Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Ole Daniel Trandheim Røn

Optimizing the Inspection Schedule of an Offshore Wind Turbine with Reinforcement Learning

TPK4950 - Sikkerhet, pålitelighet og vedlikehold

Master's thesis in Mechanical Engineering
Supervisor: Shen Yin
December 2023

NTNU
Norwegian University of
Science and Technology

Ole Daniel Trandheim Røn

Optimizing the Inspection Schedule of an Offshore Wind Turbine with Reinforcement Learning

TPK4950 - Sikkerhet, pålitelighet og vedlikehold

Master's thesis in Mechanical Engineering
Supervisor: Shen Yin
December 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

This report intends to study how reinforcement learning can be used to reduce the maintenance costs for offshore wind turbines. The explored approach is to optimize the inspection scheduling, and to see what effect this will have on the costs. Component degradation is simulated with a 4-state Weibull-Markov model and two different approaches to reliability. The degradation is approximated using Monte Carlo simulations. The work is done with Proximal Policy Optimization and programmed in Python. The performance of the suggested inspection model is measured against that of a calendar-based inspection schedule. The report finds that reinforcement learning can be used to improve on the performance of the calendar-based model by a small margin.

Keywords: Maintenance, Inspection schedule, PPO, Weibull, Markov, Monte Carlo simulation, Reinforcement Learning, Python, RAMS

Sammendrag

Denne rapporten søker å utforske hvordan forsterket læring (RL) kan anvendes til å redusere vedlikeholdskostnader for offshore vindturbiner. Den valgte tilnærmingen er å optimalisere inspeksjonsplanleggingen, og å utforske hvilken effekt dette vil ha på kostnadene. Komponentslitasje er simulert med en 4-tilstands Weibull-Markov-modell og to forskjellige tilnærminger til pålitelighet. Tilstanden til komponentene approksimeres ved hjelp av Monte Carlo-simuleringer. Arbeidet er gjennomføres med 'Proximal Policy Optimization' og er programmert i Python. Programmets prestasjon sammenlignes med en kalenderbasert inspeksjonsplanleggingsplan. Rapporten finner at forsterket læring kan anvendes til å forbedre det kalenderbaserte regimet med en liten margin.

Stikkord: Vedlikehold, Inspeksjon, PPO, Weibull, Markov, Monte Carlo simulering, Forsterket læring, Python, RAMS

Preface

This is a master's thesis in Reliability, Availability, Maintenance and Safety (RAMS) as part of a M.Sc. in Mechanical Engineering at the Department of Mechanical and Industrial Engineering – Norwegian University of Science and Technology (NTNU). The work for this report was conducted during the autumn semester of 2023. The idea for the report was thought up during the literature review. All simulation work was done on a laptop.

The report is intended for other master's students in RAMS. Some experience with programming in Python is recommended.

Trondheim, 2023-12-22

Ole Daniel Trandheim Røn

Acknowledgements

Firstly, I would like to thank Professor Shen Yin for his support throughout my work on this report. He has carved out room in his busy schedule and made himself available when needed. His insightful feedback has been a great help. This report would not be the same without his contributions.

Secondly, I would like to thank PhD student Andrie Pasca Hendradewa. Andrie has been working on a separate RL-optimization problem in parallel to my work on this report. Although he started his work earlier than I started mine, he made sure to set aside the time needed to catch me up to speed. I have learned a lot from our collaboration and am grateful for his patience and selflessness.

Thirdly, I would also like to express my gratitude to Professor Jørn Vatn and the rest of the Department of Mechanical and Industrial Engineering at NTNU for their patience and understanding during the process leading up to this report.

I would also like to thank the online reinforcement learning communities for their willingness to share their tools and experience. The tools provided by OpenAI have been instrumental to the work done in this report.

Finally, I wish to thank friends and family for their continued support.

Executive Summary

The work in this report demonstrates how reinforcement learning can be used for the maintenance planning of offshore wind turbines. The report explores the effectiveness of using reinforcement learning to optimize inspection scheduling. This is conducted in the following way:

First, the model of the system is constructed. This model is designed to describe how the system acts depending on what actions are performed. For this system, there are two actions: to do or not do an inspection. If no inspection is performed, the reliability of the system is calculated and used to give an approximation of how much power is produced on average. If an inspection is performed, the degradation is calculated for all components and used to evaluate their state. If a component is found to be degraded, a maintenance action is simulated according to the degree of degradation. The system reward is penalized according to the maintenance actions performed and estimated power production is calculated.

Second, a reinforcement learning agent is tasked to find the optimal way to solve the model. This agent performs better the longer it is allowed to simulate.

Third, a similar model is built to test different calendar-based inspection schedules. This program is designed to test a wide range of intervals to find the optimal one.

Fourth, the performances of the two models are evaluated.

Fifth, the steps above are repeated with a different approach to reliability.

Though this methodology, the reinforcement learning agent is found to be capable of outperforming the optimized calendar-based schedule by a small margin both times. Calendar-based inspection planning is quite effective, but there is still some room for improvement.

Table of Contents

List of Figures 1
List of Simplified Code 1
List of Tables 2
List of Abbreviations 2
List of Symbols 2
RAMS Theory 2
Computer Science 3
General Parameters 3
Chapter 1 - Introduction 4
1.1 Background 4
1.2 Objectives 5
1.3 Approach 6
1.4 Limitations 6
1.5 Outline 7
Chapter 2 – System Description 9
2.1 Wind Turbine Reliability 9
2.2 Wind Turbine Power Production 11
Chapter 3 – Theoretical Background 12
3.1 Reinforcement Learning (RL) 12
3.2 Deep Reinforcement Learning (DRL) 13
3.3 Proximal Policy Optimization (PPO) 13
3.4 Implementation 14
3.5 Reliability Theory 15
3.6 Maintenance Theory 16
Chapter 4 – System Model 17
4.1 Simulating Degradation 17
4.2 Observation Space and Action Space 18
4.3 The Reward Structure 19
Chapter 5 – Calculation Data 20
5.1 General Parameters 20
5.2 Weibull Parameters 20
5.3 Maintenance Costs and Durations 22
Chapter 6 – First Case Study 23
6.1 Central System 23
6.2 Control System 25

6.3 Policy Collapse 27
6.4 Results 29
Chapter 7 – Second Case Study 33
7.1 Model Changes 33
7.2 Results 36
Chapter 8 – Discussion 40
Chapter 9 – Conclusions and Future Work 41
Conclusions 41
Suggestions for Future Work 41
Bibliography 42
Appendix A i
The First Environment i
Appendix B vi
Optimizing the First Environment vi
Appendix C vii
Visualizing the First Environment vii
Appendix D xiii
Finding the First Optimal Inspection Interval xiii
Appendix E xviii
Visualizing the First Constant Inspection Interval xviii
Appendix F xxv
The Second Environment xxv
Appendix G xxx
Optimizing the Second Environment xxx
Appendix H xxxi
Visualizing the Second Environment xxxi
Appendix I xxxvii
Finding the Second Optimal Inspection Interval xxxvii
Appendix J xlii
Visualizing the Second Optimal Inspection Interval xlii

List of Figures

Figure 2.1 Example of a wind turbine and its nacelle layout showing some of the terminology
Figure 2.2 Fault tree analysis of wind turbine system failure
Figure 2.3 Power production of a 5 MW wind turbine
Figure 3.1 "The Bathtub curve"
Figure 4.1 Descriptive illustration of the Markov states
Figure 4.2 Illustration of the Markov state transition model
Figure 4.3 Example of a state probability distribution
Figure 6.1 Finding Optimal Inspection Interval, Low Precision
Figure 6.2 Finding Optimal Inspection Interval, Higher Precision
Figure 6.3 Policy collapse
Figure 6.4 New learning rate graph
Figure 6.5 Central system: Reward per simulation
Figure 6.6 Control system: Reward per simulation
Figure 6.7 Central system: System reliability
Figure 6.8 Control system: System reliability
Figure 6.9 Central system: State probability distribution
Figure 6.10 Control system: State probability distribution
Figure 7.1 Illustration of proposed component reliability and failure rate function
Figure 7.2 Descriptive illustration of terms
Figure 7.3 Finding optimal inspection interval, low precision
Figure 7.4 Finding optimal inspection interval, higher precision
Figure 7.5 Central system: Reward per simulation
Figure 7.6 Control system: Reward per simulation
Figure 7.7 Central system: System reliability
Figure 7.8 Control system: System reliability
Figure 7.9 Central system: State probability distribution
Figure 7.10 Control system: State probability distribution

List of Simplified Code

Code 6.1 Finding state probability distribution with Monte Carlo simulations
Code 6.2 Logic to find the optimal inspection interval

List of Tables

Table 5.1 Calculation values
Table 5.2 Weibull parameters
Table 5.3 Cost and duration of maintenance actions
Table 6.1 Maintenance data for wind turbine blades
Table 6.2 Weibull parameters for wind turbine blades
Table 6.3 Performance measurements of the system
Table 7.1 Performance measurements of the systems

List of Abbreviations

RAMS	Reliability, Availability, Maintenance and Safety
O&M	Operation and Maintenance
AI	Artificial Intelligence
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
PPO	Proximal Policy Optimization
G20	The Group of Twenty. Forum for international economics cooperation.
GW	Giga Watt
PM	Preventive Maintenance
CM	Corrective Maintenance

List of Symbols

RAMS Theory

i	Component index
s	State (Markov state)
$R_i(t)$	Component reliability
$R_{sys}(t)$	System reliability
α, β	Weibull shape parameter. $\alpha = \beta$
λ, η	Weibull scale parameter. $\lambda = \frac{1}{\eta}$
$z(t)$	Failure rate function
$\Pr_{i,s=X}(t)$	Probability of component i being in Markov state X at time t

Computer Science

s	State (observation)
S	State space (observation space)
a	Action
A	Action space
$R_w(t)$	Reward in week t
G_t	Total cumulative rewards including week t
γ	Discount factor
ratio_t	Probability ratio between the old and new policy
$clip(\cdot)$	Clipping function for the gradient
ε_{clip}	Clipping parameter
$\hat{\text{A}}_t$	Estimation of the average function at timestep t
$H(\cdot)$	Entropy
β	Hyperparameter to control the strength of entropy
V_ω	Critic network that estimates the state value

General Parameters

$I(t)$	Income in week t
$C_M(t)$	Maintenance costs in week t
$T(t)$	System operational time in week t
P	Maximal power production capacity
C	Average expected capacity factor
S	Average expected selling price of power
C_I	Cost of inspection
$C_{PM,i}(t)$	Cost of preventive maintenance actions for component i in week t
$C_{CM,i}(t)$	Cost of corrective maintenance actions for component i in week t
T_i	Duration of inspection
$T_{PM,i}(t)$	Duration of preventive maintenance actions for component i in week t
$T_{CM,i}(t)$	Duration of corrective maintenance actions for component i in week t

Chapter 1 - Introduction

1.1 Background

Offshore wind technology is a rapidly growing segment of power production. From 2010 to 2022 the cumulative global capacity of deployed offshore wind turbines increased from 3.1 GW to 63.2 GW, more than a twenty-fold increase. Draw factors for the industry include a vastness of available real estate and a more stable wind distribution than on land, meaning that they operate with a higher capacity factor. In Europe, the technology has an added benefit of peaking its power production during winter, coinciding with an increased demand in power [14].

Offshore wind turbines have a higher cost of construction, operation, and maintenance than onshore wind turbines. To compensate, offshore wind turbines are generally constructed at a larger scale, which significantly increases their power output. A doubling in the blade span increases potential power production by a factor of four.

Datapoints from the G20 countries report that operation and maintenance (O&M) costs typically account for 16% to 25% of the levelized cost of electricity of offshore wind turbines [14]. For comparison, onshore wind turbine O&M costs are reported to make up around 5% of its levelized electricity costs [22]. The differential is caused by a higher cost of accessing the site to perform maintenance. O&M of an offshore wind turbine requires utilizing specialized vessels and skilled crew. The harsh climate at sea can also cause significant lead times at all stages of the O&M process. There are some uncertainties connected to the expected lifetime O&M costs of an offshore wind turbine. This is caused by a lack of operational experience and a lack of available O&M cost data for the industry [14]. The high O&M costs make the industry an excellent candidate for optimization studies.

Reinforcement Learning (RL) has in the last decades emerged as promising tool for optimization. This is largely caused by a rapid increase of available processing power and data. RL is a type of machine learning that involves training an agent to act in an environment to maximize the reward signal. In the context of industrial systems, RL can be used to develop maintenance strategies based on real-time system data rather than predetermined schedules.

Problem Formulation

Offshore wind turbines have high operation and maintenance costs. Utilizing reinforcement learning to optimize the inspection schedule could have significant impact on the cost-effectiveness of the industry. Demonstrate how RL can be used to develop a predictive maintenance strategy for offshore wind turbines.

Related Work

[18] Deep reinforcement learning for cost-optimal condition-based maintenance policy of offshore wind turbine components (2023) by Cheng, J., Liu, Y., Li, W., & Li, T.:

> This paper explores different deep reinforcement learning frameworks to derive the cost-optimal condition-based maintenance policy for offshore wind turbines. It

explores different inspection intervals and repair thresholds and compares the cost-efficiency of the approaches. The scope is limited to the blade of a single wind turbine. It considers variations in wind conditions and some logistical factors. The maintenance actions considered in the report are inspection and replacement. There are three states considered: Normal working state, degraded and failed. The explored RL-models are PPO and DQN.

This paper is the main inspiration for the work done in this report.

[2] Joint optimization of preventive and condition-based maintenance for offshore wind farms (2022) by Toftaker, H., Bødal, E. F., & Sperstad, I. B.:

This paper utilizes a constrained integer linear program, maximizing for income, to explore the trade-off between maximizing power production and limiting the degradation of the turbines.

The reward structure utilized in this paper was a major inspiration during the design of the reward structure for in this report.

[3] Modelling wind turbine degradation and maintenance (2016) by Le, B., & Andrews, J.:

This paper presents an asset model for offshore wind turbine reliability. The model accounts for degradation, inspection, and maintenance actions. The purposes of the model are to predict the future condition of wind turbine components and to investigate the effect of a specified maintenance strategy.

This paper is this report's main source of statistical degradation data.

[31] *Proximal Policy Optimization Algorithms* v2 (2017) by Schulman, J., Wolski, J., Dhariwal, P., Radford, A., & Klimov, O.:

This paper is the basis for the reinforcement learning technique utilized in this report.

What Remains to Be Done?

Utilizing RL for maintenance scheduling is an emerging field of study that shows great promise. Its uses and limitations are still being explored. The complexity of industrial systems can pose significant challenges to applying RL to system maintenance. Industrial systems often have numerous interacting components and subsystems with complex dependencies and feedback loops. This complexity can make it challenging to develop RL algorithms that effectively model the system's behavior. Most articles on the subject let the RL algorithm decide the maintenance action outright, but how would the program perform if its actions were limited to inspection planning?

1.2 Objectives

The main objectives of this master thesis are to:

1 Develop a RL-based predictive maintenance model for an offshore wind turbine.
2 Compare the effectiveness of the RL-based model to that of a calendar-based inspection model.

1.3 Approach

This is the chosen approach to meet the objectives:

1 Familiarize myself with RL by solving an easy, preexisting environment (cartpole).
2 Create and solve a simple custom environment based on a Markov model.
3 Do a literature review to establish a better understanding of the system.
4 Do a literature review to establish a better understanding of RL.
5 Establish the data sets that will be used for the digital representation of the system.
6 Create a digital representation of the system in a RL model.
7 Create a digital representation of the calendar-based inspection model.
8 Make observations about the systems' performances.
9 Compare the results.

1.4 Limitations

Experience

I am a master student in mechanical engineering specializing in the field of RAMS. I have limited prior knowledge of Artificial Intelligence and no prior experience with utilizing the technology for optimization. Additionally, my experience with coding in Python is purely from recreational use. As a result of these factors, the programming in this report will not be optimized.

Field of Study

Writing a multi-disciplinary thesis means that the department is unlikely to specialize in both fields of study. Most of the computer science theory utilized in this report will be self-thought over the duration of the work.

Timeframe

This paper was produced over a timeframe of 16 weeks.

Data accuracy

The data used in this report is based on approximations and available literature. There has not been conducted any collaboration with the industry or field research.

Tools

All simulations in this report were done on an HP EliteBook 830 G6 laptop. It runs on the Intel i5-8265U @ 1.60GHz and has 8GB RAM. These are the program specs:

Visual Studio Code v1.84.2

Python v3.11.5

Gymnasium v0.29.1

Stable Baselines3 v2.1.0

1.5 Outline

Here is an overview of how the report is organized:

Abstract: Practical information about what has been done and the assumed background of the reader.

Acknowledgements: Gratitude for received support.

Executive summary: Summary of the work done in this report.

List of figures

List of simplified code

List of tables

List of abbreviations

List of symbols: List of symbols used in calculations.

Chapter 1. Introduction: Background, objectives, approach, limitations, and outline of the report.

Chapter 2. System description: System descriptions and assumptions relevant to modelling the system for maintenance optimization.

Chapter 3. Theoretical background: State of the art in RL and RAMS theory.

Chapter 4. System model: The model is explained. Observation space, action space and reward structure are described here.

Chapter 5. Calculation data: Chosen parameters to describe the system.

Chapter 6. First case study: Case study on the effectiveness of utilizing RL for inspection planning.

Chapter 7. Second case study: Second case study in the effectiveness of utilizing RL for inspection planning.

Chapter 8. Discussion: Results from the case studies are discussed.

Chapter 9. Conclusion and future work

Bibliography

Appendixes: The appendixes contain the code of the report. They are formatted in the following way:

- Appendix A contains the first case study environment. This is where the step function, the reward structure, the observation space, and the action space are defined.

- Appendix B contains the optimization code. This is where the PPO learns, and results are saved.
- Appendix C contains the code to visualize the results.
- Appendix D contains the code to find the optimal calendar-based inspection interval.
- Appendix E contains the code to visualize the calendar-based inspection interval.
- Appendix F, G, H, I and J are structured in the same way, but for the second case study.

Chapter 2 – System Description

2.1 Wind Turbine Reliability

A wind turbine is a structurally complex system comprised of thousands of components [25]. Each component has its own life expectancy, failure conditions and suitable maintenance actions. Notable maintenance actions are routine inspection, cleaning, lubrication, repair, and replacement [15]. For a meaningful representation of the system, it is important to define the scope. For this report, the scope will be limited to the major components. The maintenance actions are limited to inspection, repair, and replacement.

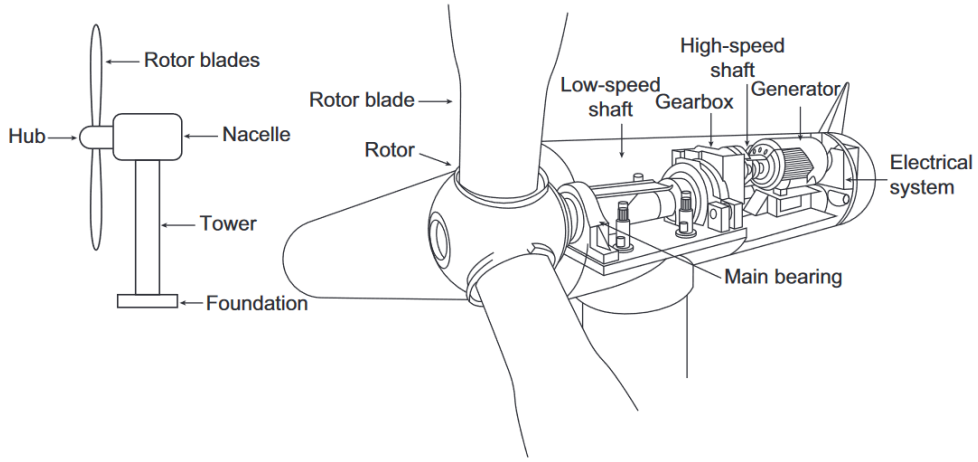


Figure 2.1: Example of a wind turbine and its nacelle layout showing some of the terminology [28]

In this report, no redundancy is assumed in the wind turbine system (see Figure 2.2 for the fault tree analysis). The components' reliabilities are also taken to be completely independent of each other. Based on these assumptions, the system reliability, $R_{sys}(t)$, is calculated by taking the product of the component reliabilities, $R_i(t)$:

$$R_{sys}(t) = \prod R_i(t) \quad \textbf{(2.1)}$$

There is not conducted a hazard analysis for this report. The risk of critical failure is considered beyond the scope of the task. As such, no additional penalty is applied for component failures during simulations.

