Mandatory Assignment STK3405

Simen Meen Haugeng

October 2024

1 1

a

b

We have from 3.1 that:

$$\phi(X) = X_i\phi(1_i, X) + (1 - X_i)\phi(0_i, X)$$

Using $i = 3$, we get:

$$\phi(X) = X_3\phi(1_3, X) + (1 - X_3)\phi(0_3, X)$$

Decomposing further using $i = 6$ when component 3 functions or not functions:

$$\phi(1_3, X) = X_6\phi(1_6, X) + (1 - X_6)\phi(0_6, X)$$
$$\phi(0_3, X) = X_6\phi(1_6, X) + (1 - X_6)\phi(0_6, X)$$

Then if both 3 and 6 functions:

$$\phi(1_6, X) = (X_1X_2) \sqcup (X_4X_5) \sqcup (X_7X_8)$$

And if $i = 3$ functions, but 6 doesn't:

$$\phi(0_6, X) = (X_1 \sqcup X_2)(X_4X_7)(X_5X_8)$$

This means now that:

$$\phi(1_3, X) = X_3(X_1 \sqcup X_2)(X_4 \sqcup X_5)(X_7 \sqcup X_8) - (1 - X_3)((X_1 \sqcup X_2)(X_4X_7)(X_5X_8))$$

If now 3 doesnt functions, but 6 does, we use $\phi(0_3, X)$
$\Rightarrow$ 3 doesn't functions, but 6 does:

$$\phi(1, X_6) = ((X_1X_4) \sqcup (X_2X_5))(X_7 \sqcup X_8)$$

Either 3 or 6 functions:

$$\phi(0_6, X) = (X_1X_4X_7) \sqcup (X_2X_5X_8)$$

This means that:

$$\phi(0_3, X) = X_6(((X_1X_4) \sqcup (X_2X_5))(X_4X_8)) - (1 - X_6)((X_1X_4X_7) \sqcup (X_2X_5X_8))$$

Combining everything together we get::

$\phi(X) =\ X_3X_6 \cdot (X_1 \sqcup X_2) \cdot (X_4 \sqcup X_5) \cdot (X_7 \sqcup X_8)$
$+ X_3(1 - X_6) \cdot (X_1 \sqcup X_2) \cdot ((X_4X_7) \sqcup (X_5X_8))$
$+ (1 - X_3)X_6 \cdot ((X_1X_4) \sqcup (X_2X_5)) \cdot (X_7 \sqcup X_8)$
$+ (1 - X_3)(1 - X_6) \cdot ((X_1X_4X_7) \sqcup (X_2X_5X_8))$

Which is what we wanted, and the statement is proved.

c

We have that $h(p) = E[\phi(x)]$. And since the components in the system are independent we get:
$\mathrm{E}[\phi(X)] = E[X_3X_6(X_1 \sqcup X_2)(X_4 \sqcup X_5)(X_7 \sqcup X_8)]$
$+ E[X_3(1 - X_6)(X_1 \sqcup X_2)((X_4X_7) \sqcup (X_5X_8))]$
$+ E[(1 - X_3)X_6((X_1X_4) \sqcup (X_2X_5))(X_7 \sqcup X_8)]$
$+ E[(1 - X_3)(1 - X_6)((X_1X_4X_7) \sqcup (X_2X_5X_8))]$
Which translates to this expression:
$= \mathrm{p}_3p_6(p_1 \sqcup p_2)(p_4 \sqcup p_5)(p_7 \sqcup_8)$
$+ p_3(1 - p_6)(p_1 \sqcup p_2)((p_4p_7) \sqcup (p_5p_8))$
$+ (1 - p_3)p_6((p_1p_4) \sqcup (p_2p_5))(p_7 \sqcup p_8) + (1 - p_3)(1 - p_6)((p_1p_4p_7) \sqcup (p_3p_5p_8))$

