KU LEUVEN

Simulation Engines for Maintenance Events in Heterogeneous Machines

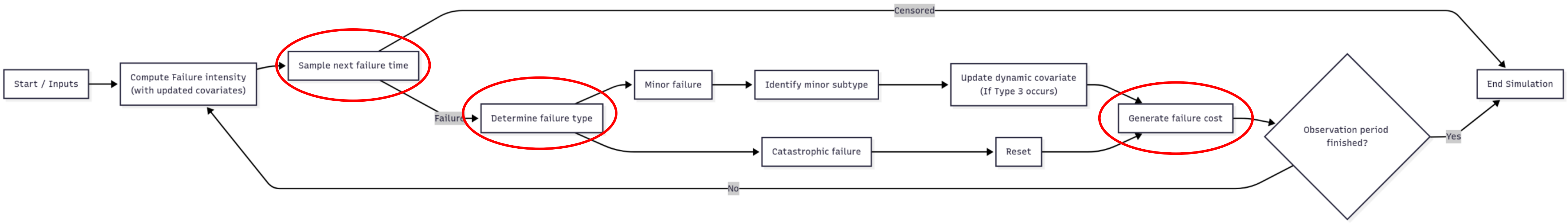
Yue Cai, Robert Boute, Laurens Deprez

A virtual laboratory for reliability research and practice

We have two versions of the simulation engine: maintenance can be time-based or condition-based

	Time-based maintenance (TBM)	Condition-based maintenance (CBM)
Focus	Failure & maintenance events	Degradation process
Model basis	Failure intensity models (e.g., NHPP)	Stochastic degradation models
Feature dependency	Hazard intensity adjusted by covariates	Degradation increments affected by covariates
PM requirement	Time based PM interval	PM policy (threshold based/time based)
Outputs	Failure/maintenance time, type, cost	Degradation paths, event log

Architecture of the Simulation Framework: The TBM simulation engine generates historical maintenance logs given periodic maintenance policy



Setup

Number of machines
Observation time & discretization

Maintenance policy

PM interval & effectiveness

Failure process models

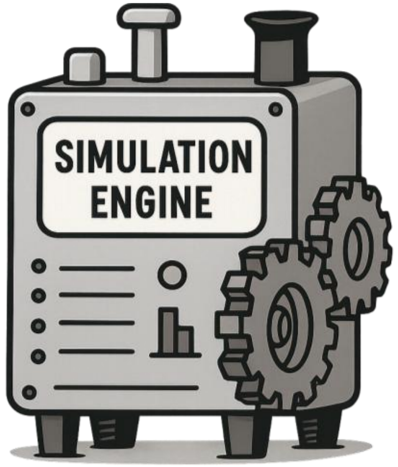
Failure intensity

Machine heterogeneity

$$X(t) = (\underbrace{x_1, x_2, \ldots, x_n}_{\text{Fixed covariates}}, \underbrace{x_{n+1}(t), x_{n+2}(t), \ldots}_{\text{Dynamic covariates}})$$

Cost structure

Failure cost + PM cost
(Parameters for gamma)



Event identification

Machine id
censor status

Event time

Failure time
PM time

Event type

PM; Catastrophic failure;
Minor failure 1, 2, 3

Event cost

Configurable inputs of the TBM simulation engine

Setup

```
results_df, all_machines_dynamic_covs = simulate_all_machines(n_machines, t_obs, m, n_dynamic_features, delta_t,
            #Preventive maintenance interval for each machine and PM effectiveness parameter ("push")
            T_machines, push,
            #Failure process
            include_minor, model_type_minor, shape_minor, scale_minor, intercept_minor, with_covariates_minor,
            include_catas, model_type_catas, shape_catas, scale_catas, intercept_catas, with_covariates_catas,
            # Machine heterogeneity
            fixed_covs, machines_dynamic_covs, beta_fixed, beta_dynamic, beta_multinom_fixed, beta_multinom_dynamic,
            n_minor_types, cov_update_fn,
            # cost-related (lists for minor types)
            gamma_coeffs_cat_fixed, gamma_coeffs_cat_dynamic,
            gamma_coeffs_minor_fixed_list, gamma_coeffs_minor_dynamic_list,
            theta_copula,
            shape_cat, scale_cat, loc_fixed_cat,
            shape_minor_list, scale_minor_list, loc_fixed_minor_list,
            use_covariates, minor_combo_map,
            # PM cost
            gamma_coeffs_pm_fixed, gamma_coeffs_pm_dynamic, shape_pm, scale_pm, loc_fixed_pm)
```

Maintenance policy

Failure process models

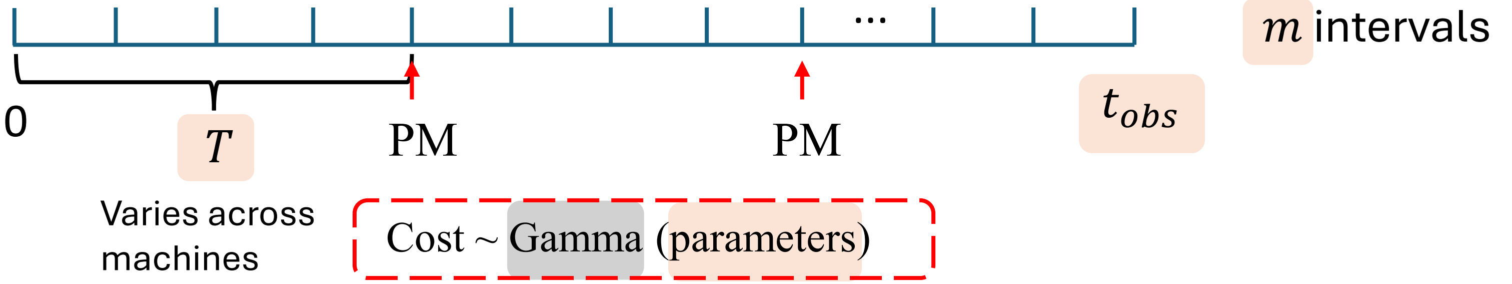
Machine heterogeneity

Cost structure

Refer to

run_tbm_example.py

The TBM engine generates failure times for machines with individual PM intervals



Failure time generation

Failure type determination

Failure cost modeling

Failure arrival ~ NHPP

Failure intensity: Catastrophic + Minor

$$\lambda_i(t) = \boxed{\lambda_{0,i}(t)} \exp(x_f' \beta_{f,i} + x_d'(t)\beta_{d,i})$$

$$\lambda(t) = \lambda_c(t) + \lambda_m(\boxed{V(t)})$$

Failure time (via Inverse Transform Sampling)

Linear, log-linear, Weibull

PM resets virtual age

$$V(t) = (t \bmod T) + (1 - \boxed{k_{\lambda_0}}) \cdot T \cdot \lfloor t/T \rfloor$$

Code for failure time generation under machine-specific PM policies

Failure arrival ~ NHPP

Failure intensity: Catastrophic + Minor

$$\lambda_i(t) = \boxed{\lambda_{0,i}(t)}\exp(x_f'\beta_{f,i} + x_d'(t)\beta_{d,i})$$

$$\lambda(t) = \lambda_c(t) + \lambda_m(\boxed{V(t)})$$

Failure time (via Inverse Transform Sampling)