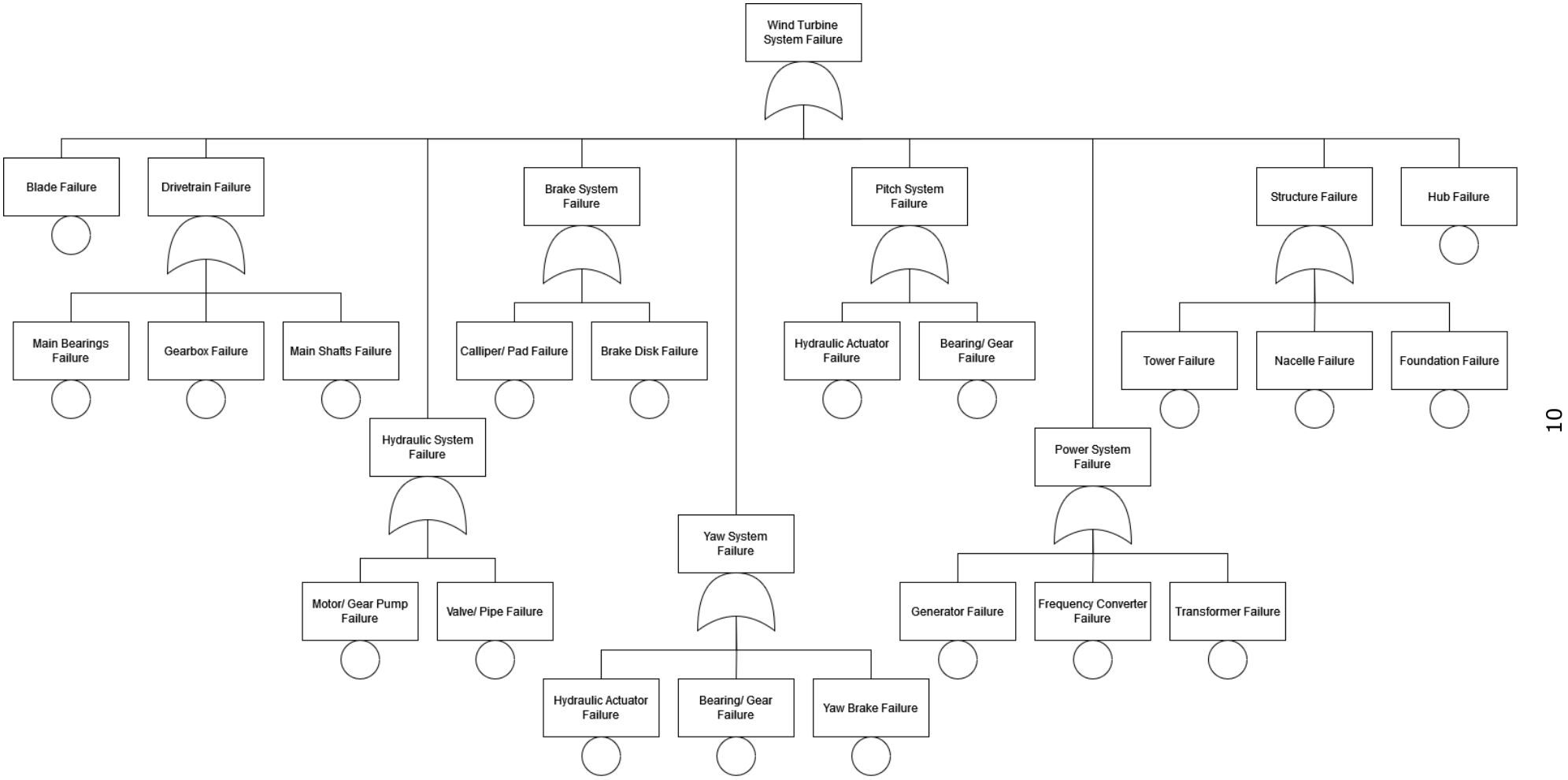


Figure 2.2: Fault tree analysis of wind turbine system failure

2.2 Wind Turbine Power Production

Wind conditions heavily influence the power production of a wind turbine (see Figure 2.3). The correlation between a wind turbine's maximum capacity and its average power output is called the capacity factor. Offshore wind turbine projects commissioned in Europe in 2022 reported a weighted average capacity factor of 49% [14]. This is the capacity factor that will be utilized in this report, though it could also be argued for using a capacity factor of 34.2%, the global average for existing offshore wind turbines [16]. The capacity factor will influence reward for keeping the system running.

A weakness of utilizing a capacity factor to estimate power production is that the model will be blind to seasonal changes in production.

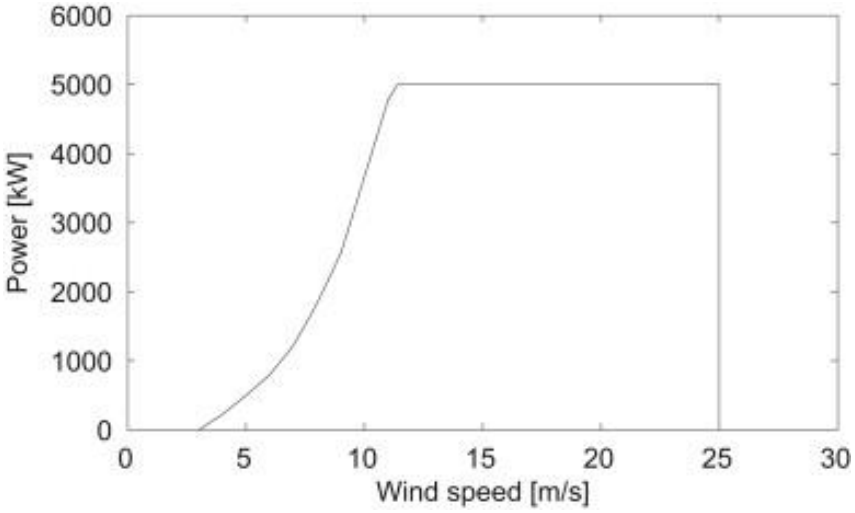


Figure 2.3: Power production of a 5 MW wind turbine [26]

Chapter 3 – Theoretical Background

3.1 Reinforcement Learning (RL)

The growth in processing power has made Artificial Intelligence (AI) technology an increasingly relevant field of science. Machine learning is a type of AI where the agent makes predictions and decides what actions to take based on past and present observations [35]. For machine learning to be effective, the future must resemble the past. With sufficient historical data, machine learning can be used to approximate component degradation, making maintenance planning an excellent candidate for machine learning optimization.

Reinforcement Learning (RL) is a type of machine learning where the algorithm has the ability to learn without explicitly being told how. The RL-agent is defined by the observation space, the action space, and the reward structure. The observation space aims to describe the current state of the system. It contains all the information required to make a prediction about the future state of the system. The action space contains all the actions the agent is allowed to take at any timestep. The reward structure assigns rewards or penalties based on the performance. The goal of the RL agent is to score as high of a reward as possible. The RL agent starts by performing random actions, and slowly develops an understanding of the optimal action throughout the observation space.

To better illustrate this structure, here is the cartpole problem [36]:

A pole is attached by an un-actuated joint to a cart. The cart moves along a frictionless track. The pendulum is placed upright on the cart. The goal is to balance the pole by applying forces to the left and right direction on the cart.

Action space:
(0) push cart left
(1) push cart right

Observation space:
(0) cart position
(1) cart velocity
(2) pole angle
(3) pole angular velocity

Reward structure:
+1 for every timestep

The simulation is terminated if the pole falls below a defined angle or if the maximal timestep is reached.

The outcome of each action is calculated in the step function and conveyed back to the RL algorithm.

3.2 Deep Reinforcement Learning (DRL)

RL algorithms learn dynamically with a trial-and-error method to maximize the reward. Mapping the optimal reward path for all states can overwhelm the algorithm for complex systems. This is where DRL comes in. DRL algorithms learn from existing knowledge and applies it to the new data set. The "deep" portion refers to the application of a neural network. This network estimates the state-action-reward correlation instead of mapping every solution [40].

The cartpole problem is an example of how this is useful. The state space for this problem is a 4-dimensional continuous vector. Mapping the optimal reward path for all states in a continuous observation space is infeasible. Traditional RL algorithms would therefore require the observation space to be discretized. This is a viable option, but the amount of states would still be extremely high.

3.3 Proximal Policy Optimization (PPO)

PPO is the DRL algorithm that is used for the optimization work in this report. PPO is the default RL algorithm at OpenAI [38] and is often referred to as the state of the art in RL. Compared with other DRL algorithms, the three main advantages of PPO are its simplicity, stability, and sample efficiency [37]. PPO has shown great performance in deriving the maintenance schedule of wind turbine systems [18].

The PPO agent starts by using a trial-and-error method to map performance based on actions, observations, and rewards. This mapping is used to establish a policy network. The policy network is used to create a probability distribution of the action space based on their expected performance from the current observed state. An action's assigned probability is correlated to its expected performance. The next action is picked with a random sampling. The policy network gets continuously updated with observed results.

A more detailed explanation of a PPO algorithm [18], [31], [37]:

The policy is expressed as a neural network $\pi_\theta(a|s)$ with the parameters θ:

$$\pi_\theta(a|s) = \Pr[a|s] \quad \forall \qquad a \in A, s \in S \tag{3.1}$$

S is the state space of the model. A is the action space of the model. s is a state. a is an action.

The policy attempts to maximize the total reward, G_t, over a chosen timeframe. G_t is expressed by:

$$G_t = R_w(t) + \gamma R_w(t+1) + \gamma^2 R_w(t+2) + \cdots = R_w(t) + G_{t+1} \tag{3.2}$$

$R_w(t)$ is the reward per timestep and γ is the discount factor.

The agent takes an action, a_t, sampled from the policy $\pi_\theta(a|s_t)$ in state s_t. The reward, $R_w(t)$, and the next state, s_{t+1}, are noted. At the end of the episode, the total reward, $G(t)$, is calculated and the transition process $\{s_t, a_t, R_w(t), G_t\}$ is stored in the replay memory. After enough samples are stored, the parameters θ and ω are updated. θ is updated by minimizing the following loss function:

$$L^{CLIP} = \mathbb{E}\left[\min\left(\text{ratio}_t(\theta), clip\left(ratio_t(\theta), 1-\varepsilon_{clip}, 1+\varepsilon_{clip}\right)\right)\hat{\text{A}}_t + \beta H(\pi_\theta)\right] \quad \textbf{(3.3)}$$

$clip(\cdot)$ is the clipping function for the gradient. It is there to prevent rapid change of parameters. ε_{clip} is the clipping parameter. $H(\cdot)$ is the entropy. β is the hyperparameter to control the strength of entropy. ratio_t is the probability ratio between the new, $\pi_\theta(a|s_t)$, and the old policy, $\pi_{\theta_{old}}(a|s_t)$:

$$\text{ratio}_t(\theta) = \frac{\pi_\theta(a|s_t)}{\pi_{\theta_{old}}(a|s_t)} \quad \textbf{(3.4)}$$

$\hat{\text{A}}_t$ is the estimation of the average function at timestep t:

$$\hat{\text{A}}_t = G_t - V_\omega(s_t) \quad \textbf{(3.5)}$$

V_ω is the critic network that estimates the state value.

The parameters ω of the critic network are found by minimizing:

$$L^V(\omega) = \mathbb{E}\left[\left(\text{G}_t - V_\omega(s_t)\right)^2\right] \quad \textbf{(3.6)}$$

The samples in the replay memory are be used to update the parameters numerous times before they are deleted, and new samples are collected.

3.4 Implementation

A challenge with PPO is that it is vulnerable to early convergence. This means that it finds a local optimum and stops exploring before it reaches the global optimum. This can be combatted by incentivizing exploration. However, if the incentive for exploration is too high, the model's ability to learn will be reduced. The model may even unlearn previous progress. Different hyperparameter initializations and search parameters can have a significant impact on performance [30].

The optimization in this report is done using a simple, vectorized wrapper for multiple environments, calling each environment in sequence. The program essentially runs multiple iterations of the same environment. This environment wrapping technique is sometimes referred to as "DummyVecEnv." The technique reduces the time spent in the network interface by parallelizing over multiple inputs when doing rollouts. Running decorrelated samples also helps with training. "DummyVecEnv" is optimal for computationally simple environments, where the overhead outweighs the computation time [33], [34].

3.5 Reliability Theory

Mechanical component failures can generally be segmented into three categories (Figure 3.1):

- **Early failures:** Failures caused by errors in installation or production.
- **Intrinsic failures:** Failures caused by short-term overload and sudden breakdowns.
- **Deterioration:** Failures caused by long-term degradation and aging.

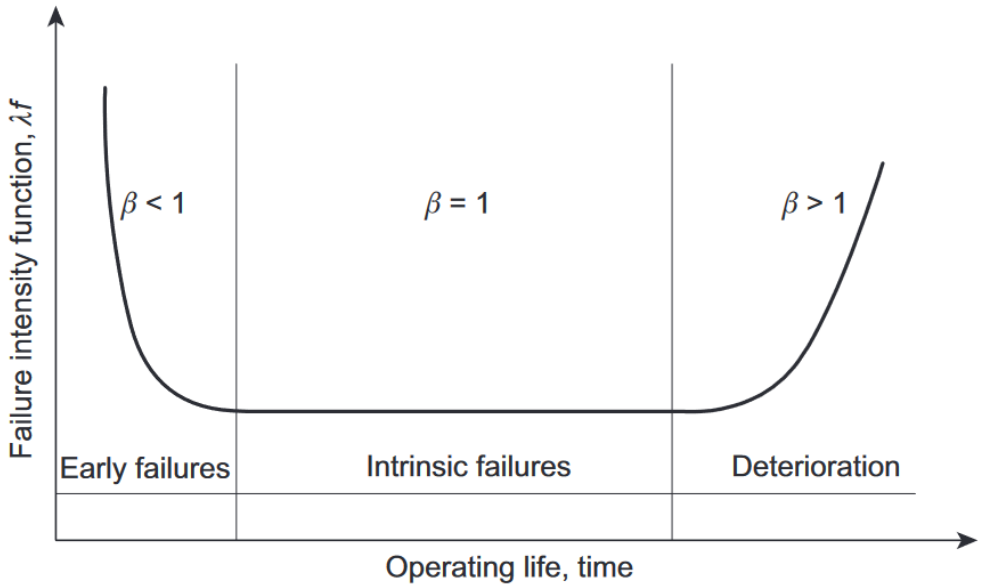


Figure 3.1: "The Bathtub curve" [27] – The shape parameter is denoted by β.

Two common ways of expressing system reliability of a mechanical system are the exponential distribution function and the Weibull distribution function. The exponential distribution function assumes a constant failure rate. The failures are considered to be evenly distributed throughout the recorded timeframe. In other words, this approach takes most failures to be intrinsic, and early failures and deterioration failures are assumed to be negligible. A strength of the exponential distribution function is that it makes expressing reliability trivial for any data set. The simplicity of the expression also simplifies calculations.

The Weibull distribution function expands upon the exponential function by introducing the shape parameter. The purpose of the shape parameter is to simulate changes in the failure function. This way degradation can be simulated. Most failures in mechanical systems are gradual processes, rather than sudden occurrences [9]. As such, deterioration should be assumed to be a notable cause of component failures. For this reason, the Weibull distribution function is chosen to simulate the system degradation. Premature (early) failures are not considered.

3.6 Maintenance Theory

The two main types of maintenance actions are preventive maintenance and corrective maintenance.

Corrective maintenance (CM) refers to maintenance actions performed after critical degradation or failure. A corrective maintenance strategy, run-to-failure maintenance, can quickly become costly due to production interruptions caused by extended equipment downtime [11]. In this report, corrective maintenance is used interchangeably with replacement.

Preventive maintenance (PM) is planned maintenance performed when an item is functioning correctly to prevent future failures. Preventative maintenance aims to prolong the lifetime of a system component. Preventive maintenance plans can be classified into the following categories [20]:

- **Age-based maintenance:** Tasks are performed at a specified component age.
- **Clock-based maintenance:** Tasks are performed at fixed calendar times.
- **Condition-based maintenance:** Tasks are performed based on measured condition variables.

The downside of a preventive maintenance plan is that it generally performs maintenance before it is necessary, reducing the lifespan of some components and resulting in additional costs [11]. Repairs or overhauls generally cannot return the system to its initial state. Practically, the lifetimes of repaired components show a significant amount of uncertainty [9]. In this report preventive maintenance is used interchangeably with repair.

The maintenance plan utilized in this report is predictive, condition-based maintenance. The program schedules inspections based on component ages. Its performance is compared to that of a clock-based inspection cycle.

Chapter 4 – System Model

4.1 Simulating Degradation

Degradations of the system components are simulated with a 4-state Markov model. Figure 4.1 illustrates the attributes of the states.

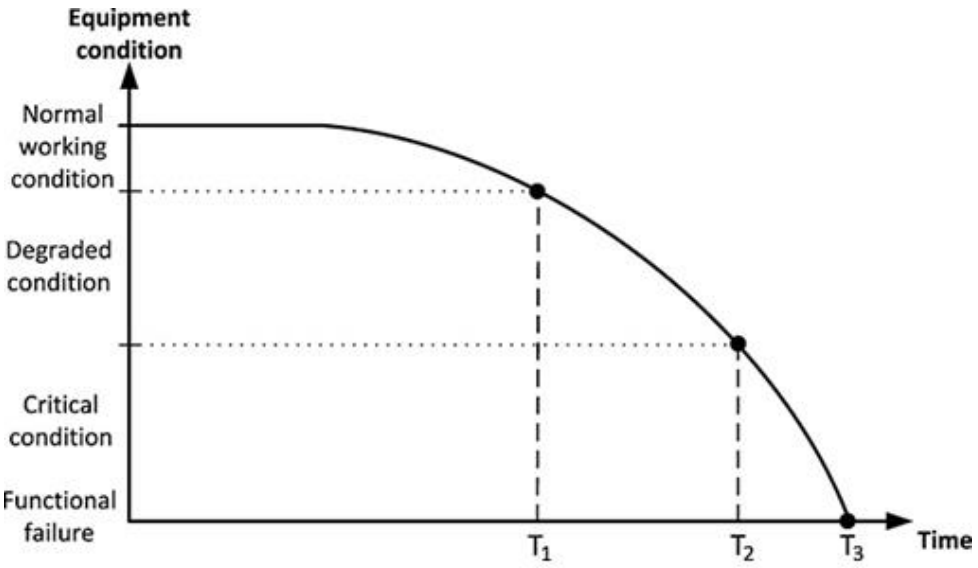


Figure 4.1: Descriptive illustration of the Markov states [3]

The program will calculate the probability distribution between the four states at each timestep. This is done by using the failure rate function for the system component, $z_i(t)$.

$$z(t) = \alpha\lambda(\lambda t)^{\alpha-1} \quad [20] \quad \textbf{(4.1)}$$

$$z_{i,s}(t) = \alpha_{i,s}\lambda_{i,s}(\lambda_{i,s}t_{i,s})^{\alpha_{i,s}-1} \quad \textbf{(4.2)}$$

λ is the scale parameter and α is the shape parameter of the Weibull distribution function. t_s denotes time spent in state s. i is the system component index.

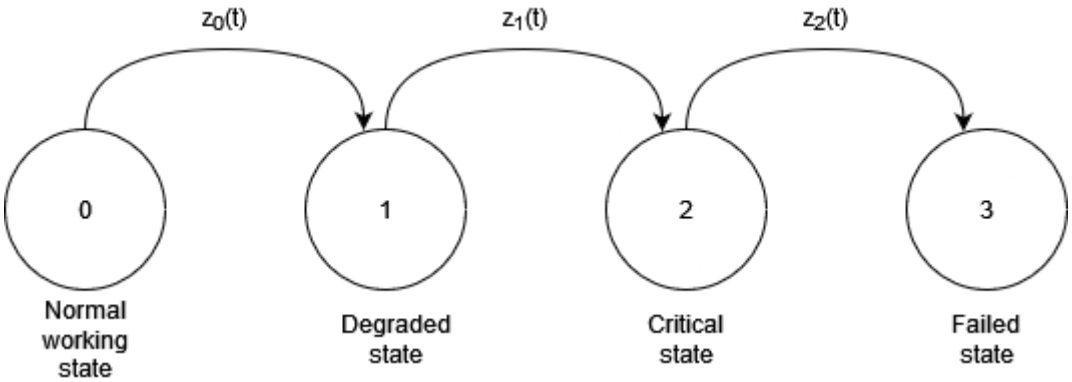


Figure 4.2: Illustration of the Markov state transition model

The reliability of a system component, $R_i(t)$, is the probability of not being in the failed state, state 3.

$$R_i(t) = 1 - \Pr_{i,s=3}(t) \quad \textbf{(4.3)}$$

4.2 Observation Space and Action Space

The observation space of the system keeps track of the age of each system component.

Observation space:
(0) Time since maintenance action for component 0.
(1) Time since maintenance action for component 1.
…

(i) Time since maintenance action for component i.

Action space:
(0) No action
(1) Inspection

If an inspection is performed, the program calculates the state probability distribution for each component. This is used to simulate the component state. The program uses a random number between 0 and 1 to simulate the expected state of the system component. Using Figure 4.3 as an example, rolling 0.00 - 0.40 would simulate a normal working state, 0.40 - 0.65 a degraded state, 0.65 - 0.85 a critical state and 0.85 - 1.00 a failed state.

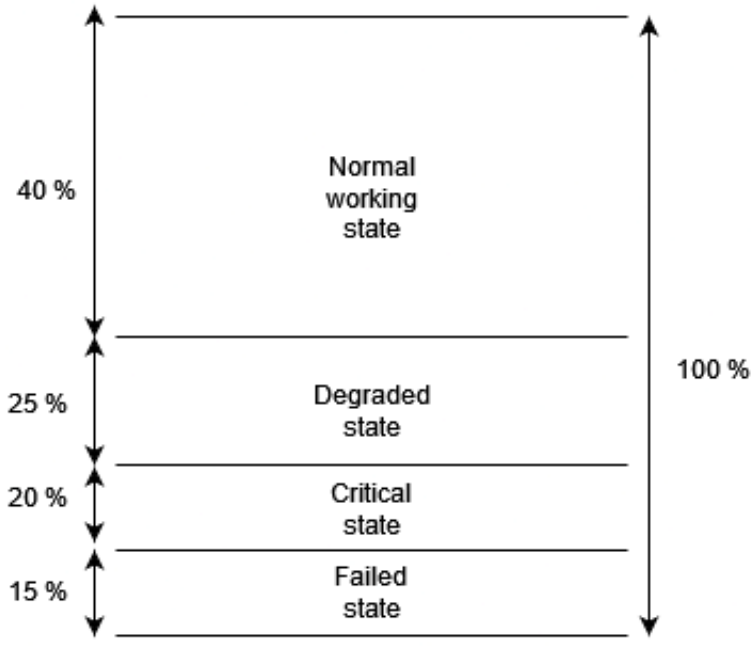


Figure 4.3: Example of a state probability distribution

For components in the normal working state, no futher maintenance actions are performed. For components in a degraded state, preventive maintenance is performed. For components in a critical state or failed state, corrective maintenance is performed.

Preventive and corrective maintenance are both assumed to return the component to "good as new," resulting in a component age of 0 weeks.

4.3 The Reward Structure

The reward structure for the program is based on expected income and expenditures. The reward is divided by 10^6 to prevent it from going out of the bounds of the PPO policy.

$$R_w(t) = I(t) - C_M(t) \quad \textbf{(4.4)}$$

Here, $R_w(t)$ represents expected income in week t and $C_M(t)$ represents maintenance costs in week t.

Expected income, $I(t)$, at any timestep is given by:

$$I(t) = R_{sys}(t) * T(t) * P * C * S \quad \textbf{(4.5)}$$

Here, $R_{sys}(t)$ is the system reliability on week t. $T(t)$ is the production hours of the system during week t. P is the maximum power capacity of the wind turbine. C is the average capacity factor for the turbine. S is the average selling price of power.

If inspection is not performed:

$$C_M(t) = 0 \quad \textbf{(4.6)}$$

$$T(t) = 24h * 7 \quad \textbf{(4.7)}$$

If inspection is performed:

$$C_M(t) = C_I + \sum C_{PM}(t) + \sum C_{CM}(t) \quad \textbf{(4.8)}$$

$$T(t) = 24h * 7 - T_I - \sum T_{PM}(t) - \sum T_{CM}(t) \quad \textbf{(4.9)}$$

C_I is the cost of inspection. $\sum C_{PM}(t)$ is the cost of all preventive maintenance actions performed in week t. $\sum C_{CM}(t)$ is the cost of all corrective maintenance actions performed in week t. T_I is the system downtime required to do an inspection, this is assumed to be 1 hour. $\sum T_{PM}(t)$ is the duration of all preventive maintenance actions performed in week t. $\sum T_{CM}(t)$ is the duration of all corrective maintenance actions performed in week t.

Chapter 5 – Calculation Data

5.1 General Parameters

The data used to simulate the system is based on approximations and available literature. The simulations will be based on a 5 MW capacity wind turbine. The wind turbine is expected to operate with an average capacity factor of 49%. The selling price of power is simplified to be static at 0,04 € / kWh.

Description	Value	Source
Power capacity (P)	5 MW	[29]
Capacity factor (C)	49 %	[14]
Selling price of power (S)	0,04 € / kWh	[18], [26]
Inspection cost (C_I)	2500 €	[18], [26]
Conversion rate	1,14 €/£	

Table 5.1: Calculation values

5.2 Weibull Parameters

Table 5.2 contains estimated Weibull parameters. They describe the behavior of the simulated failure rate function as the component degrades to the specified state. The numbers are based on estimates and inspection data for onshore wind turbines, as there currently is limited data available for offshore wind turbines [14].

Subsystem	Component	Degraded condition (years)	Critical condition (years)	Functional failure (years)	Source
Drivetrain	Main bearings	β = 1.2, η = 160	β = 1.5, η = 20	β = 1.5, η = 20	[3]
	Gearbox	β = 1.3, η = 16	β = 1.2, η = 2	β = 1.4, η = 2	[3]
	Main shafts	β = 1.2, η = 160	β = 1.5, η = 20	β = 1.5, η = 20	[3]
Hydraulic system	Motor/gear pump	-	-	β = 1.2, η = 20	[3]
	Valves/pipes	β = 1.2, η = 13.11	-	β = 1.2, η = 3.28	[3]
Brake system	Callipers/pads	β = 1.2, η = 13.11	-	β = 1.2, η = 3.28	[3]
	Brake discs	β = 1.2, η = 42.28	-	β = 1.2, η = 10.57	[3]
Yaw system	Hydraulic actuator	β = 1.2, η = 42.28	-	β = 1.2, η = 10.57	[3]
	Bearing/gear	β = 1.2, η = 29.12	β = 1.2, η = 3.64	β = 1.2, η = 3.64	[3]
	Yaw brake	β = 1.2, η = 29.12	β = 1.2, η = 3.64	β = 1.2, η = 3.64	[3]
Pitch system	Hydraulic actuator	β = 1.2, η = 29.12	-	β = 1.2, η = 7.28	[3]
	Bearing/gear	β = 1.2, η = 15.38	β = 1.2, η = 1.92	β = 1.2, η = 1.92	[3]
Hub	Hub	β = 1.2, η = 15.38	β = 1.2, η = 1.92	β = 1.2, η = 1.92	[3]
Blades	Blades	β = 1.2, η = 23.02	β = 1.2, η = 2.88	β = 1.2, η = 2.88	[3]
Power system	Generator	β = 1.2, η = 15.38	β = 1.2, η = 1.92	β = 1.2, η = 1.92	[3]
	Frequency converter	β = 1.2, η = 33.38	-	β = 1.2, η = 8.35	[3]
	Transformer	-	-	β = 1.2, η = 14.93	[3]
Structure	Tower	-	-	β = 1.2, η = 14.93	[3]
	Nacelle	β = 1.2, η = 133.33	β = 1.2, η = 16.67	β = 1.2, η = 16.67	[3]
	Foundation	β = 1.2, η = 133.33	β = 1.2, η = 16.67	β = 1.2, η = 16.67	[3]

Table 5.2: Weibull parameters

In literature, many Greek letters are used in the literature to describe the parameters of the Weibull distribution function. For clarity:

$\lambda = \frac{1}{\eta}$ (scale parameter)

$\alpha = \beta$ (shape parameter)

5.3 Maintenance Costs and Durations

The maintenance options are limited to one preventive and one corrective maintenance action per component. This is done to streamline the modelled maintenance plan. It is important to be cognizant of simplifications. An oversimplified system can give optimized solutions that are misleading or wrong. Establishing the right calculation parameters is critical to the effectiveness of the program. On a real-life implementation, the work to establish these parameters should be thorough. As this report is purely for research purposes, the data sets in this chapter serves its use, and are considered good enough.

Subsystem	Component	PM costs	PM duration	CM costs	CM duration	Source
Drivetrain	Main bearings	£5,000	3 h	£20,000	70 h	[3]
	Gearbox	£50,000	10 h	£260,000	50 h	[3]
	Main shafts	£5,000	3 h	£37,000	70 h	[3]
Hydraulic system	Motor/gear pump	-	-	£26,000	10 h	[3]
	Valves/pipes	-	-	£1,000	3 h	[3]
Brake system	Callipers/pads	-	-	£4,000	10 h	[3]
	Brake discs	-	-	£4,000	10 h	[3]
Yaw system	Hydraulic actuator	£7,000	3 h	£20,000	10 h	[3]
	Bearing/gear	£7,000	10 h	£9,000	70 h	[3]
	Yaw brake	-	-	£9,000	10 h	[3]
Pitch system	Hydraulic actuator	£8,000	3 h	£23,000	10 h	[3]
	Bearing/gear	£8,000	3 h	£23,000	10 h	[3]
Hub	Hub	£3,000	3 h	£44,000	70 h	[3]
Blades	Blades	£4,000	3 h	£200,000	70 h	[3]
Power system	Generator	£50,000	10 h	£150,000	50 h	[3]
	Frequency converter	-	-	£12,000	50 h	[3]
	Transformer	-	-	£30,000	10 h	[3]
Structure	Tower	£20,000	3 h	£264,000	70 h	[3]
	Nacelle	£5,000	3 h	£40,000	70 h	[3]
	Foundation	£15,000	3 h	£204,000	70 h	[3]

Table 5.3: Cost and duration of maintenance actions

Chapter 6 – First Case Study

6.1 Central System

See Appendix A, Appendix B and Appendix C for the code.

For the first case study, a program is designed to model the deterioration and maintenance of a single-component system consisting of solely of the blade of a wind turbine. This system is a single-component 4-state Weibull-Markov model.

The state probability distribution is calculated from the time of the last maintenance action (installation, repair, or replacement). In other words, the reliability of a system component is reset on maintenance. On inspection and maintenance, the program estimates production downtime and reduces the reward accordingly. The system is then set to state 0. This means that the simulated timeframe exclusively represents operative time. A change in degradation after repair is also not simulated.

PM cost	PM duration	CM cost	CM duration
£4,000 * 1,14 €/£	3 h	£200,000 * 1,14 €/£	70 h

Table 6.1: Maintenance data for wind turbine blades, from Table 5.1 and 5.3.

Notice the difference in the cost of preventive and corrective maintenance (see Table 6.1). Based on this observation, the PPO agent should be expected to be cautious about degrading the component to the critical state, as it would be forced perform corrective maintenance. The scale of the cost difference may be a result of an oversimplification of the data.

Degraded condition (weeks)	Critical condition (weeks)	Functional failure (weeks)
$\alpha_0 = 1.2$	$\alpha_1 = 1.2$	$\alpha_2 = 1.2$
$\lambda_0 = \frac{1}{23.02 * 52}$	$\lambda_1 = \frac{1}{2.88 * 52}$	$\lambda_2 = \frac{1}{2.88 * 52}$

Table 6.2: Weibull parameters for wind turbine blades, from Table 5.2.

The transition probabilities of the Markov chain are used to calculate the state probability distribution at each timestep. The complexity of the chosen model makes it impossible to calculate the state probability directly. For this reason, a Monte Carlo simulation is used to find an approximation of the state probability distribution.

$$z_{i,s}(t) = \alpha_{i,s}\lambda_{i,s}(\lambda_{i,s}t_{i,s})^{\alpha_{i,s}-1} \quad \textbf{(4.2)}$$

To find the distribution at time T, the system component degradation is simulated with the failure rate function (Formula 4.2) for every timestep t in range [t_0, T], where t_0 is the time of entering the current state. The result is compared to a random number between 0 and 1. If the failure rate function is larger, then the component is considered to have degraded to the next state for the rest of the calculation. For state 0, t_0 is the

time of the last maintenance action; for states 1 and 2, t_0 is the time of degradation. In the end, the component is found to be in states 0, 1, 2 or 3. The result is stored, and the simulation is repeated. A higher number of iterations produces a higher probability distribution precision. See Code 6.1 for simplified Python code.

Code 6.1: Finding state probability distribution with Monte Carlo simulations. See Appendix A for the complete code.

```
state_dist = [0,0,0,0]
for mc_loop in range (0, MC_LOOPS):
      state = 0
      time_in_state = 0
      for t in range (0, time_since_maintenance + 1):
            if state == 0:
                  Compute z_0
                  if z_0 >= random.randint(0,10**6)/10**6:
                        state = 1
                        time_in_state = 0
                  else:
                        time_in_state += 1
            elif state == 1:
                  Compute z_1
                  if z_1 >= random.randint(0,10**6)/10**6:
                        state = 2
                        time_in_state = 0
                  else:
                        time_in_state += 1
            elif state == 2:
                  Compute z_2
                  if z_2 >= random.randint(0,10**6)/10**6:
                        state = 3
                        time_in_state = 0
                  else:
                        time_in_state += 1
      state_dist[state] += 1/MC_LOOPS
```

6.2 Control System

See, Appendix E for the code.

For the control system, the same system attributes are used, but with a constant inspection interval instead. To find the optimal inspection interval, a simulation is run for every interval in a chose range, for example 1 to 200 weeks. The best-performing interval is then chosen to be compared to the reinforcement learning system.

A Monte Carlo simulation is used for the probability distribution, as well as for the expected reward for each inspection interval. In other words, the program runs a Monte Carlo simulation inside another Monte Carlo simulation. That makes this step quite time-consuming. See Code 6.2 for simplified Python code.

Code 6.2: Logic to find the optimal inspection interval. See Appendix D for the complete code.

```
sum_reward = [0] * NUMBER_OF_INTERVALS
for loop in range (0, LOOPS):
      for inspection_interval in range (MIN_INTERVAL, MAX_INTERVAL+1):
            for runtime in range (1, TIMEFRAME+1):
                  if (runtime % inspection_interval) == 0:
                        action = 1
                  else:
                        action = 0
                  Compute reward
                  sum_reward [inspection_interval – MIN_INTERVAL] += reward
sum_reward /= LOOPS
rw_max = 0
optimal_inspection_interval = 0
for interval in range (MIN_INTERVAL, MAX_INTERVAL+1):
      if sum_reward[interval - MIN_INTERVAL] > rw_max:
            rw_max = sum_reward[interval - MIN_INTERVAL]
            optimal_inspection_interval = interval
```

The program is run with low precision over an extensive range of inspection intervals (see Figure 6.1) to establish a rough approximation of the optimal interval. The resulting graph is then used to narrow the range for the next, higher-precision run (see Figure 6.2). This is done to save computing time. The high-precision run is run with 15 of each Monte Carlo loop. This is too low to get a conclusive result on the optimal inspection interval, but the resulting estimate is a good enough for our use. The simulation finds the optimal inspection interval to be 15 weeks. Based on these results, 15 weeks is used as the inspection interval of the control system.

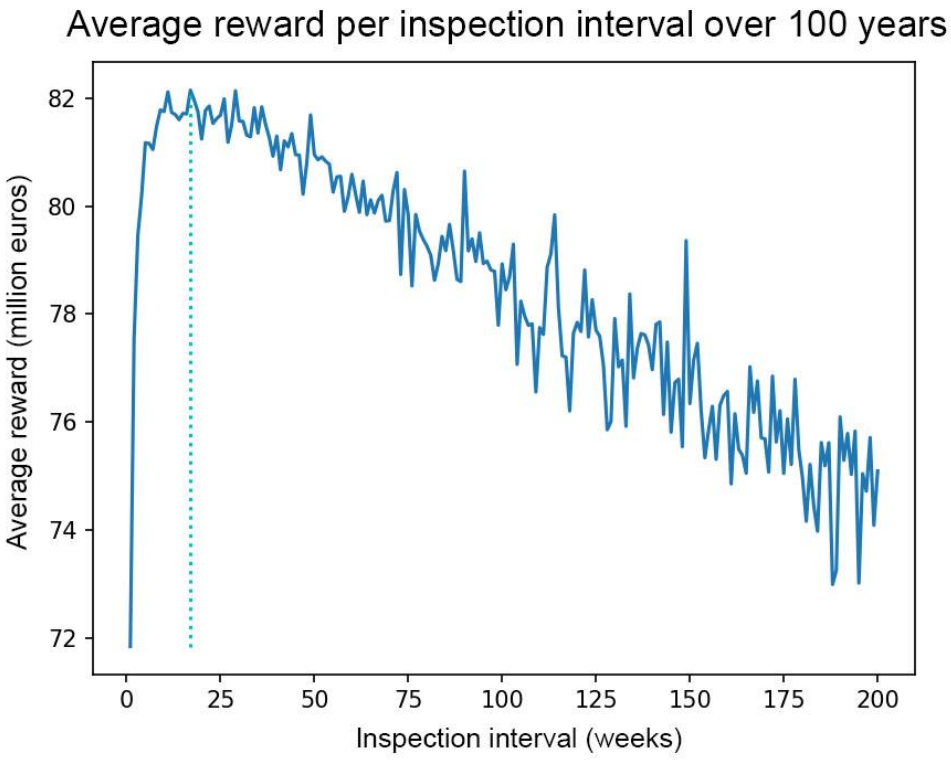


Figure 6.1: **Finding Optimal Inspection Interval, Low Precision: 17 weeks. Avg. reward: 82.145. Minimum interval: 1. Maximum interval: 200. Timeframe: 100 years. Both Monte Carlo simulations: 5 iterations.**

Figure 6.2: **Finding Optimal Inspection Interval, Higher Precision: 15 weeks. Avg. reward: 82.111. Minimum interval: 10. Maximum interval: 40. Timeframe: 100 years. Both Monte Carlo simulations: 15 iterations.**

6.3 Policy Collapse

Running the PPO simulations outright results in a policy collapse (see Figure 6.3). A policy collapse means that the policy dramatically worsen as the agent continues interacting with the environment. Policy collapse is a known phenomenon in RL, however there is currently not much research on the subject [39].

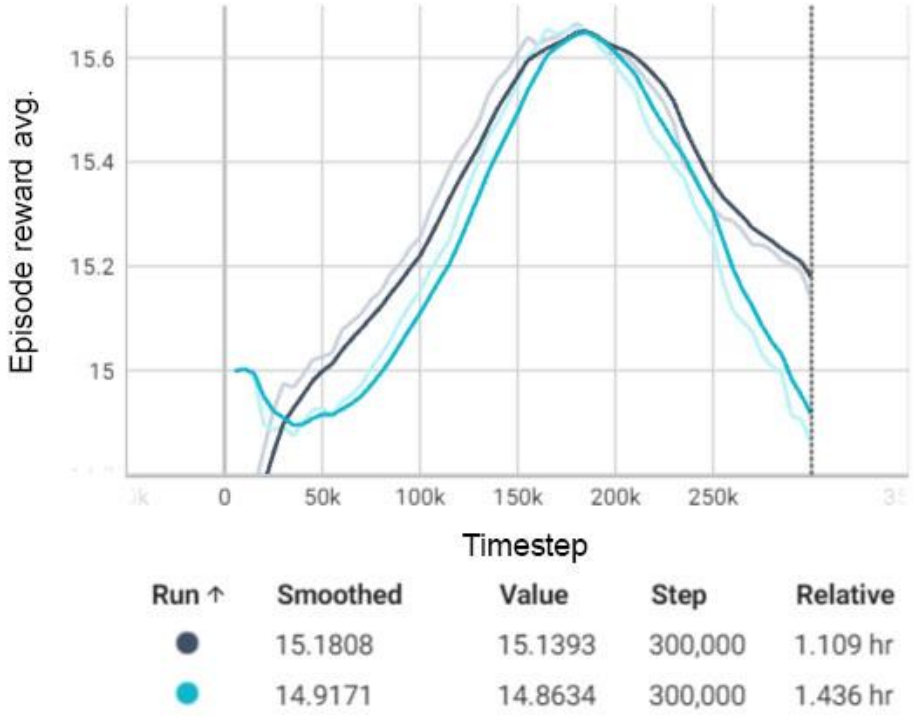


Figure 6.3: Policy collapse. Visualized with TensorBoard.

A common way of overcoming policy collapse is by modifying the parameters of the RL algorithm. This process can be time consuming as a simulation needs to be run after each modification to evaluate its impact.

The policy collapse of the case study PPO is likely caused by the high cost of corrective maintenance. These random spikes in penalty can confuse the algorithm. To reduce the impact of the spikes, the PPO algorithm parameters are altered. The agent is made to base its decision making on a larger batch size and learn at a slower pace. There is also introduced a clipping parameter, which aims to reduce the impact of sudden spikes.

Simulation and PPO algorithm parameters

n_envs = 5
This is the number of environments that are simulated in succession. It helps with exploration and run time.

n_steps = 1 000
This is the horizon. It defines how far into the future rewards influence the policy.

batch_size = 5 000
This is the minibatch size. It defines the number of samples that are stored in the replay memory. batch_size = n_envs * n_steps.

gamma = 0.998
This is the discount factor. It defines how much future rewards are weighted. This coefficient is increased to make the program consider a larger time frame for its actions.

learning_rate = $5 * 10^{-5}$
This is the learning rate. It influences how fast the neural network adapts to observations. By default, learning_rate = 0.003. This rate is reduced to minimize the impact of random spikes.

clip_range = 0.1
This is the clipping parameter. Its purpose is to prevent rapid changes in the policy from singular updates.

Effect of the Parameter Changes

A new simulation is run with the parameter changes in place (see Figure 6.4). The reward is also reduced by a factor of 10 for this test in case the reward function was running out of the bounds of the PPO algorithm. The performance can be observed to be more stable with these changes in place.

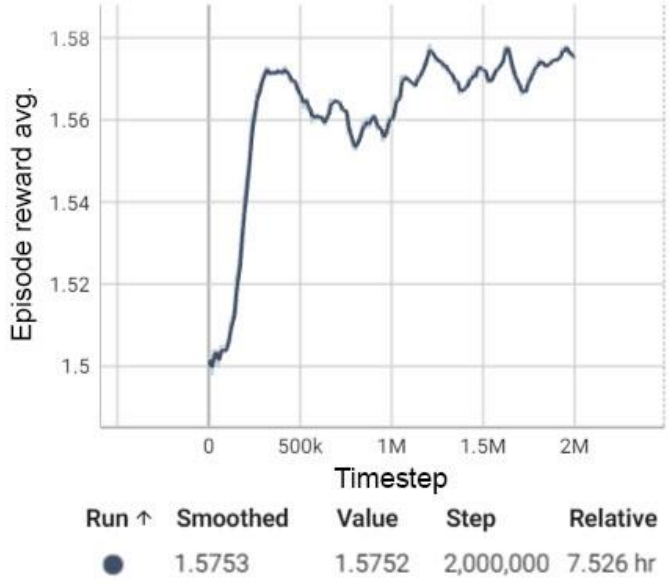


Figure 6.4: New learning rate graph. Visualized with TensorBoard.

6.4 Results

Both programs are run though 10 simulations over a timeframe 30 years with 50 Monte Carlo loops. The 30 year time frame was chosen to make the figures easier to read. These are the results:

Measurement	**Central system**	**Control system**
Average reward	24.719119	24.577995
Standard deviation	0.092868	0.108421

Table 6.3: Performance measurements of the systems

See figure 6.5 and 6.6 on the next page for the reward spreads over the 10 simulations. From these figures the performance spread can be observed. Their performances are also documented in Table 6.3. A higher number of Monte Carlo loops would increase the precision of the graphs, but 50 Monte Carlo loops per simulation is found to be sufficient to establish a trend. The control system holds up surprisingly well. The central system outperforms the control system, but only by 0.57%. This illustrates the effectiveness of calendar-based maintenance and serves as a reminder to why this is the industry standard.

Figures 6.7 and 6.8 show the simulated system reliability of the two approaches over the span of 30 years. The dashed cyan lines are times of inspection. In these figures, it can be observed a notable difference in the system reliability level. The central system is able to maintain a higher level of reliability while performing substantially fewer inspections.

Figures 6.9 and 6.10 provide some more insight into this observation. These figures describe the state probability distribution of the approaches over 30 years. Note how the probability distributions for the degraded and critical states are similar between the two approaches. The main difference is seen in the normal working state and the failed state. This means that the RL system is more effective at performing maintenance on components before they reach the failed state. This can also be observed in the spread of inspections (cyan lines).

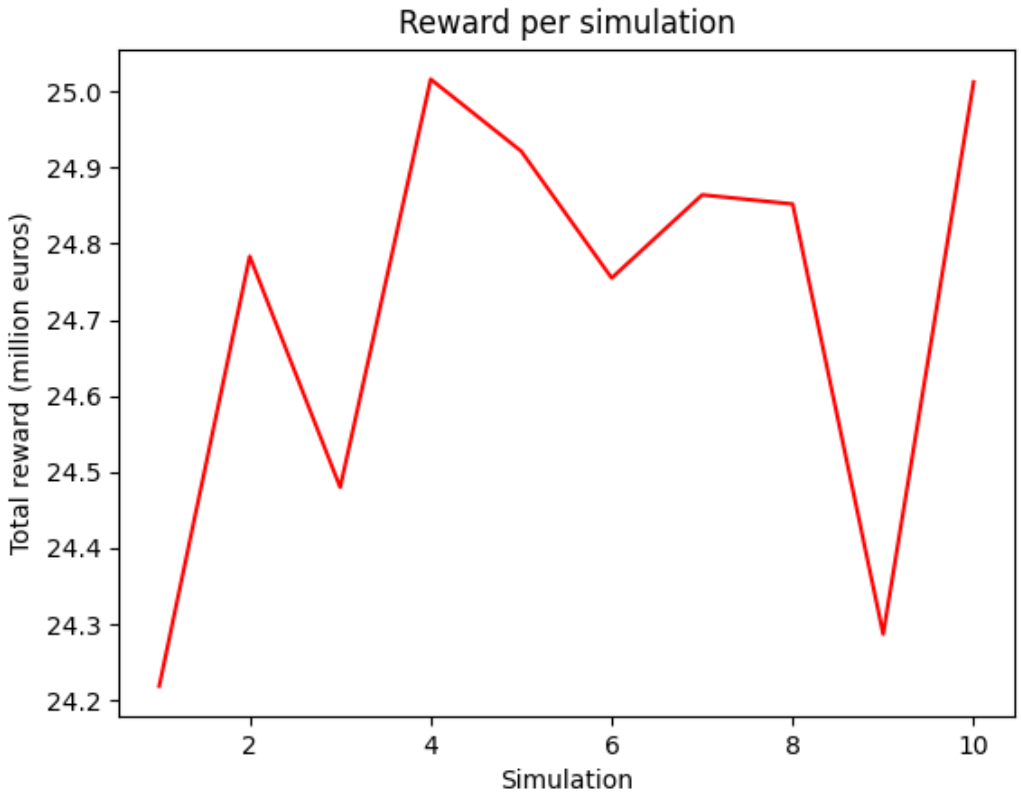


Figure 6.5: **Central system: Reward per simulation. Timeframe: 30 years. 50 Monte Carlo simulations. Avg. reward: 24.719. Standard deviation: 0.093.**

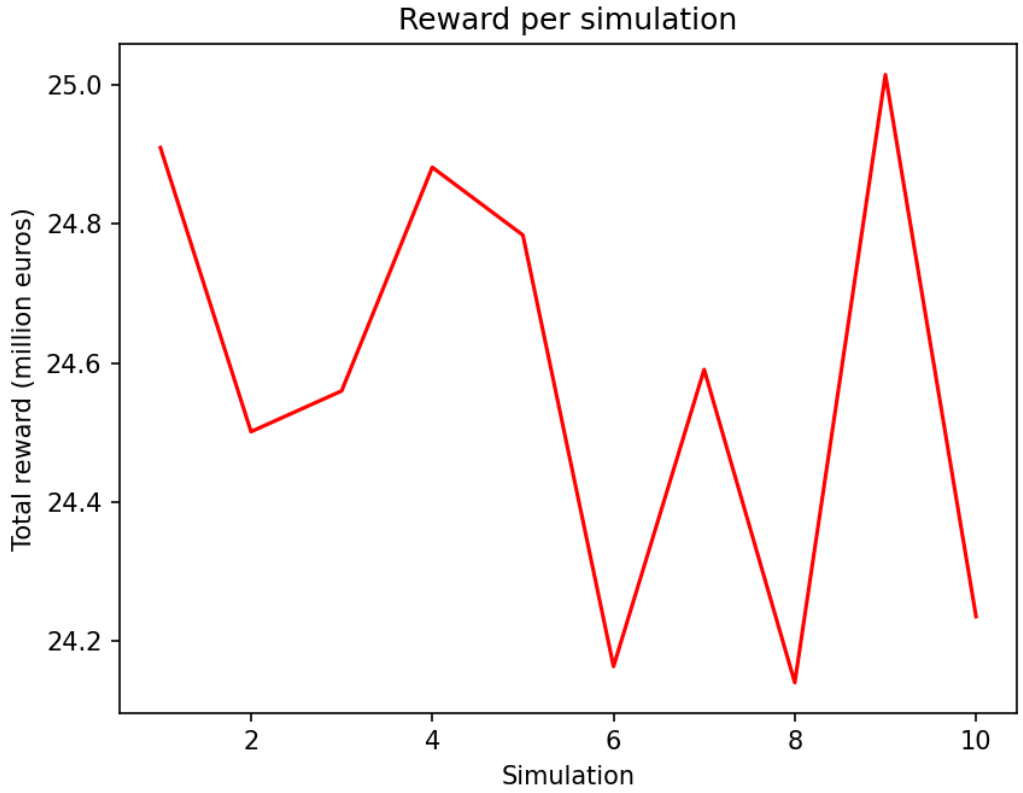


Figure 6.6: **Control system: Reward per simulation. Timeframe: 30 years. 50 Monte Carlo simulations. Avg. reward: 24.578. Standard deviation: 0.108.**

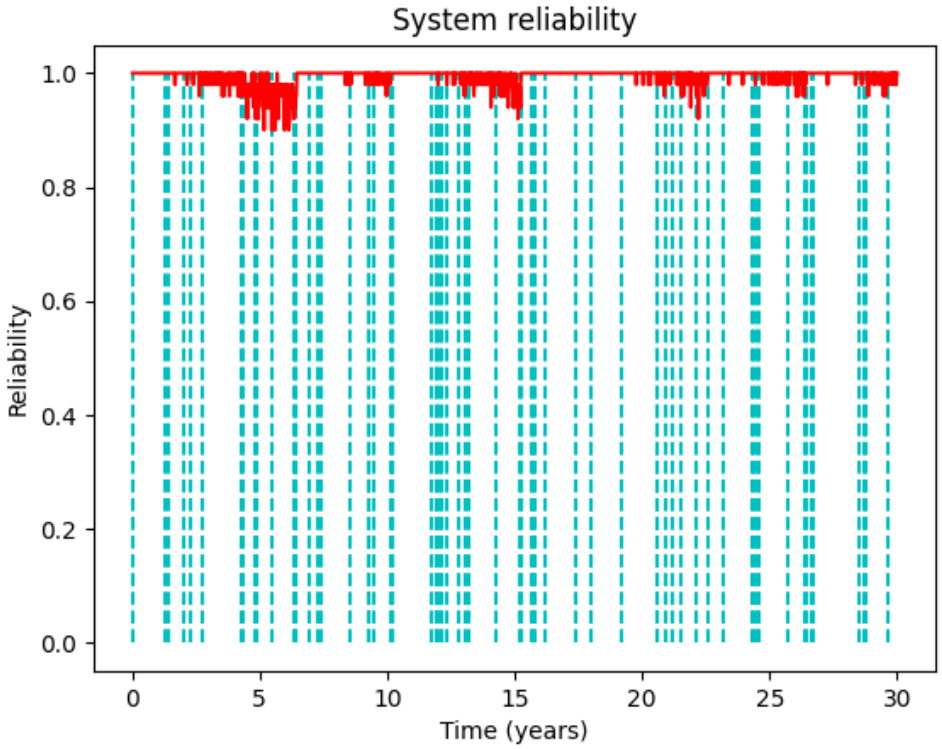


Figure 6.7: **Central system: System reliability. 65 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

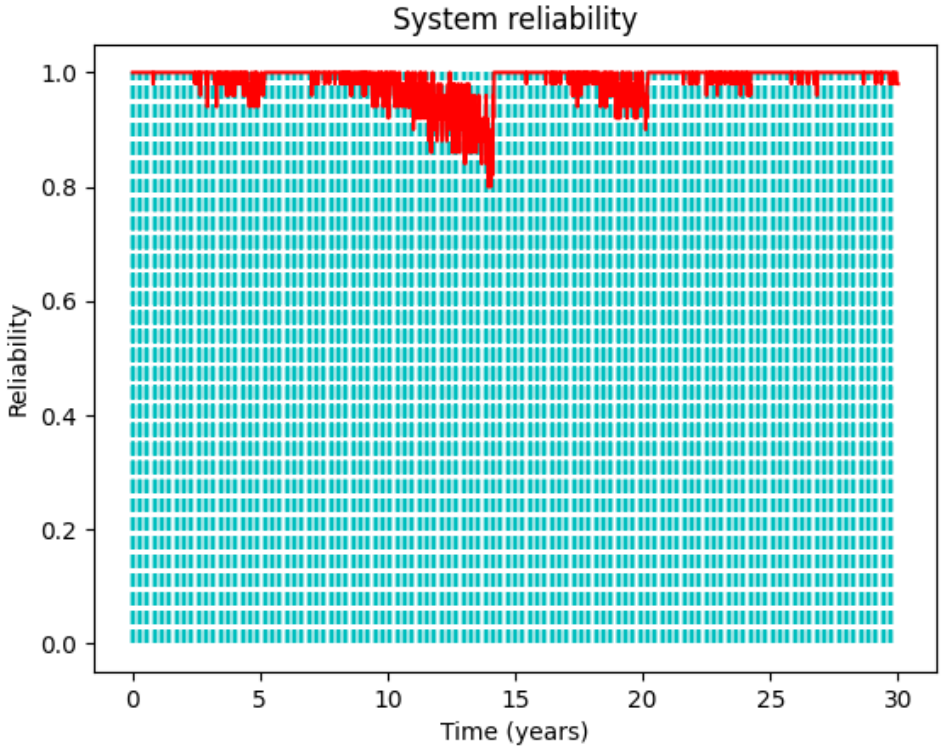


Figure 6.8: **Control system: System reliability. 104 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

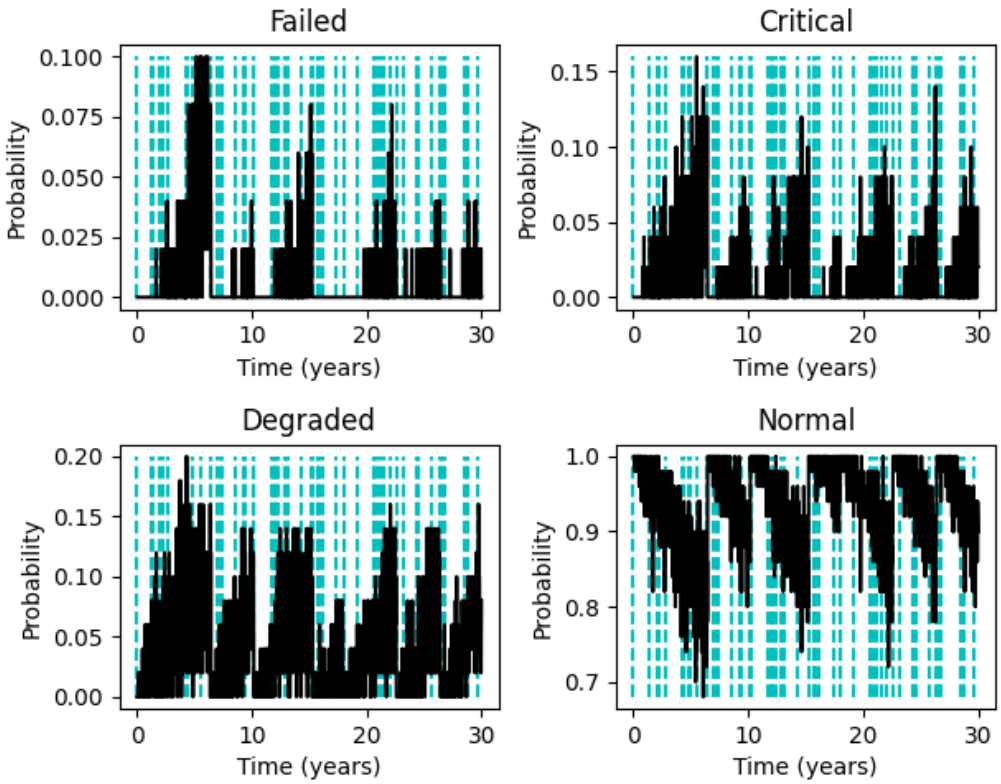


Figure 6.9: **Central system: State probability distribution. 65 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

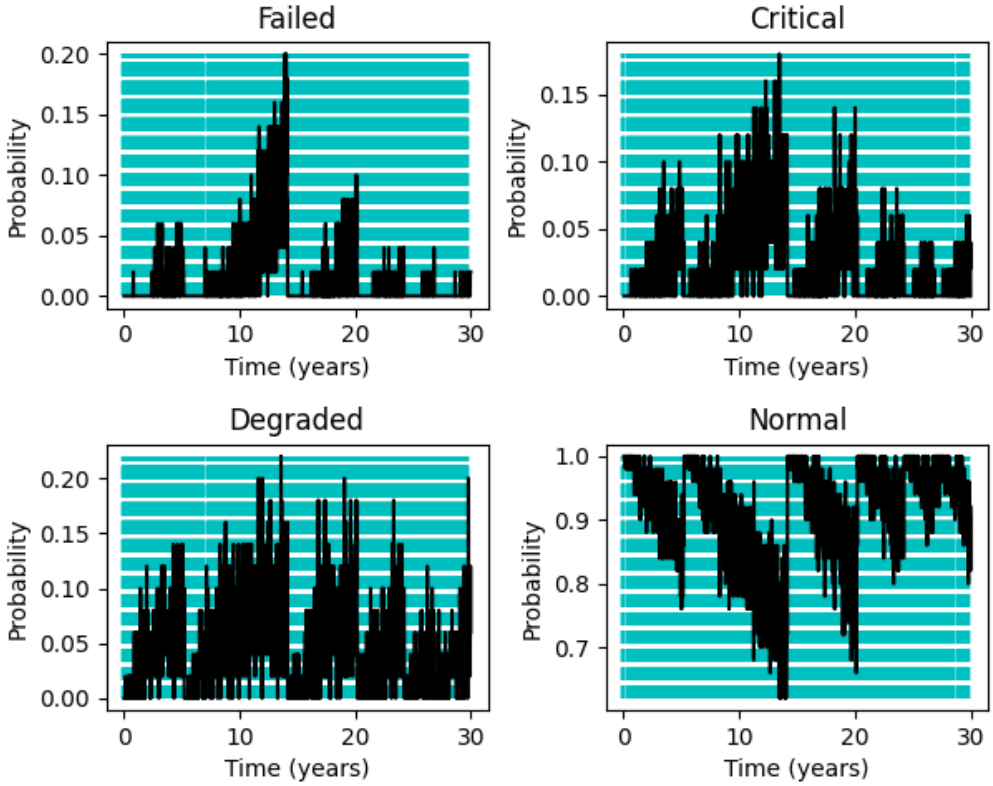


Figure 6.10: **Control system: State probability distribution. 104 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

Chapter 7 – Second Case Study

See Appendix F, Appendix G, Appendix H, Appendix I and Appendix J for the code.

7.1 Model Changes

A weakness of the model in the first case study was that an inspection without repair had no effect on the reliability of the system. In this chapter, a different model is proposed. This model will assume all components to be in the normal working state after an inspection is performed and use this to calculate the state probability distribution.

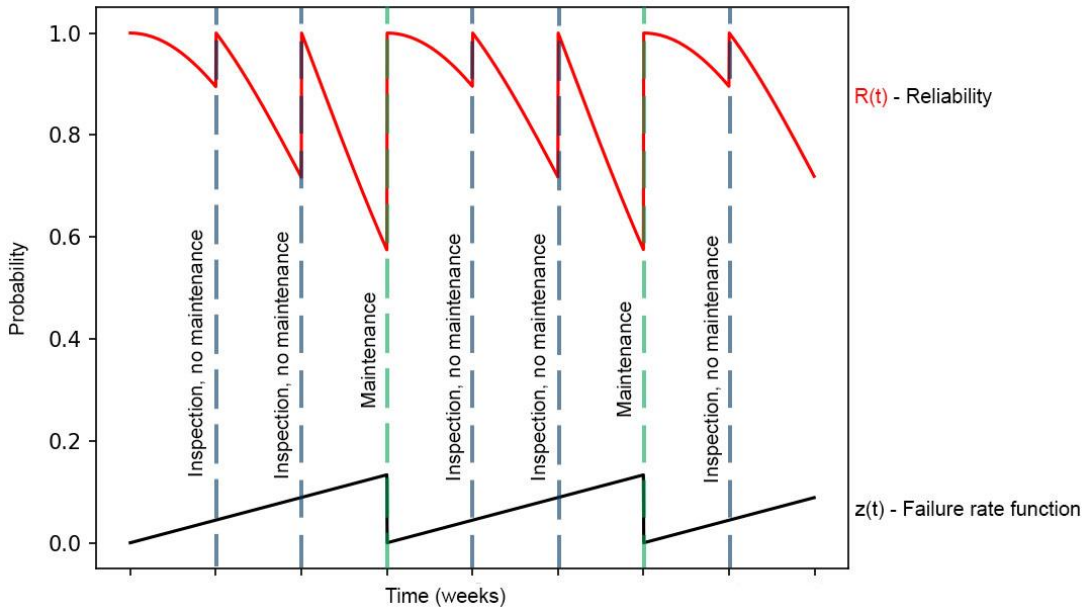


Figure 7.1: Illustration of proposed component reliability and failure rate function

The model will use the time since inspection as the basis for the component state and the component age (time since maintenance) will be used to find the failure rate function (see figure 7.1 and 7.2). The main goal of the model is to reduce the simulation processing time by lowering number of calculations. The PPO agent is also expected to stop suggesting weekly inspections for components that have surpassed a certain age.

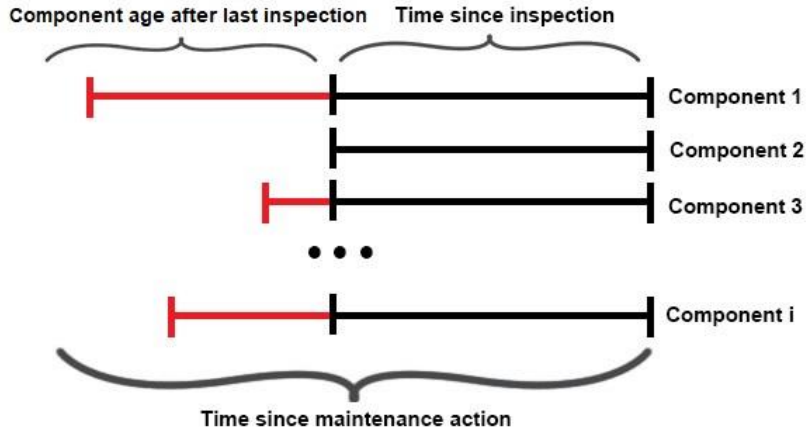


Figure 7.2: Descriptive illustration of terms

In Figure 7.2, notice how component 2 received maintenance during the last inspection. The time since inspection and time since maintenance action for this component are therefore the same.

The model in this case study also has a different way of calculating the cost of maintenance. The spikes in penalty in the previous system seemed to impact its performance. Since the system model does not simulate a change in degradation after repairs the only simulated difference between preventive and corrective maintenance is the cost and production time loss. There is also no additional penalty for system breakdowns. This means that the spikes can be evened out by extrapolating the cost of preventive and corrective maintenance into a continuous function with the state probability distribution as its variable (See formula 7.1). This approach should not have any impact on the optimal solution.

$$C_{M,i}(t) = C_{PM,i}\frac{\Pr_{i,s=1}(t)}{\Pr_{i,s=1}(t)+\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)} + C_{CM,i}\frac{\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)}{\Pr_{i,s=1}(t)+\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)} \quad \textbf{(7.1)}$$

The same approach is also used to calculate expected maintenance downtime:

$$T_{M,i}(t) = T_{PM,i}\frac{\Pr_{i,s=1}(t)}{\Pr_{i,s=1}(t)+\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)} + T_{CM,i}\frac{\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)}{\Pr_{i,s=1}(t)+\Pr_{i,s=2}(t)+\Pr_{i,s=3}(t)} \quad \textbf{(7.2)}$$

Like before, the combined maintenance costs are found by adding the cost of inspection, C_I, to the cost of all preventive/corrective maintenance actions performed at time t, $\sum C_{M,i}(t)$. The same is done for production loss.

If inspection is performed at t:

$$C_M(t) = C_I + \sum C_{M,i}(t) \quad \textbf{(7.3)}$$

$$T_M(t) = T_I + \sum T_{M,i}(t) \quad \textbf{(7.4)}$$

The optimal constant inspection interval is found with the same method as before (see figure 7.3 and 7.4).

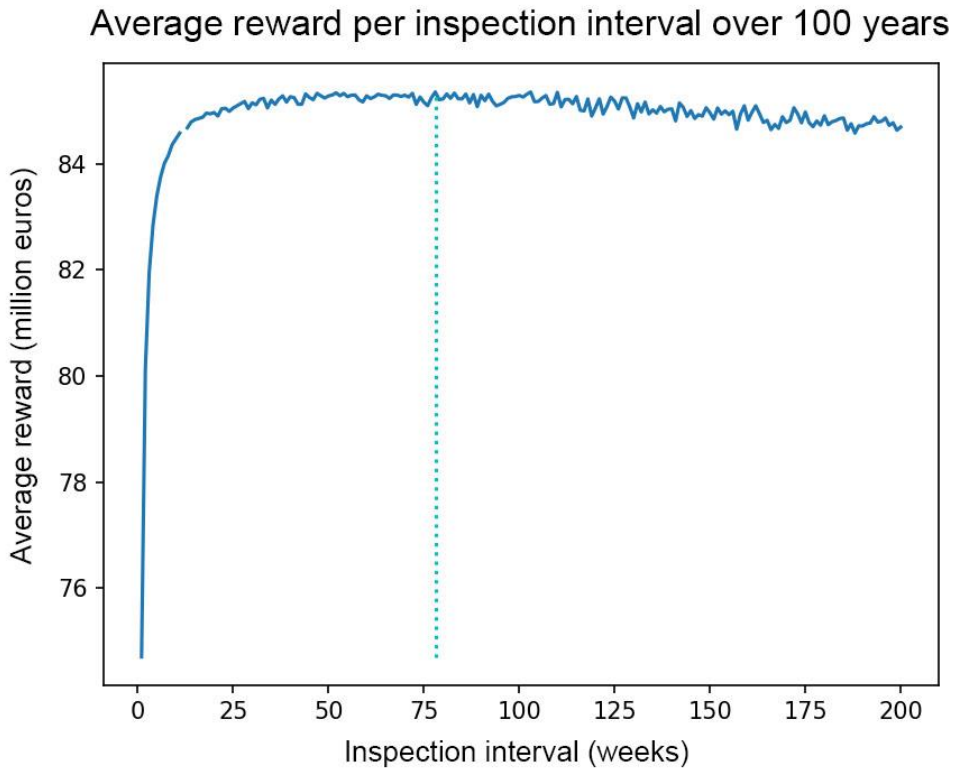


Figure 7.3: **Finding optimal inspection interval, low precision: 78 weeks. Avg. reward: 85.356. Minimum interval: 1. Maximum interval: 200. Timeframe: 100 years. Both Monte Carlo simulations: 5 iterations.**

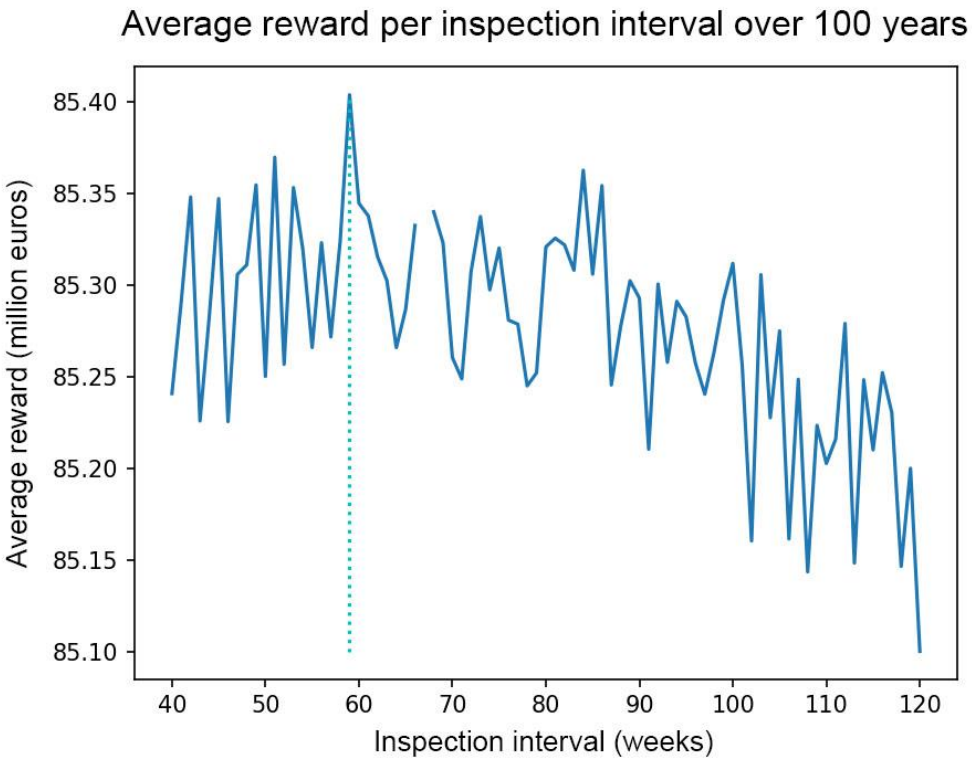


Figure 7.4: **Finding optimal inspection interval, higher precision: 59 weeks. Avg. reward: 85.404. Minimum interval: 40. Maximum interval: 120. Timeframe: 100 years. Both Monte Carlo simulations: 15 iterations.**

Policy Collapse

Like in the first case study, the RL policy struggles with policy collapse. A clipping parameter of 0.1 is introduced to combat this as well as an increase in the gamma parameter from 0.99 to 0.995. The RL-agent is able to outperform the control system after 6 000 000 timesteps of simulation. This simulation took 18 hours. With more simulation time it can be assumed that the performance would increase further.

7.2 Results

Both programs are run though 10 simulations over 30 years with 50 Monte Carlo loops. These are the results:

Measurement	**Central system**	**Control system**
Average reward	25.577634	25.551247
Standard deviation	0.025703	0.021337

Table 7.1: Performance measurements of the systems

Though the time per simulation was lowered, the system model was a lot harder to optimize than the first. The reason for this can be seen in Figure 7.3 and 7.4. The top of the curve is quite flat, meaning that the difference in performance between inspection intervals is very low. This makes establishing the pattern notably hard for the algorithm. Even the slightest of maintenance spikes would be devastating for the PPO without the clipping parameter in place.

Figures 7.5 and 7.6 show the performance spread for the central and control systems across 10 simulations. The result is also summarized in Table 7.1.

The difference in performance between the two approaches is 0.1%, even lower than in the previous case study. It should be noted that this difference likely would increase if given more simulation time. The times of inspection (cyan lines) in figures 7.7 and 7.9 visualize how the RL-agent is yet to understand the system. It would be more optimal to perform rapid inspections on an old component. The simulation starts with a completely new component, so the spike in inspections around week 1 shows this misunderstanding. It can also be observed how the correlation between degradation and inspection is yet to be established by how the peaks in degradation does not correlate to increased frequency in inspections. Still, the agent has slowly been improving with the simulations and performs on par with the optimal calendar-based approach, even before ironing out these misunderstandings.

From Figure 7.7, 7.8, 7.9 and 7.10 it can be observed that the central system performs approximately the same number of inspections (cyan lines) as the control system while maintaining a similar or perhaps even worse level of reliability.

The rewards in this case study are higher than in the previous one. This was expected because it is based on a more optimistic approach to reliability. This means that an implementation of this approach would demand a higher level of precision in the inspection data and the statistical data. The model in the second case study is based on a very theoretical understanding of degradation. It assumes rigid differences in the Markov states, perfect maintenance, and perfect inspections. In practice, the lines are generally a bit more blurred.

Because the simulation time for this case study was so limited, the first case study will be weighted higher in the conclusions.

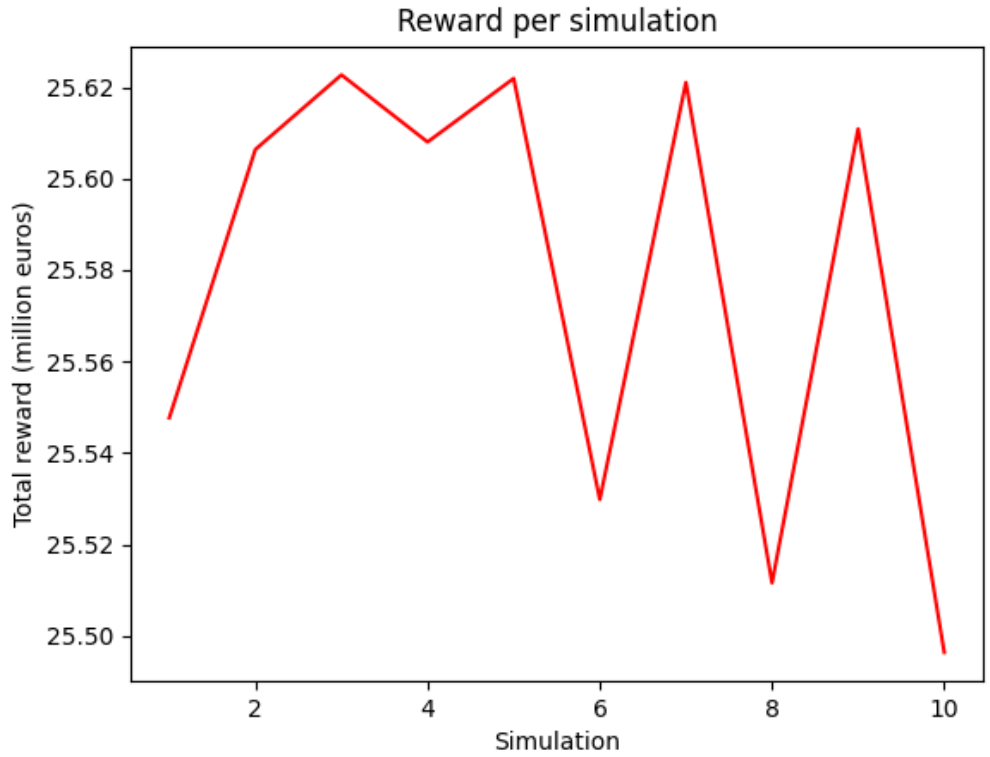


Figure 7.5: **Central system: Reward per simulation. Timeframe: 30 years. 50 Monte Carlo simulations. Avg. reward: 25.578. Standard deviation: 0.026.**

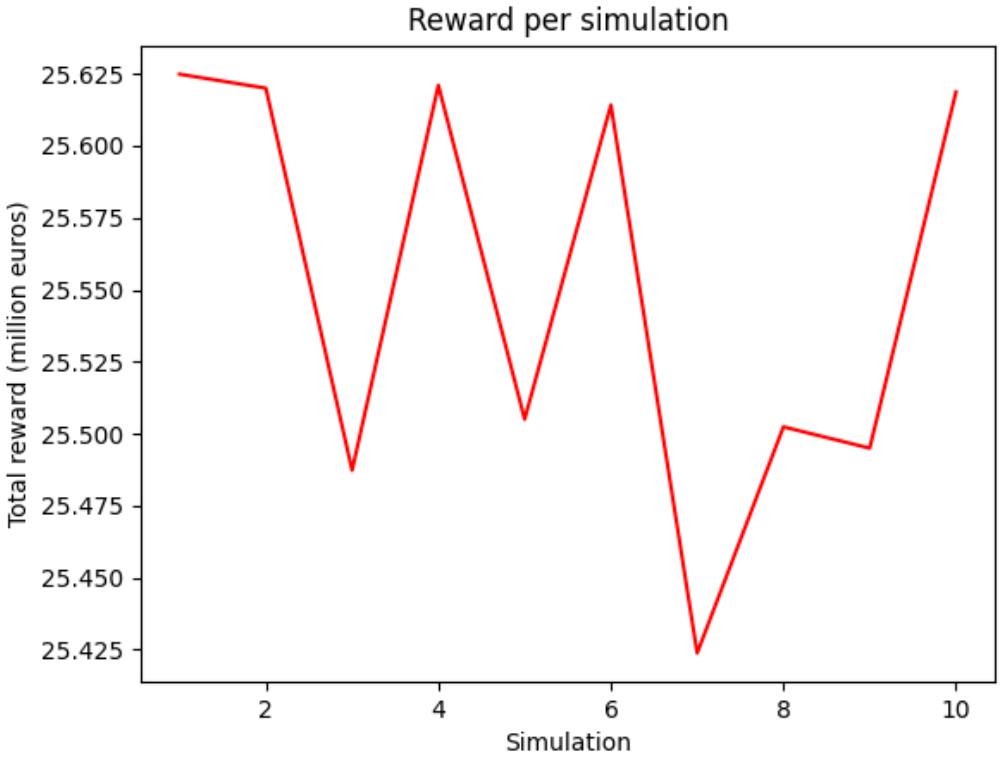


Figure 7.6: **Control system: Reward per simulation. Timeframe: 30 years. 50 Monte Carlo simulations. Avg. reward: 25.551. Standard deviation: 0.021.**

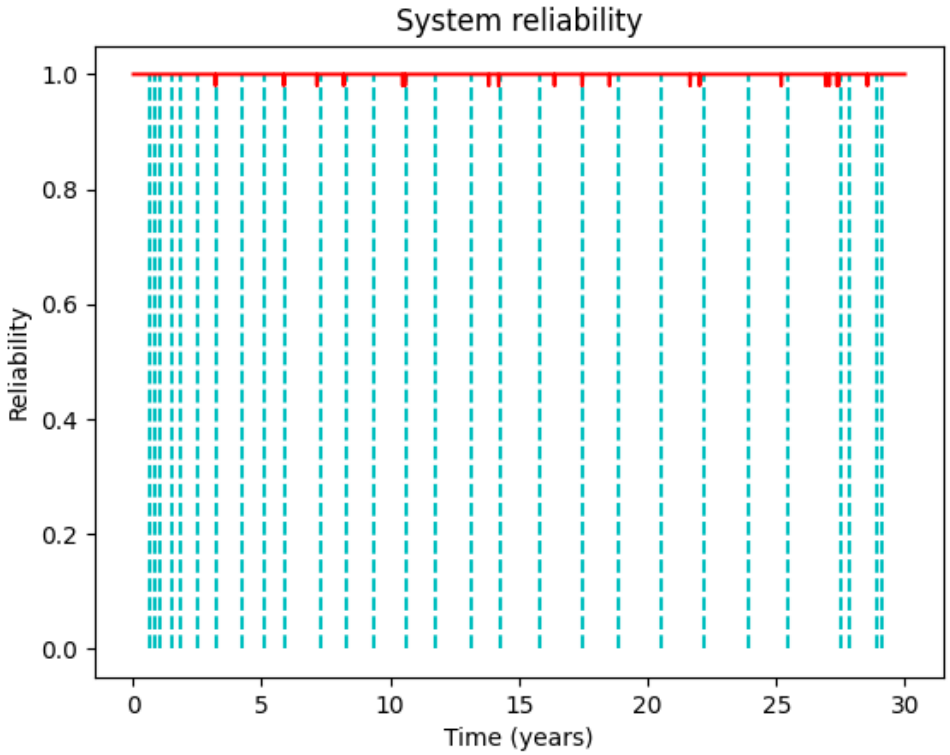


Figure 7.7: **Central system: System reliability. 28 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

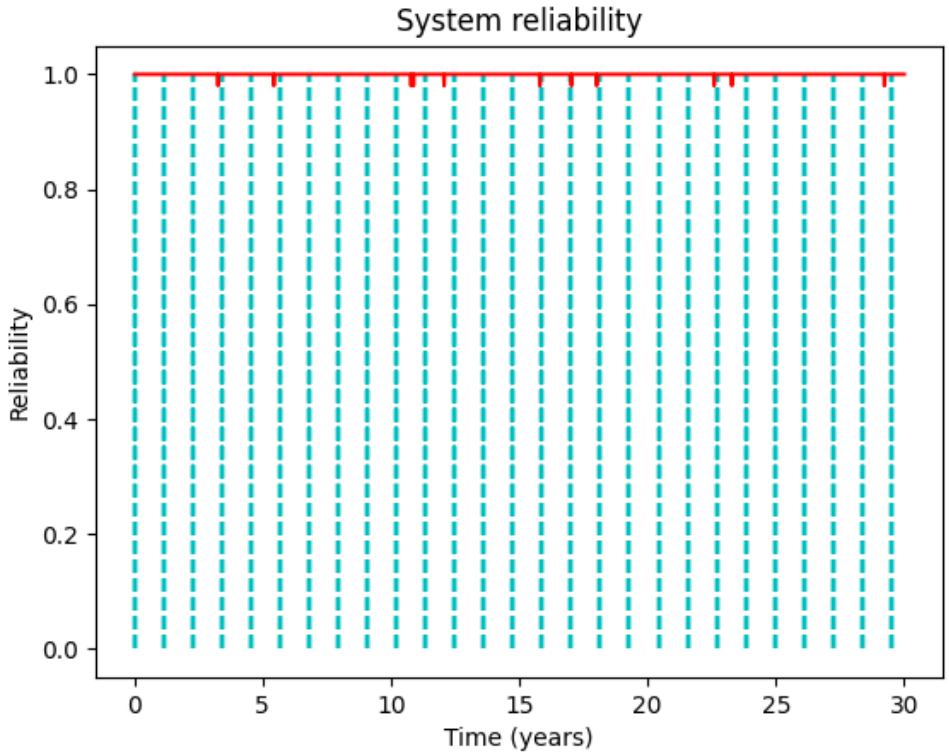


Figure 7.8: **Control system: System reliability. 27 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

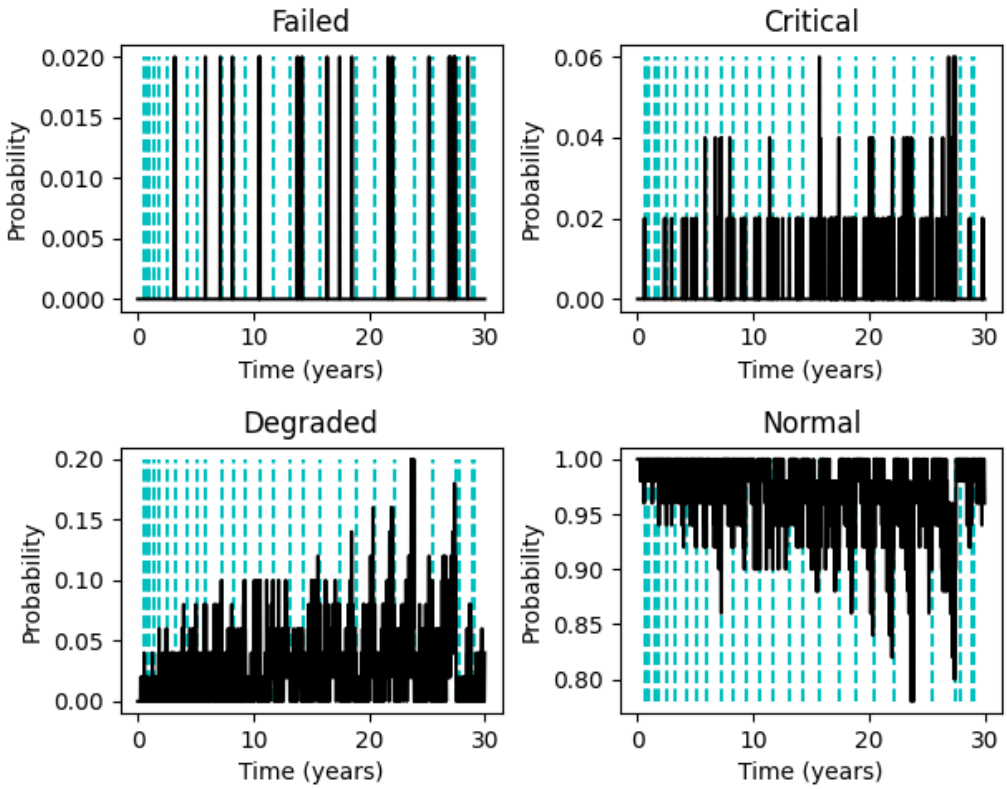


Figure 7.9: **Central system: State probability distribution. 28 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

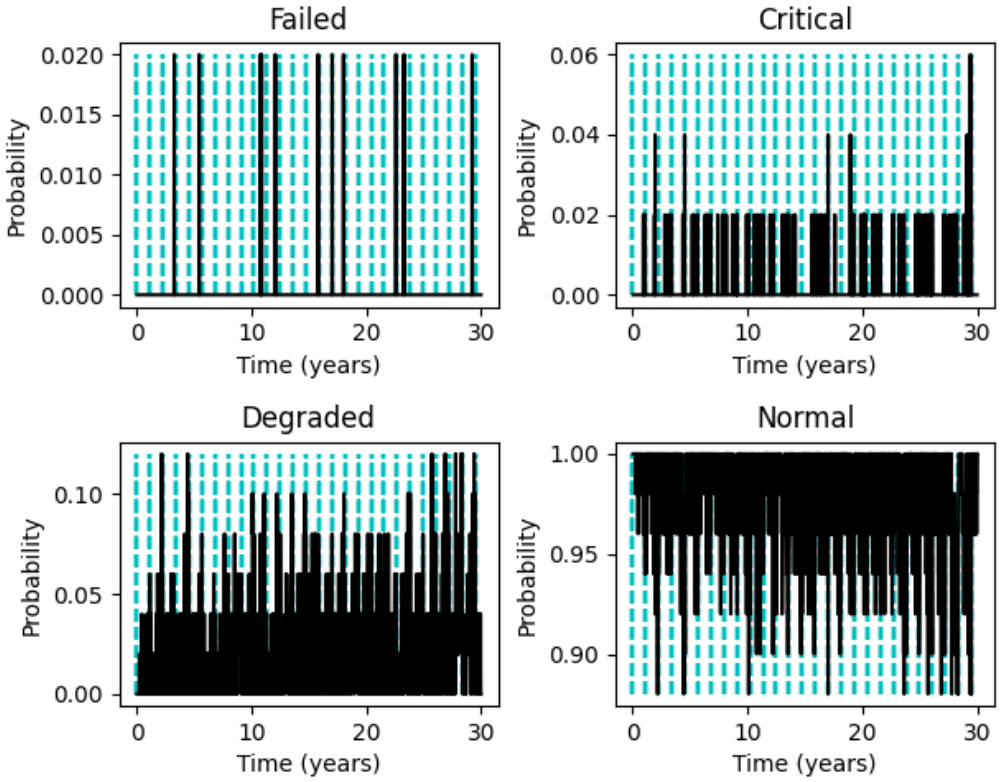


Figure 7.10: **Control system: State probability distribution. 27 inspections over 30 years. 50 Monte Carlo simulations. The dashed cyan lines are times of inspection.**

Chapter 8 – Discussion

Due to the time constraints, equipment limitations and available experience, there was no time to study how the effectiveness of RL-based vs calendar-based inspection planning developed as more components were introduced. This will have to be done in future work. The complexity of the model created a bottleneck for processing power. Using a PPO-algorithm to optimize an inspection schedule is substantially more RAM intensive than letting it decide the actions outright. Combining this problem setting with a 4-state Weibull-Markov system representation was perhaps a bit ambitious, given the limitations.

Policy collapse was the biggest time sink of the report. It did, however, teach a valuable lesson in how policy parameters of the PPO can stabilize the model with enough trial and error.

The results of the case studies have demonstrated the possibility of utilizing RL for optimizing the inspection schedules for mechanical systems. The case studies have also highlighted the effectiveness of calendar-based inspection planning.

The optimal inspection intervals were drastically different in the two case studies. The two different approaches to reliability theory would therefore have a substantial impact on the performance of the inspection schedule. It is important to be mindful of the significance of the simplifications that are made when modelling a complex system.

Chapter 9 – Conclusions and Future Work

Conclusions

This report has demonstrated how reinforcement learning can be used for inspection scheduling for offshore wind turbines. The performance of the model was compared to that of an optimized calendar-based inspection schedule. Performance was measured in cost savings, but reliability and number of inspections were also observed. There were conducted two case studies of single-component systems based on different approaches to reliability.

The reinforcement learning-based models performed marginally better than the calendar-based inspections. The first cast study was able to achieve this performance while maintaining a higher level of reliability and scheduling notably fewer inspections. Optimizing the second case study was more time intensive than the first. During the timeframe of this report, it could only be optimized to the point of performing on par with the control system.

Suggestions for Future Work

Short term: It would be interesting to investigate how the RL- and calendar-based approaches perform as more components are introduced. Can a trend be observed as more components are added?

Medium term: The RL environment could be expanded to handle logistical parameters, such as additional wind turbines, available maintenance crew, travel times, work hours and supply chain delays. How would it handle these additional parameters? This would significantly add to the complexity of the model further increasing its processing demand.

Long term: Could a complex RL-based inspection schedule be designed to be competitive in performance to that of a condition-monitoring approach to maintenance planning?

Bibliography

[1] Yürüşen, N. Y., Rowley, P. N., Watson, S. J., & Melero, J. J. (2020). Automated wind turbine maintenance scheduling. *Reliability Engineering & System Safety*, *200*, 1–14. https://doi.org/10.1016/j.ress.2020.106965

[2] Toftaker, H., Bødal, E. F., & Sperstad, I. B. (2022). Joint optimization of preventive and condition-based maintenance for offshore wind farms. *Journal of Physics. Conference Series*, *2362*(1), 12041. https://doi.org/10.1088/1742-6596/2362/1/012041

[3] Le, B., & Andrews, J. (2016). Modelling wind turbine degradation and maintenance. Wind Energy (Chichester, England), 19(4), 571–591. https://doi.org/10.1002/we.1851

[4] Belhaj Salem, M., Fouladirad, M., & Deloux, E. (2021). Prognostic and Classification of Dynamic Degradation in a Mechanical System Using Variance Gamma Process. *Mathematics (Basel)*, *9*(3), 254. https://doi.org/10.3390/math9030254

[5] Pinciroli, L., Baraldi, P., Ballabio, G., Compare, M., & Zio, E. (2022). Optimization of the Operation and Maintenance of renewable energy systems by Deep Reinforcement Learning. *Renewable Energy*, *183*, 752–763. https://doi.org/10.1016/j.renene.2021.11.052

[6] Su, J., Huang, J., Adams, S., Chang, Q., & Beling, P. A. (2022). Deep multi-agent reinforcement learning for multi-level preventive maintenance in manufacturing systems. *Expert Systems with Applications*, *192*, 116323. https://doi.org/10.1016/j.eswa.2021.116323

[7] Huang, J., Chang, Q., & Arinez, J. (2020). Deep reinforcement learning based preventive maintenance policy for serial production lines. *Expert Systems with Applications*, *160*, 113701. https://doi.org/10.1016/j.eswa.2020.113701

[8] Yao, L., Dong, Q., Jiang, J., & Ni, F. (2020). Deep reinforcement learning for long-term pavement maintenance planning. *Computer-Aided Civil and Infrastructure Engineering*, *35*(11), 1230–1245. https://doi.org/10.1111/mice.12558

[9] Zhang, Q., Hua, C., & Xu, G. (2014). A mixture Weibull proportional hazard model for mechanical system failure prediction utilising lifetime and monitoring data. *Mechanical Systems and Signal Processing*, *43*(1-2), 103–112. https://doi.org/10.1016/j.ymssp.2013.10.013

[10] Pinciroli, L., Baraldi, P., Ballabio, G., Compare, M., & Zio, E. (2022). Optimization of the Operation and Maintenance of renewable energy systems by Deep Reinforcement Learning. *Renewable Energy*, *183*, 752–763. https://doi.org/10.1016/j.renene.2021.11.052

[11] Ruiz Rodríguez, M. L., Kubler, S., de Giorgio, A., Cordy, M., Robert, J., & Le Traon, Y. (2022). Multi-agent deep reinforcement learning based Predictive Maintenance on parallel machines. *Robotics and Computer-Integrated Manufacturing*, *78*, 102406. https://doi.org/10.1016/j.rcim.2022.102406

[12] Zhang, C., Chen, H.-P., Tee, K. F., & Liang, D. (2021). Reliability-Based Lifetime Fatigue Damage Assessment of Offshore Composite Wind Turbine Blades. *Journal of Aerospace Engineering*, *34*(3). https://doi.org/10.1061/(ASCE)AS.1943-5525.0001260

[13] De Simón-Martín, M. (2022). Levelized cost of energy in sustainable energy communities : a systematic approach for multi-vector energy systems. Springer International Publishing.

[14] IRENA (2023), Renewable power generation costs in 2022, International Renewable Energy Agency, Abu Dhabi.

[15] Ren, Verma, Li, Teuwen, & Jiang. (2021). *Offshore wind turbine operations and maintenance: A state-of-the-art review*. https://www.sciencedirect.com/science/article/pii/S1364032121001805

[16] Karyotakis, A., & Bucknall, R. (2010). Planned intervention as a maintenance and repair strategy for offshore wind turbines. *Journal of Marine Engineering and Technology*, *9*(1), 27–35. https://doi.org/10.1080/20464177.2010.11020229

[17] Saleh, A., Chiachío, M., Salas, J. F., & Kolios, A. (2023). Self-adaptive optimized maintenance of offshore wind turbines by intelligent Petri nets. *Reliability Engineering & System Safety*, *231*, 109013. https://doi.org/10.1016/j.ress.2022.109013

[18] Cheng, J., Liu, Y., Li, W., & Li, T. (2023). Deep reinforcement learning for cost-optimal condition-based maintenance policy of offshore wind turbine components. *Ocean Engineering*, *283*, 115062. https://doi.org/10.1016/j.oceaneng.2023.115062

[19] Wang, Y., Deng, C., Wu, J., & Xiong, Y. (2015). Failure time prediction for mechanical device based on the degradation sequence. *Journal of Intelligent Manufacturing*, *26*(6), 1181–1199. https://doi.org/10.1007/s10845-013-0849-4

[20] Rausand, M., & Høyland, A. (2004). System Reliability Theory: Models, Statistical Methods, and Applications. Wiley, Hoboken, NJ, 2nd edition.

[21] Rausand, M. (2011). Risk Assessment: Theory, Methods, and Applications. Wiley, Hoboken, NJ.

[22] Zhao, X., & Ren, L. (2015). Focus on the development of offshore wind power in China: Has the golden period come? *Renewable Energy*, *81*, 644–657. https://doi.org/10.1016/j.renene.2015.03.077

[23] Hermelink, A., & de Jager, D. (2015). Evaluating Our Future: The crucial role of discount rates in European Commission energy system modelling. The European Council for an Energy Efficient Economy & Ecofys.

[24] B. Lu, Y. Li, X. Wu and Z. Yang, "A review of recent advances in wind turbine condition monitoring and fault diagnosis," *2009 IEEE Power Electronics and Machines in Wind Applications*, Lincoln, NE, USA, 2009, pp. 1-7, doi: 10.1109/PEMWA.2009.5208325

[25] Sun, Y., Li, H., Sun, L., & Guedes Soares, C. (2023). Failure Analysis of Floating Offshore Wind Turbines with Correlated Failures. *Reliability Engineering & System Safety*, *238*, 109485. https://doi.org/10.1016/j.ress.2023.109485

[26] Nielsen, J. J., & Sørensen, J. D. (2011). On risk-based operation and maintenance of offshore wind turbine components. *Reliability Engineering & System Safety*, *96*(1), 218–229. https://doi.org/10.1016/j.ress.2010.07.007

[27] Wang, Y., & Liu, Y. (2019). Optimal Reliability Growth Program for Repairable and Warranted Products. *2019 Annual Reliability and Maintainability Symposium (RAMS)*, 1–6. https://doi.org/10.1109/RAMS.2019.8769010

[28] Tavner, P. (2012). *Offshore Wind Turbines: Reliability, availability and maintenance* (1st ed., Vol. 13). The Institution of Engineering and Technology.

[29] McMillan, D., & Ault, G. W., (2008) Specification of reliability benchmarks for offshore wind farms. *Proceedings of the European safety and reliability*; 22–25. https://cs.stir.ac.uk/~kjt/research/prosen/pub/esrel08-mcmillan.pdf

[30] AurelianTactics. (2018, Jul 28). *PPO Hyperparameters and Ranges*. Medium.com https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe

[31] Schulman, J., Wolski, J., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms*. (v2). CoRR. arXiv:1707.06347v2

[32] Tessler, Chen. (2020). *Deep Reinforcement Learning Works - Now What?* https://tesslerc.github.io/posts

[33] Stable Baselines3. (Last accessed: 2023, Nov 11). *Vectorized Environments.* https://stable-baselines.readthedocs.io/en/master/guide/vec_envs.html

[34] Gym library. (Last accessed: 2023, Nov 11). *Vectorising your environments*. https://www.gymlibrary.dev/content/vectorising/

[35] Pugliese, R., Regondi, S., & Marini, R. (2021). *Machine learning-based approach: global trends, research directions, and regulatory standpoints.* Data Science and Management Volume 4, Pages 19-29. https://doi.org/10.1016/j.dsm.2021.12.002

[36] A. G. Barto, R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834-846, Sept.-Oct. 1983, doi: 10.1109/TSMC.1983.6313077

[37] Hugging Face (2022, Aug 5) *Proximal Policy Optimization (PPO)* https://huggingface.co/blog/deep-rl-ppo

[38] OpenAI.com (2017, Jul 20) *Research: Proximal Policy Optimization* https://openai.com/research/openai-baselines-ppo

[39] Dohare, S., Lan, Q., & Mahmood, A. R. (2023). Overcoming Policy Collapse in Deep Reinforcement Learning. *Sixteenth European Workshop on Reinforcement Learning*. https://openreview.net/forum?id=m9Jfdz4ymO

[40] Li, Y. (2018). Deep Reinforcement Learning: An Overview. *CORR*. https://doi.org/10.48550/arXiv.1810.06339

Appendix A

The First Environment

```
import numpy as np

import random

import gymnasium

from gymnasium import spaces


class Simple_Single_Component(gymnasium.Env):

    metadata = {"render_modes": ["rgb_array"], "render_fps": 4}


    def __init__(self):

        # Action space: (0) no action, (1) inspection

        self.action_space = spaces.Discrete(2)


       # Observation space:

       # | Num | Description                                          | Min | Max|

       # |-----|------------------------------------------------------|-----|----|

       # | 0   | (Blade) Time since maintenance action (weeks) | 0   | inf|


        COMPONENTS = 1

        low = np.zeros((COMPONENTS),dtype=np.float32)

        high = np.ones((COMPONENTS),dtype=np.float32)

        high *= 10**6

        self.observation_space = spaces.Box(low, high, dtype=np.float32)


        self.time_since_maintenance = None


    def step(self, action):

        COMPONENTS = 1

        MC_LOOPS = 10

        POWER_CAPACITY = 5

        CAPACITY_FACTOR = 0.49

        VALUE_OF_MWH = 40
```

```
        INSPECTION_COST = 2000
        CONVERSION_RATE = 1.14

        # Cost of repair, cost of replacement
        COST_OF_MAINTENANCE = np.array([
            [4000, 200000]], dtype=np.float32)
        COST_OF_MAINTENANCE *= CONVERSION_RATE

        # Duration of repair, duration of replacement
        DURATION_OF_MAINTENANCE = np.array([
            [3, 70]], dtype=np.float32)

        # Weibull parameters for the markov transition to degradated,
critical, failure states
        SCALE_PARAM = np.array([
            [23.02, 2.88, 2.88]], dtype=np.float32)
        SCALE_PARAM = 1/(SCALE_PARAM*52)
        SHAPE_PARAM = np.array([
            [1.2, 1.2, 1.2]], dtype=np.float32)
        z_normal = np.zeros(COMPONENTS, dtype=np.float32)
        z_degraded = np.zeros(COMPONENTS, dtype=np.float32)
        z_critical = np.zeros(COMPONENTS, dtype=np.float32)
        reward = 0
        done = False
        production_hours = 7*24
        reliability = [1] * COMPONENTS
        system_reliability = 1
        state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
                time_in_state = 0
                for t in range (0,
round(self.time_since_maintenance[component])+1):
```

```
                    if state == 0:
                        z_normal[component] =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal[component] >=
random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded[component] =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded[component] >=
random.randint(1,10**6)/10**6:
                            state = 2
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 2:
                        z_critical[component] =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical[component] >=
random.randint(1,10**6)/10**6:
                            state = 3
                            time_in_state = 0
                        else:
                            time_in_state += 1
                state_dist[component, state] += 1/MC_LOOPS
            reliability[component] = 1 - state_dist[component, 3]
            system_reliability *= reliability[component]
            self.time_since_maintenance[component] += 1
        if action == 1:
            reward -= INSPECTION_COST
            production_hours -= 1
```

```
            for component in range (0, COMPONENTS):
                random_number = random.randint(1,10**6)/10**6
                if state_dist[component, 3] >= random_number:
                    # System component has failed
                    reward -= COST_OF_MAINTENANCE[component,1]
                    self.time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                elif (state_dist[component, 3] + state_dist[component, 2])
>= random_number:
                    # System component is in critical state
                    reward -= COST_OF_MAINTENANCE[component,1]
                    self.time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                elif (state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]) >= random_number:
                    # System component is in degraded state
                    reward -= COST_OF_MAINTENANCE[component,0]
                    self.time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,0]


        reward += system_reliability * production_hours * POWER_CAPACITY *
CAPACITY_FACTOR * VALUE_OF_MWH
        reward /= 10**6
        self.runtime += 1
        if self.runtime >= 3000000:
            done=True
        truncated = False
        info = {}
        obs = self.time_since_maintenance
        return np.array(obs, dtype=np.float32), reward, done, truncated,
info


    def render(self):
```

```
        pass


    def reset(self, seed = None, options = None):
        super().reset(seed = seed)
        COMPONENTS = 1
        MAX = [2*52]    # Adds some variance in start age to encourage
exploration
        self.runtime = 0
        self.number_of_repairs_since_replacement =
np.zeros(COMPONENTS,dtype=np.float32)
        self.time_since_maintenance = np.zeros(COMPONENTS,dtype=np.float32)
        if options == None:
            self.time_since_maintenance[0] = random.randint(0, MAX[0])
        obs = self.time_since_maintenance
        info = {}
        return np.array(obs, dtype=np.float32), info
```

Appendix B

Optimizing the First Environment

```
import gymnasium

import os

from stable_baselines3 import PPO

from stable_baselines3.common.env_util import make_vec_env


gymnasium.register(

     id='Simple_single_component-v0',

     entry_point='1 Simple single component
environment:Simple_Single_Component',

     max_episode_steps=1000

)


envs = make_vec_env(env_id='Simple_single_component-v0', seed=1, n_envs=5)


RL_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple\Training', 'RL_Simple')

Log_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple\Training', 'Logs')


model = PPO('MlpPolicy', envs, verbose=0, tensorboard_log=Log_Path,
n_steps=1000, gamma=0.998, batch_size=5000, learning_rate=5*10**-5,
clip_range=0.1)

model.learn(total_timesteps=2000000, progress_bar=True)

model.save(RL_Path)

envs.close
```

Appendix C

Visualizing the First Environment

```
import gymnasium

import os

import math

from stable_baselines3 import PPO

import matplotlib.pyplot as plt

import numpy as np

import random


gymnasium.register(

     id='Simple_single_component-v0',

     entry_point='1 Simple single component
environment:Simple_Single_Component',

     max_episode_steps=1000

)

env = gymnasium.make('Simple_single_component-v0')

RL_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple\Training', 'RL_Simple')

model = PPO.load(RL_Path, env=env)

os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"


SIMULATIONS = 10

YEARS = 30

MC_LOOPS = 50

COMPONENTS = 1

SCALE_PARAM = np.array([

    [23.02, 2.88, 2.88]], dtype=np.float32)

SCALE_PARAM = 1/(SCALE_PARAM*52)

SHAPE_PARAM = np.array([

    [1.2, 1.2, 1.2]], dtype=np.float32)


x_axis = []

r_sys = []
```

```
rew = []


norm = []
crit = []
deg = []
fail = []


min_norm = 1
max_norm = 0
min_crit = 1
max_crit = 0
min_deg = 1
max_deg = 0
min_fail = 1
max_fail = 0


sum_action = 0
score = 0
score_sum = 0
time = 0
inspection_times = []
for simulation in range(0,SIMULATIONS):
    print('Running simulation {} of {}'.format((simulation+1),SIMULATIONS))
    obs, _ = env.reset(options=1)
    done = False
    truncated = False
    score = 0
    sum_action = 0
    time = 0
    while not done:
        action, state = model.predict(obs)
        if action == 1 and simulation == SIMULATIONS-1:
            inspection_times.append(time/52)
```

```
        sum_action += action
        obs, reward, done, truncated, info = env.step(action)
        time_since_maintenance = obs
        reliability = np.ones(COMPONENTS, dtype=np.float32)
        system_reliability = 1
        state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
                time_in_state = 0
                for t in range (0,
round(time_since_maintenance[component])+1):
                    if state == 0:
                        z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal >= random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded >= random.randint(1,10**6)/10**6:
                            state = 2
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 2:
                        z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical >= random.randint(1,10**6)/10**6:
                            state = 3
```

```
                    time_in_state = 0
                else:
                    time_in_state += 1
            state_dist[component, state] += 1/MC_LOOPS

    reliability[component] = 1 - state_dist[component, 3]
    system_reliability *= reliability[component]
score += reward
time += 1
if time>=52*YEARS:
    done=True
if simulation == SIMULATIONS-1:
    if min_fail > state_dist[0, 3]:
        min_fail = state_dist[0, 3]
    if max_fail < state_dist[0, 3]:
        max_fail = state_dist[0, 3]
    if min_crit > state_dist[0, 2]:
        min_crit = state_dist[0, 2]
    if max_crit < state_dist[0, 2]:
        max_crit = state_dist[0, 2]
    if min_deg > state_dist[0, 1]:
        min_deg = state_dist[0, 1]
    if max_deg < state_dist[0, 1]:
        max_deg = state_dist[0, 1]
    if min_norm > state_dist[0, 0]:
        min_norm = state_dist[0, 0]
    if max_norm < state_dist[0, 0]:
        max_norm = state_dist[0, 0]
    fail.append(state_dist[0, 3])
    crit.append(state_dist[0, 2])
    deg.append(state_dist[0, 1])
    norm.append(state_dist[0, 0])
    x_axis.append(time/52)
```

```
            r_sys.append(system_reliability)
    rew.append(score)
    score_sum += score
env.close


sims = []
for simulation in range(0,SIMULATIONS):
    sims.append(simulation+1)
    std = (rew[simulation]-(score_sum/SIMULATIONS))**2
std /= SIMULATIONS
std = math.sqrt(std)


print('Years: {} - Inspections: {} - Reward: {} - Avg interval: {:.2f} -
Reward per year: {:.6f} - Avg Reward: {} - Standard deviation:
{}'.format(time/52, sum_action, score, time/sum_action, score/YEARS,
score_sum/SIMULATIONS, std))


plt.figure(1)
figure, axis = plt.subplots(2, 2)
plt.subplots_adjust(left=0.12, bottom=0.11, right=0.95, top=0.95,
wspace=0.31, hspace=0.51)
axis[0,0].vlines(inspection_times,min_fail,max_fail, color = 'c',
linestyle='dashed')
axis[0,0].plot(x_axis, fail, color = 'k')
axis[0,0].set_title("Failed")
axis[0,0].set_xlabel('Time (years)')
axis[0,0].set_ylabel('Probability')


axis[0,1].vlines(inspection_times,min_crit,max_crit, color = 'c',
linestyle='dashed')
axis[0,1].plot(x_axis, crit, color = 'k')
axis[0,1].set_title("Critical")
axis[0,1].set_xlabel('Time (years)')
axis[0,1].set_ylabel('Probability')
```

```
axis[1,0].vlines(inspection_times,min_deg,max_deg, color = 'c',
linestyle='dashed')

axis[1,0].plot(x_axis, deg, color = 'k')

axis[1,0].set_title("Degraded")

axis[1,0].set_xlabel('Time (years)')

axis[1,0].set_ylabel('Probability')


axis[1,1].vlines(inspection_times,min_norm,max_norm, color = 'c',
linestyle='dashed')

axis[1,1].plot(x_axis, norm, color = 'k')

axis[1,1].set_title("Normal")

axis[1,1].set_xlabel('Time (years)')

axis[1,1].set_ylabel('Probability')


plt.figure(2)

figure, ax = plt.subplots()

ax.plot(sims, rew, color = 'r')

ax.set_title("Reward per simulation")

ax.set_xlabel('Simulation')

ax.set_ylabel('Total reward (million euros)')


plt.figure(3)

figure, ax = plt.subplots()

ax.vlines(inspection_times,0,1, color = 'c', linestyle='dashed')

ax.plot(x_axis, r_sys, color = 'r')

ax.set_title("System reliability")

ax.set_xlabel('Time (years)')

ax.set_ylabel('Reliability')

plt.show()
```

Appendix D

Finding the First Optimal Inspection Interval

```
import numpy as np

import random

import matplotlib.pyplot as plt


TIMEFRAME = 52*100

LOOPS = 5

MC_LOOPS = 5

MIN_INSPECTION_INTERVAL = 1

MAX_INSPECTION_INTERVAL = 200

INSPECTION_SPAN = MAX_INSPECTION_INTERVAL - MIN_INSPECTION_INTERVAL + 1

sum_reward = np.zeros(INSPECTION_SPAN, dtype=np.float32)


COMPONENTS = 1

DISCOUNT_FACTOR = 0.04

POWER_CAPACITY = 5

CAPACITY_FACTOR = 0.49

VALUE_OF_MWH = 40

INSPECTION_COST = 2000

CONVERSION_RATE = 1.14


# Cost of repair, cost of replacement

COST_OF_MAINTENANCE = np.array([

    [4000, 200000]], dtype=np.float32)

COST_OF_MAINTENANCE *= CONVERSION_RATE


# Duration of repair, duration of replacement

DURATION_OF_MAINTENANCE = np.array([

    [3, 70],], dtype=np.float32)


# Weibull parameters for the markov transition to degradated, critical,
failure states
```

```
SCALE_PARAM = np.array([
    [23.02, 2.88, 2.88]], dtype=np.float32)
SCALE_PARAM = 1/(SCALE_PARAM*52)
SHAPE_PARAM = np.array([
    [1.2, 1.2, 1.2]], dtype=np.float32)


for loop in range(0,LOOPS):
    for inspection_interval in range (MIN_INSPECTION_INTERVAL,
MAX_INSPECTION_INTERVAL+1):
        print('Running loop: {} of {} - Interval: {} of
{}'.format((loop+1),LOOPS,inspection_interval,MAX_INSPECTION_INTERVAL))
        time_since_maintenance = np.zeros(COMPONENTS,dtype=np.float32)
        for runtime in range (1,TIMEFRAME+1):
            if (runtime % inspection_interval) == 0:
                action = 1
            else:
                action = 0
            reward = 0
            done = False
            production_hours = 7*24
            reliability = [1] * COMPONENTS
            system_reliability = 1
            state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
            for component in range (0, COMPONENTS):
                for mc_loop in range (0, MC_LOOPS):
                    state = 0
                    time_in_state = 0
                    for t in range (0,
round(time_since_maintenance[component])+1):
                        if state == 0:
                            z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                            if z_normal >= random.randint(1,10**6)/10**6:
                                state = 1
```

```
                                time_in_state = 0
                            else:
                                time_in_state += 1
                        elif state == 1:
                            z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                            if z_degraded >= random.randint(1,10**6)/10**6:
                                state = 2
                                time_in_state = 0
                            else:
                                time_in_state += 1
                        elif state == 2:
                            z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                            if z_critical >= random.randint(1,10**6)/10**6:
                                state = 3
                                time_in_state = 0
                            else:
                                time_in_state += 1


                    state_dist[component, state] += 1/LOOPS


                reliability[component] = 1 - state_dist[component, 3]
                system_reliability *= reliability[component]


                time_since_maintenance[component] += 1
            if action == 1:
                reward -= INSPECTION_COST
                production_hours -= 1
                for component in range (0, COMPONENTS):
                    random_number = random.randint(1,10**6)/10**6
                    if state_dist[component, 3] >= random_number:
                        # System component has failed
```

```
                        reward -= COST_OF_MAINTENANCE[component,1]
                        time_since_maintenance[component] = 0
                        production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                    elif (state_dist[component, 3] + state_dist[component,
2]) >= random_number:
                        # System component is in critical state
                        reward -= COST_OF_MAINTENANCE[component,1]
                        time_since_maintenance[component] = 0
                        production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                    elif (state_dist[component, 3] + state_dist[component,
2] + state_dist[component, 1]) >= random_number:
                        # System component is in degraded state
                        reward -= COST_OF_MAINTENANCE[component,0]
                        time_since_maintenance[component] = 0
                        production_hours -=
DURATION_OF_MAINTENANCE[component,0]


            reward += system_reliability * production_hours *
POWER_CAPACITY * CAPACITY_FACTOR * VALUE_OF_MWH
            reward /= 1000000


            sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] +=
reward
sum_reward /= LOOPS
i = []
rw = []
optimal_inspection_interval = 0
rw_max = 0
rw_min = sum_reward[0]
for inspection_interval in range (MIN_INSPECTION_INTERVAL,
MAX_INSPECTION_INTERVAL+1):
    if sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] > rw_max:
        rw_max = sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL]
        optimal_inspection_interval = inspection_interval
```

```
    if sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] < rw_min:
        rw_min = sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL]
    i.append(inspection_interval)
    rw.append(sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL])


print('Optimal inspection interval: {} - Reward: {:.8f} - Reward per year:
{}'.format(optimal_inspection_interval, rw_max, rw_max*52/TIMEFRAME))


plt.plot(i, rw, color='k')
plt.title("Average Reward Per Inspection Interval over 100 years")
plt.xlabel('Inspection interval')
plt.ylabel('Average reward (million euros)')
plt.vlines(optimal_inspection_interval,rw_min,rw_max, color = 'c',
linestyle='dotted')
plt.show()
```

Appendix E

Visualizing the First Constant Inspection Interval

```
import numpy as np

import random

import matplotlib.pyplot as plt

import math


INSPECTION_INTERVAL = 15

SIMULATIONS = 10

YEARS = 30

MC_LOOPS = 50

COMPONENTS = 1

SCALE_PARAM = np.array([

    [23.02, 2.88, 2.88]], dtype=np.float32)

SCALE_PARAM = 1/(SCALE_PARAM*52)

SHAPE_PARAM = np.array([

    [1.2, 1.2, 1.2]], dtype=np.float32)


x_axis = []

r_sys = []

rew = []


norm = []

crit = []

deg = []

fail = []


min_norm = 1

max_norm = 0

min_crit = 1

max_crit = 0

min_deg = 1

max_deg = 0
```

```
min_fail = 1
max_fail = 0

sum_action = 0
score = 0
score_sum = 0
time = 0
inspection_times = []

DISCOUNT_FACTOR = 0.04
POWER_CAPACITY = 5
CAPACITY_FACTOR = 0.49
VALUE_OF_MWH = 40
INSPECTION_COST = 2000
CONVERSION_RATE = 1.14
COST_OF_MAINTENANCE = np.array([
# Cost of repair, cost of replacement
    [4000, 200000]], dtype=np.float32)
COST_OF_MAINTENANCE *= CONVERSION_RATE
DURATION_OF_MAINTENANCE = np.array([
# Duration of repair, duration of replacement
    [3, 70]], dtype=np.float32)

for simulation in range(0,SIMULATIONS):
    print('Running simulation {} of {}'.format((simulation+1),SIMULATIONS))
    time_since_maintenance = np.zeros(COMPONENTS,dtype=np.float32)
    done = False
    truncated = False
    score = 0
    sum_action = 0
    time = 0
    time_since_maintenance = np.zeros(COMPONENTS,dtype=np.float32)
    while not done:
```

```
        if (time % INSPECTION_INTERVAL) == 0:
            action = 1
            inspection_times.append(time/52)
            sum_action += 1
            if simulation == SIMULATIONS-1:
                inspection_times.append(time/52)
        else:
            action = 0
        reward = 0
        production_hours = 7*24
        reliability = np.ones(COMPONENTS, dtype=np.float32)
        system_reliability = 1
        state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
                time_in_state = 0
                for t in range (0,
round(time_since_maintenance[component])+1):
                    if state == 0:
                        z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal >= random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded >= random.randint(1,10**6)/10**6:
                            state = 2
                            time_in_state = 0
```

```
                        else:
                            time_in_state += 1
                    elif state == 2:
                        z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical >= random.randint(1,10**6)/10**6:
                            state = 3
                            time_in_state = 0
                        else:
                            time_in_state += 1
                state_dist[component, state] += 1/MC_LOOPS


            reliability[component] = 1 - state_dist[component, 3]
            system_reliability *= reliability[component]
            time_since_maintenance[component] += 1
        if action == 1:
            reward -= INSPECTION_COST
            production_hours -= 1
            for component in range (0, COMPONENTS):
                random_number = random.randint(1,10**6)/10**6
                if state_dist[component, 3] >= random_number:
                    # System component has failed
                    reward -= COST_OF_MAINTENANCE[component,1]
                    time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                elif (state_dist[component, 3] + state_dist[component, 2])
>= random_number:
                    # System component is in critical state
                    reward -= COST_OF_MAINTENANCE[component,1]
                    time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,1]
                elif (state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]) >= random_number:
```

```
                    # System component is in degraded state
                    reward -= COST_OF_MAINTENANCE[component,0]
                    time_since_maintenance[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,0]


        reward += system_reliability * production_hours * POWER_CAPACITY *
CAPACITY_FACTOR * VALUE_OF_MWH
        reward /= 1000000
        score += reward
        time += 1
        if time>=52*YEARS:
            done=True
        if simulation == SIMULATIONS-1:
            if min_fail > state_dist[0, 3]:
                min_fail = state_dist[0, 3]
            if max_fail < state_dist[0, 3]:
                max_fail = state_dist[0, 3]
            if min_crit > state_dist[0, 2]:
                min_crit = state_dist[0, 2]
            if max_crit < state_dist[0, 2]:
                max_crit = state_dist[0, 2]
            if min_deg > state_dist[0, 1]:
                min_deg = state_dist[0, 1]
            if max_deg < state_dist[0, 1]:
                max_deg = state_dist[0, 1]
            if min_norm > state_dist[0, 0]:
                min_norm = state_dist[0, 0]
            if max_norm < state_dist[0, 0]:
                max_norm = state_dist[0, 0]
            fail.append(state_dist[0, 3])
            crit.append(state_dist[0, 2])
            deg.append(state_dist[0, 1])
            norm.append(state_dist[0, 0])
```

```
            x_axis.append(time/52)
            r_sys.append(system_reliability)
    rew.append(score)
    score_sum += score


sims = []
for simulation in range(0,SIMULATIONS):
    sims.append(simulation+1)
    std = (rew[simulation]-(score_sum/SIMULATIONS))**2
std /= SIMULATIONS
std = math.sqrt(std)


print('Years: {} - Inspections: {} - Reward: {} - Avg interval: {:.2f} -
Reward per year: {:.6f} - Avg Reward: {} - Standard deviation:
{}'.format(time/52, sum_action, score, time/sum_action, score/YEARS,
score_sum/SIMULATIONS, std))


plt.figure(1)
figure, axis = plt.subplots(2, 2)
plt.subplots_adjust(left=0.12, bottom=0.11, right=0.95, top=0.95,
wspace=0.31, hspace=0.51)
axis[0,0].vlines(inspection_times,min_fail,max_fail, color = 'c',
linestyle='dashed')
axis[0,0].plot(x_axis, fail, color = 'k')
axis[0,0].set_title("Failed")
axis[0,0].set_xlabel('Time (years)')
axis[0,0].set_ylabel('Probability')


axis[0,1].vlines(inspection_times,min_crit,max_crit, color = 'c',
linestyle='dashed')
axis[0,1].plot(x_axis, crit, color = 'k')
axis[0,1].set_title("Critical")
axis[0,1].set_xlabel('Time (years)')
axis[0,1].set_ylabel('Probability')
```

```
axis[1,0].vlines(inspection_times,min_deg,max_deg, color = 'c',
linestyle='dashed')

axis[1,0].plot(x_axis, deg, color = 'k')

axis[1,0].set_title("Degraded")

axis[1,0].set_xlabel('Time (years)')

axis[1,0].set_ylabel('Probability')


axis[1,1].vlines(inspection_times,min_norm,max_norm, color = 'c',
linestyle='dashed')

axis[1,1].plot(x_axis, norm, color = 'k')

axis[1,1].set_title("Normal")

axis[1,1].set_xlabel('Time (years)')

axis[1,1].set_ylabel('Probability')


plt.figure(2)

figure, ax = plt.subplots()

ax.plot(sims, rew, color = 'r')

ax.set_title("Reward per simulation")

ax.set_xlabel('Simulation')

ax.set_ylabel('Total reward (million euros)')


plt.figure(3)

figure, ax = plt.subplots()

ax.vlines(inspection_times,0,1, color = 'c', linestyle='dashed')

ax.plot(x_axis, r_sys, color = 'r')

ax.set_title("System reliability")

ax.set_xlabel('Time (years)')

ax.set_ylabel('Reliability')

plt.show()
```

Appendix F

The Second Environment

```
import numpy as np

import random

import gymnasium

from gymnasium import spaces


class Inspection_Single_Component(gymnasium.Env):

    metadata = {"render_modes": ["rgb_array"], "render_fps": 4}


    def __init__(self):


        # Action space: (0) no action, (1) inspection

        self.action_space = spaces.Discrete(2)


        # Observation space:

        # | Num | Description                                                |

        # |-----|------------------------------------------------------------|

        # | 0   | Time since inspection (weeks)                              |

        # | 1   | Blade - Component age after last inspection (weeks)        |


        COMPONENTS = 1

        low = np.zeros((COMPONENTS+1),dtype=np.float32)

        high = np.ones((COMPONENTS+1),dtype=np.float32)

        high *= 10**6

        self.observation_space = spaces.Box(low, high, dtype=np.float32)


        self.time_since_inspection = None

        self.component_age_after_last_inspection = None


    def step(self, action):

        COMPONENTS = 1

        MC_LOOPS = 50
```

```
        POWER_CAPACITY = 5
        CAPACITY_FACTOR = 0.49
        VALUE_OF_MWH = 40
        INSPECTION_COST = 2000
        CONVERSION_RATE = 1.14

        COST_OF_MAINTENANCE = np.array([
            # Cost of repair, cost of replacement
            [4000, 200000]], dtype=np.float32)
        COST_OF_MAINTENANCE *= CONVERSION_RATE

        DURATION_OF_MAINTENANCE = np.array([
            # Duration of repair, duration of replacement
            [3, 70]], dtype=np.float32)

        SCALE_PARAM = np.array([
            # Weibull parameters for the markov transition to degradated,
critical, failure states
            [23.02, 2.88, 2.88]], dtype=np.float32)
        SCALE_PARAM = 1/(SCALE_PARAM*52)
        SHAPE_PARAM = np.array([
            [1.2, 1.2, 1.2]], dtype=np.float32)

        reward = 0
        done = False

        production_hours = 7*24.0
        reliability = [1] * COMPONENTS
        system_reliability = 1
        state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
```

```
                time_in_state =
round(self.component_age_after_last_inspection[component])
                for t in range (0, round(self.time_since_inspection)+1):
                    if state == 0:
                        z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal >= random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded >= random.randint(1,10**6)/10**6:
                            state = 2
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 2:
                        z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical >= random.randint(1,10**6)/10**6:
                            state = 3
                            time_in_state = 0
                        else:
                            time_in_state += 1
                state_dist[component, state] += 1/MC_LOOPS
            reliability[component] = 1 - state_dist[component, 3]
            system_reliability *= reliability[component]
        self.time_since_inspection += 1
        if action == 1:
            reward -= INSPECTION_COST
```

```
            production_hours -= 1

            for component in range (0, COMPONENTS):

                # if (state_dist[component, 3] + state_dist[component, 2])
>= random.randint(1,10**6)/10**6:

                #     # System component is in critical or failed state

                #     reward -= COST_OF_MAINTENANCE[component,1]

                #     production_hours -=
DURATION_OF_MAINTENANCE[component,1]

                #     self.component_age_after_last_inspection[component] =
0

                if (state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]) >= random.randint(1,10**6)/10**6:

                    # System component is in degraded, critical or failed
state

                    reward -= COST_OF_MAINTENANCE[component,0] *
(state_dist[component,1]/(state_dist[component,1]+state_dist[component,2]+s
tate_dist[component,3]))

                    reward -= COST_OF_MAINTENANCE[component,1] *
((state_dist[component,2]+state_dist[component,3])/(state_dist[component,1]
+state_dist[component,2]+state_dist[component,3]))

                    production_hours -=
DURATION_OF_MAINTENANCE[component,0] *
(state_dist[component,1]/(state_dist[component,1]+state_dist[component,2]+s
tate_dist[component,3]))

                    production_hours -=
DURATION_OF_MAINTENANCE[component,1] *
((state_dist[component,2]+state_dist[component,3])/(state_dist[component,1]
+state_dist[component,2]+state_dist[component,3]))

                    self.component_age_after_last_inspection[component] = 0

                else:

                    self.component_age_after_last_inspection[component] +=
self.time_since_inspection

            self.time_since_inspection = 0


        reward += system_reliability * production_hours * POWER_CAPACITY *
CAPACITY_FACTOR * VALUE_OF_MWH

        reward /= 10**6

        self.runtime += 1

        if self.runtime >= 10000000:

            done=True
```

```
        truncated = False
        info = {}
        obs = np.zeros(COMPONENTS+1,dtype=np.float32)
        obs[0] = self.time_since_inspection
        for component in range (0, COMPONENTS):
            obs[component+1] =
self.component_age_after_last_inspection[component]
        return np.array(obs, dtype=np.float32), reward, done, truncated,
info


    def render(self):
        pass


    def reset(self, seed = None, options = None):
        super().reset(seed = seed)
        COMPONENTS = 1
        # MAX = [2*52]
        self.runtime = 0
        self.component_age_after_last_inspection =
np.zeros(COMPONENTS,dtype=np.float32)
        self.time_since_inspection = 0
        obs = np.zeros(COMPONENTS+1,dtype=np.float32)
        obs[0] = self.time_since_inspection
        for component in range (0, COMPONENTS):
            obs[component+1] =
self.component_age_after_last_inspection[component]
        info = {}
        return np.array(obs, dtype=np.float32), info
```

Appendix G

Optimizing the Second Environment

```
import gymnasium

import os

from stable_baselines3 import PPO

from stable_baselines3.common.env_util import make_vec_env


gymnasium.register(

     id='Inspection_single_component-v0',

     entry_point='1 Inspection single component
environment:Inspection_Single_Component',

     max_episode_steps=5000

)


envs = make_vec_env(env_id='Inspection_single_component-v0', seed=1,
n_envs=5)


RL_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple inspection\Training', 'RL_Inspection')

Log_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple inspection\Training', 'Logs')


model = PPO('MlpPolicy', envs, verbose=0, tensorboard_log=Log_Path,
gamma=0.995, clip_range=0.1)

model.learn(total_timesteps=6000000, progress_bar=True)

model.save(RL_Path)

envs.close
```

Appendix H

Visualizing the Second Environment

```
import gymnasium

import os

import math

from stable_baselines3 import PPO

import matplotlib.pyplot as plt

import numpy as np

import random


gymnasium.register(

     id='Inspection_single_component-v0',

     entry_point='1 Inspection single component
environment:Inspection_Single_Component',

     max_episode_steps=5000

)

env = gymnasium.make('Inspection_single_component-v0')

RL_Path = os.path.join(r'C:\Users\PC\Desktop\Reinforcement
Learning\Windmill problem\Simple inspection\Training', 'RL_Inspection')

model = PPO.load(RL_Path, env=env)

os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"


SIMULATIONS = 1

YEARS = 30

MC_LOOPS = 5

COMPONENTS = 1

SCALE_PARAM = np.array([

    [23.02, 2.88, 2.88]], dtype=np.float32)

SCALE_PARAM = 1/(SCALE_PARAM*52)

SHAPE_PARAM = np.array([

    [1.2, 1.2, 1.2]], dtype=np.float32)

x_axis = []

r_sys = []

rew = []
```

```
norm = []
deg = []
crit = []
fail = []

min_norm = 1
max_norm = 0
min_crit = 1
max_crit = 0
min_deg = 1
max_deg = 0
min_fail = 1
max_fail = 0

score = 0
sum_action = 0
sum_score = 0
time = 0
inspection_times = []
for simulation in range(0, SIMULATIONS):
    print('Running simulation {} of {}'.format((simulation+1),SIMULATIONS))
    obs, _ = env.reset(options=1)
    done = False
    truncated = False
    score = 0
    sum_action = 0
    time = 0
    component_age_after_last_inspection = np.zeros(COMPONENTS,
dtype=np.float32)
    time_since_inspection = 0
    while not done:
        action, state = model.predict(obs)
        if action == 1 and simulation == SIMULATIONS-1:
            inspection_times.append(time/52)
```

```
        sum_action += action
        obs, reward, done, truncated, info = env.step(action)
        time_since_inspection = obs[0]
        for component in range(0,COMPONENTS):
            component_age_after_last_inspection[component] =
obs[component+1]
        reliability1 = np.ones(COMPONENTS, dtype=np.float32)
        system_reliability1 = 1
        state_dist1 = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
                time_in_state =
round(component_age_after_last_inspection[component])
                for t in range (0, round(time_since_inspection)+1):
                    if state == 0:
                        z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal >= random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded >= random.randint(1,10**6)/10**6:
                            state = 2
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 2:
```

```
                        z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical >= random.randint(1,10**6)/10**6:
                            state = 3
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    state_dist1[component, state] += 1/MC_LOOPS


            reliability1[component] = 1 - state_dist1[component, 3]
            system_reliability1 *= reliability1[component]
        score += reward
        time += 1
        if time>=52*YEARS:
            done=True
        if simulation == SIMULATIONS-1:
            if min_fail > state_dist1[0, 3]:
                min_fail = state_dist1[0, 3]
            if max_fail < state_dist1[0, 3]:
                max_fail = state_dist1[0, 3]
            if min_crit > state_dist1[0, 2]:
                min_crit = state_dist1[0, 2]
            if max_crit < state_dist1[0, 2]:
                max_crit = state_dist1[0, 2]
            if min_deg > state_dist1[0, 1]:
                min_deg = state_dist1[0, 1]
            if max_deg < state_dist1[0, 1]:
                max_deg = state_dist1[0, 1]
            if min_norm > state_dist1[0, 0]:
                min_norm = state_dist1[0, 0]
            if max_norm < state_dist1[0, 0]:
                max_norm = state_dist1[0, 0]
            fail.append(state_dist1[0, 3])
```

```
            crit.append(state_dist1[0, 2])
            deg.append(state_dist1[0, 1])
            norm.append(state_dist1[0, 0])
            x_axis.append(time/52)
            r_sys.append(system_reliability1)
    rew.append(score)
    sum_score += score
env.close


sims = []
for simulation in range(0,SIMULATIONS):
    sims.append(simulation+1)
    std = (rew[simulation]-(sum_score/SIMULATIONS))**2
std /= SIMULATIONS
std = math.sqrt(std)


print('Years: {} - Inspections: {} - Reward: {} - Avg interval: {:.2f} -
Reward per year: {:.6f} - Avg Reward: {} - Standard deviation:
{}'.format(time/52, sum_action, score, time/sum_action, score/YEARS,
sum_score/SIMULATIONS, std))


plt.figure(1)
figure, axis = plt.subplots(2, 2)
plt.subplots_adjust(left=0.12, bottom=0.11, right=0.95, top=0.95,
wspace=0.31, hspace=0.51)
axis[0,0].vlines(inspection_times,min_fail,max_fail, color = 'c',
linestyle='dashed')
axis[0,0].plot(x_axis, fail, color = 'k')
axis[0,0].set_title("Failed")
axis[0,0].set_xlabel('Time (years)')
axis[0,0].set_ylabel('Probability')


axis[0,1].vlines(inspection_times,min_crit,max_crit, color = 'c',
linestyle='dashed')
axis[0,1].plot(x_axis, crit, color = 'k')
axis[0,1].set_title("Critical")
```

```
axis[0,1].set_xlabel('Time (years)')

axis[0,1].set_ylabel('Probability')


axis[1,0].vlines(inspection_times,min_deg,max_deg, color = 'c',
linestyle='dashed')

axis[1,0].plot(x_axis, deg, color = 'k')

axis[1,0].set_title("Degraded")

axis[1,0].set_xlabel('Time (years)')

axis[1,0].set_ylabel('Probability')


axis[1,1].vlines(inspection_times,min_norm,max_norm, color = 'c',
linestyle='dashed')

axis[1,1].plot(x_axis, norm, color = 'k')

axis[1,1].set_title("Normal")

axis[1,1].set_xlabel('Time (years)')

axis[1,1].set_ylabel('Probability')


plt.figure(2)

figure, ax = plt.subplots()

ax.plot(sims, rew, color = 'r')

ax.set_title("Reward per simulation")

ax.set_xlabel('Simulation')

ax.set_ylabel('Total reward (million euros)')


plt.figure(3)

figure, ax = plt.subplots()

ax.vlines(inspection_times,0,1, color = 'c', linestyle='dashed')

ax.plot(x_axis, r_sys, color = 'r')

ax.set_title("System reliability")

ax.set_xlabel('Time (years)')

ax.set_ylabel('Reliability')

plt.show()
```

Appendix I

Finding the Second Optimal Inspection Interval

```
import numpy as np

import random

import matplotlib.pyplot as plt


TIME_FRAME = 52*100

LOOPS = 15

MC_LOOPS = 15

MIN_INSPECTION_INTERVAL = 40

MAX_INSPECTION_INTERVAL = 120

INSPECTION_SPAN = MAX_INSPECTION_INTERVAL - MIN_INSPECTION_INTERVAL + 1

sum_reward = np.zeros(INSPECTION_SPAN, dtype=np.float32)

COMPONENTS = 1

DISCOUNT_FACTOR = 0.04

POWER_CAPACITY = 5

CAPACITY_FACTOR = 0.49

VALUE_OF_MWH = 40

INSPECTION_COST = 2000

CONVERSION_RATE = 1.14

COST_OF_MAINTENANCE = np.array([

    # Cost of repair, cost of replacement

    [4000, 200000]], dtype=np.float32)

COST_OF_MAINTENANCE *= CONVERSION_RATE

DURATION_OF_MAINTENANCE = np.array([

    # Duration of repair, duration of replacement

    [3, 70],], dtype=np.float32)

SCALE_PARAM = np.array([

    # Weibull parameters for the markov transition to degradated, critical,
failure states

    [23.02, 2.88, 2.88]], dtype=np.float32)

SCALE_PARAM = 1/(SCALE_PARAM*52)

SHAPE_PARAM = np.array([
```

```
    [1.2, 1.2, 1.2]], dtype=np.float32)


for loop in range(0,LOOPS):

    for inspection_interval in range (MIN_INSPECTION_INTERVAL,
MAX_INSPECTION_INTERVAL+1):

        print('Running loop: {} of {} - Interval: {} of
{}'.format((loop+1),LOOPS,inspection_interval,MAX_INSPECTION_INTERVAL))

        component_age_after_last_inspection =
np.zeros(COMPONENTS,dtype=np.float32)

        time_since_inspection = 0

        for runtime in range (1,TIME_FRAME+1):

            if (runtime % inspection_interval) == 0:

                action = 1

            else:

                action = 0

            reward = 0

            done = False

            production_hours = 7*24

            reliability = [1] * COMPONENTS

            system_reliability = 1

            state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)

            for component in range (0, COMPONENTS):

                for mc_loop in range (0, MC_LOOPS):

                    state = 0

                    time_in_state =
round(component_age_after_last_inspection[component])

                    for t in range (0, round(time_since_inspection)+1):

                        if state == 0:

                            z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))

                            if z_normal >= random.randint(1,10**6)/10**6:

                                state = 1

                                time_in_state = 0

                            else:

                                time_in_state += 1
```

```
                        elif state == 1:
                            z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                            if z_degraded >= random.randint(1,10**6)/10**6:
                                state = 2
                                time_in_state = 0
                            else:
                                time_in_state += 1
                        elif state == 2:
                            z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                            if z_critical >= random.randint(1,10**6)/10**6:
                                state = 3
                                time_in_state = 0
                            else:
                                time_in_state += 1
                    state_dist[component, state] += 1/LOOPS
                reliability[component] = 1 - state_dist[component, 3]
                system_reliability *= reliability[component]
            time_since_inspection += 1
            if action == 1:
                reward -= INSPECTION_COST
                production_hours -= 1
                for component in range (0, COMPONENTS):
                    random_number = random.randint(1,10**6)/10**6
                    if (state_dist[component, 3] + state_dist[component, 2]
+ state_dist[component, 1]) >= random_number:
                        # System component is in degraded, critical or
failed state
                        reward -= COST_OF_MAINTENANCE[component,0] *
(state_dist[component, 1]/(state_dist[component, 3] + state_dist[component,
2] + state_dist[component, 1]))
                        reward -= COST_OF_MAINTENANCE[component,1] *
((state_dist[component, 2]+state_dist[component, 3])/(state_dist[component,
3] + state_dist[component, 2] + state_dist[component, 1]))
```

```
                        component_age_after_last_inspection[component] = 0

                        production_hours -=
DURATION_OF_MAINTENANCE[component,0] * (state_dist[component,
1]/(state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]))

                        production_hours -=
DURATION_OF_MAINTENANCE[component,1] * ((state_dist[component,
2]+state_dist[component, 3])/(state_dist[component, 3] +
state_dist[component, 2] + state_dist[component, 1]))

                    else:

                        component_age_after_last_inspection[component] +=
time_since_inspection

                time_since_inspection = 0

            reward += system_reliability * production_hours *
POWER_CAPACITY * CAPACITY_FACTOR * VALUE_OF_MWH

            reward /= 10**6

            sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] +=
reward

sum_reward /= LOOPS

i = []

rw = []

optimal_inspection_interval = 0

rw_max = 0

rw_min = sum_reward[0]

for inspection_interval in range (MIN_INSPECTION_INTERVAL,
MAX_INSPECTION_INTERVAL+1):

    if sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] > rw_max:

        rw_max = sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL]

        optimal_inspection_interval = inspection_interval

    if sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL] < rw_min:

        rw_min = sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL]

    i.append(inspection_interval)

    rw.append(sum_reward[inspection_interval-MIN_INSPECTION_INTERVAL])

print('Optimal inspection interval: {} - Reward: {:.8f} - Reward per year:
{}'.format(optimal_inspection_interval, rw_max, rw_max*52/TIME_FRAME))

plt.plot(i, rw)

plt.xlabel("Inspection interval (weeks)")
```

```
plt.ylabel("Average reward in million Euro")

plt.vlines(optimal_inspection_interval,rw_min,rw_max, color = 'c',
linestyle='dotted')

plt.show()
```

Appendix J

Visualizing the Second Optimal Inspection Interval

```
import numpy as np

import random

import matplotlib.pyplot as plt

import math


SIMULATIONS = 10

TIMEFRAME = 52*30

MC_LOOPS = 50

INSPECTION_INTERVAL = 59

COMPONENTS = 1

DISCOUNT_FACTOR = 0.04

POWER_CAPACITY = 5

CAPACITY_FACTOR = 0.49

VALUE_OF_MWH = 40

INSPECTION_COST = 2000

CONVERSION_RATE = 1.14


COST_OF_MAINTENANCE = np.array([

    # Cost of repair, cost of replacement

    [4000, 200000]], dtype=np.float32)

COST_OF_MAINTENANCE *= CONVERSION_RATE


DURATION_OF_MAINTENANCE = np.array([

    # Duration of repair, duration of replacement

    [3, 70],], dtype=np.float32)


SCALE_PARAM = np.array([

    # Weibull parameters for the markov transition to degradated, critical,
failure states

    [23.02, 2.88, 2.88]], dtype=np.float32)

SCALE_PARAM = 1/(SCALE_PARAM*52)
```

```
SHAPE_PARAM = np.array([
    [1.2, 1.2, 1.2]], dtype=np.float32)


x_axis = []
r_sys = []
rew = []
norm = []
crit = []
deg = []
fail = []
min_norm = 1
max_norm = 0
min_crit = 1
max_crit = 0
min_deg = 1
max_deg = 0
min_fail = 1
max_fail = 0
sum_action = 0
score = 0
score_sum = 0
time = 0
inspection_times = []
for simulation in range(0,SIMULATIONS):
    print('Running simulation {} of {}'.format((simulation+1),SIMULATIONS))
    time_since_inspection = 0
    component_age_after_last_inspection =
np.zeros(COMPONENTS,dtype=np.float32)
    done = False
    truncated = False
    score = 0
    sum_action = 0
    time = 0
    while not done:
```

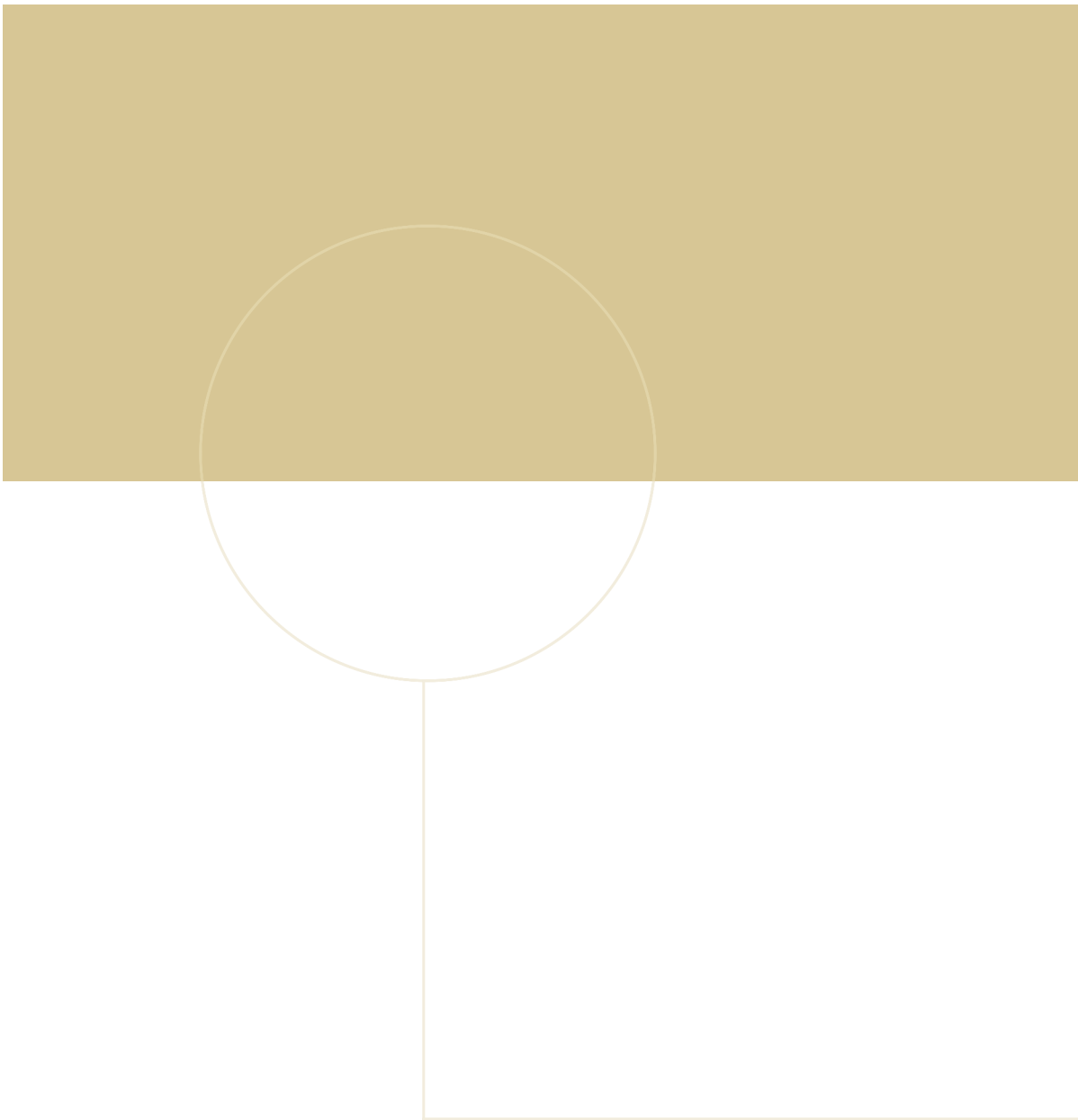
```
        if (time % INSPECTION_INTERVAL) == 0:
            action = 1
            inspection_times.append(time/52)
            sum_action += 1
            if simulation == SIMULATIONS-1:
                inspection_times.append(time/52)
        else:
            action = 0


        reward = 0
        reliability = np.ones(COMPONENTS, dtype=np.float32)
        production_hours = 7*24.0
        system_reliability = 1
        state_dist = np.zeros((COMPONENTS, 4), dtype=np.float32)
        for component in range (0, COMPONENTS):
            for mc_loop in range (0, MC_LOOPS):
                state = 0
                time_in_state =
round(component_age_after_last_inspection[component])
                for t in range (0, round(time_since_inspection)+1):
                    if state == 0:
                        z_normal =
SHAPE_PARAM[component,0]*SCALE_PARAM[component,0]*((SCALE_PARAM[component,0
]*time_in_state)**(SHAPE_PARAM[component,0]-1))
                        if z_normal >= random.randint(1,10**6)/10**6:
                            state = 1
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 1:
                        z_degraded =
SHAPE_PARAM[component,1]*SCALE_PARAM[component,1]*((SCALE_PARAM[component,1
]*time_in_state)**(SHAPE_PARAM[component,1]-1))
                        if z_degraded >= random.randint(1,10**6)/10**6:
                            state = 2
```

```
                            time_in_state = 0
                        else:
                            time_in_state += 1
                    elif state == 2:
                        z_critical =
SHAPE_PARAM[component,2]*SCALE_PARAM[component,2]*((SCALE_PARAM[component,2
]*time_in_state)**(SHAPE_PARAM[component,2]-1))
                        if z_critical >= random.randint(1,10**6)/10**6:
                            state = 3
                            time_in_state = 0
                        else:
                            time_in_state += 1
                state_dist[component, state] += 1/MC_LOOPS
            reliability[component] = 1 - state_dist[component, 3]
            system_reliability *= reliability[component]
        time_since_inspection += 1
        if action == 1:
            reward -= INSPECTION_COST
            production_hours -= 1
            for component in range (0, COMPONENTS):
                random_number = random.randint(1,10**6)/10**6
                if (state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]) >= random_number:
                    # System component is in degraded, critical or failed
state
                    reward -= COST_OF_MAINTENANCE[component,0] *
(state_dist[component, 1]/(state_dist[component, 3] + state_dist[component,
2] + state_dist[component, 1]))
                    reward -= COST_OF_MAINTENANCE[component,1] *
((state_dist[component, 2]+state_dist[component, 3])/(state_dist[component,
3] + state_dist[component, 2] + state_dist[component, 1]))
                    component_age_after_last_inspection[component] = 0
                    production_hours -=
DURATION_OF_MAINTENANCE[component,0] * (state_dist[component,
1]/(state_dist[component, 3] + state_dist[component, 2] +
state_dist[component, 1]))
                    production_hours -=
DURATION_OF_MAINTENANCE[component,1] * ((state_dist[component,
```

```
2]+state_dist[component, 3])/(state_dist[component, 3] +
state_dist[component, 2] + state_dist[component, 1]))

                else:

                    component_age_after_last_inspection[component] +=
time_since_inspection

            time_since_inspection = 0


        reward += system_reliability * production_hours * POWER_CAPACITY *
CAPACITY_FACTOR * VALUE_OF_MWH

        reward /= 10**6

        score += reward

        time += 1

        if time>=TIMEFRAME:

            done=True

        if simulation == SIMULATIONS-1:

            if min_fail > state_dist[0, 3]:

                min_fail = state_dist[0, 3]

            if max_fail < state_dist[0, 3]:

                max_fail = state_dist[0, 3]

            if min_crit > state_dist[0, 2]:

                min_crit = state_dist[0, 2]

            if max_crit < state_dist[0, 2]:

                max_crit = state_dist[0, 2]

            if min_deg > state_dist[0, 1]:

                min_deg = state_dist[0, 1]

            if max_deg < state_dist[0, 1]:

                max_deg = state_dist[0, 1]

            if min_norm > state_dist[0, 0]:

                min_norm = state_dist[0, 0]

            if max_norm < state_dist[0, 0]:

                max_norm = state_dist[0, 0]

            fail.append(state_dist[0, 3])

            crit.append(state_dist[0, 2])

            deg.append(state_dist[0, 1])
```

```
            norm.append(state_dist[0, 0])
            x_axis.append(time/52)
            r_sys.append(system_reliability)
    rew.append(score)
    score_sum += score

sims = []
for simulation in range(0,SIMULATIONS):
    sims.append(simulation+1)
    std = (rew[simulation]-(score_sum/SIMULATIONS))**2
std /= SIMULATIONS
std = math.sqrt(std)

print('Years: {} - Inspections: {} - Reward: {} - Avg interval: {:.2f} -
Reward per year: {:.6f} - Avg Reward: {} - Standard deviation:
{}'.format(time/52, sum_action, score, time/sum_action, score*52/TIMEFRAME,
score_sum/SIMULATIONS, std))

plt.figure(1)
figure, axis = plt.subplots(2, 2)
plt.subplots_adjust(left=0.12, bottom=0.11, right=0.95, top=0.95,
wspace=0.31, hspace=0.51)
axis[0,0].vlines(inspection_times,min_fail,max_fail, color = 'c',
linestyle='dashed')
axis[0,0].plot(x_axis, fail, color = 'k')
axis[0,0].set_title("Failed")
axis[0,0].set_xlabel('Time (years)')
axis[0,0].set_ylabel('Probability')

axis[0,1].vlines(inspection_times,min_crit,max_crit, color = 'c',
linestyle='dashed')
axis[0,1].plot(x_axis, crit, color = 'k')
axis[0,1].set_title("Critical")
axis[0,1].set_xlabel('Time (years)')
axis[0,1].set_ylabel('Probability')
```

```
axis[1,0].vlines(inspection_times,min_deg,max_deg, color = 'c',
linestyle='dashed')

axis[1,0].plot(x_axis, deg, color = 'k')

axis[1,0].set_title("Degraded")

axis[1,0].set_xlabel('Time (years)')

axis[1,0].set_ylabel('Probability')


axis[1,1].vlines(inspection_times,min_norm,max_norm, color = 'c',
linestyle='dashed')

axis[1,1].plot(x_axis, norm, color = 'k')

axis[1,1].set_title("Normal")

axis[1,1].set_xlabel('Time (years)')

axis[1,1].set_ylabel('Probability')


plt.figure(2)

figure, ax = plt.subplots()

ax.plot(sims, rew, color = 'r')

ax.set_title("Reward per simulation")

ax.set_xlabel('Simulation')

ax.set_ylabel('Total reward (million euros)')


plt.figure(3)

figure, ax = plt.subplots()

ax.vlines(inspection_times,0,1, color = 'c', linestyle='dashed')

ax.plot(x_axis, r_sys, color = 'r')

ax.set_title("System reliability")

ax.set_xlabel('Time (years)')

ax.set_ylabel('Reliability')

plt.show()
```



NTNU

Norwegian University of
Science and Technology