Since the states are independent we can split up the expression even more and take the expectev value at each variable, i.e. (this also counts for the component probabilities):
$\mathrm{E}[\phi(X)] = E[X_3]E[X_6]E[(X_1 \sqcup X_2)]E[(X_4 \sqcup X_5)(X_7 \sqcup X_8)]$
$+ E[X_3]E[(1 - X_6)]E[(X_1 \sqcup X_2)]E[((X_4X_7) \sqcup (X_5X_8))]$
$+ E[(1 - X_3)]E[X_6]E[((X_1X_4) \sqcup (X_2X_5))]E[(X_7 \sqcup X_8)]$
$+ E[(1 - X_3)]R[(1 - X_6)]E[((X_1X_4X_7) \sqcup (X_2X_5X_8))]$

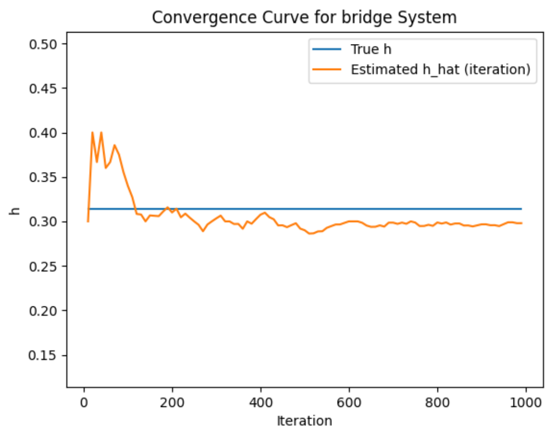
d

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

# Number of simulations
num_sims = 1000
```

```
# Interval between h_hat calculations
h_interv = 10

# Random number generator seed. Set to 'None' for a random sequence
seed_num = 1234
gen = np.random.default_rng(seed=seed_num)

# Save plot to file or not?
save_plot = True

# Name of system
sys_name = "bridge"

# Number of components:
n = 8

# Component reliabilities
px = [0.0, 0.6, 0.3, 0.5, 0.4, 0.7, 0.5, 0.3, 0.6]  # p = (0.6, 0.3, 0.5, 0.4, 0.7, 0.5, 0.3

def coprod(x, y):
    """Coproduct for probabilities."""
    return x + y - x * y

def phi(x):
    """Calculate the reliability function phi(X) based on the system's structure."""
    return x[3] * x[6] * coprod(x[1], x[2]) * coprod(x[4], x[5]) * coprod(x[7], x[8]) + x[3

def hh(p):
    """True reliability function based on the component reliabilities."""
    component_prob = (
                p[3] * p[6] * coprod(p[1], p[2]) * coprod(p[4], p[5]) * coprod(p[7], p[8])
                + p[3] * (1 - p[6]) * coprod(p[1], p[2]) * coprod(p[4] * p[7], p[5] * p[8])
                + (1 - p[3]) * p[6] * coprod(p[1] * p[4], p[2] * p[5]) * coprod(p[7], p[8])
                + (1 - p[3]) * (1 - p[6]) * coprod(p[1] * p[4] * p[7], p[2] * p[5] * p[8])
    )
    return component_prob

X = np.zeros(n + 1, dtype=int)  # Component state variables (X[0] is not used)

I = []
H = []
H_hat = []
T = 0

# Calculate the true system reliability
h = hh(px)
```

```
for sim in range(num_sims):
    U = gen.uniform(0.0, 1.0, n + 1)  # Uniform variables (U[0] is not used)
    for m in range(1, n + 1):
        X[m] = 1 if U[m] <= px[m] else 0
    T += phi(X)
    if sim > 0 and sim % h_interv == 0:
        h_hat = T / sim
        I.append(sim)
        H.append(h)
        H_hat.append(h_hat)

# Estimate final system reliability
h_hat = T / num_sims

print("True system reliability h = ", h)
print("Estimated system reliability h_hat = ", h_hat)

# Plotting the convergence curve
plt.plot(I, H, label='True h')
plt.plot(I, H_hat, label='Estimated h_hat (iteration)')
plt.ylim(h - 0.2, h + 0.2)
plt.xlabel('Iteration')
plt.ylabel('h')
plt.title('Convergence Curve for ' + sys_name + " System")
plt.legend()

if save_plot:
    plt.savefig(f"./crudegen/{sys_name}.pdf")
plt.show()