Refer to

time_based/hazard.py

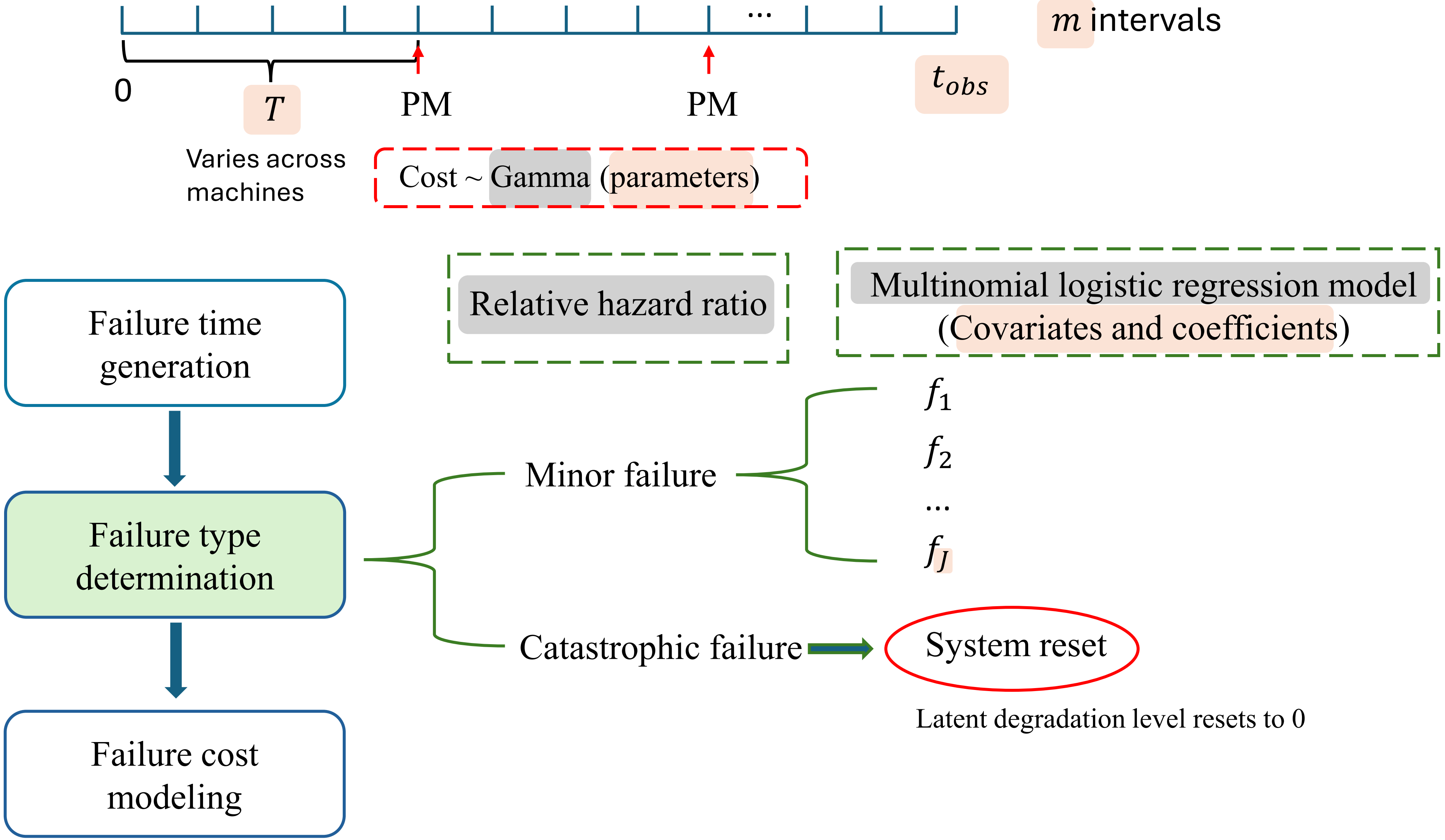
```python
"""
Hazard function implementations for failure intensity modeling.

This module contains:
- lambda_f: Unified hazard function for both minor and catastrophic failures
- integrated_lambda_closed: Closed-form cumulative hazard
- integrated_lambda_numeric: Numerical integration of hazard
- compute_cumulative_integrals: Update cumulative integrals
"""

import numpy as np
from scipy.integrate import quad


def lambda_f(
    machine_id, t, pm_affects=True, T=None, push=0.0, scale=None, intercept=None, shape=None,
    fixed_covs=None, dynamic_cov_t=None, beta_fixed=None, beta_dynamic=None,
    model_type="weibull", with_covariates=True
```

Refer to

time_based/failure_time.py

```python
"""
Failure time generation module.

This module contains the getFailureTime function that determines when
failures occur based on cumulative hazard functions.
"""

import numpy as np
from .hazard import compute_cumulative_integrals


def getFailureTime(
    s, cumulative_integrals_minor, cumulative_integrals_catas, cumulative_integrals,
    dynamic_covs_changed, machine_id, valid_indices, m, delta_t,
    # Minor failure parameters
    include_minor=True,
    model_type_minor="linear", shape_minor=None, scale_minor=None, intercept_minor=None,
    fixed_covs=None, dynamic_covs=None, beta_fixed=None, beta_dynamic=None,
    with_covariates_minor=True, T=None, push=0.0,
    # Catastrophic failure parameters
    include_catas=True,
    model_type_catas="linear", shape_catas=None, scale_catas=None, intercept_catas=None,
    with_covariates_catas=False
):
```

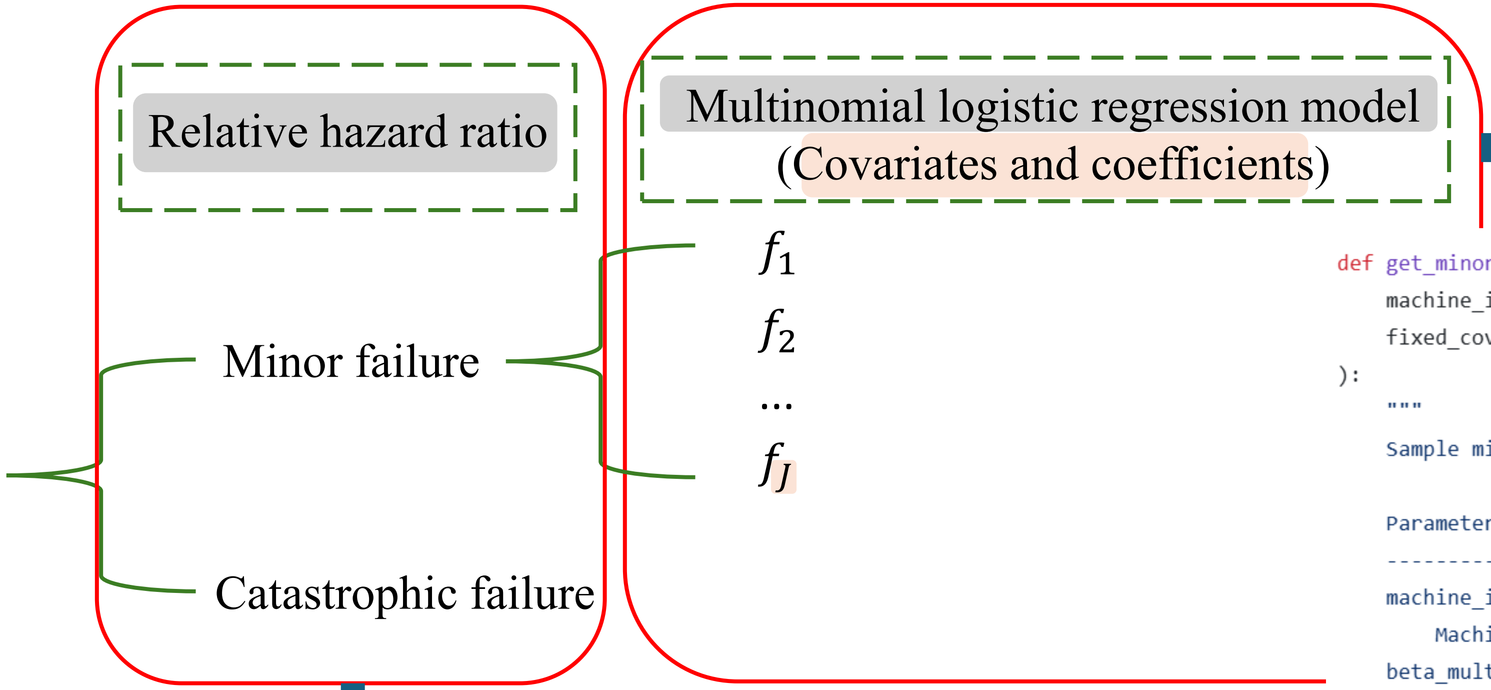
The TBM engine determines the failure type using covariate-driven competing risks and multinomial logistic models



