STK3405 – Lecture 17

A. B. Huseby

Department of Mathematics
University of Oslo, Norway



The Barlow-Proschan importance measure for repairable systems



The Barlow-Proschan importance measure

Consider a repairable binary monotone system (C, ϕ). For all $i \in C$ we let:

$$U_{ij} = \text{The } j\text{th lifetime of the } i\text{th component, } j = 1, 2, \ldots$$

$$D_{ij} = \text{The } j\text{th repair time of the } i\text{th component, } j = 1, 2, \ldots$$

For all $i \in C$ we assume that:

$$U_{i1}, U_{i2}, \ldots \text{ are identically distributed with mean value } 0 < \mu_i < \infty$$

$$D_{i1}, D_{i2}, \ldots \text{ are identically distributed with mean value } 0 < \nu_i < \infty$$

All lifetimes and repair times are assumed to be independent random variables.

For all $i \in C$ we also let T_{ij} denote the point of time of the jth event affecting component i. Thus, the points of time when component i fails are $T_{i1}, T_{i3}, T_{i5}, \ldots$, where:

$$T_{i,(2m+1)} = U_{i1} + \sum_{j=1}^{m} [D_{ij} + U_{i,(j+1)}], \quad m = 0, 1, 2, \ldots$$



The Barlow-Proschan importance measure (cont.)

MAIN IDEA: A component is said to cause a system failure, if it fails at the same time as the system. Thus, the most important component is the one that causes most system failures.

$I_{B-P}^{(i)}$ is the asymptotic fraction of system failures caused by component i.

We then let:

$$N_i(t) = \text{The number of times component } i \text{ fails in } [0, t], \quad i \in C.$$

The first failure of component i occurs at time U_{i1}, the second failure occurs at time $U_{i1} + D_{i1} + U_{i2}$, and so on. The expected time between the jth and the $(j+1)$th failure is:

$$E[D_{ij} + U_{i,(j+1)}] = \nu_i + \mu_i, \quad j = 1, 2, \ldots, \; i \in C.$$

By the strong law of large numbers it follows that with probability one:

$$\lim_{m\to\infty} \frac{1}{m} \sum_{j=1}^{m} [D_{ij} + U_{i,(j+1)}] = \nu_i + \mu_i, \quad i \in C.$$



The Barlow-Proschan importance measure (cont.)

We then note that:

$$N(T_{i,(2m+1)}) = N_i(U_{i1} + \sum_{j=1}^{m}[D_{ij} + U_{i,(j+1)}]) = m+1, \quad m = 1, 2, \ldots$$

Hence, with probability one:

$$\frac{N(T_{i,(2m+1)})}{T_{i,(2m+1)}} = \frac{N_i(U_{i1} + \sum_{j=1}^{m}[D_{ij} + U_{i,(j+1)}])}{U_{i1} + \sum_{j=1}^{m}[D_{ij} + U_{i,(j+1)}])}$$

$$= \frac{m+1}{U_{i1} + \sum_{j=1}^{m}[D_{ij} + U_{i,(j+1)}])}$$

$$= \frac{\frac{1}{m}+1}{\frac{1}{m}U_{i1} + \frac{1}{m}\sum_{j=1}^{m}[D_{ij} + U_{i,(j+1)}])} \to \frac{1}{\nu_i + \mu_i}$$

when m tends to infinity.



The Barlow-Proschan importance measure (cont.)

More generally, it can be shown that:

$$\lim_{t\to\infty} \frac{N_i(t)}{t} = \frac{1}{\nu_i + \mu_i}, \quad i \in C.$$

with probability one.

If component i fails at time t, this only causes a system failure if i is critical at time t.

In order to take this into account we let:

$$\tilde{N}_i(t) = \text{The number of times a failure in component } i \text{ causes system failure in } [0, t], \quad i \in C.$$



The Barlow-Proschan importance measure (cont.)

The probability that i is critical at time t is $I_B^{(i)}(t)$.

When t is large, the fraction of component failures that result in system failures in the interval $[0, t]$ is approximately $I_B^{(i)}(t)$, Thus, we have:

$$\tilde{N}_i(t) \approx I_B^{(i)}(t) \cdot N_i(t), \quad i \in C.$$

Using this, it can be shown that:

$$\lim_{t\to\infty} \frac{\tilde{N}_i(t)}{t} = \lim_{t\to\infty} I_B^{(i)}(t) \cdot \frac{N_i(t)}{t} = \frac{I_B^{(i)}}{\nu_i + \mu_i}, \quad i \in C.$$

with probability one.



The Barlow-Proschan importance measure (cont.)

We then let:

$$N_\phi(t) = \text{The number of times the system fails in } [0, t]$$

Since the probability that two or more components fail at the same time is zero, each system failure is almost surely caused by the failure of a single component.

Thus, it follows that:

$$N_\phi(t) = \sum_{r=1}^{n} \tilde{N}_r(t), \text{ almost surely.}$$



The Barlow-Proschan importance measure (cont.)

Hence, it follows that:

$$\lim_{t\to\infty}\frac{N_\phi(t)}{t} = \lim_{t\to\infty}\sum_{r=1}^{n}\frac{\tilde{N}_r(t)}{t} = \sum_{r=1}^{n}\frac{I_B^{(r)}}{\nu_r+\mu_r}, \text{ almost surely.}$$

By combining these results we get:

$$\lim_{t\to\infty}\frac{\tilde{N}_i(t)}{N_\phi(t)} = \lim_{t\to\infty}\frac{\tilde{N}_i(t)/t}{N_\phi(t)/t} = \frac{I_B^i/(\nu_i+\mu_i)}{\sum_{r=1}^{n} I_B^r/(\nu_r+\mu_r)}, \text{ almost surely.}$$



The Barlow-Proschan importance measure (cont.)

Hence, it follows that the Barlow-Proschan importance measure of component $i \in C$ in a repairable system, denoted $I_{B-P}^{(i)}$, can be computed by the following formula:

$$I_{B-P}^{(i)} = \frac{I_B^i/(\nu_i + \mu_i)}{\sum_{r=1}^{n} I_B^r/(\nu_r + \mu_r)}, \quad i \in C.$$

NOTE: Just as in the case of non-repairable components, we have:

$$\sum_{i=1}^{n} I_{B-P}^{(i)} = \frac{\sum_{i=1}^{n} I_B^i/(\nu_i + \mu_i)}{\sum_{r=1}^{n} I_B^r/(\nu_r + \mu_r)} = 1 \tag{1}$$



Section 7.4.1

Estimating availability and importance



Estimates of availability and importance

Goal: Estimate $A_\phi(t)$ and $I_B^{(1)}(t), \ldots, I_B^{(n)}(t)$ for $0 \leq t \leq t_N$

Solution: Pointwise estimates of $A_\phi(t)$ and $I_B^{(1)}(t), \ldots, I_B^{(n)}(t)$ for $t \in \{t_1, \ldots, t_N\}$, and use interpolation between these points.

In each simulation we sample the values of ϕ and $\psi_1, \ldots, \psi_n$ at each sampling point $t_1, \ldots, t_N$. We denote the sth simulated result of the component state vector process by $\{\boldsymbol{X}_s(t)\}$, $s = 1, \ldots, M$, and obtain the following estimates:

$$\hat{A}_\phi(t_j) = \frac{1}{M} \sum_{s=1}^{M} \phi(\boldsymbol{X}_s(t_j)), \quad j = 1, \ldots, K.$$

$$\hat{I}_B^{(i)}(t_j) = \frac{1}{M} \sum_{s=1}^{M} \psi_i(\boldsymbol{X}_s(t_j)), \quad j = 1, \ldots, K.$$



Estimates of availability and importance

We are also interested in the mean values of the availability and importance measures in the interval $[0, t]$, where $0 \leq t \leq t_K$, denoted respectively $A_\phi([0,t])$ and $I_B^{(1)}([0,t]), \ldots, I_B^{(n)}([0,t])$, and defined as follows:

$$A_\phi([0,t]) = \frac{1}{t}\int_0^t A_\phi(t)dt, \quad 0 \leq t \leq t_K$$

$$I_B^{(i)}([0,t]) = \frac{1}{t}\int_0^t I_B^{(i)}(t)dt, \quad 0 \leq t \leq t_K,\ i = 1, \ldots, n.$$



Estimates of availability and importance

These mean values can be estimated using the following estimators:

$$\hat{A}_{\phi}([0,t]) = \frac{1}{M}\sum_{s=1}^{M}\frac{1}{t}\int_0^t \phi(\boldsymbol{X}_s(t))dt, \quad 0 \le t \le t_K$$

$$\hat{l}_B^{(i)}([0,t]) = \frac{1}{M}\sum_{s=1}^{M}\frac{1}{t}\int_0^t \psi_i(\boldsymbol{X}_s(t))dt, \quad 0 \le t \le t_K,\ i = 1,\ldots,n,$$

where the integrals of $\{\phi(\boldsymbol{X}_s(t))\}, \{\psi_1(\boldsymbol{X}_s(t))\}, \ldots, \{\psi_n(\boldsymbol{X}_s(t))\}$ are calculated using the formula for integrals of pure jump processes.