#Output:
True system reliability h =  0.31356
Estimated system reliability h_hat =  0.295
```

Which gives this plot:



e

Since $\hat{\theta}_0, ..., \hat{\theta}_8$ are unbiased and therefore $E[\hat{\theta}_s] = \hat{\theta}_s$, we get that:
E$[h_C \hat{M}C] = E[\sum_{s=0}^{8} \hat{\theta}_s * P(S = s)]$
$= \sum_{s=0}^{8} E[\hat{\theta}_s] * P(S = s) = \sum_{s=0}^{8} \theta_s * P(S = s)$

By (6.4) in the compendium we know that:
$\sum_{s=0}^{8} \theta_s * P(S = s) = h$
Which concludes the proof.

f

We know that if the states $\theta_i = 0 \ \forall i \leq d$, the system dont work. And if $\theta_i = 1$ $\forall i > n - c$, then the system works. Since we have that d = 3, c = 2 and n = 8, we get that $\theta_0, \theta_1, \theta_2 = 0$ and since 8-2 = 6, $\theta_7, \theta_8 = 1$.
This means that if less than 3 states are functioning, then the system will always fail. And if more than 6 states are functioning, then the system will always work. I.e., we dont need to run a MC for $\theta_0, \theta_1, \theta_2, \theta_7 and \theta_8$ and we can focus on the middle values and therefore make our simulations computationally more efficient.

g

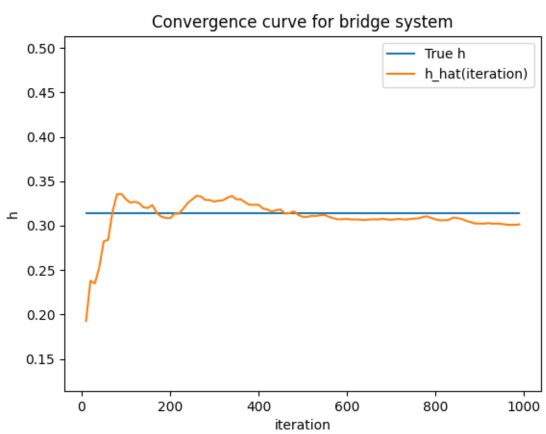
```
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 1000

# Interval between h_hat calculations
```

```
h_interv = 10

# Random number generator seed. Set to 'None' for a random sequence
seed_num = 1234
gen = np.random.default_rng(seed = seed_num)

# Save plot to file or not?
save_plot = True

# Name of system
sys_name = "bridge"

# Number of components:
n = 8

# Length of shortest path
d = 3

# Length of shortest cut
c = 2

# Component reliabilities
px = [0.0, 0.6, 0.3, 0.5, 0.4, 0.7, 0.5, 0.3, 0.6]      # P(X_i = px[i]), i = 1, .., n. px[0

def coprod(x, y):
    return x + y - x*y

def phi(x):
    return x[3] * x[6] * coprod(x[1], x[2]) * coprod(x[4], x[5]) * coprod(x[7], x[8]) + x[3


def hh(p):
    component_prob = (
                p[3] * p[6] * coprod(p[1], p[2]) * coprod(p[4], p[5]) * coprod(p[7], p[8])
                + p[3] * (1 - p[6]) * coprod(p[1], p[2]) * coprod(p[4] * p[7], p[5] * p[8])
                + (1 - p[3]) * p[6] * coprod(p[1] * p[4], p[2] * p[5]) * coprod(p[7], p[8])
                + (1 - p[3]) * (1 - p[6]) * coprod(p[1] * p[4] * p[7], p[2] * p[5] * p[8])
    )
    return component_prob


# Compute the distributions of S_1, ... , S_n, where S_m = X_m + ... + X_n, m = 1, ..., n
ps = np.zeros([n+1, n+1])                   # P(S_m = s) = ps[m,s], s = 0, 1, ..., (n-m+1).