Code for failure type determination

Refer to

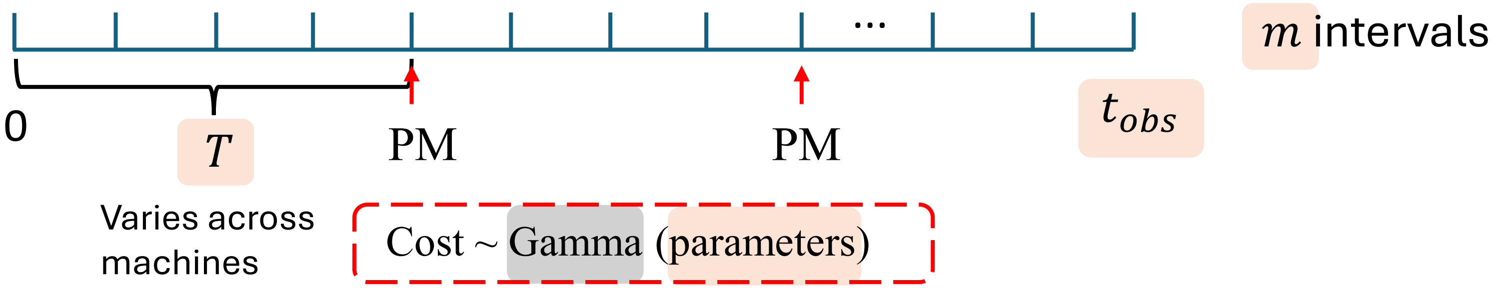
time_based/failure_type.py




```python
def get_minor_failure_type(
    machine_id, beta_multinom_fixed, beta_multinom_dynamic,
    fixed_covs, dynamic_cov_t, n_minor_types
):
    """
    Sample minor failure subtype using multinomial logistic regression with covariates

    Parameters:
    -----------
    machine_id : int
        Machine identifier
    beta_multinom_fixed : array-like
        Fixed covariate coefficients, shape (n_fixed_features, n_minor_types-1)
    beta_multinom_dynamic : array-like
        Dynamic covariate coefficients, shape (n_dynamic_features, n_minor_types-1)
    fixed_covs : array-like
        Fixed covariates for all machines, shape (n_machines, n_fixed_features)
    dynamic_cov_t : array-like
        Dynamic covariates at current time, shape (n_dynamic_features,)
    n_minor_types : int
        Number of minor failure types

    Returns:
    --------
    int : Minor failure subtype (1 to n_minor_types)
    """
```

```python
def get_failure_type(
    machine_id, ft, T, push, scale, intercept, shape, with_covariates_minor, model_type_m
    fixed_covs, dynamic_cov_t, beta_fixed, beta_dynamic,
    scale_c, shape_c, intercept_c, model_type_catas, with_covariates_catas
):
    """
    Determine whether failure at time ft is minor or catastrophic
```

The TBM Engine models failure costs using covariate-dependent gamma



Failure time generation

Failure type determination

Failure cost modeling

Gamma sampling (with covariates)

Frank Copula (for dependent costs)

Parameters

Code for failure costs modelling

Gamma sampling (with covariates)

```
def generate_gamma_cost(fixed_covs, dynamic_cov_t, gamma_coeffs_fixed, gamma_coeffs_dynamic,
                        machine_id, a, b, loc_fixed, use_covariates):
    """
    Generate cost from Gamma distribution with optional covariate effects
```

Frank Copula (for dependent costs)

```
# Correlated cost generation using Frank Copula
def get_failure_costs_with_frank_copula(machine_id, dynamic_cov_t, fixed_covs,
    gamma_coeffs_y1_fixed, gamma_coeffs_y1_dynamic,
    gamma_coeffs_y2_fixed, gamma_coeffs_y2_dynamic,
    theta, a1=2.0, a2=2.0, b1=1.0, b2=1.0,
    loc_fixed1=0.0, loc_fixed2=0.0, use_covariates=True):
    """
    Generate two correlated failure costs using Frank Copula
    Supports both fixed and dynamic covariates
```

Refer to

time_based/failure_cost.py

```
"""
Cost simulation module.

This module handles:
- Gamma distribution cost generation with covariates
- Frank Copula for correlated costs
- Cost simulation for all failure types
- PM cost generation
"""
```

Flow of multi-machine simulation in the TBM engine

```
"""
Main simulation orchestration module.

This module contains the high-level simulation functions:
- simulate_single_cycle: Simulate from AGAN state to failure or observation end
- simulation_complete_observed_period: Handle catastrophic resets
- simulate_machine_full_observed_period: Complete machine simulation with costs
- simulate_all_machines: Multi-machine simulation
"""
..
def simulate_all_machines(n_machines, t_obs, m, n_dynamic_features, delta_t, T_machines, push,
                          include_minor, model_type_minor, shape_minor, scale_minor, intercept_minor, with_covariates_minor,
                          include_catas, model_type_catas, shape_catas, scale_catas, intercept_catas, with_covariates_catas,
                          fixed_covs, machines_dynamic_covs, beta_fixed, beta_dynamic, beta_multinom_fixed, beta_multinom_dynamic,
                          n_minor_types, cov_update_fn,
                          # cost-related (lists for minor types)
                          gamma_coeffs_cat_fixed, gamma_coeffs_cat_dynamic,

    for machine_id in range(1, n_machines + 1):
        T = T_machines[machine_id]
        # Get the per-machine initial dynamic covariates
        dynamic_covs = machines_dynamic_covs.get(machine_id) if machines_dynamic_covs else None  #initial dynamic covs

        # Run single-machine simulation
        df_failures, machine_dynamic_covs, pm_index, pm_times, pm_costs = simulate_machine_full_observed_period(
            t_obs, machine_id, m, n_dynamic_features, delta_t, T, push,

        # Aggregate all machines
        final_df = pd.concat(all_results, ignore_index=True)
```