Estimates of Barlow-Proschan importance

We let $\tilde{N}_{is}(t)$ denote the number of times a failure in component i causes system failure in $[0, t]$ in the sth simulation, $s = 1, \ldots, M$, $i = 1, \ldots, n$.

Similarly, we let $N_{\phi s}(t)$ denote the number of times the system fails in $[0, t]$ in the sth simulation, $s = 1, \ldots, M$.

The fraction of system failures caused by component i in $[0, t]$ converges towards $I_{B-P}^{(i)}$ as t tends to infinity.

Thus, we propose the following estimate for $I_{B-P}^{(i)}$:

$$\hat{I}_{B-P}^{(i)} = \frac{\sum_{s=1}^{M} \tilde{N}_{is}(t_K)}{\sum_{s=1}^{M} N_{\phi s}(t_K)}, \quad i = 1, \ldots, n.$$



Estimates of availability and importance

Alternatively, we may use the following estimator:

$$\tilde{I}_{B-P}^{(i)} = \frac{1}{M} \sum_{s=1}^{M} \frac{\tilde{N}_{is}(t_K)}{N_{\phi s}(t_K)}, \quad i = 1, \ldots, n.$$

When t_K is large, the two estimators will be approximately equal.

In fact, if we use a different *stopping rule* for the simulations, where we run each of the simulations until a predefined number, N_ϕ, of system failures have occurred, the two estimators are equal.

We let $\tilde{N}_{is}$ denote the number of times component i causes system failure in the sth simulation, $s = 1, \ldots, M$, $i = 1, \ldots, n$, and get:

$$\hat{I}_{B-P}^{(i)} = \frac{\sum_{s=1}^{M} \tilde{N}_{is}}{M N_\phi} = \frac{1}{M} \sum_{s=1}^{M} \frac{\tilde{N}_{is}}{N_\phi} = \tilde{I}_{B-P}^{(i)}, \quad i = 1, \ldots, n$$



Discrete event simulation – Event model

Let (C, ϕ) be a binary monotone system with component state processes: $\{X_1(t)\}, \ldots, \{X_n(t)\}$.

- $E_{i1}, E_{i2}, \ldots$ are the events affecting the process $\{X_i(t)\}$
- $T_{i1}, T_{i2}, \ldots$ are the corresponding points of time for these events

Assuming that all lifetimes and repair times have *absolutely continuous distributions*, all the events happen at *distinct* points of time almost surely, i.e., all the T_{ij}s are distinct numbers.

We assume that the events are sorted with respect to their respective points of time, so that $T_{i1} < T_{i2} < \cdots$.

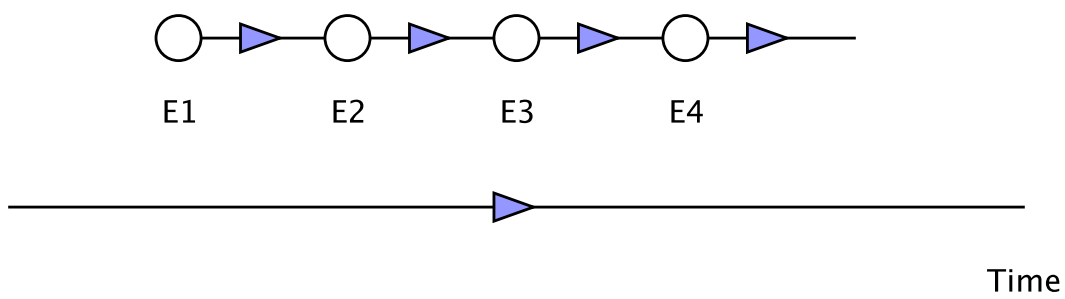


Event model (cont.)

At the system level the event set is the *union* of all the component event sets.

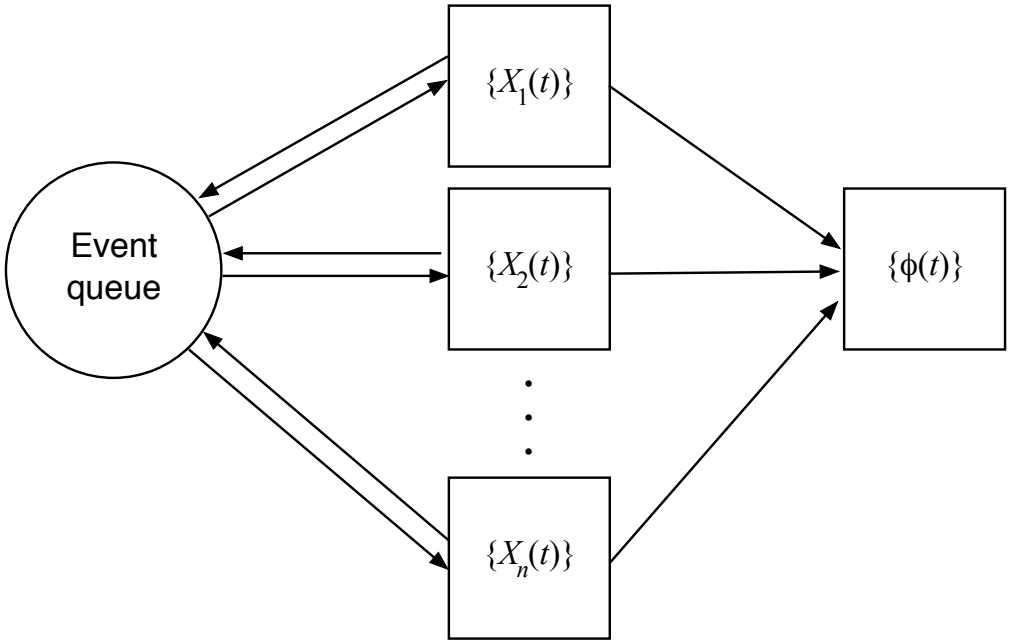
- Let $E^{(1)}, E^{(2)}, \ldots$ denote the system events sorted with respect to their respective points of time
- Let $T^{(1)} < T^{(2)} < \cdots$ be the corresponding points of time

Each system event corresponds to a unique component event, organised in a dynamic queue sorted with respect to the points of time of the events:





Program flow



- The components post initial events to the event queue
- The event queue processes events in chronological order, and notifies the components when the events occur. As soon as an event is processed, it is removed from the queue.
- The component updates its state, posts a new event to the queue, and notifies the system about the state change



Sampling events

Although the system state and component states stay constant between events, it is of interest to sample the state values at *predefined* points of time. Thus, we introduce yet another type of event, called a *sampling events* spread out evenly on the timeline.

- Let $e_1, e_2, \ldots$ denote the sampling events
- Let $t_1 < t_2 < \cdots$ are the corresponding points of time

Typically $t_j = j \cdot \Delta$ for some suitable $\Delta > 0$, $j = 1, 2, \ldots$.

The sampling events are placed into the queue in the same way as for the ordinary events.