ps[n,0] = 1.0 - px[n]                       # P(S_n = 0) = 1 - P(X_n = 1)
ps[n,1] = px[n]                             # P(S_n = 1) = P(X_n = 1)
```

```
for j in range(1, n):
    m = n-j
    ps[m,0] = ps[m+1,0] * (1.0 - px[m])                              # P(S_m = 0) = P(S_{m+1]

    for s in range(1, n-m+1):
        ps[m,s] = ps[m+1,s-1] * px[m] + ps[m+1,s] * (1.0 - px[m])   # P(S_m = s) = P(S_{m+1]
                                                                     # + P(S_{m+1} = s) * P(X

    ps[m,n-m+1] = ps[m+1,n-m] * px[m]                                # P(S_m = n-m+1) = P(S_

# Print the distribution of S = S_1
for s in range(n+1):
    print("P(S = " + str(s) + ") =", ps[1,s])

# Calculate pdc = P(d <= S_1 <= n-c)
pdc = 0
for s in range(d, n-c+1):
    pdc += ps[1,s]

# Calculate pcn = P(n-c < S_1 <= n)
pcn = 0
for s in range(n-c+1, n+1):
    pcn += ps[1,s]

# Sample S_1 from the set {d, ... , n-c}
def sampleS():
    u = gen.uniform(0.0, pdc)
    for s in range(d, n-c):
        if u < ps[1,s]:
            return s
        else:
            u -= ps[1,s]
    return n-c

X = np.zeros(n+1, dtype = int)       # The component state variables (X[0] is not used)
T = np.zeros(n+1, dtype = int)       # T[s] counts random path sets of size s = d, 1, ...,
V = np.zeros(n+1, dtype = int)       # V[s] counts random sets of size s = d, 1, ..., n-c

I = []
H = []
H_hat = []

# Calculate the true system reliability
h = hh(px)
```

```
# Run the simulations
for sim in range(num_sims):
    s = sampleS()
    V[s] += 1
    U = gen.uniform(0.0, 1.0, n+1)    # Uniform variables (U[0] is not used)
    sumx = 0
    for m in range(1, n):
        if sumx < s:
            p = px[m] * ps[m+1, s-sumx-1] / ps[m, s-sumx]
            if U[m] <= p:
                X[m] = 1
            else:
                X[m] = 0
            sumx += X[m]
        else:
            X[m] = 0
    if sumx < s:
        X[n] = 1
    else:
        X[n] = 0
    T[s] += phi(X)
    if sim > 0 and sim % h_interv == 0:
        h_hat = pcn
        for s in range(d, n-c+1):
            if V[s] > 0:
                h_hat += ps[1,s] * T[s] / V[s]
        I.append(sim)
        H.append(h)
        H_hat.append(h_hat)

# Print the estimated conditional reliabilities, theta_s, s = d, ..., n-c
for s in range(d, n-c+1):
    print("theta_" + str(s) + " =", T[s] / V[s])

# Estimate final system reliability
h_hat = pcn

for s in range(d, n-c+1):
    h_hat += ps[1,s] * T[s] / V[s]

print("h = ", h, ", h_hat = ", h_hat)

plt.plot(I, H, label='True h')
plt.plot(I, H_hat, label='h_hat(iteration)')
plt.ylim(h - 0.2, h + 0.2)
plt.xlabel('iteration')
```

```
plt.ylabel('h')
plt.title('Convergence curve for ' + sys_name + " system")
plt.legend()
if save_plot:
    plt.savefig("cmcgen/" + sys_name + ".pdf")
plt.show()