Refer to

time_based/simulation.py

Loop over machines

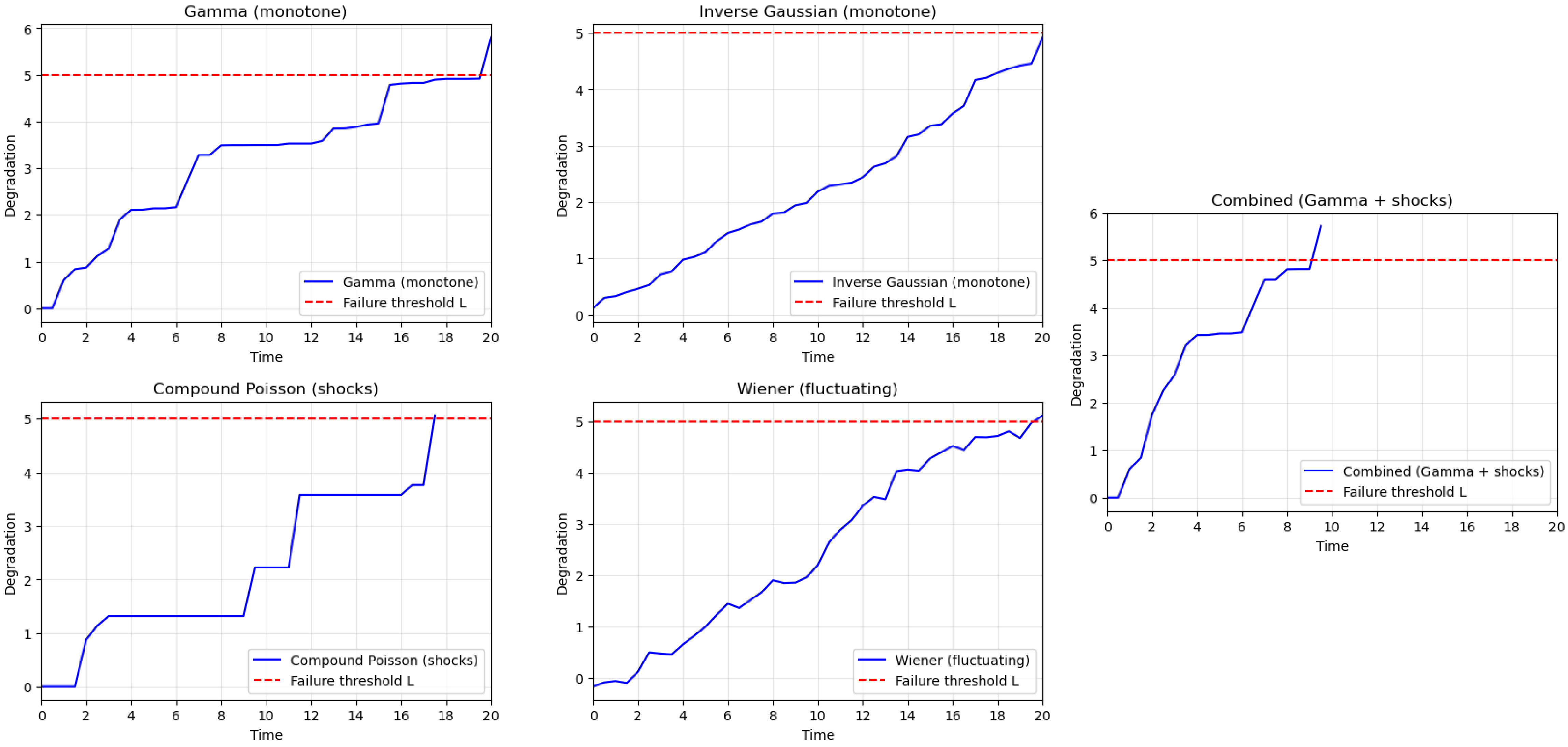
Simulate one machine

Aggregate results

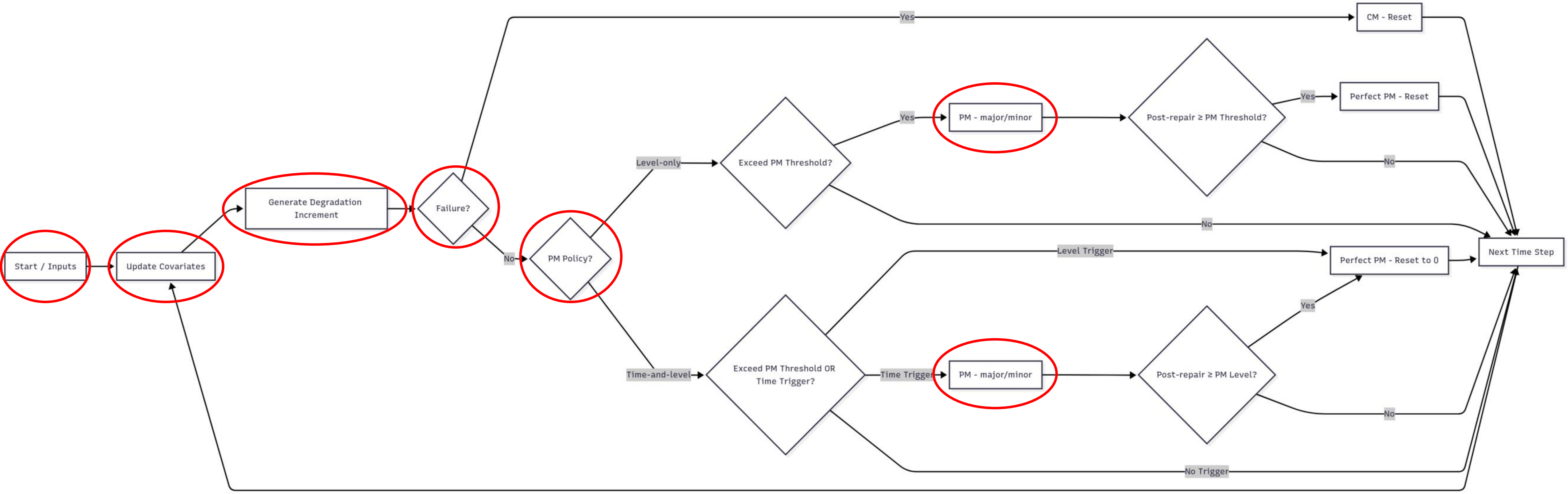
The result is a log table with maintenance event data and its feature values

	machine_id	event_time	event_type	censor_status	event_cost	fixed_cov_1	fixed_cov_2	fixed_cov_3	fixed_cov_4	dynamic_cov_1_at_event
0	1	0.910	f_2	1	68.763955	1.0	1.0	0.0	1.0	0.0
1	1	1.000	pm	1	60.465617	1.0	1.0	0.0	1.0	0.0
2	1	1.105	f_2	1	124.086685	1.0	1.0	0.0	1.0	0.0
3	1	2.000	pm	1	59.990668	1.0	1.0	0.0	1.0	0.0
4	1	3.000	pm	1	45.915114	1.0	1.0	0.0	1.0	0.0
5	1	4.000	pm	1	54.933842	1.0	1.0	0.0	1.0	0.0
6	1	4.300	c	1	410.878490	1.0	1.0	0.0	1.0	0.0
7	1	5.000	pm	0	165.807104	1.0	1.0	0.0	1.0	0.0
8	2	0.525	f_1	1	150.726853	0.0	0.0	1.0	0.0	0.0
9	2	1.000	pm	1	45.336625	0.0	0.0	1.0	0.0	0.0
10	2	2.000	pm	1	33.130589	0.0	0.0	1.0	0.0	0.0
11	2	2.865	f_2	1	88.565947	0.0	0.0	1.0	0.0	0.0
12	2	3.000	pm	1	48.741522	0.0	0.0	1.0	0.0	0.0
13	2	3.665	f_3	1	91.236713	0.0	0.0	1.0	0.0	1.0
14	2	3.750	f_1	1	314.555790	0.0	0.0	1.0	0.0	1.0
15	2	4.000	pm	1	80.021697	0.0	0.0	1.0	0.0	1.0
16	2	4.360	f_1	1	204.948800	0.0	0.0	1.0	0.0	1.0
17	2	4.385	f_1	1	346.866142	0.0	0.0	1.0	0.0	1.0
18	2	4.440	f_2	1	90.656132	0.0	0.0	1.0	0.0	1.0
19	2	4.585	f_2	1	80.993307	0.0	0.0	1.0	0.0	1.0
20	2	4.935	f_1	1	133.096948	0.0	0.0	1.0	0.0	1.0
21	2	5.000	pm	0	56.546781	0.0	0.0	1.0	0.0	1.0