Discrete event simulation in Python

```
#############################################################
#                                                           #
#    This file contains various classes and methods         #
#    needed to handle discrete event simulations of         #
#    of binary repairable components in a binary            #
#    monotone system                                        #
#                                                           #
#############################################################

from random import expovariate, weibullvariate
from math import gamma
import numpy as np
import matplotlib.pyplot as plt

# Event codes
FAILURE_EVENT = 100
REPAIR_EVENT = 200
SAMPLE_EVENT = 300

# State codes
FUNCTIONING = 1
FAILED = 0

# Criticality codes
CRITICAL = 1
NONCRITICAL = 0

# Run simulation up to maxTime + EPS to make sure that we get all events
EPS = 0.1

class Distribution:
    def generate(self):
        pass

    def mean(self):
        pass
```



The Distribution class

```
class Distribution:
    def generate(self):
        pass

    def mean(self):
        pass


class Exponential(Distribution):
    def __init__(self, rate):
        self.rate = rate    # The rate parameter (inverse of expectation)

    def generate(self):
        return expovariate(self.rate)

    def mean(self):
        return 1 / self.rate


class Weibull(Distribution):
    def __init__(self, alpha, beta):
        self.alpha = alpha  # The shape parameter
        self.beta = beta    # The scale parameter

    def generate(self):
        return weibullvariate(self.beta, self.alpha)

    def mean(self):
        return self.beta * gamma(1 + 1/self.alpha)
```



The Event class

```
##################################################
#                                                #
#    Event class                                 #
#                                                #
##################################################
class Event:
    def __init__(self, eventTime, eventCode, eventHandler, eventData = None):
        self.eventTime = eventTime
        self.eventCode = eventCode
        self.eventHandler = eventHandler
        self.eventData = eventData
        self.next = None

    def push(self, event):
        if self.next == None:
            self.next = event
        elif event.eventTime < self.next.eventTime:
            event.next = self.next
            self.next = event
        else:
            self.next.push(event)

    def processEvent(self):
        self.eventHandler.processEvent(self)
```



The EventHandler class

```
################################################
#                                              #
#    EventHandler class                        #
#                                              #
################################################
class EventHandler:
    def __init__(self, eventQueue):
        self.eventQueue = eventQueue

    def prepare(self):
        pass

    def processEvent(self, event):
        pass
```



The EventQueue class

```
################################################
#                                              #
#    EventQueue class                          #
#                                              #
################################################
class EventQueue:
    def __init__(self, maxTime):
        self.first = None
        self.time = 0
        self.maxTime = maxTime

    def reinit(self):
        self.first = None
        self.time = 0

    def push(self, event):
        event.eventTime += self.time
        if self.first == None:
            self.first = event
        elif event.eventTime < self.first.eventTime:
            event.next = self.first
            self.first = event
        else:
            self.first.push(event)

    def pop(self):
        if self.first != None:
            current_event = self.first
            self.time = current_event.eventTime
            self.first = current_event.next
            current_event.processEvent()

    def processAllEvents(self):
        while (self.first != None) and (self.time < self.maxTime + EPS):
            self.pop()
```



The EventSystem class

```
##################################################
#                                                #
#    EventSystem class                           #
#                                                #
##################################################
class EventSystem(EventHandler):
    def __init__(self, eventQueue, structureFunction, eventComponents,
                 sampleInterval, calculateBirnbaum=False, calculateBarlowProschan=False):
        super().__init__(eventQueue)
        self.structureFunction = structureFunction
        self.eventComponents = eventComponents
        self.numSimulations = 0

        index = 0
        for comp in self.eventComponents:
            comp.setEventSystem(self, index)
            index += 1

        self.numComponents = len(eventComponents)
        self.compStates = np.ones(self.numComponents)

        self.sampleInterval = sampleInterval
        self.numPoints = int(eventQueue.maxTime / sampleInterval) + 1
        self.sampleTimes = np.linspace(0, eventQueue.maxTime, self.numPoints)

        self.systemStates = np.zeros(self.numPoints)
        self.meanSystemStates = np.zeros(self.numPoints)

        self.failures = 0
```



The EventSystem class (cont.)

```
        self.systemStates = np.zeros(self.numPoints)
        self.meanSystemStates = np.zeros(self.numPoints)

        self.failures = 0

        # Birnbaum criticality
        self.calculateBirnbaum = calculateBirnbaum
        if self.calculateBirnbaum:
            self.criticalityStates = []
            self.meanCriticalityStates = []

            for i in range(self.numComponents):
                self.criticalityStates.append(np.zeros(self.numPoints))
                self.meanCriticalityStates.append(np.zeros(self.numPoints))

            self.criticalTime = np.zeros(self.numComponents)
            self.noncriticalTime = np.zeros(self.numComponents)
            self.criticalityState = np.zeros(self.numComponents)

        # Barlow & Proschan criticality
        self.calculateBarlowProschan = calculateBarlowProschan
        if self.calculateBarlowProschan:
            self.barlowProschanCriticality = np.zeros(self.numComponents)

            for i in range(self.numComponents):
                self.barlowProschanCriticality[i] = 0
```



The EventSystem class (cont.)

```
    def criticalityFunction(self, index):
        tmpCompState = self.compStates[index]
        if tmpCompState == FUNCTIONING:
            phi_1 = self.state
            self.compStates[index] = FAILED
            phi_0 = self.structureFunction(self.compStates)
        else:
            phi_0 = self.state
            self.compStates[index] = FUNCTIONING
            phi_1 = self.structureFunction(self.compStates)
        self.compStates[index] = tmpCompState
        return phi_1 - phi_0

    def prepare(self):
        self.numSimulations += 1
        for comp in self.eventComponents:
            comp.prepare()
        self.state = self.structureFunction(self.compStates)

        self.systemStates[0] += self.state
        self.meanSystemStates[0] += self.state

        self.upTime = 0
        self.downTime = 0

        if self.calculateBirnbaum:
            for i in range(self.numComponents):
                self.criticalityState[i] = self.criticalityFunction(i)

                self.criticalityStates[i][0] += self.criticalityState[i]
                self.meanCriticalityStates[i][0] += self.criticalityState[i]

                self.criticalTime[i] = 0
                self.noncriticalTime[i] = 0

        self.pointCount = 1
        self.lastEventTime = 0

        if self.pointCount < self.numPoints:
            sampleEvent = Event(self.sampleInterval, SAMPLE_EVENT, self)
            self.eventQueue.push(sampleEvent)
```



The EventSystem class (cont.)

```
def processEvent(self, event):
    if event.eventCode == FAILURE_EVENT:
        self.processFailureEvent(event)
    elif event.eventCode == REPAIR_EVENT:
        self.processRepairEvent(event)
    elif event.eventCode == SAMPLE_EVENT:
        self.processSampleEvent(event)
```



The EventSystem class (cont.)

```
    def processFailureEvent(self, event):
        index = int(event.eventData)
        self.compStates[index] = FAILED
        if self.state == FUNCTIONING:
            self.upTime += (event.eventTime - self.lastEventTime)
            self.state = self.structureFunction(self.compStates)
            if self.state == FAILED:
                self.failures += 1
                if self.calculateBarlowProschan:
                    self.barlowProschanCriticality[index] += 1
        else:
            self.downTime += (event.eventTime - self.lastEventTime)

        if self.calculateBirnbaum:
            for i in range(self.numComponents):
                if self.criticalityState[i] == CRITICAL:
                    self.criticalTime[i] += (event.eventTime - self.lastEventTime)
                else:
                    self.noncriticalTime[i] += (event.eventTime - self.lastEventTime)
                self.criticalityState[i] = self.criticalityFunction(i)

        self.lastEventTime = event.eventTime
```



The EventSystem class (cont.)

```
    def processRepairEvent(self, event):
        index = int(event.eventData)
        self.compStates[index] = FUNCTIONING
        if self.state == FAILED:
            self.downTime += (event.eventTime - self.lastEventTime)
            self.state = self.structureFunction(self.compStates)
        else:
            self.upTime += (event.eventTime - self.lastEventTime)

        if self.calculateBirnbaum:
            for i in range(self.numComponents):
                if self.criticalityState[i] == CRITICAL:
                    self.criticalTime[i] += (event.eventTime - self.lastEventTime)
                else:
                    self.noncriticalTime[i] += (event.eventTime - self.lastEventTime)
                self.criticalityState[i] = self.criticalityFunction(i)

        self.lastEventTime = event.eventTime
```