#Output:
P(S = 0) = 0.003528
P(S = 1) = 0.031248000000000005
P(S = 2) = 0.11588200000000001
P(S = 3) = 0.23502200000000006
P(S = 4) = 0.28492000000000006
P(S = 5) = 0.211212
P(S = 6) = 0.093402
P(S = 7) = 0.022518000000000003
P(S = 8) = 0.002268
theta_3 = 0.028112449799196786
theta_4 = 0.1752873563218391
theta_5 = 0.6462585034013606
theta_6 = 0.8899082568807339
h =  0.31356 , h_hat =  0.30095267976950774
```



Comparing g) to d) we see that $\hat{h}$ is closer to the blue line, i.e. it is a better approximation. We se that both the estimations stabilizes at the approximately same number of iteration, but g) is stable at a closer level. Both uses a bit under 200 iterations before it stabilizes, but the difference before that is that the approximation in g starts a bit belove the true h before stbailizing and in d, it is the opposite.

h

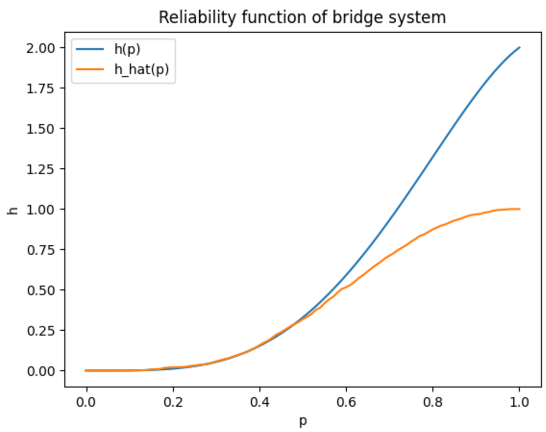
We know that:
h(p) = p$_3p_6(p_1 \sqcup p_2)(p_4 \sqcup p_5)(p_7 \sqcup_8)$
$+ p_3(1-p_6)(p_1 \sqcup p_2)((p_4p_7) \sqcup (p_5p_8))$
$+ (1-p_3)p_6((p_1p_4) \sqcup (p_2p_5))(p_7 \sqcup p_8)$
$+ (1-p_3)(1-p_6)((p_1p_4p_7) \sqcup (p_3p_5p_8))$
Setting $p_1, ..., p_8 = p$ gives us:
h(p) = p*p((p $\sqcup p)(p \sqcup p)(p \sqcup p))$
$+ p(1-p)(p \sqcup p)((p * p) \sqcup (p * p))$
$+ (1-p)p((p * p) \sqcup (p * p))(p \sqcup p)$
$+ (1-p)(1-p)((p * p * p) \sqcup (p * p * p))$
$= p^2((p \sqcup p)(p \sqcup p)(p \sqcup p))$
$+ (p-p^2)(p \sqcup p)((p^2) \sqcup (p^2))$
$+ (p-p^2)((p^2 \sqcup p^2))(p \sqcup p)$
$+ (1-p)^2((p^3 \sqcup p^3))$
$= p^2((p \sqcup p)(p \sqcup p)(p \sqcup p)) + 2p(1-p)(p \sqcup p)(p^2 \sqcup p^2) + (1-p)^2((p^3 \sqcup p^3))$
Which is what we wanted, and the statement is proved.

i

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

# Number of simulations
num_sims = 1000

# Random number generator seed. Set to 'None' for a random sequence
seed_num = 1234
gen = np.random.default_rng(seed = seed_num)

# Number of points
num_points = 101

# Save plot to file or not?
save_plot = True

# Name of system
sys_name = "bridge"

# Number of components:
n = 8

def coprod(x, y):
    return x + y - x*y
```

```
def phi(x):
    return x[3] * x[6] * coprod(x[1], x[2]) * coprod(x[4], x[5]) * coprod(x[7], x[8]) + x[3]


def hh(p):
    component_prob = p**2 * coprod(p, p) * coprod(p, p) * coprod(p, p) + 2 * p * (1 - p) * c
    return component_prob

X = np.zeros(n + 1, dtype = int)        # The component state variables (X[0] is not used)

p = np.linspace(0, 1, num_points)
T = np.zeros(num_points, dtype = int)

for _ in range(num_sims):
    U = gen.uniform(0.0, 1.0, n + 1)    # Uniform variables (U[0] is not used)
    for j in range(num_points):
        for i in range(1, n + 1):
            if U[i] <= p[j]:
                X[i] = 1
            else:
                X[i] = 0
        T[j] += phi(X)

h_hat = [T[i] / num_sims for i in range(num_points)]

h = np.zeros(num_points)                                # True reliability function

for j in range(num_points):
    h[j] = hh(p[j])


plt.plot(p, h, label='h(p)')
plt.plot(p, h_hat, label='h_hat(p)')
plt.xlabel('p')
plt.ylabel('h')
plt.title('Reliability function of ' + sys_name + " system")
plt.legend()
if save_plot:
    plt.savefig("crude/" + sys_name + ".pdf")
plt.show()
```

Which produced this plot:

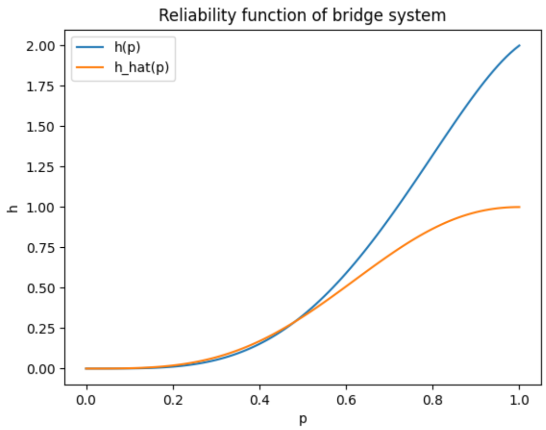


j

```
    from scipy.stats import binom
import matplotlib.pyplot as plt
import numpy as np

# Number of simulations
num_sims = 1000

# Random number generator seed. Set to 'None' for a random sequence
seed_num = 1234
gen = np.random.default_rng(seed = seed_num)

# Number of points
num_points = 101

# Save plot to file or not?
save_plot = True

# Name of system
sys_name = "bridge"

# Number of components:
n = 8

def coprod(x, y):
    return x + y - x*y

def phi(x):
    return x[3] * x[6] * coprod(x[1], x[2]) * coprod(x[4], x[5]) * coprod(x[7], x[8]) + x[3]
```

```
def hh(p):
    component_prob = p**2 * coprod(p, p) * coprod(p, p) * coprod(p, p) + 2 * p * (1 - p) * c
    return component_prob


C = list(range(1, n + 1))               # The component set [1, 2, ..., n]
X = np.zeros(n + 1, dtype = int)        # The component state variables (X[0] is not used)
T = np.zeros(n + 1, dtype = int)        # T[s] counts the number of random path sets of size

for _ in range(num_sims):
    for i in range(1, n + 1):
        X[i] = 0
    sys_state = phi(X)                  # phi(0,...0) will always be zero unless we have a t
    T[0] += sys_state
    gen.shuffle(C)                      # Generate a random permutation of the component set
    for i in range(1, n + 1):
        X[C[i-1]] = 1
        if sys_state == 0:              # If sys_state = 1 already, we know phi(X) = 1 since
            sys_state = phi(X)
        T[i] += sys_state

s_values = list(range(n + 1))                           # Set of possible values of S = X[1]

theta_hat = [T[s] / num_sims for s in s_values]         # theta_hat[s] = Estimated condition
print(theta_hat)

p = np.linspace(0, 1, num_points)
h = np.zeros(num_points)                                # True reliability function
h_hat = np.zeros(num_points)                            # Estimated reliability function

for j in range(num_points):
    h[j] = hh(p[j])
    dist = [binom.pmf(s, n, p[j]) for s in s_values]
    for s in range(n + 1):
        h_hat[j] += theta_hat[s] * dist[s]

plt.plot(p, h, label='h(p)')
plt.plot(p, h_hat, label='h_hat(p)')
plt.xlabel('p')
plt.ylabel('h')
plt.title('Reliability function of ' + sys_name + " system")
plt.legend()
if save_plot:
    plt.savefig("cmc/" + sys_name + ".pdf")
plt.show()
```

Which produces this output:



This is almost exactly the same as in i, but around p = 0.6, j is a bit smoother.