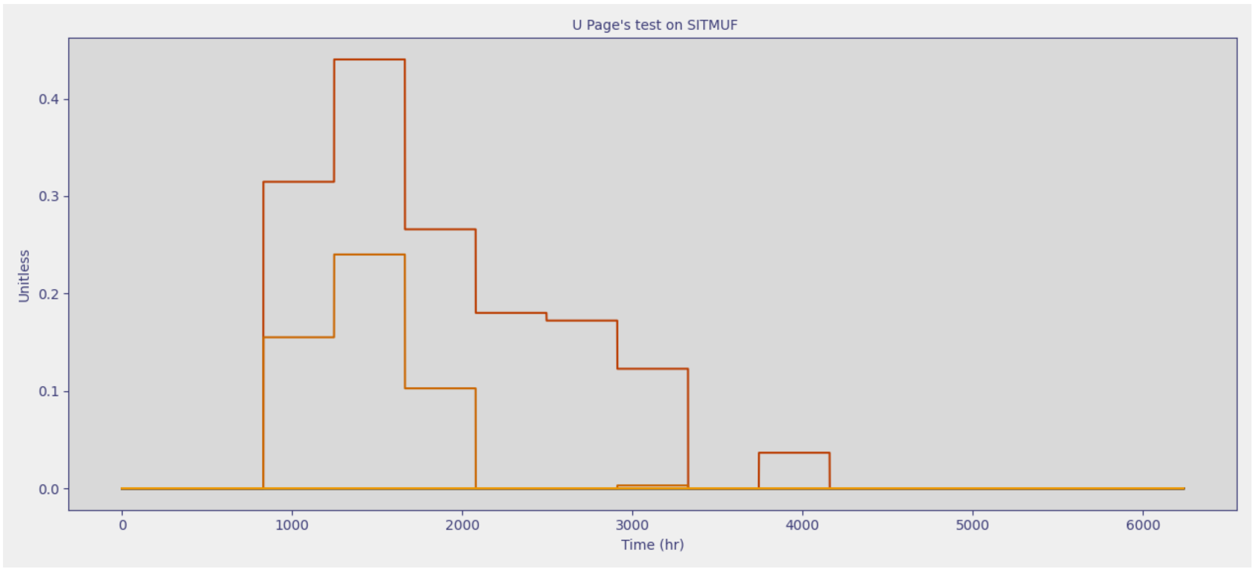


Figure 5-13: Page scores for several SITMUF iterations

- Use the `Statistical Thresholds` box to determine an appropriate threshold (i.e. 5% FAP)
  - This is performed by entering values in the `Enter Threshold` box
  - MAPIT returns the number of iterations that exceed this threshold
    - MAPIT operates on the entire dataset, not just the iterations that are plotted. This can be important in cases where many iterations are calculated and only a few are plotted.