The CBM engine makes use of degradation paths



Architecture of the Simulation Framework: Condition-based maintenance events are generated based on the degradation paths



The simulation engine generates historical degradation paths and maintenance logs based on user-specified inputs

Setup

Number of machines
Observation time & discretization

Maintenance policy

PM threshold (and PM time)
(level or time-and-level)
Repair strategy (major:p / minor:$(1-p)$)

Degradation process

Machine heterogeneity

$$X(t) = (\underbrace{x_1, x_2, \ldots, x_n}_{\text{Fixed covariates}}, \underbrace{x_{n+1}(t), x_{n+2}(t), \ldots}_{\text{Dynamic covariates}})$$

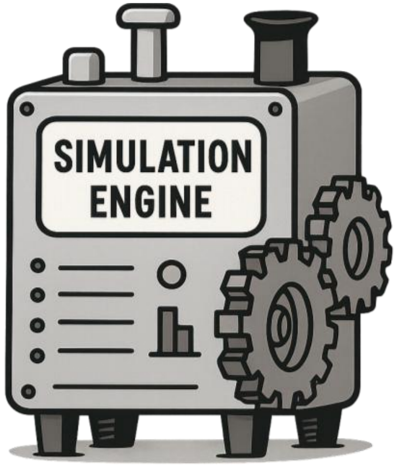
Parameter adjustment

- Compound Poisson

$$\lambda(t) = \lambda_0 \exp(x_f' \beta_f + x_d'(t)\beta_d)$$

- Gamma
- IG
- Wiener

$$\theta(t) = \theta_0 \exp(x_f' \beta_f + x_d'(t)\beta_d)$$



Degradation path

Event identification

Machine id
censor status

Event time

Failure time
PM time

Event type

PM; CM

Configurable inputs of the CBM simulation engine

```
fleet_results['Scenario2_Gamma'] = simulate_multiple_machines(
    n_machines=N_MACHINES,
    degradation_type=scenario2_degradation_type,
    degradation_params=scenario2_degradation_params,
    covariate_specs=scenario2_covariates,
    covariate_effects=scenario2_covariate_effects,
    dt=DT,
    PM_level=scenario2_pm_level,
    PM_interval=scenario2_pm_interval,
    L=L,
    x0=X0,
    repair_func=sample_post_repair_mixed,
    repair_params=repair_params_shared,
    obs_time=OBS_TIME,
    random_seed_base=RANDOM_SEED_BASE + 1000,
    noise = noise_params_shared,
    cost_params=cost_params_shared,
    cost_covariate_specs=cost_covariates_shared,
    cost_covariate_effects=scenario2_cost_covariate_effects
)
```

Setup

Degradation process

Machine heterogeneity

Maintenance policy

Cost structure

Refer to

run_cbm_example.py

Modeling degradation with PHM intensity and covariate-dependent increments

Refer to *condition_based/degradation.py*

Degradation process

Parameter adjustment

- Compound Poisson

$$\lambda(t) = \lambda_0 \exp(x_f'\beta_f + x_d'(t)\beta_d)$$

- Gamma
- IG
- Wiener

$$\theta(t) = \theta_0 \exp(x_f'\beta_f + x_d'(t)\beta_d)$$

Machine heterogeneity

$$X(t) = (\underbrace{x_1, x_2, \ldots, x_n}_{\text{Fixed covariates}}, \underbrace{x_{n+1}(t), x_{n+2}(t), \ldots}_{\text{Dynamic covariates}})$$

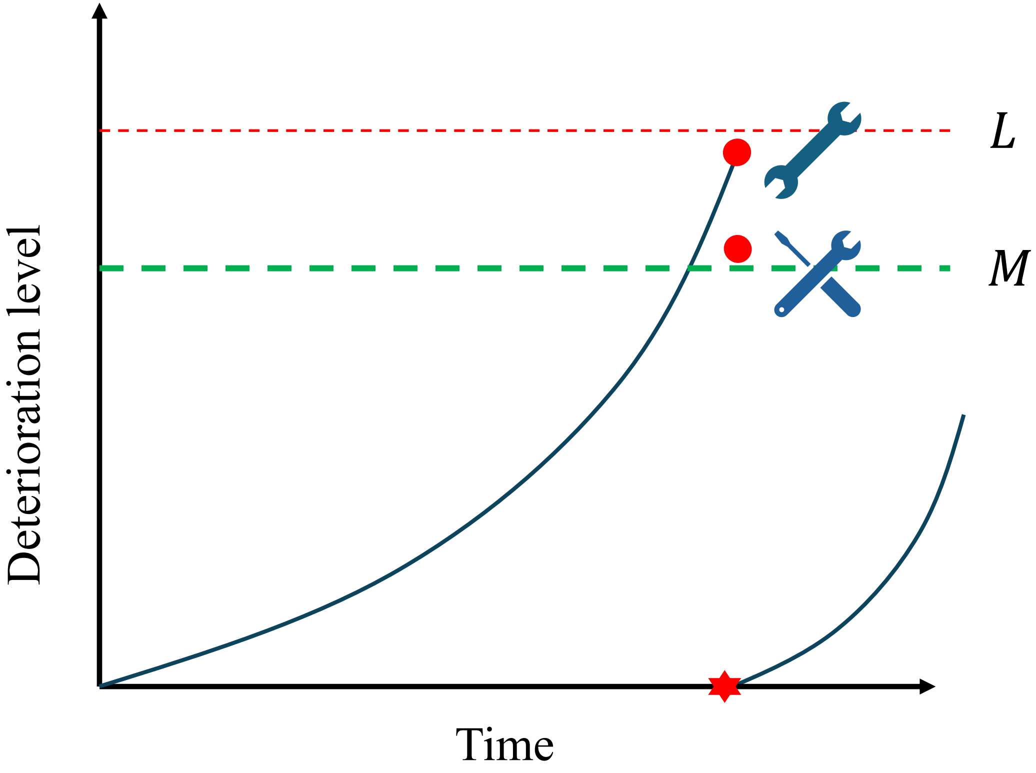
```python
def compute_phm_intensity(
    base_intensity: float,
    covariates: np.ndarray,
    beta_coeffs: np.ndarray,
    t: float = None
) -> float:
    """
    Compute Proportional Hazards Model (PHM) intensity function.

    Formula: λ(t|Z(t)) = λ₀(t) × exp(β'Z(t))
```

```python
def generate_degradation_increment_with_covariates(
    degradation_type: str,
    dt: float,
    params: Dict[str, Any],
    covariates: np.ndarray = None,
    covariate_effects: Dict[str, np.ndarray] = None
) -> float:
    """
    Generate degradation increment with covariate effects.

    Supports multiple stochastic processes:
    - gamma: Gamma process
    - inverse_gaussian: Inverse Gaussian process
    - wiener: Wiener process (Brownian motion with drift)
    - compound_poisson: Compound Poisson process with PHM for shock arrival
    - combined: Combination of base process and compound Poisson shocks
```

```python
def add_observation_noise(
    x: float,
    dt: float,
    noise: Optional[Dict[str, Any]]
) -> float:
    """
    Add observation noise to a scalar degradation level.

    Supports two types of noise:
    - additive_normal: x_obs = x + N(0, σ²)
    - brownian_increment: x_obs = x + N(0, σ²xdt)
```