The EventSystem class (cont.)

```
def processSampleEvent(self, event):
    if self.state == FUNCTIONING:
        self.upTime += (event.eventTime - self.lastEventTime)
    else:
        self.downTime += (event.eventTime - self.lastEventTime)

    if self.calculateBirnbaum:
        for i in range(self.numComponents):
            if self.criticalityState[i] == CRITICAL:
                self.criticalTime[i] += (event.eventTime - self.lastEventTime)
            else:
                self.noncriticalTime[i] += (event.eventTime - self.lastEventTime)

    self.lastEventTime = event.eventTime

    currPoint = self.pointCount
    self.pointCount += 1
    self.systemStates[currPoint] += self.state
    self.meanSystemStates[currPoint] += (self.upTime / self.lastEventTime)

    if self.calculateBirnbaum:
        for i in range(self.numComponents):
            self.criticalityStates[i][currPoint] += self.criticalityState[i]
            self.meanCriticalityStates[i][currPoint] += (self.criticalTime[i] / self.lastEventTime)

    if self.pointCount < self.numPoints:
        sampleEvent = Event(self.sampleInterval, SAMPLE_EVENT, self)
        self.eventQueue.push(sampleEvent)
```



The EventSystem class (cont.)

```
def calculateAverages(self):
    for i in range(self.numPoints):
        self.systemStates[i] /= self.numSimulations
        self.meanSystemStates[i] /= self.numSimulations

    if self.calculateBirnbaum:
        for i in range(self.numComponents):
            self.criticalityStates[i] /= self.numSimulations
            self.meanCriticalityStates[i] /= self.numSimulations
```



The EventSystem class (cont.)

```
    ###### Various plot methods related to system states and mean system states ######
    def plotSystemStates(self, file=None):
    def plotMeanSystemStates(self, file=None):
    def plotComboSystemStates(self, file=None):
    def plotBirnbaumImportance(self, file=None):
    def plotMeanBirnbaumImportance(self, file=None):

    def plotBarlowProschanImportance(self, file=None):
    def printSystemStatistics(self):
```



The EventComponent class

```
################################################
#                                              #
#    EventComponent class                      #
#                                              #
################################################
class EventComponent(EventHandler):
    def __init__(self, eventQueue, failureDistribution, repairDistribution):
        super().__init__(eventQueue)
        self.eventSystem = None
        self.index = 0
        self.failureDistribution = failureDistribution
        self.repairDistribution = repairDistribution
        self.state = FUNCTIONING

    def setEventSystem(self, eventSystem, index):
        self.eventSystem = eventSystem
        self.index = index

    def prepare(self):
        self.state = FUNCTIONING
        failureEvent = Event(self.failureDistribution.generate(), FAILURE_EVENT, self, self.index)
        self.eventQueue.push(failureEvent)

    def processEvent(self, event):
        if event.eventCode == FAILURE_EVENT:
            self.state = FAILED
            repairEvent = Event(self.repairDistribution.generate(), REPAIR_EVENT, self, self.index)
            self.eventQueue.push(repairEvent)
        elif event.eventCode == REPAIR_EVENT:
            self.state = FUNCTIONING
            failureEvent = Event(self.failureDistribution.generate(), FAILURE_EVENT, self, self.index)
            self.eventQueue.push(failureEvent)
        if self.eventSystem != None:
            self.eventSystem.processEvent(event)
```



Example – A bridge structure

The lifetimes of component $i \in C$, i.e., $U_{i1}, U_{i1}, \ldots$, are Weibull-distributed with parameters α_i^u and β_i^u, where:

$$\alpha_1^u = 1.5, \quad \alpha_2^u = 1.5, \quad \alpha_3^u = 1.5, \quad \alpha_4^u = 1.5, \quad \alpha_5^u = 1.5,$$
$$\beta_1^u = 30.0, \quad \beta_2^u = 60.0, \quad \beta_3^u = 20.0, \quad \beta_4^u = 40.0, \quad \beta_5^u = 50.0.$$

The repair times of component $i \in C$, i.e., $D_{i1}, D_{i1}, \ldots$, are Weibull-distributed with parameters α_i^d and β_i^d, where:

$$\alpha_1^d = 1.0, \quad \alpha_2^d = 1.0, \quad \alpha_3^d = 1.0, \quad \alpha_4^d = 1.0, \quad \alpha_5^d = 1.0,$$
$$\beta_1^d = 15.0, \quad \beta_2^d = 30.0, \quad \beta_3^d = 10.0, \quad \beta_4^d = 20.0, \quad \beta_5^d = 25.0.$$