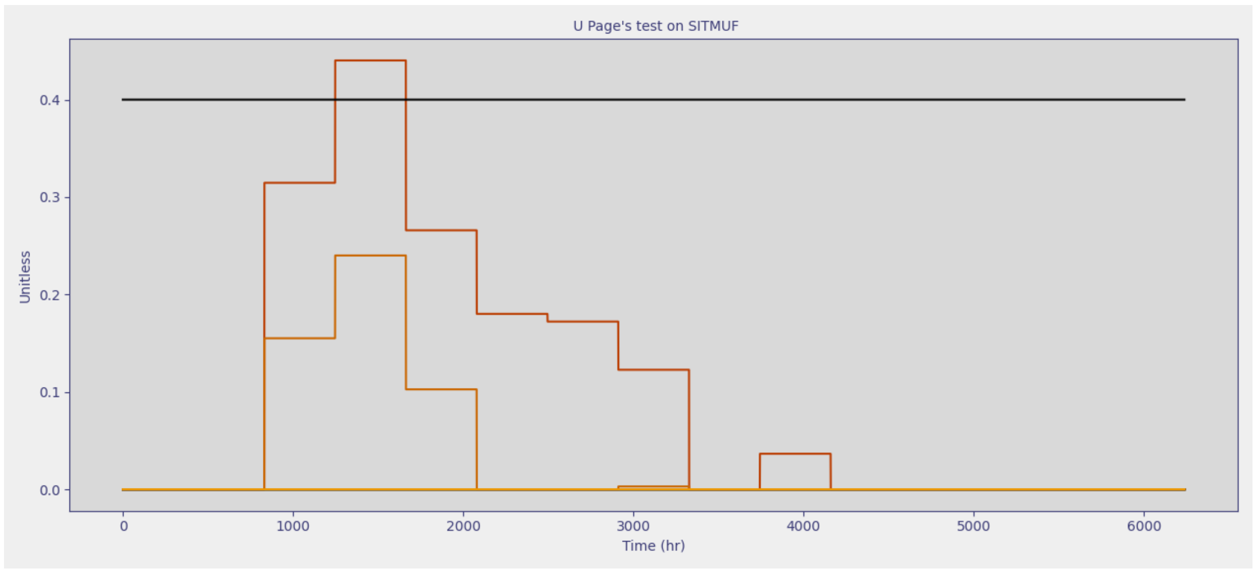


Figure 5-14: Setting a threshold on Page's scores

- Your threshold should be near 0.5
    - Due to the randomness of the calculations, your threshold might be slightly different
    - Increasing the number of iterations can help obtain a more precise estimate of the threshold
        - The uncertainty in the threshold itself should roughly decrease with sqrt(iterations).

5.4.4. *Evaluating probability of detection*

- Load the Abrupt dataset.
- Select the same options from steps 2, 3, and 4 then run MAPIT.
    - Select all the options in the Tests/Uranium box
    - Select a uncertainty a value of **0.5% for all random and systematic errors**
    - `MBP`: **416**
    - `Iterations`: **50**
        - If running on a lower performance device, try running with `Iterations`:**20**
    - `Analysis Element/ Index`: **U**
    - `Temporal Offset`: ***Empty***
- Plot Page's trend test on U SITMUF
    - This is labeled as U Page's test on SITMUF in MAPIT

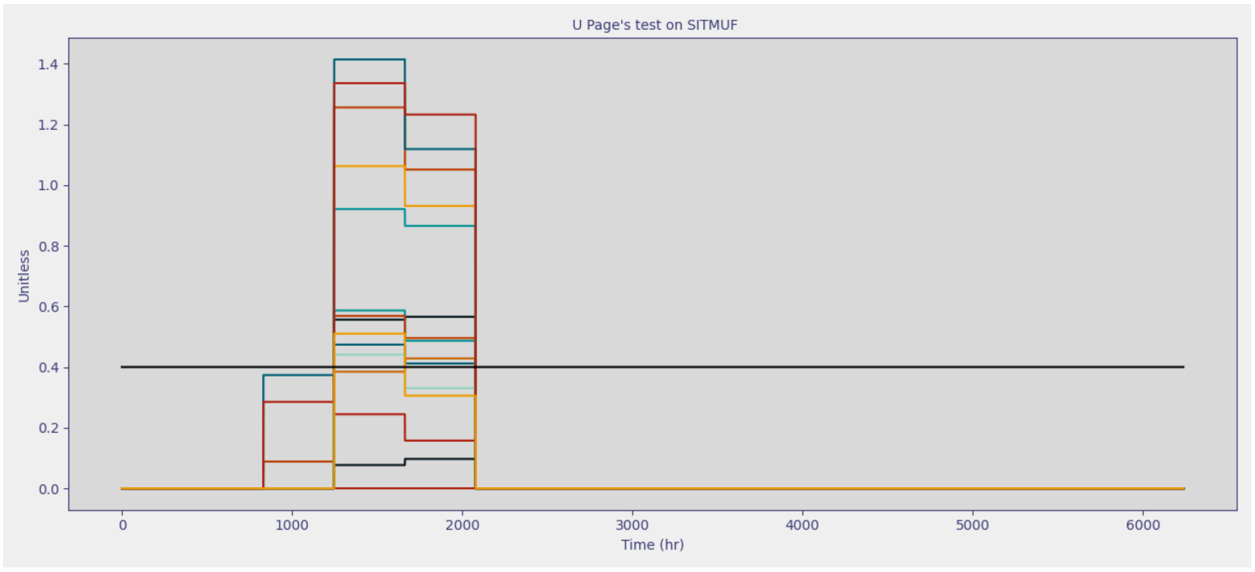


Figure 5-15: Threshold evaluation on Page's trend test on SITMUF for the abrupt loss dataset

- Enter in the previously determined threshold to determine the probability of detection for this material loss.
    - The value should be approximately 50-60%

> **Note**
>
> In this exercise, we changed the measurement uncertainties for all locations to be the same value. However, in practice, different measurement technologies are deployed based on safeguards need. On your own, try changing just some high impact measurement locations to 0.5% while leaving others at higher levels to see if you can reach similar results.

5.4.5. Summary

This exercise introduced the the SITMUF test in MAPIT along with the capability to set threshold and evaluate probability of detection. In this exercise you:

- Set a threshold based on a false alarm probability
- Evaluated probabilities of detection
- Explored the SITMUF transform
- Explored Page's trend test on SITMUF

6. API

Page intentionally left blank.
API documentation starts on the next page due to formatting.

API home

Added in version 1.40.0: Added parallel capabilities. Note that we do not yet provide guidance on optimal settings for parallel batching.

Changed in version 1.40.0: API breaking changes! Instead of accessing statistical tests directly, a new object, `MBArea`, is the preferred way to perform analyses using the API. The `StatsTests` module is provided for educational purposes only and it is expected that users instead user `MBArea` functions. See the API example notebooks for more details.

core

AuxFunctions

`MAPIT.core.AuxFunctions.`**`trapSum`**`(`*relevantIndex*, *time*, *data*, *IT=None*, *baseline_zero=1e-10*`)`

Function performs trapezoidal integration on a dataset segment. This is required for bulk facility flows that might need integration before use within statistical tests.

In some cases, flows might be represented as discontinuous pulses of material, in which case, special care is needed to identify the non-zero regions of the dataset to enable proper integration.

This function first identified a list of non-zero pulses of material before performing trapezoidal integration on each non-zero pulse segment. *np.trapz* is used to perform the integration. See the numpy documentation for more information.

$\int y(x)dx$

Parameters

- **`relevantIndex`** (*`ndarray`*) – An array that expresses the relevant time slice, with boolean values (0 = not relevant, 1 = relevant), with shape $[m, j]$ where m is the total number of relevant samples and j is the total number of iterations.
- **`time`** (*`ndarray`*) – An array containing the total number of timestep values under consideration for the analysis. Has shape $[n, j]$ where n is the total number of samples under consideration and $n > m$ for m in *relevantIndex* and j is the total number of iterations.
- **`data`** (*`ndarray`*) – An array containing the total number of samples under consideration for the analysis. Should have the same shape as **time**.
- **`baseline_zero`** (*`float`*) – A float that expresses the threshold below which values are considered zero. Important as datasets often do not represent zero as exactly zero for a variety of reasons.

Returns

An array of shape $[1, j]$ containing the integrated total for each iteration over the time specified by **time[relevantIndex]**.

Return type

ndarray

Preprocessing

`MAPIT.core.Preprocessing.`**`SimErrors`**`(`*rawData*, *ErrorMatrix*, *iterations*, *GUIObject=None*, *doTQDM=True*, *batchSize=10*, *dopar=False*, *bar=None*`)`

Function to add simulated measurement error. Supports variable sample rates. Assumes the traditional multiplicative measurement error model:

$M_{i,j} = T(1 + R_{i,j} + S_j)$

Random errors: $R_{i,j} \sim \mathcal{N}(0, {\delta_{R}}_j^2)$

Systematic errors: $S_j \sim \mathcal{N}(0, {\delta_{S}}_j^2)$

where i is the measurement time and j is the location

Parameters

- **`rawData`** (*`list`*) – Raw data to apply errors to, list of 2D ndarrays. Each entry in the list should correspond to a different location and the shape of ndarray in the list should be [MxN] where M is the sample dimension (number of samples) and N is the elemental dimension, if applicable. If only considering one element, each ndarray in the rawData list should be [Mx1].
- **`ErrorMatrix`** (*`ndarray`*) – 2D ndarray of shape [Mx2] describing the relative standard deviation to apply to `rawData`. M sample dimension in each input array and should be identical to M described in `rawData`. The second dimension (e.g., 2) refers to the random and systematic error respectively such that `ErrorMatrix[0,0]` refers to the random relative standard deviation of the first location and `ErrorMatrix[0,1]` refers to the systematic relative standard deviation.
- **`iterations`** (*`int`*) – Number of iterations to calculate
- **`GUIObject`** (*`obj, default=None`*) – GUI object for internal MAPIT use
- **`doTQDM`** (*`bool, default=True`*) – Controls the use of TQDM progress bar for command line or notebook operation.