Preventive Maintenance Policies: Threshold-based and scheduled triggering

Level-only PM

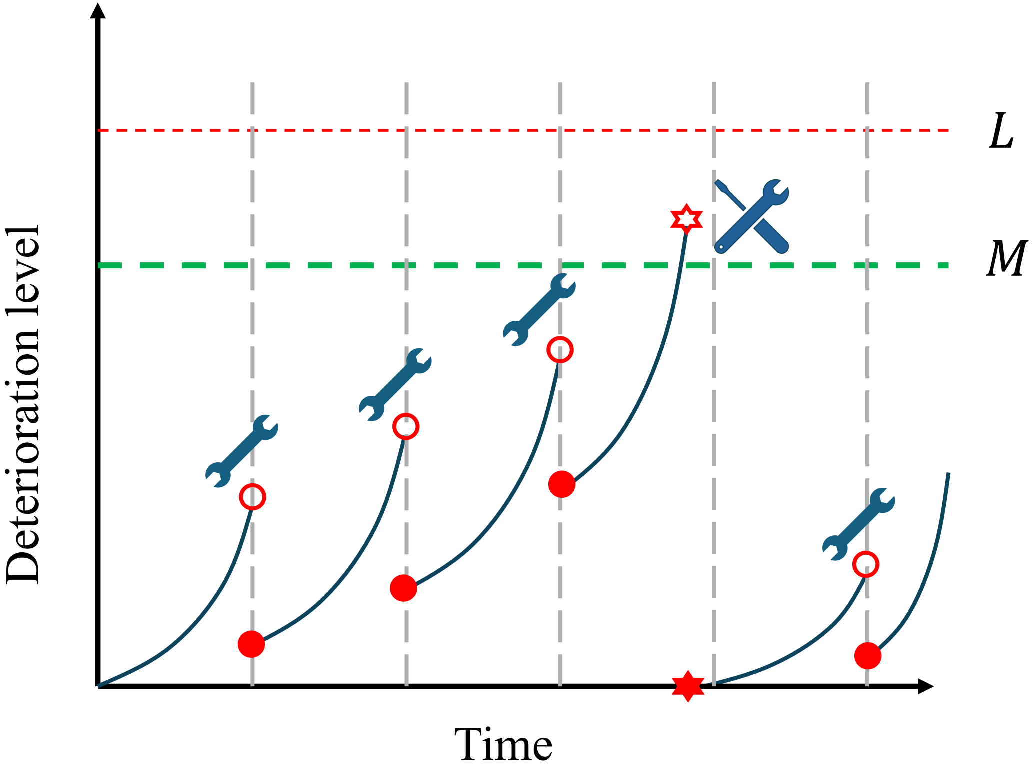


If $X'(t) \geq M$

$X(t) \geq M \rightarrow$ PM $\rightarrow X'(t) \rightarrow$ Perfect PM

PM resets degradation level

Time-and-level PM



$t \in \{T_{PM}, 2T_{PM}, 3T_{PM}, \ldots\} \rightarrow$ PM

$X(t) \geq M \rightarrow$ Perfect PM

Code for implementation of level-only and time-and-level PM policies

```python
def simulate_path_with_covariates(
    degradation_type: str = "compound_poisson",
    degradation_params: Dict[str, Any] = None,
    covariate_specs: List[CovariateSpec] = None,
    covariate_effects: Dict[str, np.ndarray] = None,
    dt: float = 0.01,
    PM_level: float = 2.0,
    PM_interval: float = None,
    L: float = 5.0,
    x0: float = 0.0,
    repair_func: Callable = None,
    repair_params: Dict = None,
    obs_time: float = 100.0,
    random_seed: int = None,
    noise: Optional[Dict[str, Any]] = None,
    cost_params: CostParams = None,
    cost_covariate_specs: List[CovariateSpec] = None,
    cost_covariate_effects: Dict[str, np.ndarray] = None
) -> Dict[str, Any]:
    """
    Simulate degradation path with covariates, distinguishing latent true wear (X_latent)
    and noisy observations (X_obs), and compute maintenance costs.

    Decision Logic:
    - CM (Corrective Maintenance): Triggered when x_latent >= L (true failure)
    - PM (Preventive Maintenance): Triggered when x_observed >= PM_level (decision based on observable info)

    Maintenance Strategies:
    - Level-only: PM triggered when observed degradation reaches PM_level
    - Time-and-level: PM triggered by either time (PM_interval) or level (PM_level)
```

Level-only PM

```python
# --- PM Logic (based on observed degradation level) ---
if not has_scheduled_pm:
    # Level-only strategy (based on observed level)
    if x_obs >= PM_level:
        z_lat = x_lat
        y_lat = repair_func(last_repair_post_lat, z_lat, params=repair_params)
```

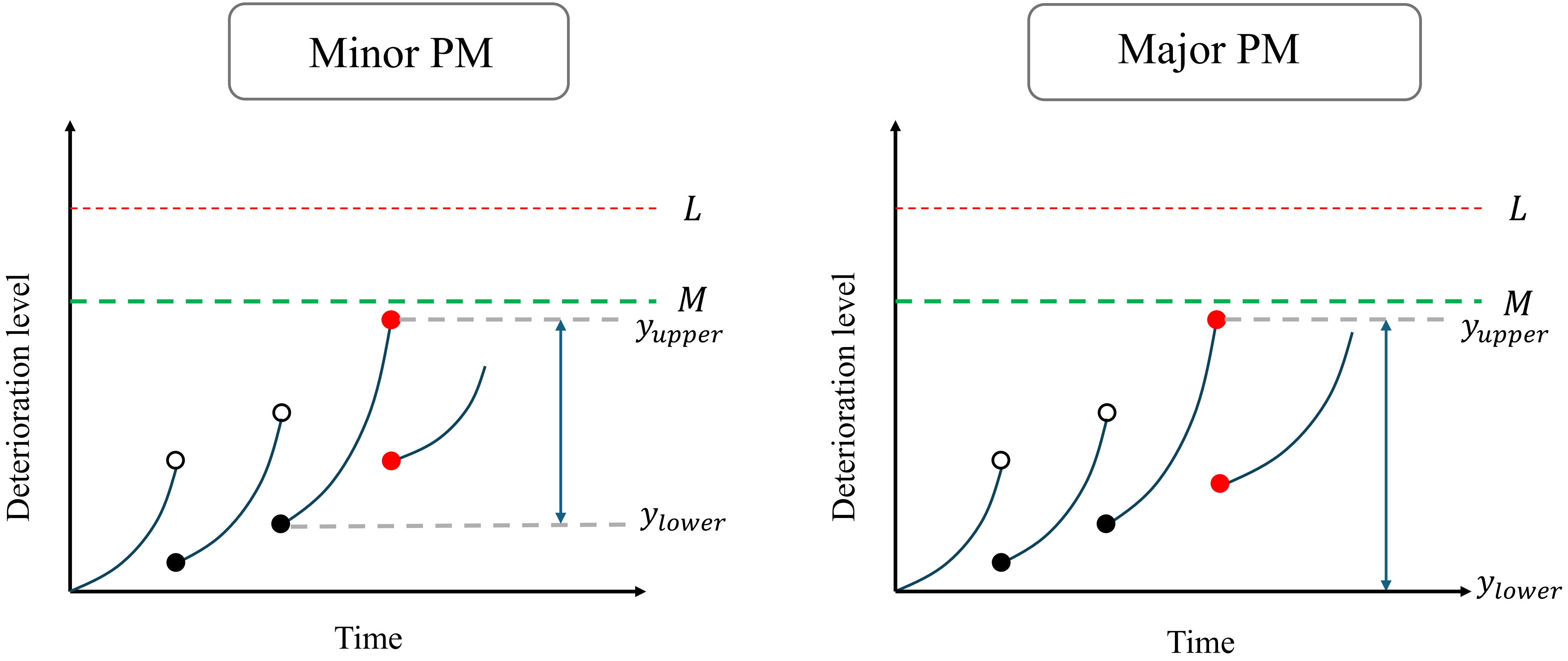
Time-and-level PM

```python
else:
    # Time-and-level strategy (based on observed level)
    level_triggered = (x_obs >= PM_level)
    time_triggered = (t - last_pm_time >= PM_interval)
```