Estimates of availability and mean availability

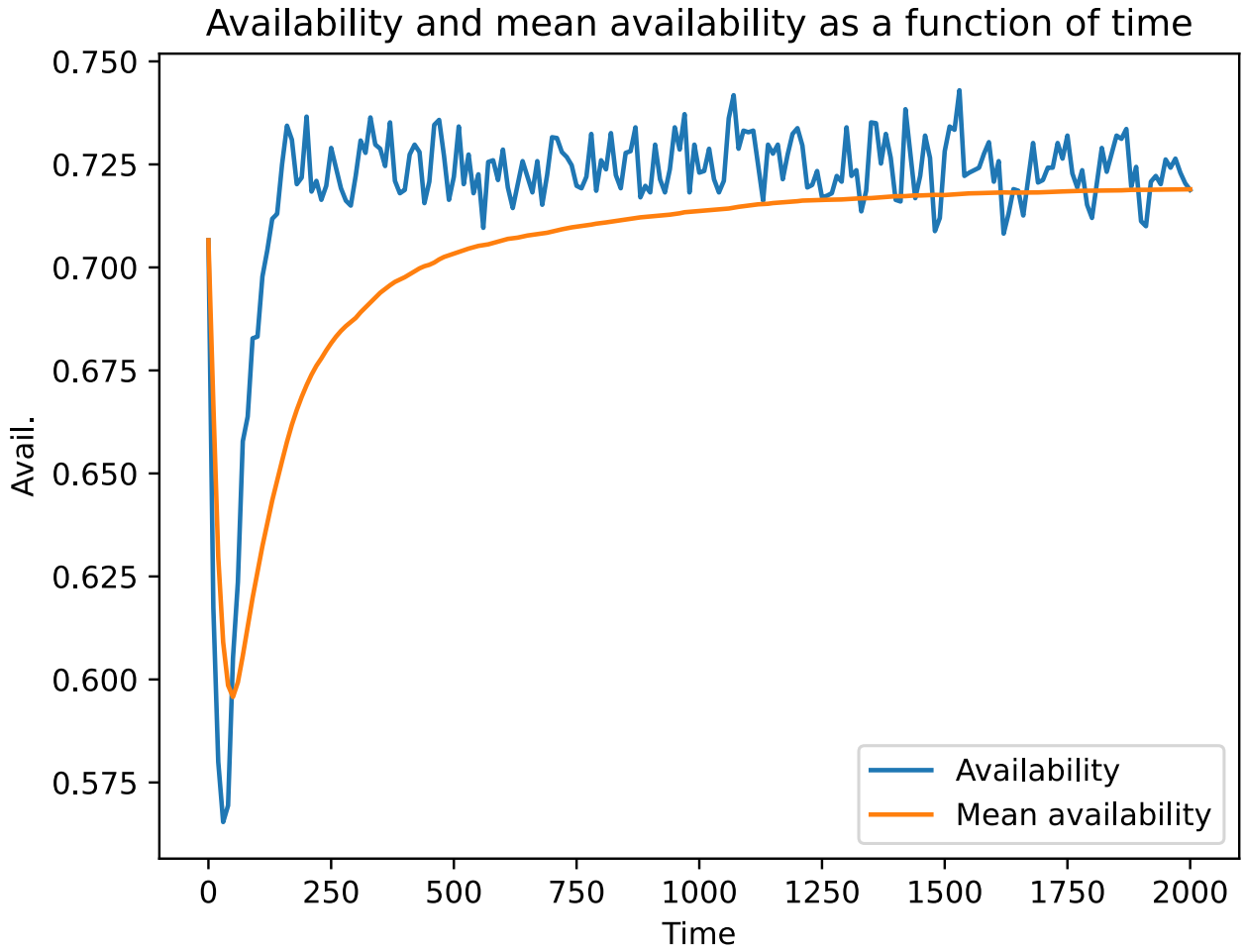


Figure: Estimated availability and mean availability of a bridge system

Estimates of Birnbaum importance

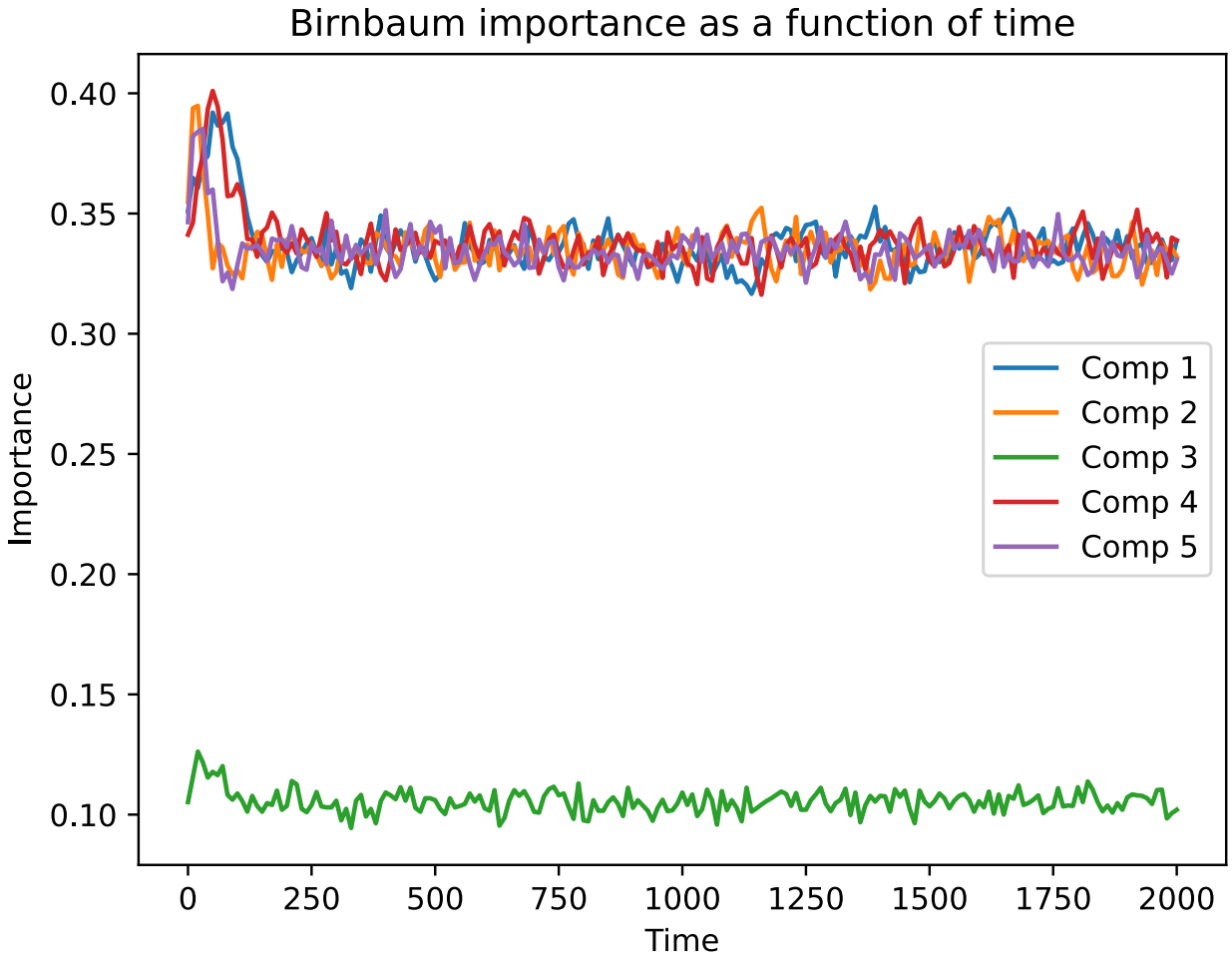


Figure: Estimated Birnbaum importance of components in a bridge system

Estimates of mean Birnbaum importance

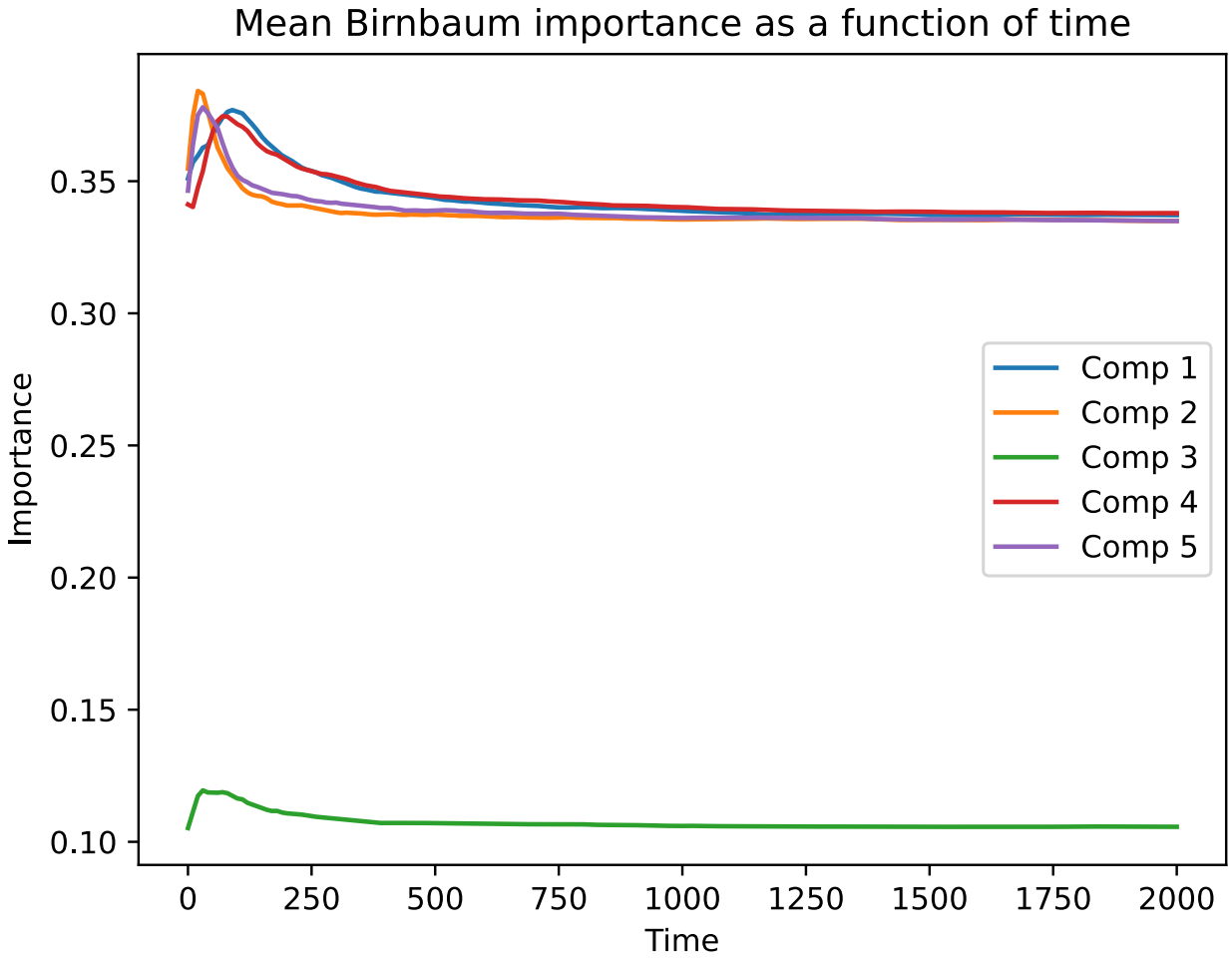


Figure: Estimated mean Birnbaum importance of components in a bridge



Estimates of Barlow-Proschan importance

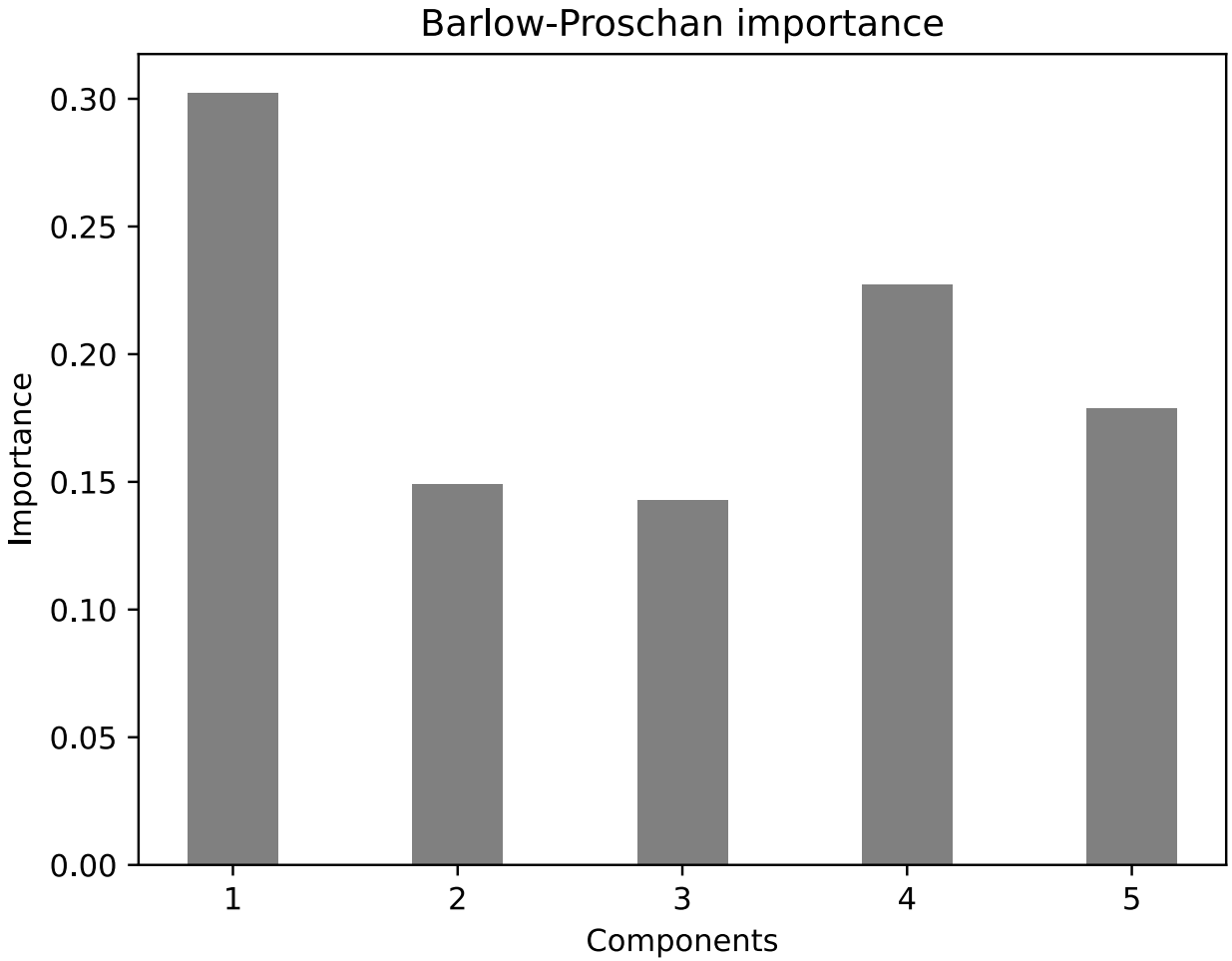


Figure: Estimated Barlow-Proschan importance of components in a bridge

Example – A threshold system

The threshold value is 5.0, while the component weights $\omega_1, \ldots, \omega_5$ are:

$$\omega_1 = 2.0, \quad \omega_2 = 4.0, \quad \omega_3 = 2.3, \quad \omega_4 = 3.5, \quad \omega_5 = 1.9$$

The lifetimes of component $i \in C$, i.e., $U_{i1}, U_{i1}, \ldots$, are Weibull-distributed with parameters α_i^u and β_i^u, where:

$$\alpha_1^u = 1.5, \quad \alpha_2^u = 1.5, \quad \alpha_3^u = 1.5, \quad \alpha_4^u = 1.5, \quad \alpha_5^u = 1.5,$$
$$\beta_1^u = 30.0, \quad \beta_2^u = 60.0, \quad \beta_3^u = 20.0, \quad \beta_4^u = 40.0, \quad \beta_5^u = 50.0.$$

The repair times of component $i \in C$, i.e., $D_{i1}, D_{i1}, \ldots$, are Weibull-distributed with parameters α_i^d and β_i^d, where:

$$\alpha_1^d = 1.0, \quad \alpha_2^d = 1.0, \quad \alpha_3^d = 1.0, \quad \alpha_4^d = 1.0, \quad \alpha_5^d = 1.0,$$
$$\beta_1^d = 15.0, \quad \beta_2^d = 30.0, \quad \beta_3^d = 10.0, \quad \beta_4^d = 20.0, \quad \beta_5^d = 25.0.$$



Estimates of availability and mean availability

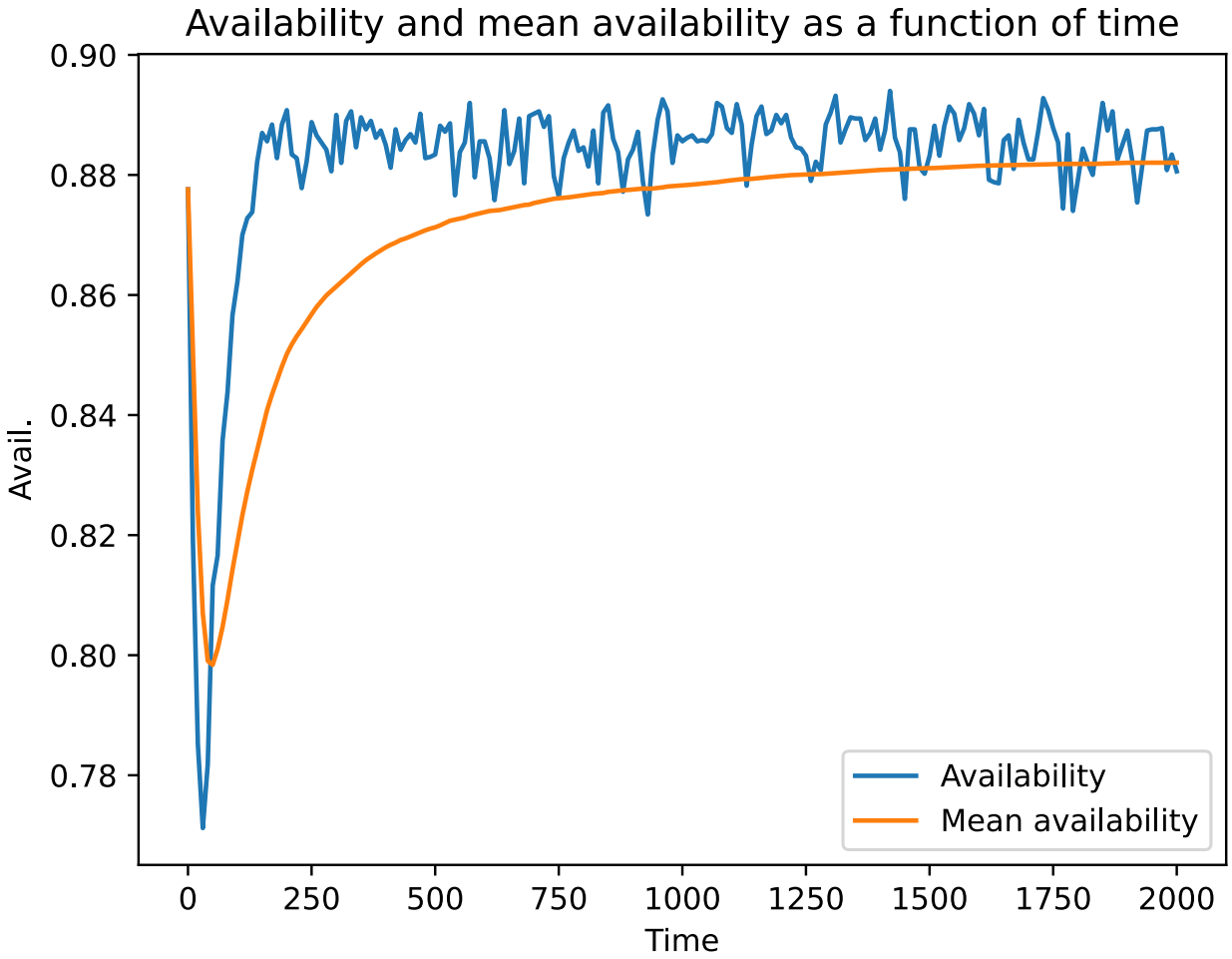


Figure: Estimated availability and mean availability of a threshold system

Estimates of Birnbaum importance

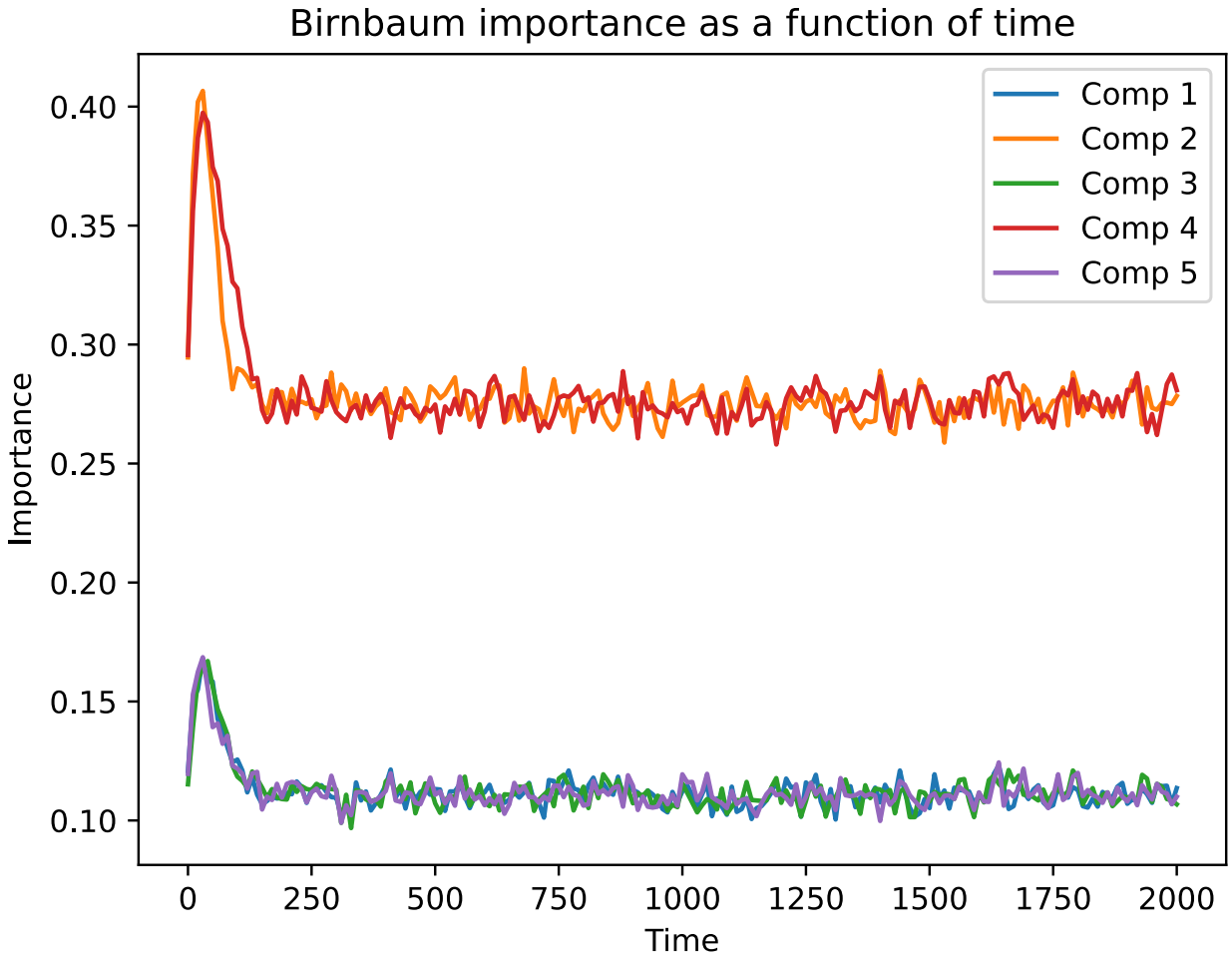


Figure: Estimated Birnbaum importance of components in a threshold system

Estimates of mean Birnbaum importance

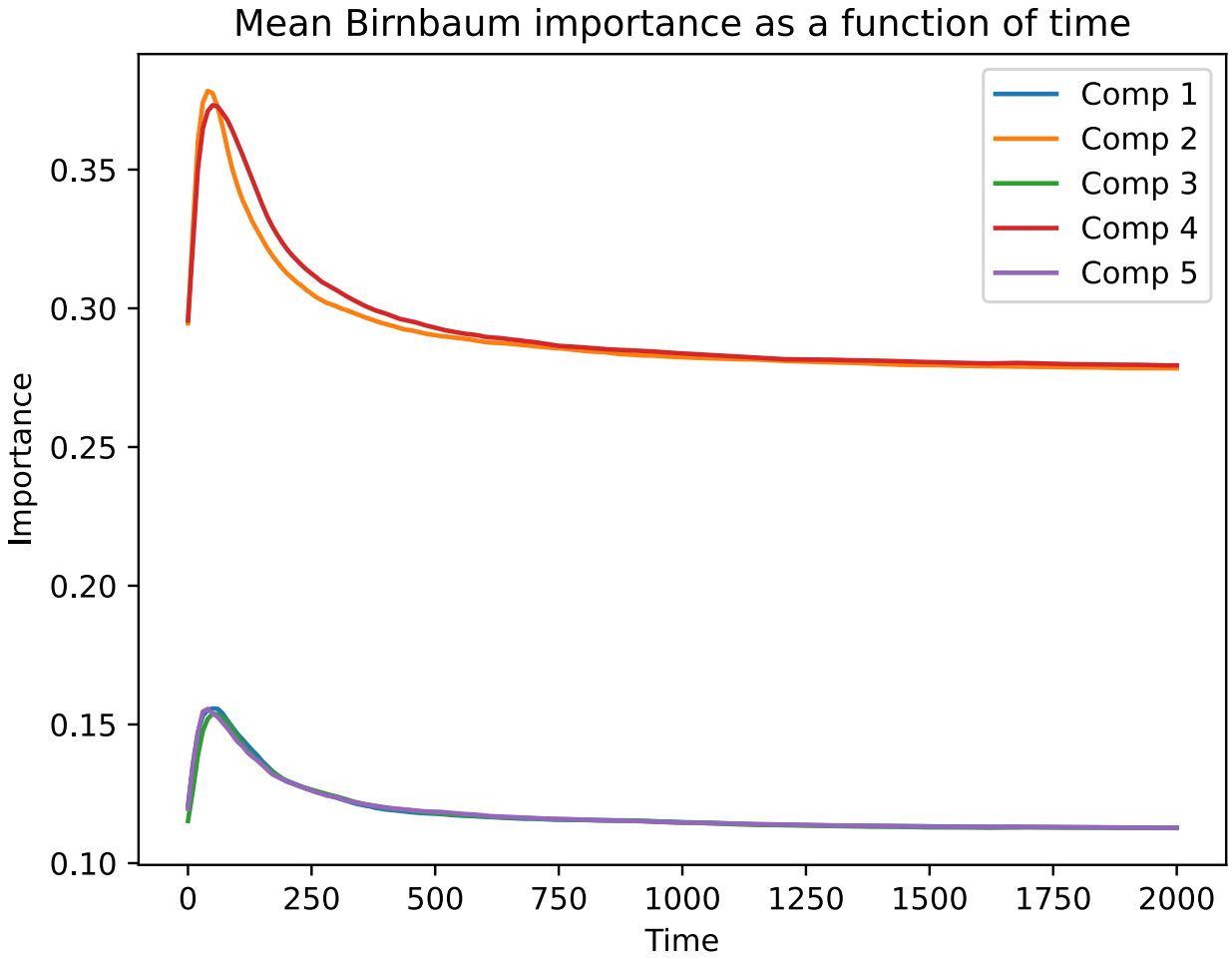


Figure: Estimated mean Birnbaum importance of components in a threshold

Estimates of Barlow-Proschan importance

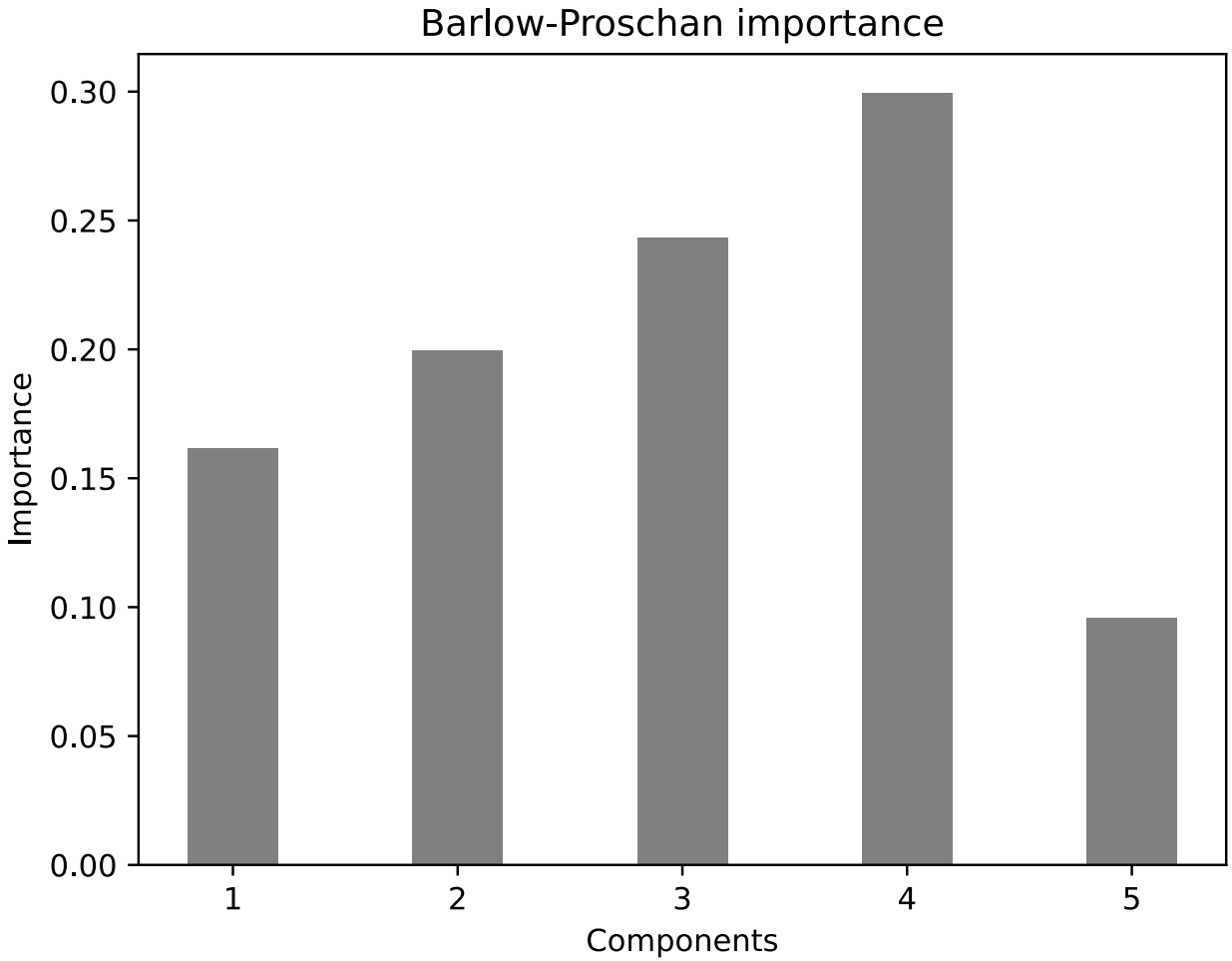


Figure: Estimated Barlow-Proschan importance of components in a threshold

Section 7.5

Exercises



Exercise 7.3.

Consider a binary monotone system (C, ϕ) with minimal path sets $P_1, \ldots, P_p$ and minimal cut sets $K_1, \ldots, K_k$. We assume that the components cannot be repaired, and let T_i denote the lifetime of component $i \in C$. Moreover, we let S denote the lifetime of the system. Prove that:

$$S = \min_{1 \le j \le k} \max_{i \in K_j} T_i = \max_{1 \le j \le p} \min_{i \in P_j} T_i.$$

SOLUTION: Let V_j denote the lifetime of the jth minimal cut parallel structure. Then V_j equals the lifetime of the longest living component in K_j. That is, we have:

$$V_j = \max_{i \in K_j} T_i$$

Moreover, the lifetime of the system equals the lifetime of the shortest living minimal cut parallel structure. That is, we have:

$$S = \min_{1 \le j \le k} V_j = \min_{1 \le j \le k} \max_{i \in K_j} T_i$$



Exercise 7.3 (cont.)

Let U_j denote the lifetime of the jth minimal path series structure. Then U_j equals the lifetime of the shortest living component in P_j. That is, we have:

$$U_j = \min_{i \in P_j} T_i$$

Moreover, the lifetime of the system equals the lifetime of the longest living minimal path series structure. That is, we have:

$$S = \max_{1 \leq j \leq p} U_j = \max_{1 \leq j \leq p} \min_{i \in P_j} T_i$$



Exercise 7.4.

Assume that the component lifetimes have so-called *proportional hazards*, that is the survival function is given by:

$$\bar{F}_i(t) = 1 - F_i(t) = \exp(-\lambda_i R(t)), \quad \lambda_i > 0, t \geq 0, \quad i = 1, \ldots, n,$$

where R is a strictly increasing, differentiable function such that $R(0) = 0$, and $\lim_{t\to\infty} R(t) = \infty$. Prove that for a series structure, we have:

$$I_{B-P}^{(i)} = \frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}.$$



SOLUTION: By the definition of $\bar{F}_i(t)$ we have:

$$f_i(t) = -\frac{d\bar{F}_i(t)}{dt} = \lambda_i R'(t) \exp(-\lambda_i R(t)) = \lambda_i R'(t) \bar{F}_i(t)$$

Furthermore, since we are considering a series system, we have:

$$I_B^{(i)}(t) = \prod_{j \neq i} p_i(t) = \prod_{j \neq i} \bar{F}_j(t)$$



$$I_{B-P}^{(i)} = \int_0^\infty I_B^{(i)}(t) f_i(t) dt = \int_0^\infty \prod_{j \neq i} \bar{F}_j(t) \lambda_i R'(t) \bar{F}_i(t) dt$$

$$= \int_0^\infty \prod_{j=1}^{n} \bar{F}_j(t) \lambda_i R'(t) dt = \lambda_i \int_0^\infty R'(t) e^{-(\sum_{j=1}^n \lambda_j) R(t)} dt$$

$$= \frac{\lambda_i}{\sum_{j=1}^n \lambda_j} \int_0^\infty (\sum_{j=1}^{n} \lambda_j) R'(t) e^{-(\sum_{j=1}^n \lambda_j) R(t)} dt$$

$$= \frac{\lambda_i}{\sum_{j=1}^n \lambda_j}$$

NOTE: $(\sum_{j=1}^n \lambda_j) R'(t) e^{-(\sum_{j=1}^n \lambda_j) R(t)}$ is the density of a distribution with cumulative failure rate function $(\sum_{j=1}^n \lambda_j) R(t)$.



Exercise 7.5.

Consider a binary non-repairable monotone system (C, ϕ), and assume that the i'th component is irrelevant for the system ϕ. Then, what is $I_B^{(i)}(t)$ and $I_{B-P}^{(i)}$?



SOLUTION: By pivotal decomposition we get that:

$$I_B^{(i)}(t) = \frac{\partial h(\mathbf{p}(t))}{\partial p_i(t)} = h(1_i, \mathbf{p}(t)) - h(0_i, \mathbf{p}(t))$$

$$= E[\phi(1_i, \mathbf{X}(t)) - \phi(0_i, \mathbf{X}(t))]$$

$$= \sum_{(\cdot_i, \mathbf{x})} (\phi(1_i, \mathbf{x}) - \phi(0_i, \mathbf{x}))P(\mathbf{X}(t) = \mathbf{x}) = 0$$

since the i'th component is irrelevant.

Moreover, since $I_B^{(i)}(t) = 0$ for all $t > 0$, it follows that:

$$I_{B-P}^{(i)} = \int_0^\infty f_i(t) I_B^{(i)}(t) dt = 0$$

Thus, according to both these measures, the reliability importance of an irrelevant component is 0 (which is intuitive).