Returns
List of arrays identical in shape to `rawData`. A list is returned so that each location can have a different sample rate.

Return type
list

StatsProcessor

Overview

The `StatsProcessor` module contains the `MBArea` object, which is the foundation of the MAPIT API. The first step in using MAPIT is to define a material balance area (i.e., `MBArea`). This object takes a number of parameters that are used to define a material balance area. The initial properties can be later modified by accessing the specific object properties.

After the `MBArea` is successfully defined, different statistical tests can be applied to the `MBArea` by calling object methods. The results are returned after calling the method, but results are also stored as object attributes that can be easily accessed.

Tip: The MBArea object is designed to streamline the analysis experience while providing flexibility. For example, a MBArea could be initialized, copied, then have a few properties modified to compare "what-if" scenarios.

```
# initalize with some variables
MBA0 = MBArea(...)

# clone MBArea
MBA1 = copy.copy(MB0)

# modify input term errors
MBA1.inputErrorMatrix = otherErrorMatrix

# calculate sigma MUF
MBA0.SEMUF()
MBA1.SEMUF()

# do comparison between baseline and modified input error cases
# ...
```

Important: If modifying the error matrix *after* having calculated errors or a statistical quantity, the errors must be recalculated using the calcErrors method.

Parallel Processing

MAPIT provides parallel processing capabilities through the (Ray)[https://www.ray.io/] library. By default, Ray provides a local dashboard at 127.0.0.1:8265 which can be used to monitor progress and view job related statistics. Two key parameters are used for parallel processing; ncpu and nbatch. ncpu controls the number of CPUs provided to Ray whereas nbatch is the number of iterations to process for each task. Once provided, each Ray worker (total is equal to ncpu) works through a queue of tasks. Each task returns some of the iterations requested by the user (defined by nbatch) until all results are processed. A table showing the relationship between user specified variables iterations, ncpu, and nbatch the number of tasks performed by each worker is shown below. Workers process tasks in the queue until the queue is completed.

iterations	ncpu	nbatch	total number of tasks	tasks completed per worker
100	5	1	100	20
100	5	5	20	4
100	5	20	5	1

nbatch is provided as a parameter as there is overhead incurred when copying data to/from workers. If nbatch is too small, then parallel processing might be slower than sequential processing if the calculation time is small compared to the memory copying time. We do not provide guidance on setting these parameters as performance will be system specific.

Classes

class MAPIT.core.StatsProcessor.**MBArea**(*rawInput*, *rawInventory*, *rawOutput*, *rawInputTimes*, *rawInventoryTimes*, *rawOutputTimes*, *inputErrorMatrix*, *inventoryErrorMatrix*, *outputErrorMatrix*, *mbaTime*, *iterations=1*, *dopar=False*, *ncpu=1*, *nbatch=1*, *GUIObject=None*, *dataOffset=0*, *rebaseToZero=True*, *doTQDM=True*)

Object representing a material balance area.

Parameters

- **rawInput** (*list of ndarrays*) – Raw input data for the material balance area, list of 2D ndarrays. Each entry in the list should correspond to a different location and the shape of ndarray in the list should be [MxN] where M is the sample dimension (number of samples) and N is the isotopic dimension, if applicable. If only considering one isotope, each ndarray in the rawData list should be [Mx1]. It is expected that M will have rate units (i.e., kg/hr) as this quantity will be integrated.
- **rawInventory** (*list of ndarrays*) – Raw inventory data for the material balance area, list of 2D ndarrays. Shape structure is the same as `rawInput`. It is expected that M will have mass units (i.e., kg) as this quantity will not be integrated.
- **rawOutput** (*list of ndarrays*) – Raw output data for the material balance area, list of 2D ndarrays. Shape structure is the same as `rawInput`. It is expected that M will have rate units (i.e., kg/hr) as this quantity will be integrated.
- **rawInputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(rawInputTimes)* and the shape of each list entry (ndarray) should be the same as for *rawInput*. Each entry in each ndarray should correspond to a timestamp indicating when the value was taken.
- **rawInventoryTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Shape structure is the same as `rawInputTimes`.
- **rawOutputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Shape structure is the same as `rawInputTimes`.
- **inputErrorMatrix** (*ndarray*) – 2D ndarray of shape [Mx2] describing the relative standard deviation to apply to `rawInput`. M sample dimension in each input array and should be identical to M described in `rawInput`. The second dimension (e.g., 2) refers to the random and systematic error respectively such that `ErrorMatrix[0,0]` refers to the random relative standard deviation of the first location and `ErrorMatrix[0,1]` refers to the systematic relative standard deviation.
- **inventoryErrorMatrix** (*ndarray*) – 2D ndarray with the same shape structure as `inputErrorMatrix` describing errors to apply to `rawInventory`.
- **outputErrorMatrix** (*ndarray*) – 2D ndarray with the same shape structure as `inputErrorMatrix` describing errors to apply to `rawOutput`.
- **mbaTime** (*int*) – The material balance period.
- **iterations** (*int, default=1*) – Number of statistical realizations.
- **doPar** (*bool, default=False*) – Controls the use of parallel processing provided by Ray. If used, progress can be monitored on a local dashboard that is accessible at http://127.0.0.1:8265.

- **ncpu** (*int, default=1*) – The number of CPUs to use if parallel processing is enabled.
- **nbatch** (*int, default=1*) – The number of batches to process for each job.
- **GUIObject** (*object, default=None*) – An object containing MAPIT GUI parameters. Only used interally by the GUI.
- **dataOffset** (*int, default=0*) – Offset to apply to the data. If specified, data before this value in time will be removed. For example, if dataOffset=273, then any data with a corresponding time before 273 will be excluded from calculations.
- **rebaseToZero** (*bool, default=False*) – Used in conjunction with dataOffset. If true, then times after `dataOffset` will be rebased to start at zero (i.e., if dataOffset=273, then t=274 will be rebased to be t=1).
- **doTQDM** (*bool, default=True*) – Boolean used to control progress bar of calculations.

Returns
None

calcCUMUF()

Calculates cumulative MUF using `StatsTests.CUMUF`. The result is returned and stored as an attribute after the calculation is complete. Automatically calculates MUF if not present as an attribute.

Returns
CUMUF sequence with identical shape to the input MUF.

Return type
ndarray

calcErrors()

Function that applies the specified error matrices to the supplied raw data and stores the results as object attributes. Uses the `Preprocessing.SimErrors` implementation.

Returns
None

calcMUF()

Calculates MUF using `StatsTests.MUF`. The result is returned and stored as an attribute after the calculation is complete.

Returns

MUF sequence with shape $[n, j]$ where n length equal to the maximum time based on the number of material balances that could be constructed given the user provided `mbaTime` and number of samples in the input data. j is the number of iterations given as input. The term n is calculated by finding the minimum of each of the provided input times.

For example:

```
import numpy as np

time1[-1] = 400
time2[-1] = 300
time3[-1] = 800

n = np.floor(
        np.min(
        (time1,time2,time3)))
```

Tip: MAPIT doesn't assume that time series provided have zero value if unspecified. For example, if a time series starts at t=800, it is assumed that values before t=800 are undefined so MUF cannot be calculated before t=800. The user can modify input data such that values before t=800 are present, but zero, if that assumption is valid.

Return type
ndarray

`calcPageTT``()`

Calculates Page's trend test on SITMUF using `StatsTests.PageTrendTest`. The result is returned and stored as an attribute after the calculation is complete. Automatically calculates SITMUF if not present as an attribute.

Returns
The results of the trend test which has shape $[m, n]$.

Return type
ndarray

`calcSEMUF``()`

Calculates σ MUF using `StatsTests.SEMUF`. The result is returned and stored as an attribute after the calculation is complete. Automatically calculates MUF if not present as an attribute.

Returns

- SEID (ndarray): sequence with shape $[n, j, 1]$ where n is the number of material balances and j is the number of iterations given as input. The term n is calculated by finding the minimum of each of the provided input times.
- SEMUFContribR (ndarray): the random contribution to the overall SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.
- SEMUFContribS (ndarray): the systematic contribution to the overall SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.
- ObservedValues (ndarray): the observed values used to calculate SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.

Return type
tuple (ndarray, ndarray, ndarray, ndarray)

`calcSITMUF``()`

Calculates SITMUF using `StatsTests.SITMUF`. The result is returned and stored as an attribute after the calculation is complete. Automatically calculates MUF if not present as an attribute.

Returns
SITMUF sequence with shape $[n, j]$ where n length equal to the maximum time based on the number of material balances that could be constructed given the user provided MBP and number of samples in the input data and j is the number of iterations given as input. As is the case with MUF, the term n is calculated by finding the minimum of each of the provided input times.

Return type
ndarray

StatsTests

`MAPIT.core.StatsTests.`**`CUMUF`**`(`*MUF*, *GUIObject=None*, *doTQDM=True*, *ispar=False*`)`

This function performs the cumulative MUF test. This is simply the sum of all previous MUF values at a particular time.

$\text{CUMUF}_t = \sum_{t=0}^{t} \text{MUF}_t$

Parameters

- **`MUF`** (*`ndarray`*) – MUF sequence with shape $[n, j]$ where n is the number of iterations and j is the temporal dimension. Expects a continuous valued MUF sequence that is similar in format to what is returned by *core.StatsTests.MUF*.
- **`GUIParams`** (*`object, default=None`*) – An optional object that carries GUI related parameters when the API is used inside the MAPIT GUI.
- **`doTQDM`** (*`bool, default=True`*) – Controls the use of TQDM progress bar for command line or notebook operation.

Returns

CUMUF sequence with identical shape to the input MUF.

Return type

ndarray

`MAPIT.core.StatsTests.`**`MUF`**`(`*inputAppliedError*, *processedInputTimes*, *inventoryAppliedError*, *processedInventoryTimes*, *outputAppliedError*, *processedOutputTimes*, *MBP*, *GUIObject=None*, *GUIparams=None*, *doTQDM=True*, *ispar=False*`)`

Function to calculate Material Unaccounted For (MUF), which is sometimes also called ID (inventory difference). Specifically calculates the material balance sequence given some input time series.

$\text{MUF}_t = I_t - O_t - (C_t - C_{t-1})$

I_t is input at time t

O_t is output at t

C_t is inventory at time t (note C is used to denote *container* to have clearer notation rather than using I with subscripts for both inventory and input)

Important: The lengths and shapes of appliedErrors and processedTimes should be the same. For example:

```
assert(len(inputAppliedError) == len(processedInputTimes)
assert(inputAppliedError[0].shape == processedInputTimes[0].shape)
```

See the Input guide for more information.

Parameters

- **`inputAppliedError`** (*`list of ndarrays`*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites (as opposed to ground truths). Inputs are assumed to be flows in units of $\frac{1}{s}$ and will be integrated.

- **processedInputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInputTimes)* and the shape of each list entry (ndarray) should be the same as for *inputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **inventoryAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Inventories are assumed to be in units of mass and will *not* be integrated.
- **processedInventoryTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInventoryTimes)* and shape of each list entry (ndarray) should be the same as for *inventoryAppliedError*. Each entry in the ndarray should corresond to a timestamp indicating when the value was taken.
- **outputAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Outputs are assumed to be in flows with units of $\frac{1}{s}$ and will be integrated.
- **processedOutputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedOutputTimes)* and shape of each list entry (ndarray) should be the same as for *outputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **MBP** (*float*) – Defines the material balance period.
- **GUIObject** (*object, default=None*) – An optional object that carries GUI related references when the API is used inside the MAPIT GUI.
- **GUIParams** (*object, default=None*) – An optional object that carries GUI related parameters when the API is used inside the MAPIT GUI.
- **doTQDM** (*bool, default=True*) – Controls the use of TQDM progress bar for command line or notebook operation.

Returns

MUF sequence with shape $[n, j]$ where n length equal to the maximum time based on the number of material balances that could be constructed given the user provided MBP and number of samples in the input data and j is the number of iterations given as input. The term n is calculated by finding the minimum of each of the provided input times.

For example:

```python
import numpy as np

time1[-1] = 400
time2[-1] = 300
time3[-1] = 800

n = np.floor(
      np.min(
        (time1,time2,time3)))
```

Return type

ndarray

MAPIT.core.StatsTests.**PageTrendTest** (*inQty*, *MBP*, *MBPs*, *K=0.5*, *GUIObject=None*, *doTQDM=True*)

Function for calculating Page's trend test, which is commonly applied to the SITMUF sequence. Formally compares the null hypothesis that there is no trend versus the alternate trend where there is a trend.

Parameters

- **inQty** (*ndarray*) – A ndarray with shape $[m, n]$ where m is the number of iterations and n is the total number of timesteps.
- **MBP** (*float*) – A float expressing the material balance period.
- **MBPs** (*float*) – The total number of material balance periods present in **inQty**.
- **K** (*float, default = 0.5*) – Parameter in the trend test.
- **GUIObject** (*object, default=None*) – An optional object that carries GUI related references when the API is used inside the MAPIT GUI.
- **GUIParams** (*object, default=None*) – An optional object that carries GUI related parameters when the API is used inside the MAPIT GUI.
- **doTQDM** (*bool, default=True*) – Controls the use of TQDM progress bar for command line or notebook operation.

Returns

The results of the trend test which has shape $[m, n]$.

Return type

ndarray

MAPIT.core.StatsTests.**SEMUF** (*inputAppliedError*, *processedInputTimes*, *inventoryAppliedError*, *processedInventoryTimes*, *outputAppliedError*, *processedOutputTimes*, *MBP*, *ErrorMatrix*, *GUIObject=None*, *doTQDM=True*, *ispar=False*)

Function for calculating standard error of the material balance sequence (often called SEID or Standard Error of Inventory Difference; σ_{ID}). This is accomplished by assuming the error incurred at each location (specified in the ErrorMatrix) rather than estimating it emperically, which is difficult in practice. The equation used here is suitable for most traditional bulk facilities such as enrichment or reprocessing where input and output flows are independent. This function should **not** be used for facilitiy types where there are more complex statistical dependencies between input, inventory, and output terms (e.g., molten salt reactors). See guide XX for more information.

Parameters

- **inputAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites (as opposed to ground truths). Inputs are assumed to be flows in units of $\frac{1}{s}$ and will be integrated.
- **processedInputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInputTimes)* and the shape of each list entry (ndarray) should be the same as for *inputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **inventoryAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Inventories are assumed to be in units of mass and will *not* be integrated.
- **processedInventoryTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInventoryTimes)* and shape of each list entry

(ndarray) should be the same as for *inventoryAppliedError*. Each entry in the ndarray should corresond to a timestamp indicating when the value was taken.

- **outputAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Outputs are assumed to be in flows with units of $\frac{1}{s}$ and will be integrated.
- **processedOutputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedOutputTimes)* and shape of each list entry (ndarray) should be the same as for *outputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **MBP** (*float*) – Defines the material balance period.
- **ErrorMatrix** (*ndarray*) – mx1 A ndarray shaped $[M, 2]$ where M is the *total* number of locations across inputs, inventories, and outputs stacked together (in that order) and 2 refers to the relative random and systematic errors. For example with 2 inputs, 2 inventories, and 2 outputs, ErrorMatrix[3,1] would be the relative systematic error of inventory 2. See guide XX for more information.
- **GUIObject** (*object, default=None*) – An optional object that carries GUI related references when the API is used inside the MAPIT GUI.
- **GUIParams** (*object, default=None*) – An optional object that carries GUI related parameters when the API is used inside the MAPIT GUI.
- **doTQDM** (*bool, default=True*) – Controls the use of TQDM progress bar for command line or notebook operation.

Returns

tuple containing:

SEID (ndarray): sequence with shape $[n, j, 1]$ where n is the number of material balances and j is the number of iterations given as input. The term n is calculated by finding the minimum of each of the provided input times.

SEMUFContribR (ndarray): the random contribution to the overall SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.

SEMUFContribS (ndarray): the systematic contribution to the overall SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.

ObservedValues (ndarray): the observed values used to calculate SEMUF with shape $[j, l, n]$ where j is the number of iterations given as input, l is the total number of locations stacked in the order [inputs, inventories, outputs] and n is the number of material balances.

Return type

(tuple)

`MAPIT.core.StatsTests.`**`SITMUF`**(*inputAppliedError*, *processedInputTimes*, *inventoryAppliedError*, *processedInventoryTimes*, *outputAppliedError*, *processedOutputTimes*, *ErrorMatrix*, *MUF*, *MBP*, *GUIObject=None*, *doTQDM=True*, *ispar=False*)

Function that carries out the standardized independent transformation of MUF. More detailed information can be found in the guide XX.

Parameters

- **inputAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites (as opposed to ground truths). Inputs are assumed to be flows in units of $\frac{1}{s}$ and will be integrated.
- **processedInputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of input locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInputTimes)* and the shape of each list entry (ndarray) should be the same as for *inputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **inventoryAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Inventories are assumed to be in units of mass and will *not* be integrated.
- **processedInventoryTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of inventory locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedInventoryTimes)* and shape of each list entry (ndarray) should be the same as for *inventoryAppliedError*. Each entry in the ndarray should corresond to a timestamp indicating when the value was taken.
- **outputAppliedError** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. This array should reflect observed quantites. Outputs are assumed to be in flows with units of $\frac{1}{s}$ and will be integrated.
- **processedOutputTimes** (*list of ndarrays*) – A list of ndarrays that has length equal to the total number of output locations. Each array should be $[m, 1]$ in shape where m is the number of samples. *len(processedOutputTimes)* and shape of each list entry (ndarray) should be the same as for *outputAppliedError*. Each entry in the ndarray should correspond to a timestamp indicating when the value was taken.
- **MBP** (*float*) – Defines the material balance period.
- **GUIObject** (*object, default=None*) – An optional object that carries GUI related references when the API is used inside the MAPIT GUI.
- **GUIParams** (*object, default=None*) – An optional object that carries GUI related parameters when the API is used inside the MAPIT GUI.
- **doTQDM** – Controls the use of TQDM progress bar for command line or notebook operation.

REFERENCES

[1] I. A. E. Agency, "The structure and content of agreements between the agency and states required in connection with the tready of the non-proliferation of nuclear weapons." https://www.iaea.org/sites/default/files/publications/documents/infcircs/1972/infcirc153.pdf, June 1972.

[2] D. of Energy, "Nuclear material control and accountability." https://www.directives.doe.gov/directives-documents/400-series/0474.2-BOrder-a, February 2023.

[3] N. R. Commission, "Material control and accounting of special nuclear material." https://www.nrc.gov/reading-rm/doc-collections/cfr/part074/index.html, September 2015.

[4] T. Burr and M. S. Hamada, "Revisiting statistical aspects of nuclear material accounting," *Science and Technology of Nuclear Installations*, March 2013.

[5] O. Alique, Y. Aregbe, R. Bencardino, R. Binner, T. Burr, J. A. Chapman, S. Croft, A. Fellerman, T. Krieger, K. Martin, *et al.*, "Statistical error model-based and gum-based analysis of measurement uncertainties in nuclear safeguards–a reconciliation," *ESARDA BULLETIN*, vol. 64, no. FZJ-2023-00429, pp. 10–29, 2022.

[6] I. A. E. Agency, *International Target Values for Measurement Uncertainties in Safeguarding Nuclear Materials*, December 2022.

[7] D. G. Cacuci, *Handbook of Nuclear Engineering: Vol. 1: Nuclear Engineering Fundamentals; Vol. 2: Reactor Design; Vol. 3: Reactor Analysis; Vol. 4: Reactors of Generations III and IV; Vol. 5: Fuel Cycles, Decommissioning, Waste Disposal and Safeguards*, vol. 1. Springer Science & Business Media, 2010.

[8] R. Avenhaus and J. Jaech, "On subdividing material balances in time and/or space," *Journal of Nuclear Materials Management*, vol. 10, 1981.

[9] T. Speed and D. Culpin, "The role of statistics in nuclear materials accounting: issues and problems," *Journal of the Royal Statistical Society: Series A (General)*, vol. 149, no. 4, pp. 281–300, 1986.

[10] R. R. Picard, "Sequential analysis of material balances," *Journal of Nuclear Materials Management*, vol. 15, 1987.

[11] K. K. Pillay, "Fundamentals of materials accounting for nuclear safeguards," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 1989.

[12] B. Jones, "Calculation of diversion detection using the sitmuf sequence and page's test," in *Nuclear safeguards technology 1986*, 1987.

[13] B. Jones, "Near real time material accountancy," *ESARDA Bulletin*, vol. 7, pp. 19–22, 1984.

[14] B. Jones, "Near real time material accountancy using SITMUF and a joint page's test: comparison with MUF and CUMUF tests," 1988.

[15] E. S. Page, "Continuous inspection schemes," *Biometrika*, June 1954.

[16] R. J. Jones, E. V. Weinstock, and W. R. Kane, "Detailed description of an ssac at the facility level for a low-enriched uranium conversion and fuel fabrication facility," tech. rep., International Atomic Energy Agency, 1984.

DISTRIBUTION

Email-External [redacted]

Name	Company Email Address	Company Name
Mike Browne	mcbrowne@lanl.gov	LANL

Email-Internal [redacted]

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov
Nathan Shoman	8845	nshoman@sandia.gov