Refer to

condition_based/single_machine_sim.py

Imperfect Repair Model: major and minor repairs



Post-repair degradation level ~ $U(y_{lower}, y_{upper})$

$$y_{lower} + (y_{upper} - y_{lower})B(a, b)$$

$$y_{upper} - (1-\rho)(y_{upper} - y_{lower})$$

Code for major and minor repairs

```python
def sample_post_repair_mixed(
    y_lower: float,
    y_upper: float,
    params: Optional[Dict[str, Any]] = None
) -> float:
    """
    Mixed/relaxed version of post-repair sampling for imperfect maintenance.

    This function models the degradation level after an imperfect repair.
    With probability p_major, performs major repair supporting [0, y_upper].
    With probability 1-p_major, performs minor repair supporting [y_lower, y_upper].

    Special Case Handling:
    - If y_upper <= y_lower: degradation level after last repair is still above PM threshold,
      execute perfect preventive maintenance (complete restoration to 0).
    - If y_upper <= 0: already at perfect condition, return 0.

    Args:
        y_lower: Lower bound (typically degradation level after last repair)
        y_upper: Upper bound (typically current degradation level before repair)
        params: Dictionary containing repair parameters:
            - p_major: Probability of major repair (default: 0.2)
            - dist_minor: Distribution for minor repair ('uniform'/'beta'/'proportional')
            - dist_major: Distribution for major repair ('uniform'/'beta'/'proportional')
            - a_minor, b_minor: Beta distribution parameters for minor repair
            - a_major, b_major: Beta distribution parameters for major repair
            - rho_minor: Proportional parameter for minor repair (reduction ratio)
            - rho_major: Proportional parameter for major repair (reduction ratio)

    Returns:
        Post-repair degradation level
```

Refer to

condition_based/repair.py

Cost modeling in the CBM simulation engine

Perfect PM & CM Cost

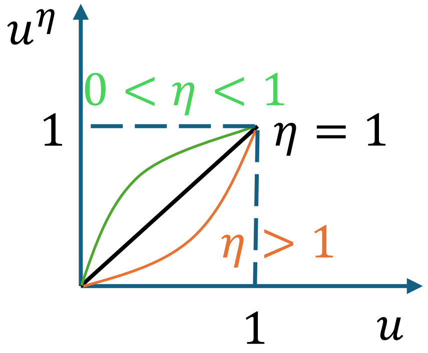
$$C_r(t)|W(t) \sim Gamma(k_r, \theta_r, l_r(t))$$

- $l_r(t)$: covariate-dependent location parameter
- $W(t)$: cost-related operational factors (operating environment + logistic factors)
- r ∈{PM, CM}; k_r: shape parameter; θ_r: scale parameter

Imperfect PM Cost

$$u = \frac{X_{before} - X_{after}}{X_{before}} \in [0,1]$$

$$C_{IPM}(t) = c_{fix}(t) + c_0 u^{\eta} + \varepsilon$$



- $c_{fix}(t)$: fixed cost affected by operational covariates $W(t)$; fixed value if no covariates
- c_0 : largest possible variable cost (maintenance cost)
- η: controls the curvature of the cost–maintenance effect relationship
  - $\eta = 1$: Linear
  - $\eta > 1$: Increasing marginal cost
  - $0 < \eta < 1$: Diminishing returns
- ε : Noise
- Implemented with: $C_{IPM} = \max(0, C_{IPM})$

Code for cost modeling in the CBM simulation engine

```python
def compute_maintenance_cost(
    maintenance_type: str,
    cost_params: CostParams,
    cost_covariates: np.ndarray = None,
    cost_covariate_effects: Dict[str, np.ndarray] = None,
    repair_effectiveness: float = None
) -> float:

    if maintenance_type == 'perfect_pm':
        # Perfect PM cost follows 3-parameter gamma distribution
        shape = cost_params.pm_shape
        scale = cost_params.pm_scale

        # Compute location parameter from covariates (linear effect)
        location = 0.0
        if cost_covariates is not None and cost_covariate_effects is not None:
            if 'pm_location' in cost_covariate_effects:
                beta = cost_covariate_effects['pm_location']
                location = np.dot(beta, cost_covariates)

        # Generate from 3-parameter gamma: shape, scale, location
        cost = gamma_dist.rvs(a=shape, scale=scale, loc=location)
        return cost

    elif maintenance_type == 'cm':
        # CM cost follows 3-parameter gamma distribution (higher than PM)
        shape = cost_params.cm_shape
        scale = cost_params.cm_scale

        # Compute location parameter from covariates (linear effect)
        location = 0.0
        if cost_covariates is not None and cost_covariate_effects is not None:
            if 'cm_location' in cost_covariate_effects:
                beta = cost_covariate_effects['cm_location']
                location = np.dot(beta, cost_covariates)

        # Generate from 3-parameter gamma: shape, scale, location
        cost = gamma_dist.rvs(a=shape, scale=scale, loc=location)
        return cost

    elif maintenance_type == 'imperfect_pm':
        # Imperfect PM: C_IPM = c_fix + c_0*u + ε
        if repair_effectiveness is None:
            raise ValueError("repair_effectiveness must be provided for imperfect_pm")
```

Refer to

condition_based/cost.py

Flow of multi-machine simulation in the CBM engine

```
"""
Simulate multiple machines with the same or different maintenance policies.

This function simulates a fleet of machines operating under the same
degradation parameters but potentially different maintenance policies,
with independent random realizations.

IMPORTANT:
- Fixed covariates (type='fixed') are independently sampled for each machine
- PM_level and PM_interval can be specified per machine or shared across fleet

def simulate_multiple_machines(
    n_machines: int,
    degradation_type: str = "compound_poisson",
    degradation_params: Dict[str, Any] = None,
    covariate_specs: List[CovariateSpec] = None,
    covariate_effects: Dict[str, np.ndarray] = None,
    dt: float = 0.01,
    PM_level: Union[float, List[float], Dict[int, float]] = 2.0,
    PM_interval: Union[float, List[float], Dict[int, float], None] = None,
    L: float = 5.0,
    x0: float = 0.0,
    repair_func: Any = None,
    repair_params: Dict = None,
    obs_time: float = 100.0,
    random_seed_base: int = None,
    noise: Optional[Dict[str, Any]] = None,
    cost_params: CostParams = None,
    cost_covariate_specs: List[CovariateSpec] = None,
    cost_covariate_effects: Dict[str, np.ndarray] = None
):
```

Loop over machines

```
for i in range(n_machines):
    # Set random seed for this machine
    seed = random_seed_base + i if random_seed_base is not None else None

    # Get machine-specific maintenance policy
    machine_pm_level = pm_levels[i]
    machine_pm_interval = pm_intervals[i]

    # Deep copy covariate specs so each machine can have independent fixed values
    machine_covariate_specs = copy.deepcopy(covariate_specs) if covariate_specs else None
    machine_cost_covariate_specs = copy.deepcopy(cost_covariate_specs) if cost_covariate_specs else None

    # Run simulation for this machine
    result = simulate_path_with_covariates(
        degradation_type=degradation_type,
        degradation_params=degradation_params,
        covariate_specs=machine_covariate_specs,
        covariate_effects=covariate_effects,
        dt=dt,
        PM_level=machine_pm_level,
        PM_interval=machine_pm_interval,
        L=L,
        x0=x0,
        repair_func=repair_func,
```

PM policy varies across machines

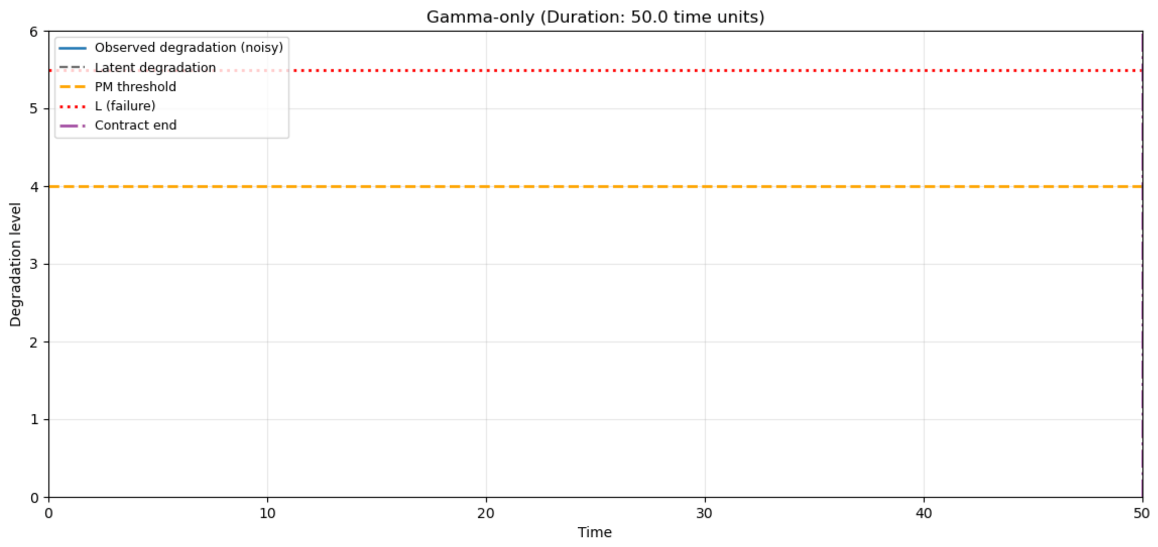
Simulate one machine

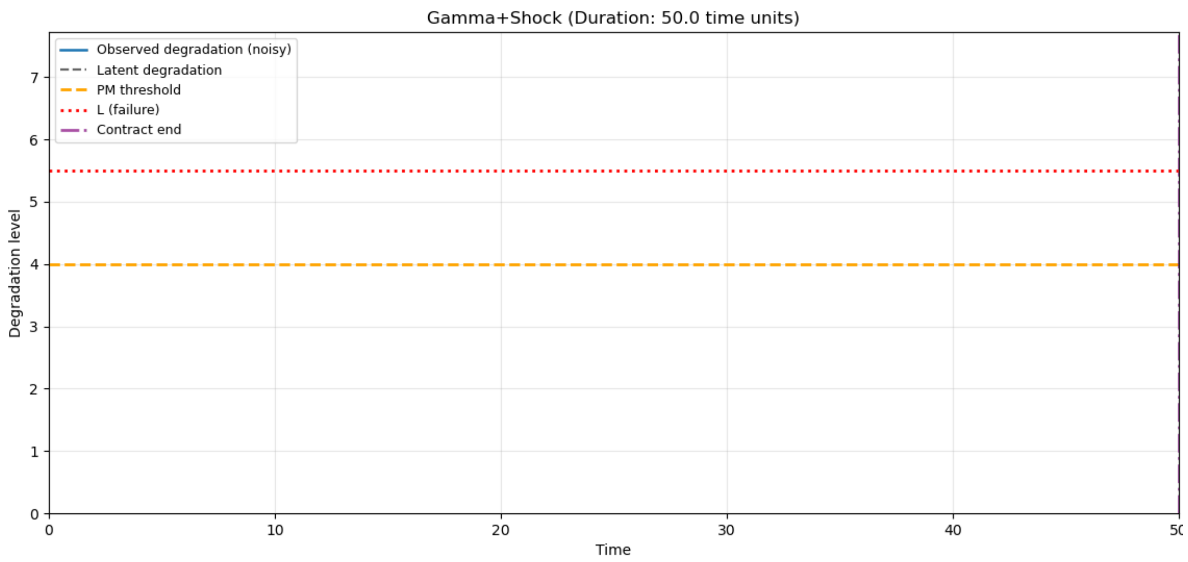
```
# Compute fleet-level cost analysis
fleet_costs = compute_fleet_costs(machine_results)

# Record events of multiple machines
event_logs = export_event_level_df(machine_results)

# Record events of multiple machines
machine_logs = export_machine_level_df(machine_results)
```

Aggregate results

Refer to ***condition_based/multi_machine_sim.py***

CBM Outputs: Degradation paths and maintenance logs





	machine_id	PM_level	PM_interval	time	type	trigger_reason	level_before_latent	level_before_observed	level_after_latent	level_af
0	0	4.0	15.0	10.4	perfect_preventive_maintenance	level_threshold_observed	4.751474	4.796827	0.000000	
1	0	4.0	15.0	15.1	imperfect_repair	scheduled_time	1.176669	1.301039	0.995994	
2	0	4.0	15.0	30.1	imperfect_repair	scheduled_time	2.613625	2.405190	1.804810	
3	0	4.0	15.0	42.5	perfect_preventive_maintenance	level_threshold_observed	4.364901	4.432757	0.000000	
4	0	4.0	15.0	45.1	imperfect_repair	scheduled_time	0.134978	0.163415	0.067489	
5	1	3.5	12.0	10.5	perfect_preventive_maintenance	level_threshold_observed	3.541137	3.563063	0.000000	
6	1	3.5	12.0	12.1	imperfect_repair	scheduled_time	0.000000	0.000000	0.000000	
7	1	3.5	12.0	24.1	imperfect_repair	scheduled_time	1.779304	1.641513	0.889652	
8	1	3.5	12.0	30.9	catastrophic_failure_replacement	N/A	5.509543	5.465794	0.000000	

Access the simulation engine repository

Scan the QR code below to access the GitHub repository for the TBM and CBM simulation engines: