Ron S. Kenett, Shelemyahu Zacks, Peter Gedeck

Industrial Statistics: A Computer Based Approach with Python

Solutions

October 30, 2022

Springer Nature

Contents

Chapter 1
Introduction to industrial statistics

Solution 1.1 Examples of 100% inspection:

1. Airplane pilots rely heavily on checklists before takeoff. These lists provide a systematic check on safety and functional items that can be verified by relatively simple pilot inspection. The lists are used in every flight.
2. The education system relies on tests to evaluate what students learn in various courses and classes. Tests are administered to all students attending a class. The grades received on these tests reflect the student's ability, the course syllabus and training material, the instructor's ability, the classroom environment, etc.
3. Production lines of electronic products typically include automatic testers that screen products as PASS or FAIL. Failed products are directed to rework departments where the problems are diagnosed and fixed before the products are retested.

Solution 1.2 Possible articles to evaluate are:

- `https://www.industryweek.com/operations/quality/article/21213860/quality-initiatives-deserve-better-from-industry-40`
- `https://www.nytimes.com/2017/03/13/upshot/lousy-customer-service-a-better-way-in-health-care.html`
- `https://www.nytimes.com/2007/02/08/business/08scene.html`
- `https://www.wsj.com/articles/BL-CIOB-6959`
- `https://www.newsweek.com/trends-opportunities-threats-manufacturing-operations-management-software-market-1696962`

Solution 1.3 Examples of production systems:

a) *Continuous flow production*: Polymerization is used in the petroleum industry and in the manufacture of synthetic rubber. In polymerization, a reaction occurs in which two or more molecules of the same substance combined to form a compound from which the original substance may or may not be regenerated. Typical parameters affecting the process are feed rate, polymerizer temperature, and sludge levels in the separator kettles. Typical measured responses are yield, concentration, color, and texture.

b) *Discrete mass production*: A college or university is, in fact, a discrete mass production system where students are acquiring knowledge and experience through various combinations of classes, laboratories, projects, and homework assignments.
c) *Job shop*: Modern print houses are typical job shop operations. Customers provide existing material, computer files, or just an idea. The print house is responsible for producing hard copies in various sizes, colours, and quantities. The process involves several steps combining human labor with machine processing.

Solution 1.4 Sample inspection for acceptance sampling procedures determining the quality of batches of units.

Solution 1.5 a) *A school system?*: In most cases, school systems rely on individual learning combined with 100% inspection. Many schools also encourage cooperative learning efforts such as team projects.
b) *A military unit?*: Traditionally, this is a classical organization operating with strict regulations and 100% inspection. Improvements are achieved through training and exercises with comprehensive performance reviews.
c) *A football team?*: Serious sports teams invest huge efforts in putting together a winning team combination. This includes screening and selection, team building exercises, on-going feedback during training and games, and detailed analysis and review of self and other teams performance.

Solution 1.6 Examples of how you personally apply the four management approaches:

a) As a student:With an old car that you maintain with a “don’t fix what ain’t broke" strategy you are fire fighting. Reviewing your bank account statements is typically done on a 100% basis. Serious learning involves on going process control to assure success in the final exam. Proper preparation, ahead of time, through self study and learning from the experience of others is a form of ensuring quality by design.
b) In your parent’s home: This is a personal exercise for self thinking.
c) With your friends: This is also a personal exercise for self thinking.

Solution 1.7 Three case studies are provided in slides 20, 21 and 22 in `https://ceeds.unimi.it/wp-content/uploads/2020/02/Kenett_Analytics_2020-1.pdf`. Their information quality assessment is provided in slides 45-59.

Chapter 2
Basic Tools and Principles of Process Control

Import required modules and define required functions

```
import numpy as np
import pandas as pd
import pingouin as pg
from scipy import stats
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.stats as sms
from statsmodels.graphics.mosaicplot import mosaic
import seaborn as sns
import matplotlib.pyplot as plt
import pwlf

import mistat
```

Solution 2.1 The following Python code creates the chart shown in Fig. 2.1.

```
oelect = mistat.load_data('OELECT')
qcc = mistat.QualityControlChart(oelect, qcc_type='xbarone',
                                 std_dev='SD')
qcc.plot()
plt.show()
```

It seems that the data are randomly distributed around a mean value of 219.25.

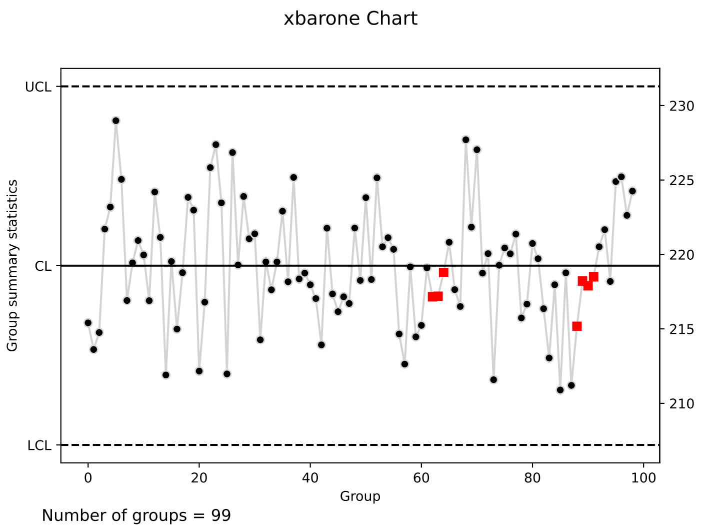
Solution 2.2 The following Python code creates the chart shown in Fig. 2.2.

```
steelrod = mistat.load_data('STEELROD')
qcc = mistat.QualityControlChart(steelrod, qcc_type='xbarone',
                                 std_dev='SD')
qcc.plot()
plt.show()
```

It seems that the data are randomly distributed around a mean value of 19.89.

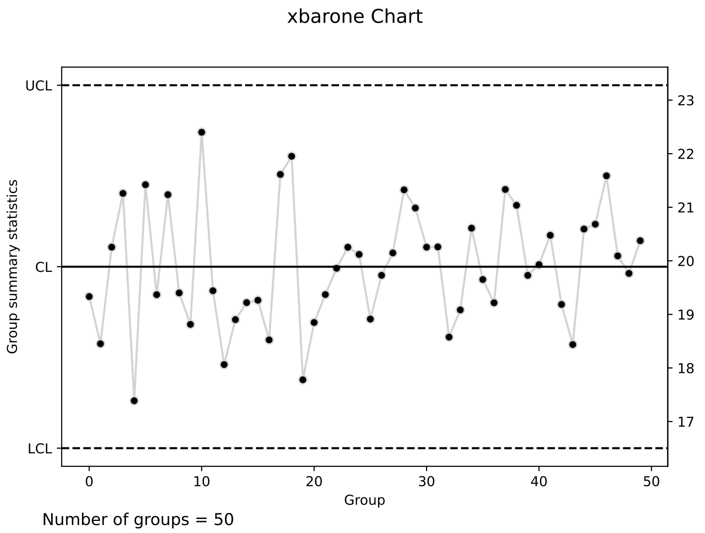
Solution 2.3 No patterns of non-randomness are apparent.

Solution 2.4 The following Python code creates the chart shown in Fig. 2.3.



```
Number of groups = 99
Center = 219.25          LCL = 207.21        Number beyond limits = 0
StdDev = 4.0142          UCL = 231.29        Number violating runs = 7
```

Fig. 2.1: X-barone chart of **OELECT.csv** dataset



```
Number of groups = 50
Center = 19.89           LCL = 16.503        Number beyond limits = 0
StdDev = 1.1291          UCL = 23.278        Number violating runs = 0
```

Fig. 2.2: X-barone chart of the **STEELROD.csv** dataset

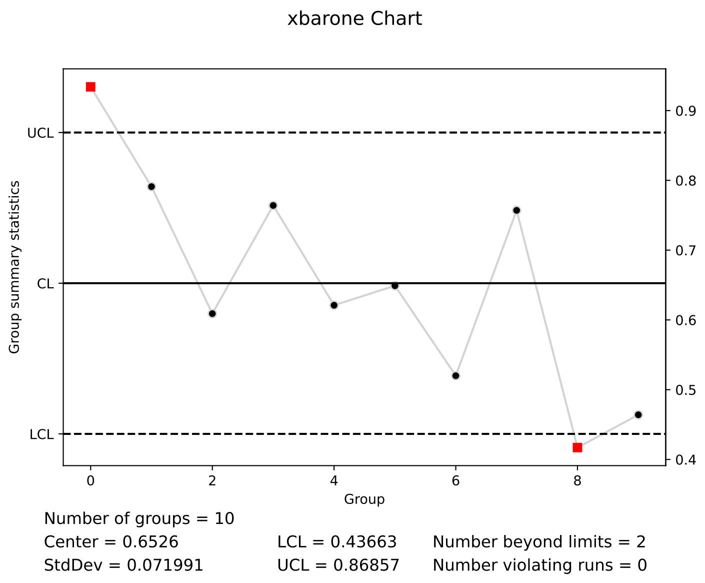


Fig. 2.3: X-barone chart of the **OTURB2.csv** dataset

```
oturb2 = mistat.load_data('OTURB2')
# print(oturb2)
sd = np.sqrt(oturb2['xbar'].var() / 5)
center = oturb2['xbar'].mean()
print(sd, center)
qcc = mistat.QualityControlChart(oturb2['xbar'], qcc_type='xbarone',
        center=center, std_dev=sd)
qcc.plot()
plt.show()
```

```
0.07199111056234651 0.6526
```

We can see a downward trend in the data.

Solution 2.5 If $\bar{X}_n \sim N(\mu - \sigma, \frac{\sigma}{\sqrt{n}})$ the proportion of points expected to fall outside the control limits is

$$\pi_n \equiv \Pr\left\{\bar{X}_n < \mu - \frac{3\sigma}{\sqrt{n}}\right\} + \Pr\left\{\bar{X}_n > \mu + \frac{3\sigma}{\sqrt{n}}\right\}.$$

(a) For $n = 4$, $\pi_n = 0.159$; **(b)** For $n = 6$, $\pi_n = 0.291$; **(c)** For $n = 9$, $\pi_n = 0.500$.

Solution 2.6 In Python:

```
oelect = mistat.load_data('OELECT')

qcc = mistat.QualityControlChart(oelect, qcc_type='xbarone',
                                 std_dev='SD')
```

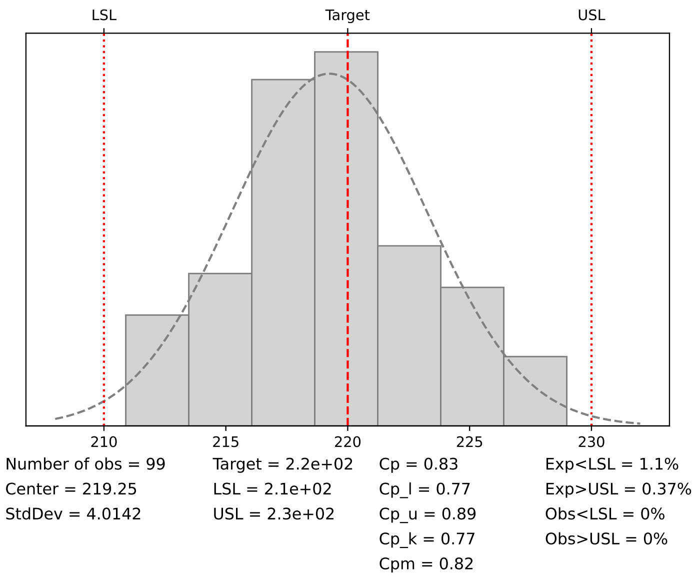


Fig. 2.4: Process capability analysis of **OELECT.csv** dataset

```
pc = mistat.ProcessCapability(qcc, spec_limits = [210, 230])
pc.plot()
plt.show()
pc.summary()
```

```
Process Capability Analysis

Number of obs = 99           Target = 220.00
       Center = 219.25          LSL = 210.00
       StdDev = 4.014219        USL = 230.00

Capability indices:

        Value     2.5%   97.5%
Cp      0.8304  0.7142  0.9463
Cp_l    0.7679  0.6622  0.8737
Cp_u    0.8928  0.7743  1.0113
Cp_k    0.7679  0.6420  0.8939
Cpm     0.8162  0.7007  0.9315

Exp<LSL   1%   Obs<LSL   0%
Exp>USL   0%   Obs>USL   0%
```

Performing the capability analysis (see Fig. 2.4) we found that $C_p = 0.83$. Although $C_p = 0.83$, there is room for improvement by designing the process with reduced variability. See Chapter 5.

Solution 2.7 Using the `mistat` package, we get:

```
oelect = mistat.load_data('OELECT')
qcc = mistat.QualityControlChart(oelect, qcc_type='xbarone',
    std_dev=4.004, center=219.25)
pc = mistat.ProcessCapability(qcc, spec_limits = [210, 230],
                              confidence_level=0.975)
pc.summary()
pc = mistat.ProcessCapability(qcc, spec_limits = [210, 230],
                              confidence_level=0.95)
pc.summary()
```

```
Process Capability Analysis

Number of obs = 99           Target = 220.00
       Center = 219.25          LSL = 210.00
       StdDev = 4.004000        USL = 230.00

Capability indices:

        Value     2.5%   97.5%
Cp      0.8325  0.7002  0.9663
Cp_l    0.7701  0.6438  0.8963
Cp_u    0.8949  0.7535  1.0364
Cp_k    0.7701  0.6257  0.9144
Cpm     0.8183  0.6868  0.9513

Exp<LSL   1%   Obs<LSL   0%
Exp>USL   0%   Obs>USL   0%
Process Capability Analysis

Number of obs = 99           Target = 220.00
       Center = 219.25          LSL = 210.00
       StdDev = 4.004000        USL = 230.00

Capability indices:

        Value     2.5%   97.5%
Cp      0.8325  0.7161  0.9488
Cp_l    0.7701  0.6641  0.8760
Cp_u    0.8949  0.7762  1.0136
Cp_k    0.7701  0.6438  0.8963
Cpm     0.8183  0.7025  0.9339

Exp<LSL   1%   Obs<LSL   0%
Exp>USL   0%   Obs>USL   0%
```

With $\bar{X} = 219.25$, $S = 4.004$, $n = 99$, $\xi_U = 230$ and $\xi_L = 210$ we obtained $\hat{C}_{pl} = 0.77$, $\hat{C}_{pu} = 0.895$ and $\hat{C}_{pk} = 0.77$.

The `mistat` implementation uses the normal distribution to calculate confidence intervals. Using the F-distribution with $F_{.975}[1, 98] = 5.1982$, we get:

```
def confidenceLimitSL(Cp, n, alpha):
    F = stats.f(1, n-1).ppf(1-(1-alpha)/2)
    a = np.sqrt(F/n) * np.sqrt(Cp**2/2 + (1 - F/(2*n))/9)
    b = 1 - F/(2*n)
    return (Cp - a) / b, (Cp + a )/ b

n = len(oelect)
rho_1L, rho_1U = confidenceLimitSL(pc.Cp_l, n, 0.95)
rho_2L, rho_2U = confidenceLimitSL(pc.Cp_u, n, 0.95)
print(rho_1L, rho_1U)
print(rho_2L, rho_2U)
```

```
0.6413037077789147 0.9402124117777945
0.7514350333838329 1.0865431596145048
```

The confidence intervals for C_{pl} and C_{pu}, at 0.95 confidence level, are:

	Lower Limit	Upper Limit
C_{pl}	0.6413	0.9402
C_{pu}	0.7514	1.0865
C_{pk}	0.6413	0.9402

The 0.95-Confidence interval for C_{p_k} is $(0.6413, 0.9402)$.

Solution 2.8 In Python:

```
steelrod = mistat.load_data('STEELROD')
qcc = mistat.QualityControlChart(steelrod, qcc_type='xbarone',
                                 std_dev='SD')
pc = mistat.ProcessCapability(qcc, spec_limits = [19, 21],
                              confidence_level=0.95)
pc.summary()
```

```
Process Capability Analysis

Number of obs = 50           Target = 20.00
       Center = 19.89           LSL = 19.00
       StdDev = 1.129109        USL = 21.00

Capability indices:

        Value     2.5%   97.5%
Cp      0.2952  0.2369  0.3534
Cp_l    0.2628  0.1738  0.3518
Cp_u    0.3276  0.2329  0.4224
Cp_k    0.2628  0.1568  0.3688
Cpm     0.2938  0.2361  0.3514

Exp<LSL  22%   Obs<LSL  22%
Exp>USL  16%   Obs>USL  20%
```

Using the **STEELROD.csv** data with $\xi_L = 19$ and $\xi_U = 21$ we obtained $\hat{C}_p = 0.295$, $\hat{C}_{pu} = 0.328$, $\hat{C}_{pl} = 0.263$ and $\hat{C}_{pk} = 0.263$. We use the function from the previous exercise to calculat the confidence interval.

```
n = len(steelrod)
rho_1L, rho_1U = confidenceLimitSL(pc.Cp_l, n, 0.95)
rho_2L, rho_2U = confidenceLimitSL(pc.Cp_u, n, 0.95)
print(rho_1L, pc.Cp_l, rho_1U)
print(rho_2L, pc.Cp_u, rho_2U)
```

```
0.14851207708654524 0.26280004915567645 0.40677992256113854
0.2084480020651775 0.32763581095436156 0.4838408510212421
```

A 0.95-confidence interval for C_{pk} is $(0.1485, 0.4068)$.

Solution 2.9 [1] Simulate 20 observations at the low level of the 7 control parameters.

```
settings = {'m': 30, 's': 0.005, 'k': 1000, 't': 290,
            'p0': 90_000, 'v0': 0.002, 't0': 340}
```

[1] TODO: Adjusted the exercise to version 2 of the piston simulator - please check that it is reasonable.

```
simulator = mistat.PistonSimulator(n_simulation=20, n_replicate=1, seed=1,
                                   **settings)
Ps = simulator.simulate()
cycleTime = mistat.qcc_groups(Ps['seconds'], Ps['group'])

qcc = mistat.QualityControlChart(cycleTime, std_dev= Ps['seconds'].std())
print(f'Mean    {qcc.center:.4f}')
print(f'Std.Dev {qcc.std_dev:.4f}')
```

```
Mean    0.0257
Std.Dev 0.0078
```

If all the 7 control parameters are at their low level, the mean cycle time of 20 observations is $\bar{X}_{20} = 0.0257$. This value is outside the specification limits, 0.04 and 0.06. Thus the process, at the low parameter values is incapable of satisfying the specs. The standard deviation is $S_{20} = 0.0078$.

(a) Calclate the capability index C_p

```
pc = mistat.ProcessCapability(qcc, spec_limits = [0.04, 0.06],
                              confidence_level=0.95)
print(f'C_p {pc.Cp:.3f}')
```

```
C_p 0.425
```

If the process mean can be moved to 0.05, the C_p value is 0.425.

(b) [2]

(c) Under the high values of the seven control parameters we get

```
settings = {'m': 60, 's': 0.02, 'k': 5_000, 't': 296,
            'p0': 110_000, 'v0': 0.01, 't0': 360}

simulator = mistat.PistonSimulator(n_simulation=20, n_replicate=1, seed=1, **settings)
Ps = simulator.simulate()
cycleTime = mistat.qcc_groups(Ps['seconds'], Ps['group'])
qcc = mistat.QualityControlChart(cycleTime, std_dev= Ps['seconds'].std())
print(f'Mean    {qcc.center:.4f}')
print(f'Std.Dev {qcc.std_dev:.4f}')
```

```
Mean    0.0568
Std.Dev 0.0037
```

$\bar{X}_{20} = 0.0568$ and $S_{20} = 0.0037$. The mean is within the specification limit 0.05 ± 0.01.

For these values, we get the following for the capability indices.

```
pc = mistat.ProcessCapability(qcc, spec_limits = [0.04, 0.06],
                              confidence_level=0.95)
pc.summary()
```

```
Process Capability Analysis

Number of obs = 20           Target = 0.05
       Center = 0.06            LSL = 0.04
       StdDev = 0.003720        USL = 0.06
```

[2] TODO: Is 0.425 low? The limits of C_{pk} are -0.3657 -0.8503. What does this mean?

```
Capability indices:

        Value     2.5%   97.5%
Cp      0.8960  0.6135  1.1782
Cp_l    1.5090  1.0881  1.9298
Cp_u    0.2830  0.1390  0.4270
Cp_k    0.2830  0.1115  0.4546
Cpm     0.4281  0.2543  0.6020

Exp<LSL   0%   Obs<LSL   0%
Exp>USL  20%   Obs>USL  10%
```

The values are: $C_p = 0.890$ and $C_{pk} = 0.283$.

(d) Under the normal approximation the confidence interval for C_{pk}, at level 0.95, is (0.1115, 0.4546).

(e) As mentioned above, the gas turbine at low control levels is incapable of satisfying the specifications of 0.05 ± 0.01 seconds. At high levels its C_{pk} is not greater than 0.282. In Chapter 5 we study experimental methods for finding the combination of control levels, which maximizes the capability.

Solution 2.10 **(a)** 1.8247 calculated in Python

```
std = 0.015

nominal_value = 1.8 - stats.norm(0, std).isf(0.95)
nominal_value
```

```
1.8246728044042722
```

(b) 1.8063 and 1.8430 [3]

(c) $C_{pk} = 0.44$.

```
Cpk = (1.82 - 1.80) / (3 * std)
print(Cpk)
```

```
0.44444444444444486
```

Solution 2.11 [4] **(a)** $\bar{\bar{X}} = \frac{10950}{30} = 365$, $\bar{S} = 1.153$, $n = 4$, $\hat{\sigma} = \bar{S}/c(n) = 1.252$. The control limits for $\bar{X}_4$ are

$$UCL = 365 + 3 \times \frac{1.252}{\sqrt{4}} = 366.878 \quad \text{and} \quad LCL = 365 - 3 \times \frac{1.252}{\sqrt{4}} = 363.122.$$

(b) When $X \sim N(\mu = 365, \sigma = 1.252)$,

$$1 - \Pr\{365 < X < 375\} = 1 - \Phi\left(\frac{375 - 365}{1.252}\right) + \Phi\left(\frac{365 - 365}{1.252}\right) = 0.5.$$

(c) When $X \sim N(\mu = 370, \sigma = 1.252)$,

[3] TODO: how is this calculated?

[4] TODO: Check how this is calculated.

$$1 - \Pr\{365 < X < 375\} = 2 \times \left(1 - \Phi\left(\frac{5}{1.252}\right)\right) = 0.000065.$$

(d) $UCL = 370 + 3 \times \frac{1.252}{\sqrt{4}} = 371.878$ and $LCL = 370 - 3 \times \frac{1.252}{\sqrt{4}} = 368.122$.

(e) When $\bar{X}_4 \sim N(\mu = 365, \sigma = 1.252/\sqrt{4})$,

$\Pr\{368.122 < \bar{X}_4 < 371.878\} =$

$$\Phi\left(\frac{371.878 - 365}{1.252/\sqrt{4}}\right) - \Phi\left(\frac{368.122 - 365}{1.252/\sqrt{4}}\right) = 0.$$

(f) The process capability before and after the adjustment is

	Before	After
C_p	1.331	1.331
C_{pk}	0	1.331

Solution 2.12 **(a)** Fig. 2.5 is created using the following code

```
october = pd.DataFrame([
  ['Missing component', 293],
  ['Wrong component', 431],
  ['Too much solder', 120],
  ['Insufficient solder', 132],
  ['Failed component', 183],
], columns=['Issue', 'Count'])
november = pd.DataFrame([
  ['Missing component', 34],
  ['Wrong component', 52],
  ['Too much solder', 25],
  ['Insufficient solder', 34],
  ['Failed component', 18],
], columns=['Issue', 'Count'])

def makeParetoChart(data, ax, title):
  paretoChart = mistat.ParetoChart(data['Count'], labels=data['Issue'])
  paretoChart.plot(rotation=30, ha='right', ax=ax)
  ax.set_title(title)

fig, axes = plt.subplots(ncols=2, figsize=(8,4))
makeParetoChart(october, axes[0], 'October')
makeParetoChart(november, axes[1], 'November (2nd week)')
fig.suptitle('')
plt.tight_layout()
plt.show()
```

(b) Using the test described in Section 2.6 we find that the improvement team has produced significant differences in the type of nonconformities. In particular, the difference in the 'Insufficient solder' category is significant at the 1% level and the difference in the 'Too much solder' category is significant at the 10% level (see Table 2.4). The computations involved are summarized in Table 2.1.

Solution 2.13 From the data, $\bar{\bar{X}} = 15.632$ ppm and $\bar{R} = 3.36$. An estimate of the process standard deviation is $\hat{\sigma} = \bar{R}/d(5) = 1.4446$.

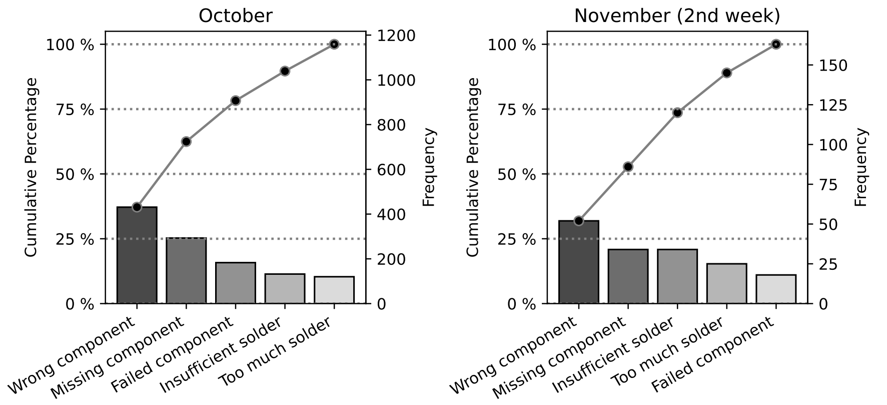


Fig. 2.5: Pareto chart noconformities in October and the second week of November

Table 2.1: Computations for Pareto significance analysis (Exercise 2.12)

Category	October	p	November	Expected	(Obs. - Exp.)	Z
Missing	293	0.253	34	41.239	-7.239	-1.304
Wrong	431	0.372	52	60.636	-8.636	-1.399
Too Much	120	0.103	25	16.789	8.211	2.116
Insufficient	132	0.114	34	18.582	15.418	3.800
Failed	183	0.158	18	25.754	-7.754	-1.665
Total	1159		163			

(a) $C_{pu} = \frac{18-15.632}{3*\hat{\sigma}} = 0.546$.
(b) The proportion expected to be out of spec is 0.051.
(c) The control limits for $\bar{X}$ are $LCL = 13.694$ and $UCL = 17.570$. The control limits for R_5 are $LCL = 0$ and $UCL = 7.106$.

Solution 2.14 Part I (a) Fig. 2.6 is created using the following code.

```
settings = {'m': 30, 's': 0.005, 'v0': 0.002, 'k': 1000,
            'p0': 90_000, 't': 290, 't0': 340}

simulator = mistat.PistonSimulator(n_simulation=20, n_replicate=5, seed=1, **settings)
Ps = simulator.simulate()

# Add 0.02 seconds to last 50 simulation results
Ps.loc[50:,'seconds'] = Ps.loc[50:,'seconds'] + 0.02

def makeQCCplot(data, reference, qcc_type, title):
  # convert to groups
  data = mistat.qcc_groups(data['seconds'], data['group'])
  reference = mistat.qcc_groups(reference['seconds'], reference['group'])
  # calculate control limits based on reference data
  qcc_ref = mistat.QualityControlChart(reference, qcc_type=qcc_type)
  qcc = mistat.QualityControlChart(data, qcc_type=qcc_type,
    center=qcc_ref.center, limits=qcc_ref.limits)
  return qcc.plot(title=title)

# create xbar and R control charts
reference = Ps.iloc[:50, ]

makeQCCplot(Ps, reference, 'xbar', 'for cycleTime')
plt.show()
makeQCCplot(Ps, reference, 'R', 'for cycleTime')
plt.show()
```

See the control charts in Fig. 2.6. The increase in cycle times after group 9 is so large that the means of the subgroup are immediately out of the control limits. The shift has no influence on the ranges within subgroups.

Part II (b) Fig. 2.7 is created using the following code.

```
# Ps['seconds'] = random.sample(Ps['seconds'], len(Ps['seconds']))

# alternative approach
# Ps['seconds'] = Ps['seconds'].sample(frac=1).values

makeQCCplot(Ps, Ps, 'xbar', 'for cycleTime')
plt.show()
makeQCCplot(Ps, Ps, 'R', 'for cycleTime')
plt.show()
```

See the control charts in Fig. 2.7. Due to the randomization of the cycle times, the increased values are distributed over all groups and the mean cycle time increases by 0.01 The variability as shown in the R chart is is increased too.

Solution 2.15 Part I (a): We run the simulation with all 7 control parameters at their lowest value. We obtained $\bar{\bar{X}} = 0.0234$, $\bar{S} = 0.0068$.

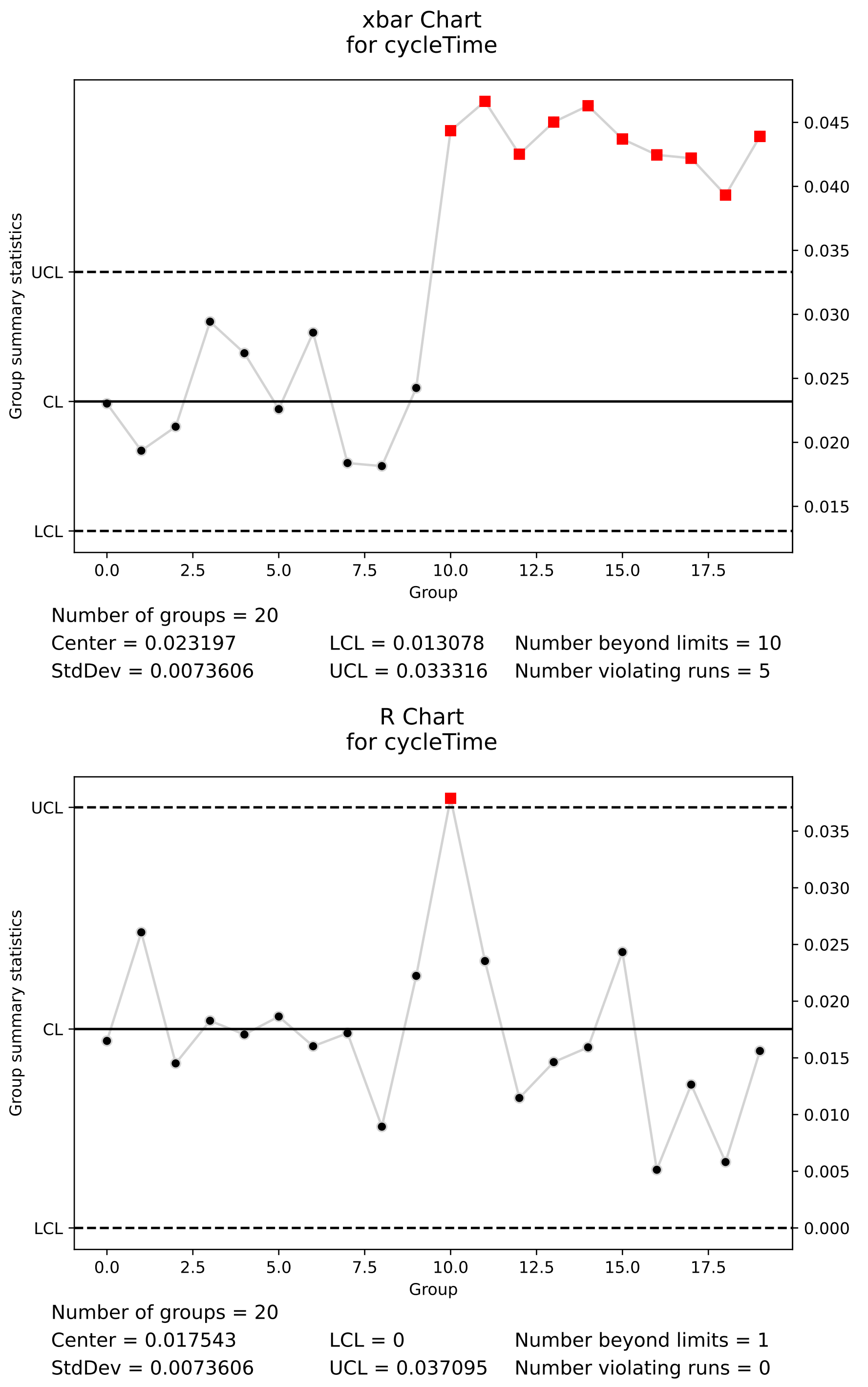
```
settings = {'m': 30, 's': 0.005, 'v0': 0.002, 'k': 1000,
            'p0': 90_000, 't': 290, 't0': 340}

simulator = mistat.PistonSimulator(n_simulation=20, n_replicate=5, seed=1, **settings)
Ps = simulator.simulate()
data = mistat.qcc_groups(Ps['seconds'], Ps['group'])
for qcc_type in ('xbar', 'S'):
  qcc = mistat.QualityControlChart(data, qcc_type=qcc_type)
  print(f'{qcc_type:4s} Center {qcc.center:.4f} Control limits ' +
        f'[{qcc.limits.LCL[0]:.5f}, {qcc.limits.UCL[0]:.4f}]')
```

```
xbar Center 0.0234 Control limits [0.01354, 0.0333]
S    Center 0.0068 Control limits [0.00000, 0.0141]
```

Part II (b): We obtained $\bar{\bar{X}} = 0.0253$, $\bar{S} = 0.0093$.

```
simulator = mistat.PistonSimulator(n_simulation=20, n_replicate=10, seed=1, **settings)
Ps = simulator.simulate()
data = mistat.qcc_groups(Ps['seconds'], Ps['group'])
for qcc_type in ('xbar', 'S'):
  qcc = mistat.QualityControlChart(data, qcc_type=qcc_type)
  print(f'{qcc_type:4s} Center {qcc.center:.4f} Control limits ' +
        f'[{qcc.limits.LCL[0]:.5f}, {qcc.limits.UCL[0]:.4f}]')
```

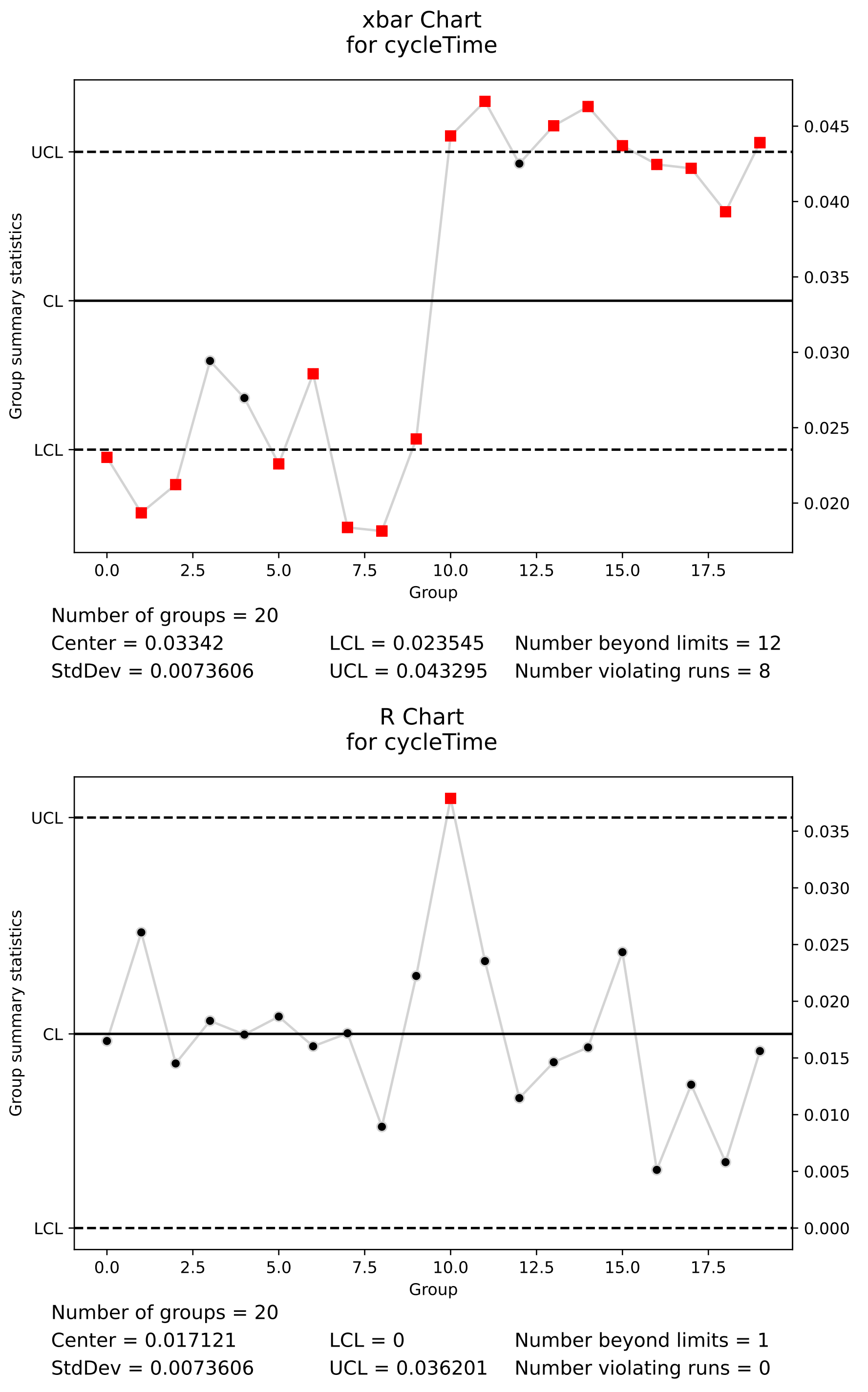
```
xbar Center 0.0253 Control limits [0.01623, 0.0344]
S    Center 0.0093 Control limits [0.00264, 0.0159]
```



```
Number of groups = 20
Center = 0.023197          LCL = 0.013078     Number beyond limits = 10
StdDev = 0.0073606         UCL = 0.033316     Number violating runs = 5
```

```
Number of groups = 20
Center = 0.017543          LCL = 0            Number beyond limits = 1
StdDev = 0.0073606         UCL = 0.037095     Number violating runs = 0
```

Fig. 2.6: Control charts for time shift Piston simulation (Excercise 2.14)



```
Number of groups = 20
Center = 0.03342          LCL = 0.023545    Number beyond limits = 12
StdDev = 0.0073606        UCL = 0.043295    Number violating runs = 8
```

```
Number of groups = 20
Center = 0.017121         LCL = 0           Number beyond limits = 1
StdDev = 0.0073606        UCL = 0.036201    Number violating runs = 0
```

Fig. 2.7: Control charts for randomized cycle time in Piston simulation (Excercise 2.14)

(c) We see that the control limits of $\bar{X}_{10}$ and of S_{10} are closer to the centerline than those of $\bar{X}_5$ and S_5, respectively.

Solution 2.16 (a) Use code like the following to explore piecewise linear fits to the sensor data using different number of knots.

```
data = mistat.load_data('PROCESS_SEGMENT')

def sensorData(data, label):
    series = data[label]
    return pd.DataFrame({
        'Time': np.arange(len(series)),
        'values': series,
    })

sensorX = sensorData(data, 'X')
sensorZ = sensorData(data, 'Z')

def fitPiecewiseLinearFit(sensor, knots):
    model = pwlf.PiecewiseLinFit(sensor['Time'], sensor['values'], degree=1)
    model.fit(knots)
    return model

modelX = fitPiecewiseLinearFit(sensorX, 6)
modelZ = fitPiecewiseLinearFit(sensorZ, 3)

def plotPiecewiseLinearFit(sensor, model, ax, label):
    for bp in model.fit_breaks[1:-1]:
        ax.axvline(bp, color='lightgrey')
    ax.scatter(sensor['Time'], sensor['values'], color='grey', alpha=0.5)
    ax.plot(sensor['Time'], model.predict(sensor['Time']), color='black')
    ax.set_xlabel('Time')
    ax.set_ylabel(label)
    return ax
fig, axes = plt.subplots(ncols=2, figsize=(8,4))
plotPiecewiseLinearFit(sensorX, modelX, axes[0], 'Sensor X')
plotPiecewiseLinearFit(sensorZ, modelZ, axes[1], 'Sensor Z')
plt.tight_layout()
plt.show()
```

The figure shows results for sensor X and sensor Z data. By trying different number of segments, we can see that we require 6 segments for the sensor X data while 3 segments are sufficient for sensor Z. In contrast to the step function fit, the piecewise linear function doesn't enforce the linear segments to have a slope of 0. While visually, we can see that the segment that is characteristics for a progress under control are essentially horizontal, it is useful to analyze the individual segments. The slope can be extracted from the model using the method *modelX.calc_slopes()*. We get:

Sensor X

	Range	slope	prediction
Segment 1	Time < 894.3	35.30	26.3–4634.1
Segment 2	130.5 ≤ Time < 894.3	-0.02	4634.1–4619.8
Segment 3	894.3 ≤ Time < 910.7	32.89	4619.8–5158.9
Segment 4	910.7 ≤ Time < 1426.3	-0.43	5158.9–4936.1
Segment 5	1426.3 ≤ Time < 1502.6	-63.67	4936.1–77.9
Segment 6	1502.6 ≤ Time	-0.02	77.9–70.0

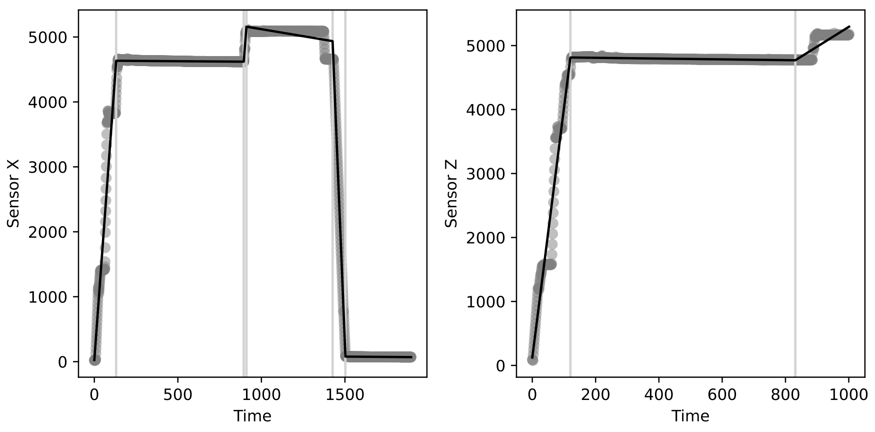


Fig. 2.8: Piecewise linear fits of sensor X and sensor Z data.

Sensor Z

	Range	slope	prediction
Segment 1	Time < 831.1	38.97	125.2–4813.7
Segment 2	120.3 ≤ Time < 831.1	-0.06	4813.7–4770.3
Segment 3	831.1 ≤ Time	3.09	4770.3–5295.3

The slope of the longer second segment is almost zero for both sensor data. The segment analysis can therefore be used.

(b) The step function fits of Section 2.8.2 required 9 segments for sensor X and 6 segments for sensor Z to describe the stable phase sufficiently. With the piecewise linear fit, we require fewer segments.

(c) The piecewise linear fit can model a linear ramp-up phase like we see at the beginning of both run charts better than a step function fit. It also allows us identifying and estimating drifts in the run charts.

Chapter 3
Advanced Methods of Statistical Process Control

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats
from scipy.special import factorial
import statsmodels.formula.api as smf
import mistat
import matplotlib.pyplot as plt
```

Solution 3.1 First define a function to calculate the probability of runs:

```
from scipy.special import binom
def probRuns(m1, m2, R):
    n = m1 + m2
    k = R // 2
    if R % 2:
        denom = binom(m1-1,k-1) * binom(m2-1,k) + binom(m1-1,k) * binom(m2-1,k-1)
        return denom / binom(n, m2)
    else:
        return 2 * binom(m1-1, k-1) * binom(m2-1, k-1) / binom(n, m2)
```

For the case of $n = 25$ and $m_2 = 10$ calculate the distribution

```
n = 25
m2 = 10
m1 = n - m2

df = pd.DataFrame({
    'R': range(0, n+1),
    'p.d.f': [probRuns(m1, m2, R) for R in range(0, n+1)],
})
df['c.d.f'] = np.cumsum(df['p.d.f'])
```

(a) We can determine the median and quantiles from the calculated c.d.f

```
cdf = df['c.d.f'].values

Q1 = np.where(cdf < 0.25)[0][-1]
```

```
Me = np.where(cdf < 0.5)[0][-1]
Q3 = np.where(cdf < 0.75)[0][-1]
print(Q1, Me, Q3)
```

```
10 12 14
```

$Q_1 = 10$, $M_e = 12$ and $Q_3 = 14$.

(b) From Eq. (9.1.3) and Eq. (9.1.4) we get $\mu_R = 13$ and $\sigma_R = 2.3452$.

```
mu_R = 1 + 2 * m1 * m2 / n
sigma_R = np.sqrt(2*m1*m2*(2*m1*m2 - n) / (n*n*(n-1)))
print(mu_R, sigma_R)
```

```
13.0 2.345207879911715
```

(c) $\Pr\{10 \le R \le 16\} = 0.8657$.

```
p = df['c.d.f'][16] - df['c.d.f'][9]
print(p)
```

```
0.8656750572082381
```

(d) Using the normal approximation $\Pr\{10 \le R \le 16\} \approx 0.8644$. Note that we set the limits to [9.5, 16.5].

```
print(stats.norm.cdf((16.5-mu_R)/sigma_R) - stats.norm.cdf((9.5-mu_R)/sigma_R))
```

```
0.8644069987336978
```

Solution 3.2 Load the data and determine number of runs information:

```
data = mistat.load_data('CYCLT')
# convert to up (1) or down (0) information relative to mean
mean_ct = np.mean(data)
runs = [1 if ct > mean_ct else 0 for ct in data]

# determine number of runs
obs_Runs = 0
current = None
for r in runs:
    if r != current:
        obs_Runs += 1
        current = r
print(f'Observed number of runs: {obs_Runs}')

# calculate expected number of runs
m1 = sum(data > mean_ct)
m2 = sum(data <= mean_ct)
n = m1 + m2
mu_R = 1 + 2 * m1 * m2 / n
print(f'Expected number of runs {mu_R:.2f}')

# determine if difference is significant
mistat.runsTest(data, cutoff=np.mean(data))
```

```
Observed number of runs: 26
Expected number of runs 25.64
```

```
Result(statistic=0.1044134517056721, pval=0.9168412481142088,
method='Runs Test', alternative='two.sided')
```

The test shows no significant deviations from randomness.

Solution 3.3 **(i)** For $n = 50$, $E\{R^*\} = \frac{2n-1}{3} = 33$.

```
n = 50
mu_Rstar = (2*n-1)/3
print(mu_Rstar)
```

```
33.0
```

(ii) Using the cycle time data, $R^* = 34$.

```
# determine direction of change up (1) or down (-1)
y = [1 if xi < xip1 else -1 for xi, xip1 in zip(data[:-1], data[1:])]

# count number of up and down runs
up = 0
down = 0
current = None
for yi in y:
    if yi == current: # no change of direction
        continue
    if yi < 0:
        down += 1
    else:
        up += 1
    current = yi
Rstar = up + down
print(Rstar, up, down)
```

```
34 17 17
```

(iii) We have $\sigma^* = 2.9269$, $\alpha_u^* = 1 - \Phi\left(\frac{1}{2.9269}\right) = 0.3663$. The deviation is not significant.

```
n = 50
mu_Rstar = (2*n-1)/3
sigma_Rstar = np.sqrt((16*n-29)/90)
print(sigma_Rstar)
alpha_L = stats.norm.cdf((Rstar-mu_Rstar)/sigma_Rstar)
alpha_U = 1 - alpha_L
print(alpha_U, alpha_L)
```

```
2.9268868558020253
0.3663034098961011 0.6336965901038989
```

(iv) The probability that a sample of size 50 will have at least one run of length 5 or longer is 0.102.

```
def expected_R_k(n, k):
    return 2 *(n*(k+1) - k*k - k + 1)/factorial(k+2)

print(expected_R_k(50, 5))
# probability to have run greater than 5
print(1 - np.exp(-expected_R_k(50, 5)))
```

```
0.10753968253968255
0.10195911479934461
```

The function *mistat.runStatistics* calculates a variety of statistics for runs.

```
mistat.runStatistics(data)
```

```
{'count': {'mu_R': 25.64, 'sigma_R': 3.4478316167038754, 'observed':
26},
 'direction': {'mu_Rstar': 33.0,
  'sigma_Rstar': 2.9268868558020253,
  'up': 17,
  'down': 17,
  'Rstar': 34,
  'alpha': [0.3663034098961011, 0.6336965901038989]}}
```

Solution 3.4 We make use of the *mistat.runStatistics* function.

```
data = mistat.load_data('YARNSTRG')
mistat.runStatistics(data)
```

```
{'count': {'mu_R': 50.92, 'sigma_R': 4.966642668235696, 'observed':
49},
 'direction': {'mu_Rstar': 66.33333333333333,
  'sigma_Rstar': 4.1779846284489315,
  'up': 32,
  'down': 32,
  'Rstar': 64,
  'alpha': [0.28825730949895256, 0.7117426905010474]}}
```

```
mistat.runsTest(data, cutoff=data.mean())
```

```
Result(statistic=-0.3865790491189181, pval=0.6990678707195341,
method='Runs Test', alternative='two.sided')
```

Solution 3.5 Define the parameter settings and create 50 simulations with 5 replications each

```
parameter = pd.DataFrame({
    'm': [60],
    's': [0.02],
    'k': [5_000],
    't': [296],
    'p0': [110_000],
    'v0': [0.01],
    't0': [360],
})
simulator = mistat.PistonSimulator(parameter=parameter, n_simulation=50, n_replicate=5, seed=1236)
# simulator = mistat.PistonSimulator(n_simulation=50, n_replicate=5, seed=1)
Ps = simulator.simulate()

# get grouped cycle times
cycleTime = mistat.qcc_groups(Ps['seconds'], Ps['group'])
mistat.runsTest(np.mean(cycleTime, axis=1), np.mean(cycleTime))
```

```
Result(statistic=-0.5610330087428953, pval=0.5747750350233831,
method='Runs Test', alternative='two.sided')
```

The test shows no significant runs. The resulting $\bar{X}$ and S charts are shown in Fig. 3.1.

```
qcc = mistat.QualityControlChart(cycleTime, qcc_type='xbar')
ax = qcc.plot(title='for piston simulation at upper level')
plt.show()
qcc = mistat.QualityControlChart(cycleTime, qcc_type='S')
ax = qcc.plot(title='for piston simulation at upper level')
plt.show()
```

Solution 3.6 **(i)** See solution of Exercise 3.5 for this part of the exercise. Fig. 3.1 shows the resulting control charts.

(ii) In simulating the piston cycle times, the ambient temperature around the piston is increased 10% per cycle after the shift point, which is after the 16th sample.

```
parameter = pd.DataFrame({
    'm': [60]*50,
    's': [0.02]*50,
    'k': [5_000]*50,
    't': [296] * 16 + [296 * 1.1**i for i in range(1, 35)],
    'p0': [110_000]*50,
    'v0': [0.01]*50,
    't0': [360]*50,
})

simulator = mistat.PistonSimulator(parameter=parameter, n_simulation=50,
                                   n_replicate=5, seed=1, check=False)
Ps = simulator.simulate()
cycleTimeTshift = mistat.qcc_groups(Ps['seconds'], Ps['group'])
```

```
qcc = mistat.QualityControlChart(cycleTimeTshift, qcc_type='xbar')
ax = qcc.plot(title='for contact lens data')
plt.show()
qcc = mistat.QualityControlChart(cycleTimeTshift, qcc_type='S')
ax = qcc.plot(title='for contact lens data')
plt.show()
```

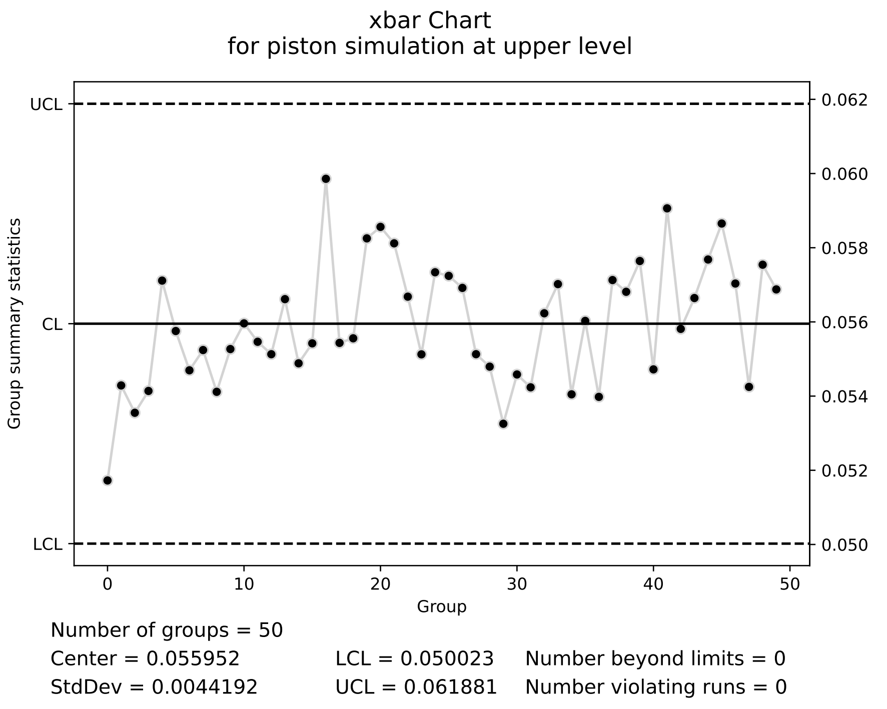
The control charts for this simulation are shown in Fig. 3.2. Both show clearly unrandom behavior.

(iii) We calculate runs statistics for both simulations.

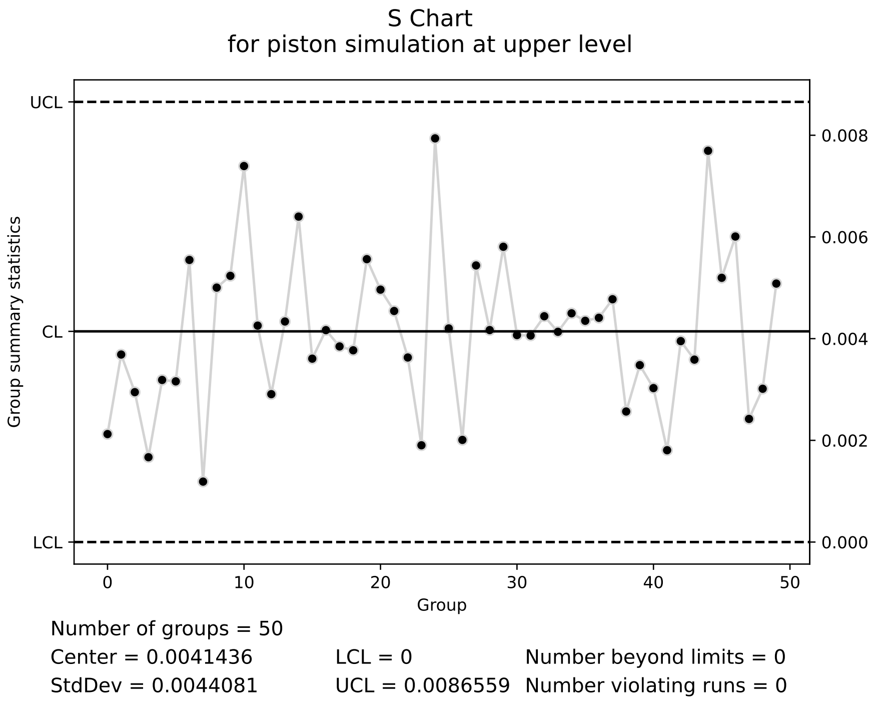
```
mistat.runStatistics(np.mean(cycleTime, axis=1))
```

```
{'count': {'mu_R': 25.96, 'sigma_R': 3.493555583105112, 'observed':
24},
 'direction': {'mu_Rstar': 33.0,
  'sigma_Rstar': 2.9268868558020253,
  'up': 16,
  'down': 16,
  'Rstar': 32,
  'alpha': [0.36630340989610105, 0.6336965901038989]}}
```

```
mistat.runStatistics(np.mean(cycleTimeTshift, axis=1))
```

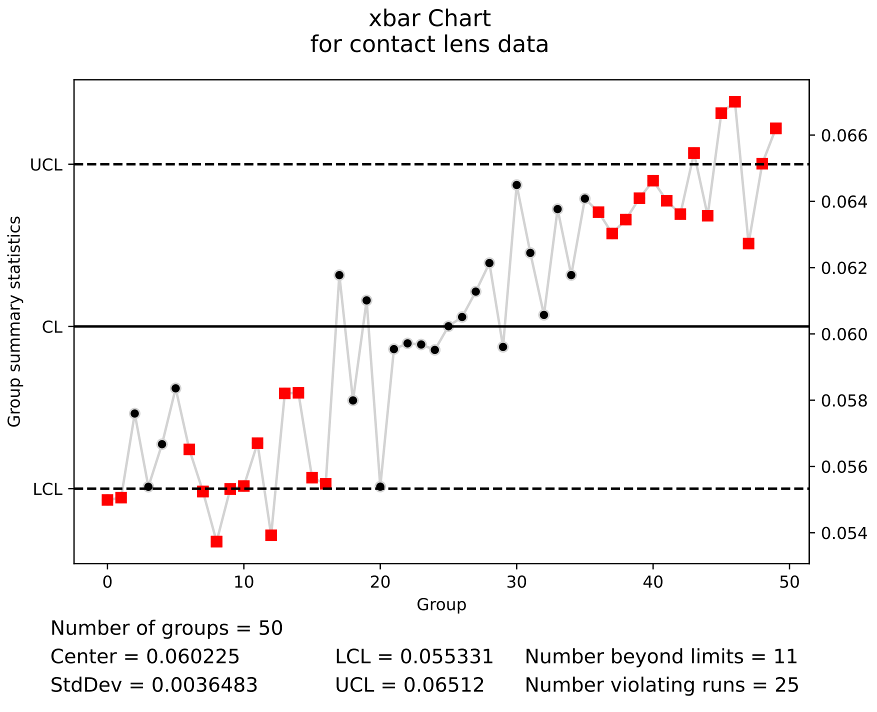


```
Number of groups = 50
Center = 0.055952         LCL = 0.050023    Number beyond limits = 0
StdDev = 0.0044192        UCL = 0.061881    Number violating runs = 0
```



```
Number of groups = 50
Center = 0.0041436        LCL = 0           Number beyond limits = 0
StdDev = 0.0044081        UCL = 0.0086559   Number violating runs = 0
```

Fig. 3.1: $\bar{X}$ and S chart of simulated Piston data (Exercises 3.5 and 3.6)



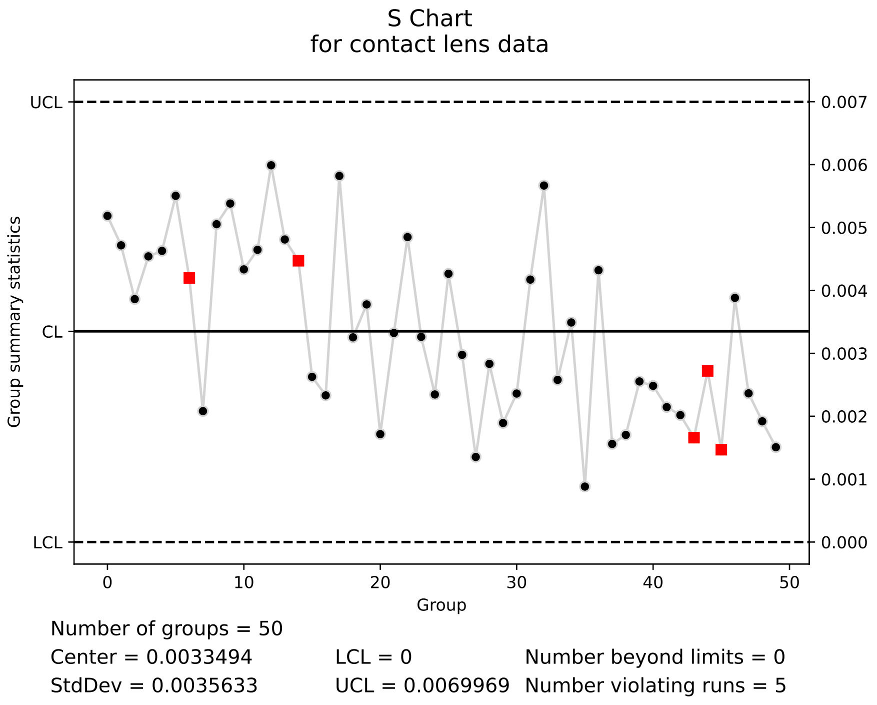


Fig. 3.2: $\bar{X}$ and S chart of simulated Piston data (Exercise 3.6)

```
{'count': {'mu_R': 25.96, 'sigma_R': 3.493555583105112, 'observed':
8},
 'direction': {'mu_Rstar': 33.0,
  'sigma_Rstar': 2.9268868558020253,
  'up': 15,
  'down': 14,
  'Rstar': 29,
  'alpha': [0.08586912144778469, 0.9141308785522153]}}
```

The number of runs in the first simulation is within the expected range. For the second simulation we observe a much smaller number of runs. The statistics for the direction of runs, shows no strong deviation from expected values.

```
mistat.runStatistics(np.mean(cycleTime, axis=1))
```

```
{'count': {'mu_R': 25.96, 'sigma_R': 3.493555583105112, 'observed':
24},
 'direction': {'mu_Rstar': 33.0,
  'sigma_Rstar': 2.9268868558020253,
  'up': 16,
  'down': 16,
  'Rstar': 32,
  'alpha': [0.36630340989610105, 0.6336965901038989]}}
```

```
mistat.runStatistics(np.mean(cycleTimeTshift, axis=1))
```

```
{'count': {'mu_R': 25.96, 'sigma_R': 3.493555583105112, 'observed':
8},
 'direction': {'mu_Rstar': 33.0,
  'sigma_Rstar': 2.9268868558020253,
  'up': 15,
  'down': 14,
  'Rstar': 29,
  'alpha': [0.08586912144778469, 0.9141308785522153]}}
```

We can also perform runs tests for both simulations. For the first simulation we get:

```
print('mean:', mistat.runsTest(np.mean(cycleTime, axis=1), np.mean(cycleTime)).pval)
STD = np.std(cycleTime, axis=1)
print('std:', mistat.runsTest(STD, np.mean(STD)).pval)
```

```
mean: 0.5747750350233831
std: 0.2529990614746843
```

This confirms our observation that the simulation shows no non-random behavior. For the second simulation we get:

```
print('mean:', mistat.runsTest(np.mean(cycleTimeTshift, axis=1), np.mean(cycleTimeTshift)).pval)
STD = np.std(cycleTimeTshift, axis=1)
print('std:', mistat.runsTest(STD, np.mean(STD)).pval)
```

```
mean: 2.7343384749349e-07
std: 0.09297787599834818
```

The run tests reflect the nonrandomness of the sequence, after the change point.

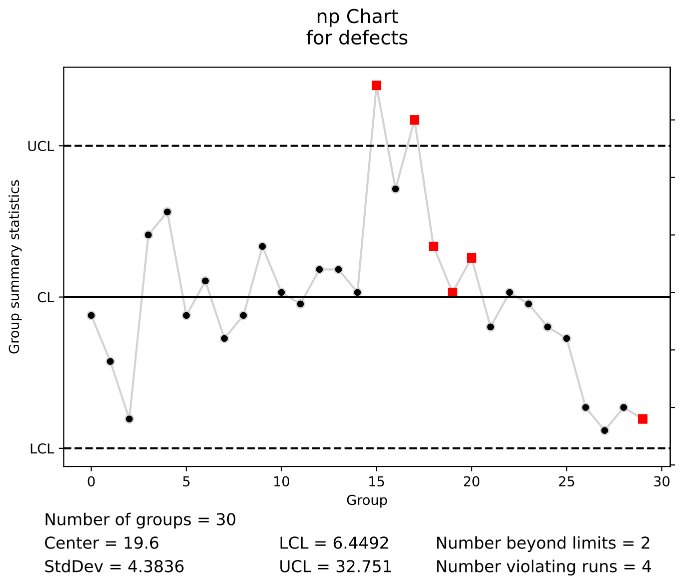


Fig. 3.3: *np*-Chart for defective substrates

Solution 3.7 Construct the *np*-chart of the data using *mistat.QualityControlChart*. It is shown in Fig. 3.3.

```
data = pd.Series([18, 14,  9, 25, 27, 18, 21, 16, 18, 24, 20, 19, 22, 22, 20,
        38, 29, 35, 24, 20, 23, 17, 20, 19, 17, 16, 10,  8, 10,  9])

qcc = mistat.QualityControlChart(data, qcc_type='np', sizes=1000)
ax = qcc.plot(title='for defects')
plt.show()
```

As we see, the data points of week 16 and 18 fall outside the upper control limit. Also, the last 8 weeks show a significant trend down (improvement).

After removing the data points for week 16 and 18 (note that Python arrays are 0-indexed), we get a revised *np*-chart (Fig. 3.4). The points are now all in control, but the pattern of the last 8 weeks remains.

```
revised = data[data < 32.751]

qcc = mistat.QualityControlChart(revised, qcc_type='np', sizes=1000)
ax = qcc.plot(title='for number of defects')
plt.show()
```

Solution 3.8 The average proportion of defectives for line 1 is $\hat{p}_1 = 0.005343$, while that for line 2 is $\hat{p}_2 = 0.056631$. The difference is very significant. Indeed,

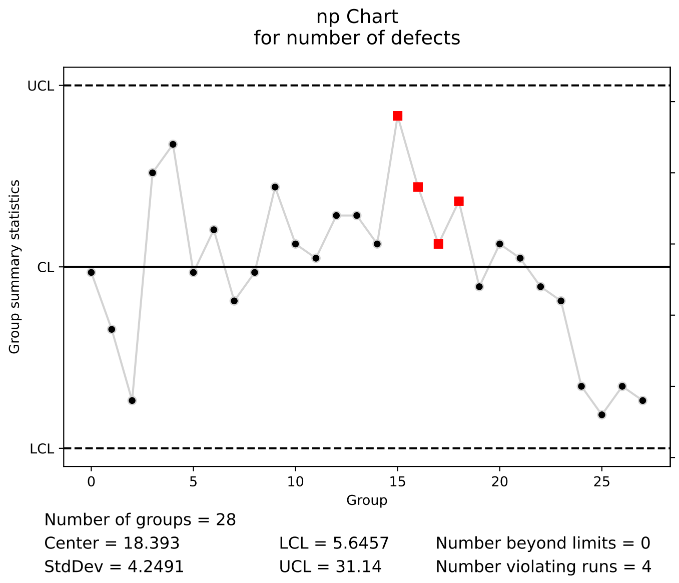


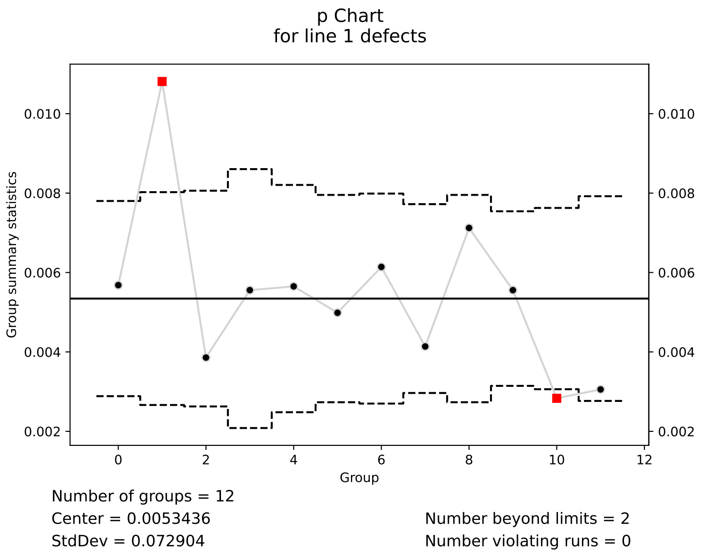
Fig. 3.4: Revised np-chart for defective substrates

$$Z = \frac{\hat{p}_2 - \hat{p}_1}{\sqrt{\frac{\hat{p}_1(1-\hat{p}_1)}{N_1} + \frac{\hat{p}_2(1-\hat{p}_2)}{N_2}}} = 34.31,$$

where $N_1 = \sum_{i=1}^{12} n_1(i)$ and $N_2 = \sum_{i=1}^{12} n_2(i)$.

In Fig. 3.5 we present the p-charts for these two production lines. We see that the chart for line 1 reveals that the process was at the beginning (point 2), out of control.

```
data = pd.DataFrame([
    [1, 45, 7920, 135, 2640],
    [2, 72, 6660, 142, 2160],
    [3, 25, 6480,  16,  240],
    [4, 25, 4500,   5,  120],
    [5, 33, 5840, 150, 2760],
    [6, 35, 7020, 156, 2640],
    [7, 42, 6840, 140, 2760],
    [8, 35, 8460, 160, 2980],
    [9, 50, 7020, 195, 2880],
    [10,55, 9900, 132, 2160],
    [11,26, 9180,  76, 1560],
    [12,22, 7200,  85, 1680]],
    columns=['Week', 'Line 1 X', 'Line 1 n', 'Line 2 X', 'Line 2 n']
)

qcc = mistat.QualityControlChart(data['Line 1 X'], qcc_type='p', sizes=data['Line 1 n'])
ax = qcc.plot(title='for line 1 defects')
plt.show()

qcc = mistat.QualityControlChart(data['Line 2 X'], qcc_type='p', sizes=data['Line 2 n'])
ax = qcc.plot(title='for line 2 defects')
plt.show()
```



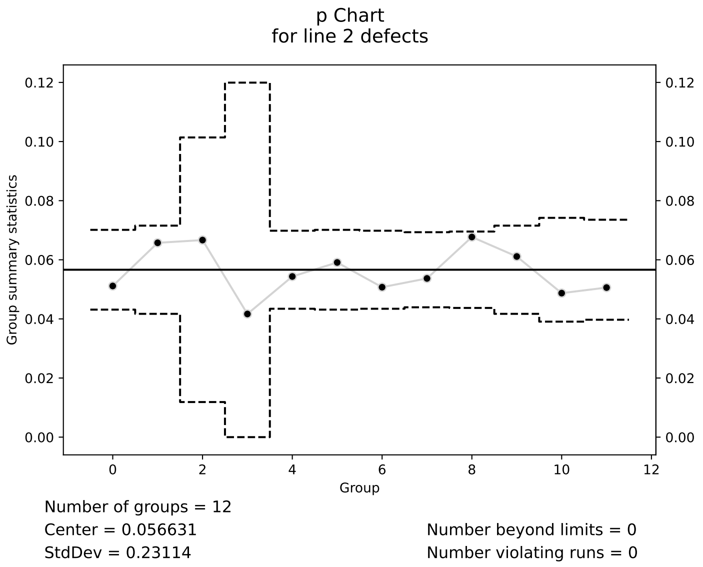


Fig. 3.5: *p*-Charts for production line defects (Exercise 3.8)

Solution 3.9 Let θ be the probability of not detecting the shift in a given day. Solving $\theta^5 = 0.2$ we get $\theta = 0.72478$. We find the smallest n for which

$$\theta \geq \Phi\left(\frac{0.01 + 3\sqrt{\frac{0.01\times 0.99}{n}} - 0.05}{\sqrt{\frac{0.05\times 0.95}{n}}}\right) - \Phi\left(\frac{0.01 - 3\sqrt{\frac{0.01\times 0.99}{n}} - 0.05}{\sqrt{\frac{0.05\times 0.95}{n}}}\right).$$

The solution is $n = 16$. Using Eq. (3.3.7) we get $n \approx 18$. This solution and the one shown above both use the normal approximation to the binomial. Another approach, which may be preferable for small sample sizes, is to use binomial distribution directly. We find the smallest n for which

$$B(n\times 0.01+3\sqrt{n \times 0.01 \times 0.99}; n, 0.05) - B(n\times 0.01-3\sqrt{n \times 0.01 \times 0.99}; n, 0.05) \leq \theta.$$

In this case the solution is $n = 7$.

Table 3.1: Dock to Stock Cycle Times

Day	Times				
1	27	43	49	32	36
2	34	29	34	31	41
3	36	32	48	35	33
4	31	41	51	51	34
5	43	35	30	32	31
6	28	42	35	40	37
7	38	37	41	34	44
8	28	44	44	34	50
9	44	36	38	44	35
10	30	43	37	29	32
11	36	40	50	37	43
12	35	36	44	34	32
13	48	49	44	27	32
14	45	46	40	35	33
15	38	36	43	38	34
16	42	37	40	42	42
17	44	31	36	42	39
18	32	28	42	39	27
19	41	41	35	41	44
20	44	34	39	30	37
21	51	43	36	50	54
22	52	50	50	44	49
23	52	34	38	41	37
24	40	41	40	23	30
25	34	38	39	35	33

Solution 3.10 (i) In Fig. 3.6 we see the $\bar{X}$ and S control charts. Points for days 21 and 22 are outside the UCL for $\bar{X}$. Excluding these points, the recalculated control chart limits for $\bar{X}$ are LCL = 29.731 and UCL = 45.781. Also, $\bar{\bar{X}}$ = 37.757.

```
cycleTime = pd.DataFrame([
    [1, 27, 43, 49, 32, 36],      [2, 34, 29, 34, 31, 41],
    [3, 36, 32, 48, 35, 33],      [4, 31, 41, 51, 51, 34],
    [5, 43, 35, 30, 32, 31],      [6, 28, 42, 35, 40, 37],
    [7, 38, 37, 41, 34, 44],      [8, 28, 44, 44, 34, 50],
    [9, 44, 36, 38, 44, 35],      [10, 30, 43, 37, 29, 32],
    [11, 36, 40, 50, 37, 43],     [12, 35, 36, 44, 34, 32],
    [13, 48, 49, 44, 27, 32],     [14, 45, 46, 40, 35, 33],
    [15, 38, 36, 43, 38, 34],     [16, 42, 37, 40, 42, 42],
    [17, 44, 31, 36, 42, 39],     [18, 32, 28, 42, 39, 27],
    [19, 41, 41, 35, 41, 44],     [20, 44, 34, 39, 30, 37],
    [21, 51, 43, 36, 50, 54],     [22, 52, 50, 50, 44, 49],
    [23, 52, 34, 38, 41, 37],     [24, 40, 41, 40, 23, 30],
    [25, 34, 38, 39, 35, 33]],
    columns=['Week', 'S1', 'S2', 'S3', 'S4', 'S5'])
cycleTime = cycleTime.set_index('Week')

qcc = mistat.QualityControlChart(cycleTime, qcc_type='xbar')
ax = qcc.plot(title='for dock-to-stock cycle times')
plt.show()

qcc = mistat.QualityControlChart(cycleTime, qcc_type='S')
ax = qcc.plot(title='for dock-to-stock cycle times')
plt.show()

# exclude points for week 21 and 22
qcc = mistat.QualityControlChart(cycleTime.drop(labels=[21, 22]), qcc_type='xbar')
qcc.center, qcc.limits
```

```
(37.756521739130434,
          LCL        UCL
 0  29.731454  45.781589)
```

(ii) We can calculate the significance of the day 21 and 22 cycle times relative to the UCL of the recalculated control chart.

```
statistic, pvalue = stats.ttest_1samp(cycleTime.loc[21,], 45.781, alternative='greater')
print(pvalue)
statistic, pvalue = stats.ttest_1samp(cycleTime.loc[22,], 45.781, alternative='greater')
print(pvalue)
```

```
0.3846525285951686
0.03720636409199298
```

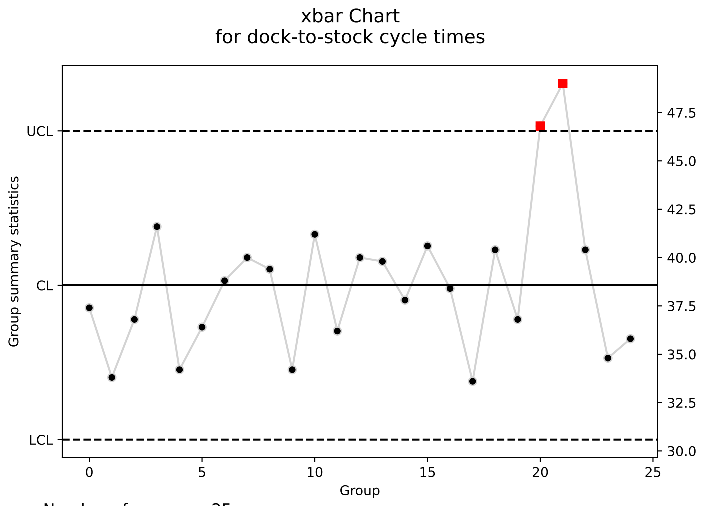
We find that only $\bar{X}_{22}$ is significantly larger than 45.781.

(iii) Unusual long cycle times can be due to

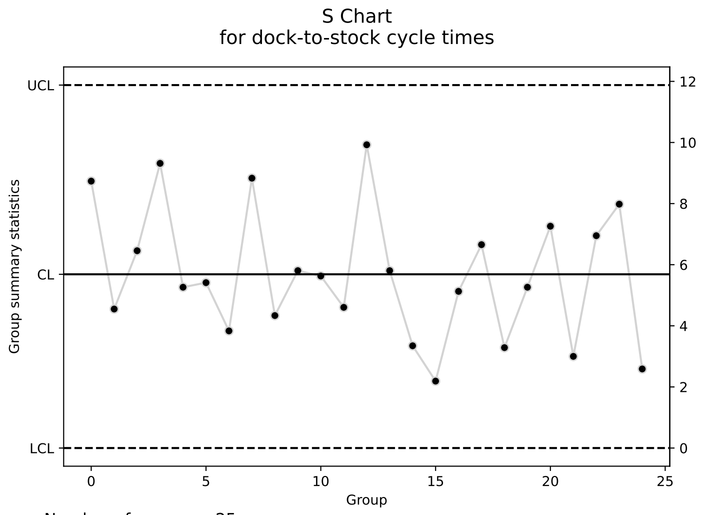
1. Missing or misplaced information in accompanying paperwork
2. Missing or misplaced marks on package
3. Defective package
4. Non standard package
5. Wrong information on package destination
6. Overloaded stock room so that packages cannot be accepted
7. Misplaced packages.

Additional data that can be collected to explain long cycle times:

1. Package destination
2. Stock room



```
Number of groups = 25
Center = 38.568          LCL = 30.585       Number beyond limits = 2
StdDev = 5.9501          UCL = 46.551       Number violating runs = 0
```



```
Number of groups = 25
Center = 5.6863          LCL = 0            Number beyond limits = 0
StdDev = 6.0493          UCL = 11.879       Number violating runs = 0
```

Fig. 3.6: Xbar and S chart of dock to stock cycle times (Exercise 3.10)

3. Package size
4. Package weight
5. Package origin (country, supplier)
6. Package courier.

Solution 3.11 [1] The ARL of the modified Shewhart control chart with $a = 3$, $w = 2$ and $r = 4$ when $n = 10$ and $\delta = 0$ is 370.3. When $\delta = 1$, the ARL is 1.75 and when $\delta = 2$ the ARL is 1.0.

Solution 3.12 [2] **9.2.2** For $a = 3$, $w = 1$ and $r = 15$ when $n = 5$ the ARL is 33.4.

Solution 3.13 [3] Samples should be taken every $h^0 = 34$ [min] = 0.57 [hr].

Solution 3.14 With $n = 20$ and $p_0 = 0.10$,

$$OC(p) = B(np_0 + 3\sqrt{np_0(1-p_0)}; n, p) - B(np_0 - 3\sqrt{np_0(1-p_0)}; n, p).$$

we get the following table of values of $OC(p)$, for $p = 0.05, 0.5$ (0.05).

p	$OC(p)$
0.05	0.999966
0.10	0.997614
0.15	0.978065
0.20	0.913307
0.25	0.785782
0.30	0.608010
0.35	0.416625
0.40	0.250011
0.45	0.129934
0.50	0.057659

Solution 3.15 The sample size n_0 is the smallest n for which

$$B(n \times 0.01 + 0.2985\sqrt{n}; n, 0.05) - B(n \times 0.01 - 0.2985\sqrt{n}; n, 0.05) \leq 0.1.$$

From this we obtain $n_0 = 184$.

```
n = 1
while OC_p_chart(0.05, n, 0.01) > 0.1:
    n += 1
n
```

```
184
```

[1] TODO: require clarification on how to calculate this
[2] TODO: require clarification on how to calculate this
[3] TODO: require clarification on how to calculate this

Using Eq. (3.3.5), which is based on the normal approximation to the binomial, gives $n_0 \approx 209$.

```
# normal approximation
def OC_p_chart_normal(p, n, p0):
    loc = n * p
    scale = np.sqrt(n*p*(1-p))
    c = n * p0
    delta = 3 * np.sqrt(n * p0 * (1 - p0))
    return (stats.norm(loc, scale).cdf(c + delta) -
            stats.norm(loc, scale).cdf(c - delta))

# alternative implementation
def OC_p_chart_normal_2(p, n, p0):
    delta = 3 * np.sqrt(p0 * (1 - p0) / n)
    UCL = p0 + delta
    LCL = p0 - delta
    denom = np.sqrt(p * (1 - p) / n)
    return (stats.norm().cdf((UCL - p)/denom) -
            stats.norm().cdf((LCL - p)/denom))

n = 1
while OC_p_chart_normal(0.05, n, 0.01) > 0.1:
    n += 1
print(n, OC_p_chart_normal(0.05, n, 0.01))
```

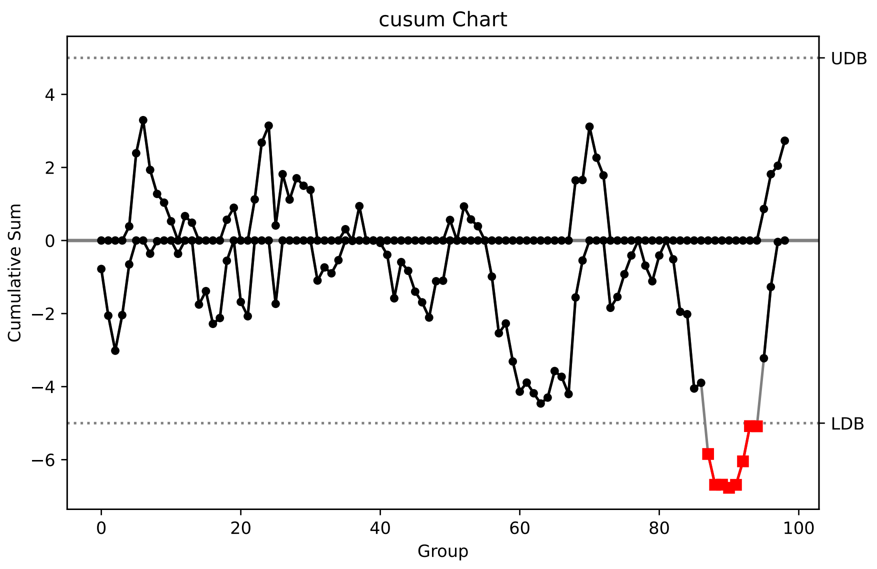
```
209 0.09959381959080052
```

Solution 3.16 **(i)** 4.5 [hr]; **(ii)** 7; **(iii)** For a shift of size $\delta = 1$, option 1 detects it on the average after 4.5 [hr]. Option 2 detects it on the average after $2 \times 1.77 = 3.5$ [hr]. Option 2 is preferred from the point of view of detection speed.

Solution 3.17 [4] **9.4.1** Here $K^+ = 221$ and $h^+ = 55.3724$. Figure 9.4.1 shows the "up" and "down" CUSUM charts. We see that the upper or lower limits are not crossed. There is no signal of change.

```
data = mistat.load_data('OELECT')
analysis = mistat.Cusum(data, center=220)
analysis.plot()
plt.show()
```

[4] TODO: the cusum charts use z-transformed data. How do I get UCL and LCL from K^+ and h^+ and how are these actually determined?



```
Number of groups = 99        Decision interval (std. err.) = 5
Center = 220                 Shift detection (std. err.) 1
StdDev = 3.5914              No. of points beyond limits = 8
```

Solution 3.18 [5] Using $K^+ = 19.576$, $h^+ = 13.523$, $K^- = 10.497$ and $h^- = -9.064$ we obtained $PFA = 0.014$ and $CED = 3.01 \pm 0.235$. Here the value of λ changed from 15 to 25.

```
lambda0 = 15
lambda1_p = 25
lambda1_m = 7
alpha = 0.001
tau = 30
kp = (lambda1_p - lambda0) / np.log(lambda1_p/lambda0)
hp = - np.log(alpha) / np.log(lambda1_p/lambda0)

km = (lambda1_m - lambda0) / np.log(lambda1_m/lambda0)
hm = - np.log(alpha) / np.log(lambda1_m/lambda0)

arl = mistat.cusumPfaCed(randFunc1=stats.poisson(mu=15),
                         randFunc2=stats.poisson(mu=25),
                         tau=tau,
                         kp=kp, km=km,
                         hp=hp, hm=hm,
                         N=4000, limit=1000, seed=1)
result = arl['statistic']
```

```
PFA 0.01075  CED 2.1663  Std. Error 0.51072
```

[5] TODO: I understand how K+ is derived in this case. The values for K^- and h^- are obtained using a λ_1^- of 7. This should go into the exercise description. Please check if h^- (hm) is calculated correctly.

Solution 3.19 [6] **(i)** Determine the control parameters:

```
mu0 = 100
mu1p = 110
mu1m = 90
sigma = 20
n = 5
alpha = 0.001

Kp = (mu0+mu1p)/2
hp = -(sigma**2 / n) * np.log(alpha) / (mu1p-mu0)
Km = (mu0+mu1m)/2
hm = -(sigma**2 / n) * np.log(alpha) / (mu1m-mu0)

pd.Series({'Kp': Kp, 'hp': hp, 'Km': Km, 'hm': hm})
```

```
Kp    105.000000
hp     55.262042
Km     95.000000
hm    -55.262042
dtype: float64
```

(ii) Estimate PFA and CED for $\tau = 10, 20, 30$.

```
results = []
for tau in [10, 20, 30]:
    arl = mistat.cusumPfaCed(randFunc1=stats.norm(loc=mu0, scale=sigma/np.sqrt(5)),
                             randFunc2=stats.norm(loc=mu1p, scale=sigma/np.sqrt(5)),
                             tau=tau, kp=Kp, km=Km, hp=hp, hm=hm,
                             N=300, limit=1000, seed=1, verbose=False)
    results.append({
        'tau': tau,
        **arl['statistic'], # copy all results from arl['statistic']
    })
pd.DataFrame(results)
```

```
   tau       PFA        CED  Std. Error
0   10  0.000000  10.090000    1.050516
1   20  0.003333  10.076923    1.671984
2   30  0.006667  10.449664    2.289643
```

(iii) Increase n to 20 and estimate PFA and CED for $\tau = 10, 20, 30$.

```
n = 20
hp = -(sigma**2 / n) * np.log(alpha) / (mu1p-mu0)
hm = -(sigma**2 / n) * np.log(alpha) / (mu1m-mu0)

results = []
for tau in [10, 20, 30]:
    arl = mistat.cusumPfaCed(randFunc1=stats.norm(loc=mu0, scale=sigma/np.sqrt(5)),
                             randFunc2=stats.norm(loc=mu1p, scale=sigma/np.sqrt(5)),
                             tau=tau, kp=Kp, km=Km, hp=hp, hm=hm,
                             N=300, limit=1000, seed=1, verbose=False)
    results.append({
        'tau': tau,
        **arl['statistic'], # copy all results from arl['statistic']
    })
pd.DataFrame(results)
```

[6] TODO: I assume that mu1m is symmetric here. This should go into the exercise description. Please check if h^- (hm) is calculated correctly.

```
   tau       PFA       CED  Std. Error
0   10  0.586667  1.709677    1.056345
1   20  0.806667  2.172414    2.910490
2   30  0.920000  2.250000    6.583663
```

The probability of a false alarm increases considerably.

Solution 3.20 According to Eq. (3.5.6), in the Poisson case,

$$\begin{aligned}
W_m &= \sum_{i=1}^{m-1} \prod_{j=i+1}^{m} R_j \\
&= \sum_{i=1}^{m-1} \exp\left\{-(m-i)\delta + \sum_{j=i+1}^{m} X_j \log(\rho)\right\} \\
&= e^{-\delta + X_m \log\rho} + \sum_{i=1}^{m-2} \exp\left\{-(m-i)\delta + \sum_{j=i+1}^{m} X_j \log(\rho)\right\} \\
&= e^{-\delta + X_m \log\rho} + e^{-\delta + X_m \log\rho} \sum_{i=1}^{m-2} \exp\left\{-(m-1-i)\delta + \sum_{j=i+1}^{m-1} X_j \log(\rho)\right\} \\
&= (1 + W_{m-1}) e^{-\delta + X_m \log\rho} \\
&= (1 + W_{m-1}) R_m.
\end{aligned}$$

Solution 3.21 The mean of the data is $\bar{X} = 219.25$, and its standard deviation is $S = 4.004$.

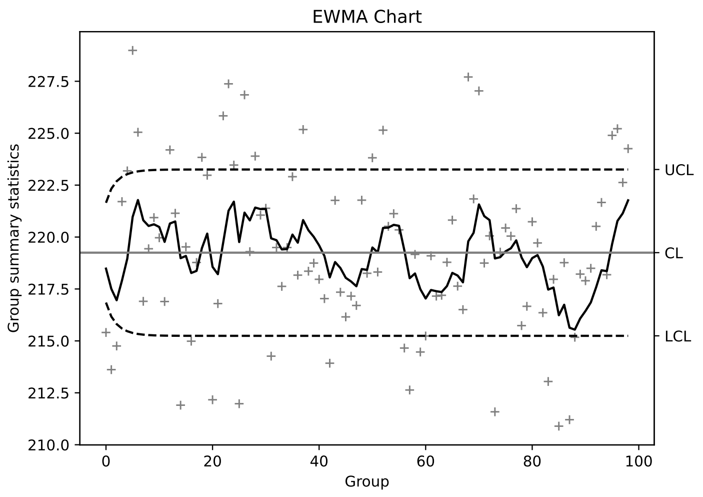
```
data = mistat.load_data('OELECT')
ewma = mistat.EWMA(data, center=data.mean(), std_dev=data.std(),
                   sizes=1, smooth=0.2, nsigmas=3)
ewma.plot()
plt.show()
```

Fig. 3.7 shows that there is no significant shift in the data.

Solution 3.22 In Fig. 3.8 we see the EWMA control chart for Diameter 1. The mean and standard deviation are $\bar{X} = 9.993$ and $S = 0.0164$. The target value for Diameter 1 is 10 mm. Notice in the chart that after a drop below 9.98 mm the machine corrects itself automatically, and there is a significant run upwards towards the target value. Using the EWMA chart an automatic control can be based on Eq. **9.8.11**.

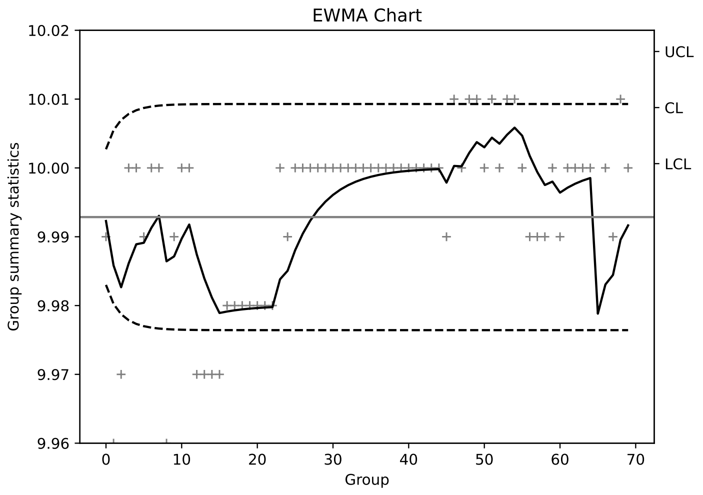
```
data = mistat.load_data('ALMPIN')
data = data['diam1']
ewma = mistat.EWMA(data, center=data.mean(), std_dev=data.std(),
                   sizes=1, smooth=0.2, nsigmas=3)
ax = ewma.plot()
ax.set_ylim(9.96, 10.02)
plt.show()
```

Solution 3.23 Using the first 50 values of **DOW1941** perform the regression method outlined in the text,



```
Number of groups = 99          Smoothing parameter = 0.2
Center = 219.25                Control limits at 3*sigma
StdDev = 4.004                 No. of points beyond limits = 0
```

Fig. 3.7: EWMA chart for the **OELECT** dataset



```
Number of groups = 70          Smoothing parameter = 0.2
Center = 9.9929                Control limits at 3*sigma
StdDev = 0.016431              No. of points beyond limits = 0
```

Fig. 3.8: EWMA chart for the Diameter 1

```python
dow1941 = mistat.load_data('DOW1941')
# solve the regression equation
m = 50
sqrt_t = np.sqrt(range(1, m + 1))
df = pd.DataFrame({
    'Ut': dow1941[:m]/sqrt_t,
    'x1t': 1 / sqrt_t,
    'x2t': sqrt_t,
})
model = smf.ols(formula='Ut ~ x1t + x2t - 1', data=df).fit()
mu0, delta = model.params
var_eta = np.var(model.resid, ddof=2)
pd.Series({'mu0': mu0, 'delta': delta, 'Var(eta)': var_eta})
```

```
mu0          132.808555
delta         -0.255630
Var(eta)       0.297616
dtype: float64
```

The least squares estimates of the initial parameter values are $\hat{\mu}_0 = 132.809$, $\hat{\delta} = -0.2556$ and $\hat{\sigma}^2_\epsilon + \hat{\sigma}^2_2 = 0.297616$.

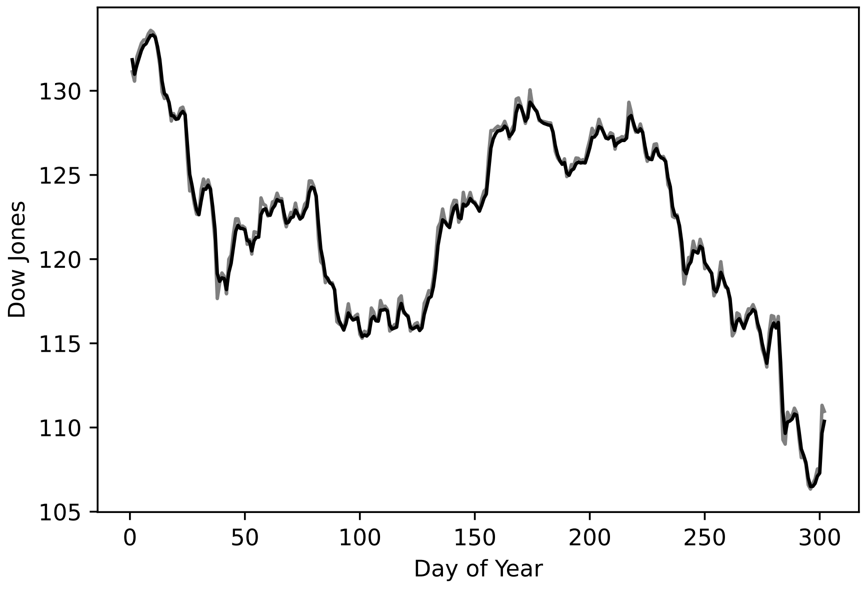
For the Kalman filter, we choose $\hat{\sigma}^2_\epsilon = 0.15$ and for w^2_0 the value 0.0015. The Python commands used to obtain the Kalman filter estimates of the **DOW1941** data are as follows:

```python
# choose sig2e and w20
sig2e = 0.15
w20 = 0.0015
# apply the filter
results = []
mu_tm1 = mu0
w2_tm1 = w20
y_tm1 = mu0
for i in range(0, len(dow1941)):
    y_t = dow1941[i]
    B_t = sig2e / (var_eta + w2_tm1)
    mu_t = B_t * (mu_tm1 + delta) + (1 - B_t) * y_t # X
    results.append({
        't': i + 1,
        'y_t': y_t,
        'mu_t': mu_t,
        'B_t': B_t,
        'W2_t': w2_tm1,
    })
    w2_tm1 = B_t * (var_eta - sig2e + w2_tm1)
    mu_tm1 = mu_t
    y_tm1 = y_t
results = pd.DataFrame(results)
```

The Table 3.4 shows the first 25 values of the DOW 1941 data as well as the values of B_t, w^2_t, and $\hat{\mu}_t$. Starting with $w^2_0 = 0.0015$ we see that w^2_t converges fast to 0.0923. The values of $\hat{\mu}_t$ are very close to the data values. The result is shown graphically in Table 3.4 with actual values of Y_t in grey and estimated $\hat{\mu}_t$ in black.

Table 3.2: Results for Kalman filter applied to the Dow-Jones daily 1941 dataset

	Dow 1941	$\hat{\mu}_t$	B_t	w_t^2
0	131.1300	131.8436	0.5015	0.0015
1	130.5700	130.9800	0.4028	0.0748
2	132.0100	131.5120	0.3874	0.0896
3	132.4000	131.9596	0.3851	0.0919
4	132.8300	132.3967	0.3848	0.0922
5	133.0200	132.6819	0.3847	0.0923
6	133.0200	132.7916	0.3847	0.0923
7	133.3900	133.0614	0.3847	0.0923
8	133.5900	133.2883	0.3847	0.0923
9	133.4900	133.3141	0.3847	0.0923
10	133.2500	133.1763	0.3847	0.0923
11	132.4400	132.6249	0.3847	0.0923
12	131.5100	131.8406	0.3847	0.0923
13	129.9300	130.5667	0.3847	0.0923
14	129.5400	129.8366	0.3847	0.0923
15	129.7500	129.6850	0.3847	0.0923
16	129.2400	129.3128	0.3847	0.0923
17	128.2000	128.5298	0.3847	0.0923
18	128.6500	128.5054	0.3847	0.0923
19	128.3400	128.3053	0.3847	0.0923
20	128.5200	128.3391	0.3847	0.0923
21	128.9600	128.6228	0.3847	0.0923
22	129.0300	128.7750	0.3847	0.0923
23	128.6000	128.5690	0.3847	0.0923
24	126.0000	126.8900	0.3847	0.0923



Chapter 4
Multivariate Statistical Process Control

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns

import mistat
```

Solution 4.1 The values of T^2 were computed, with $\mathbf{m}$ being the mean of the first 48 vectors. Fig. **??** shows the T^2 values for all boards with the calculated control limit of UCL = 17.1953.

```
# use eqn 4.1.6 to calculate the upper control limit for monitoring
n = 48; p = 3
UCL = (n-1)*(n+1)*p/(n*(n-p)) * stats.f(p, n-p).ppf(0.997)

tsq = mistat.load_data('TSQ')
ax = tsq.plot()
ax.axhline(UCL, color='black')
ax.axhline(0, color='black')
```

```
<matplotlib.lines.Line2D at 0x7f98ab3136a0>
```

The side by side boxplots in Fig. 4.2 aggregate the information of the T^2 values by board number.

```
df = pd.DataFrame({
    'T2': tsq,
    'board': mistat.load_data('PLACE')['crcBrd'][48:],
})
ax = df.groupby('board').boxplot(column='T2', subplots=False, rot=90, grid=False)
ax.axhline(UCL, color='grey')
plt.show()
```

We see that even on the first 9 cards there are a few outliers, that is, points whose T^2 is outside the control limits. All points from card 13 on and a majority of points from boards 10, 11 and 12 have T^2 values greater than UCL.

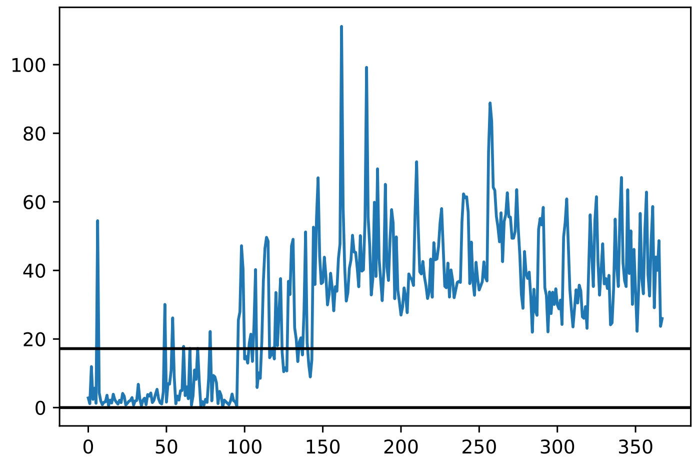


Fig. 4.1: Plot of T^2 values with control limit

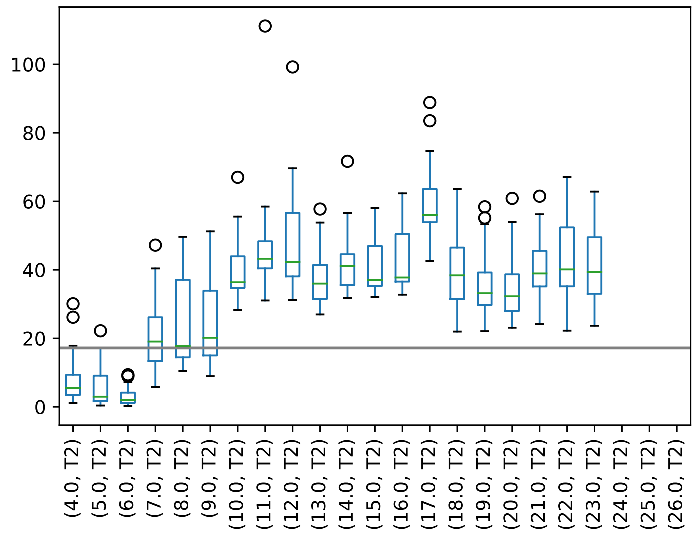


Fig. 4.2: Plot of T^2 values with control limit

Solution 4.2 As $(N_v(\boldsymbol{\mu}, \boldsymbol{\sigma})$ is a multivariate normal distribution, $\boldsymbol{\Sigma}$ is positive definite. Therefore there exists a nonsingular matrix $\boldsymbol{P}$ so that $\boldsymbol{\Sigma} = \boldsymbol{P}\boldsymbol{P}'$.

Using this we get,

$$\begin{aligned}(\mathbf{X}-\boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\mathbf{X}-\boldsymbol{\mu}) &= (\mathbf{X}-\boldsymbol{\mu})'(\boldsymbol{P}')^{-1}\boldsymbol{P}^{-1}(\mathbf{X}-\boldsymbol{\mu})\\ &= (\mathbf{X}-\boldsymbol{\mu})'(\boldsymbol{P}^{-1})'\boldsymbol{P}^{-1}(\mathbf{X}-\boldsymbol{\mu})\\ &= \left(\boldsymbol{P}^{-1}(\mathbf{X}-\boldsymbol{\mu})\right)'\boldsymbol{P}^{-1}(\mathbf{X}-\boldsymbol{\mu})\\ &= \boldsymbol{Y}'\boldsymbol{Y}\end{aligned}$$

with $\boldsymbol{Y} = \boldsymbol{P}^{-1}(\mathbf{X}-\boldsymbol{\mu})$.
1

Solution 4.3 Create T^2 chart using separate computations by cylinders. The chart is shown in Fig. 4.3.

```
car = mistat.load_data('CAR')
car = car.sort_values('cyl')
columns = ['turn', 'hp', 'mpg']

fig, axes = plt.subplots(nrows=3, figsize=[8, 8])
for cyl, ax in zip([4, 6, 8], axes):
  base = car.loc[car['cyl'] == cyl, columns]
  newdata = car.loc[car['cyl'] != cyl, columns]
  mqcc = mistat.MultivariateQualityControlChart(base, qcc_type='T2single',
            confidence_level=0.99, newdata=newdata)
  mqcc.plot(ax=ax, show_legend=False)
  ax.set_ylabel(f'{cyl} cylinders')
  plt.tight_layout()
plt.show()
```

The T^2 charts for the internally derived targets for 4 or 6 cylinders, show now relevant differences towards the remaining data. For 8 cylinders on the other hand, many of other cars show strong differences.

Solution 4.4 Create T^2 chart using separate computations based on origin. The chart is shown in Fig. 4.4.

```
car = mistat.load_data('CAR')
car = car.sort_values('cyl')
columns = ['turn', 'hp', 'mpg']
origins = [None, 'US', 'Europe', 'Asia']

fig, axes = plt.subplots(nrows=3, figsize=[8, 8])
for origin, ax in zip([1, 2, 3], axes):
  base = car.loc[car['origin'] == origin, columns]
  newdata = car.loc[car['origin'] != origin, columns]
  mqcc = mistat.MultivariateQualityControlChart(base, qcc_type='T2single',
            confidence_level=0.99, newdata=newdata)
  mqcc.plot(ax=ax, show_legend=False)
  ax.set_ylabel(f'Origin {origins[origin]}')
plt.tight_layout()
plt.show()
```

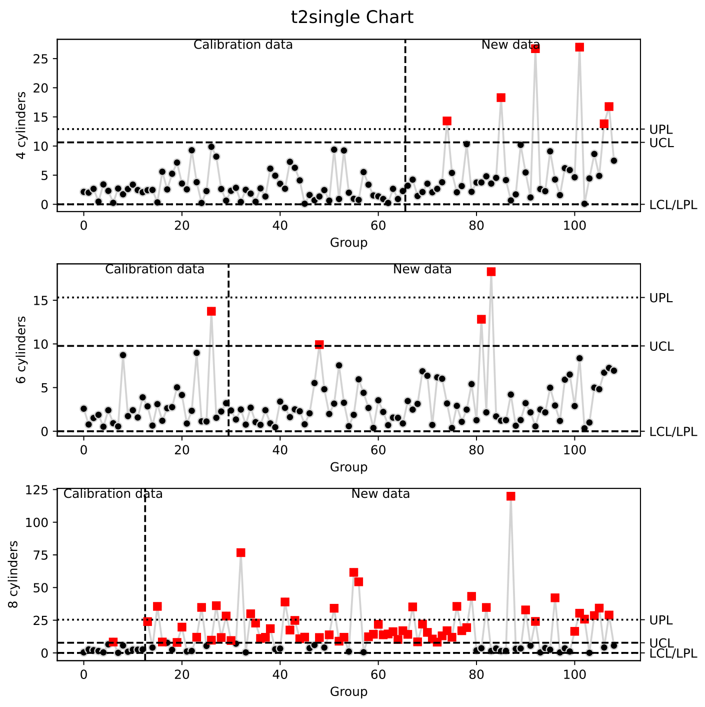


Fig. 4.3: T^2 charts for the **CAR** dataset using subsets based on number of cylinders to derive internal targets (Excercise 4.3)

Using the European cars to derive internal targets, we can see that many of the other cars are above the UCL. This is less frequent in the other two plots.

The differences are also obvious when the data are visualized in a scatterplot matrix. See Fig. 4.5

```
sns.pairplot(car[[*columns, 'cyl']], hue='cyl', height=1.5)
sns.pairplot(car[[*columns, 'origin']], hue='origin', height=1.5)
plt.show()
```

Solution 4.5 Create T^2 chart for full **GASOL** dataset.

```
gasol = mistat.load_data('GASOL')
mqcc = mistat.MultivariateQualityControlChart(gasol, qcc_type='T2single',
           confidence_level=0.99)
ax = mqcc.plot()
ax.plot((11, 23), (1, 1), color='red')
plt.show()
```

[1] TODO: proof required

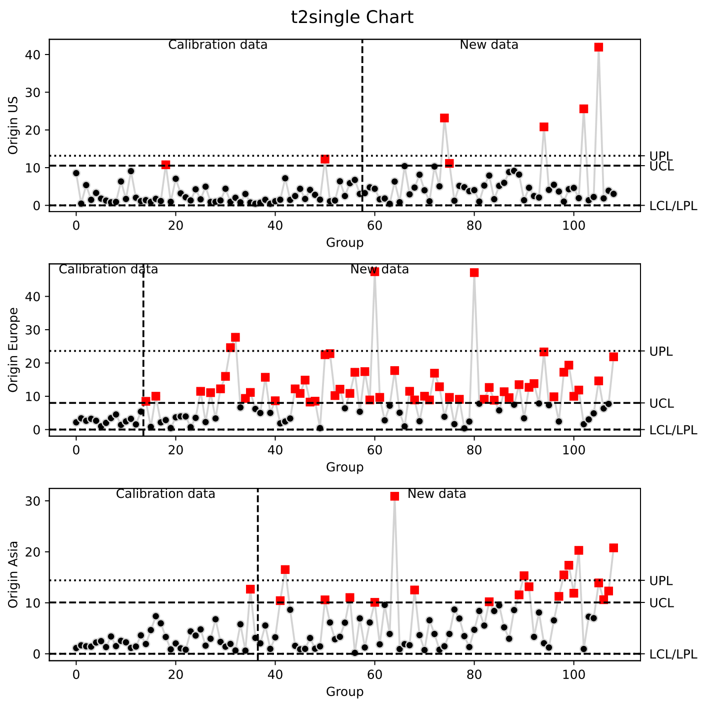


Fig. 4.4: T^2 charts for the **CAR** dataset using subsets based on number of cylinders to derive internal targets (Excercise 4.4)

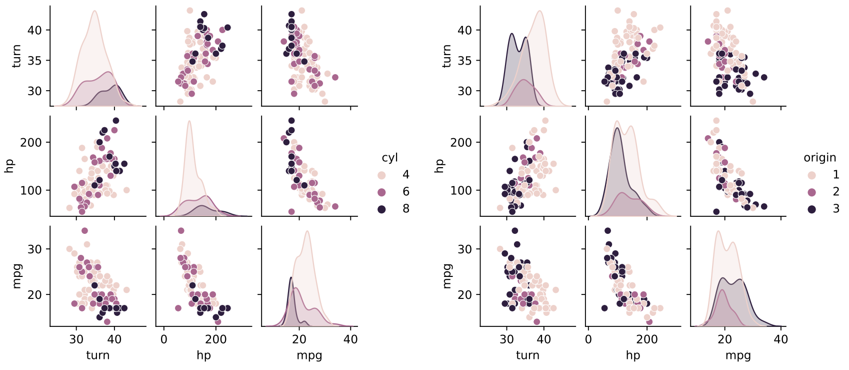


Fig. 4.5: Scatterplot matrix for the **CAR** dataset colored by cylinders (left) and origin (right).

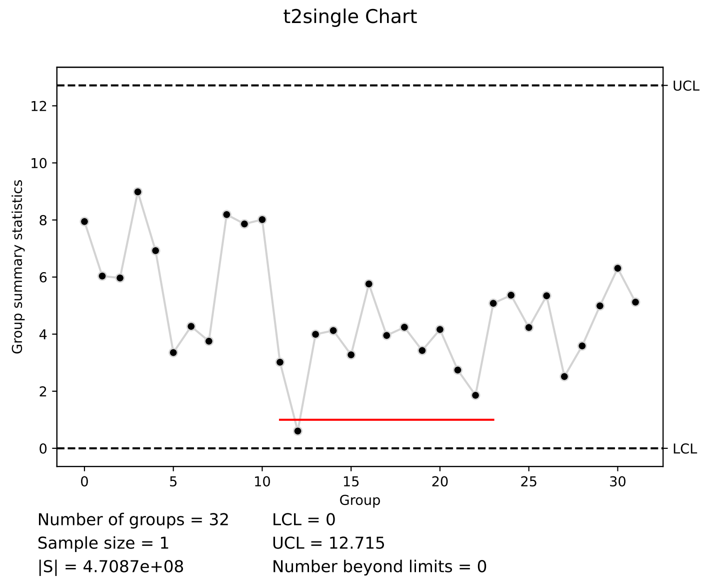


Fig. 4.6: T^2 charts for the **GASOL** dataset. The horizontal line highlights observations 12 to 24 are highlighted

Using observations 12 to 24 as an external assigned target, we get the T^2 chart in Fig. 4.7. All other data points have now T^2 value greater than UCL. The scatterplot matrix shows that the observations 12 to 24 have very distinct values especially for $x1$.

```
gasol = mistat.load_data('GASOL')
base = gasol.iloc[11:24]
mqcc_base = mistat.MultivariateQualityControlChart(base, qcc_type='T2single',
            confidence_level=0.99)
mqcc = mistat.MultivariateQualityControlChart(gasol, qcc_type='T2single',
          center=mqcc_base.stats.center, cov=mqcc_base.stats.cov,
          confidence_level=0.99)
ax = mqcc.plot()
plt.show()
gasol['color'] = ['red' if 11 <= i < 24 else 'black'
                        for i in range(len(gasol))]
sns.pairplot(gasol, hue='color', height=1.75)
plt.show()
```

Solution 4.6 The T^2 chart and a scatterplot matrix are shown in Fig. 4.8. Compared to Exercise 4.5, the effect is even stronger, as the variation of $x1$ of the subset is even tighter and closer to 0. This results in very large values of T^2.

```
gasol = mistat.load_data('GASOL')
base = gasol.iloc[24:32]
mqcc_base = mistat.MultivariateQualityControlChart(base, qcc_type='T2single',
```

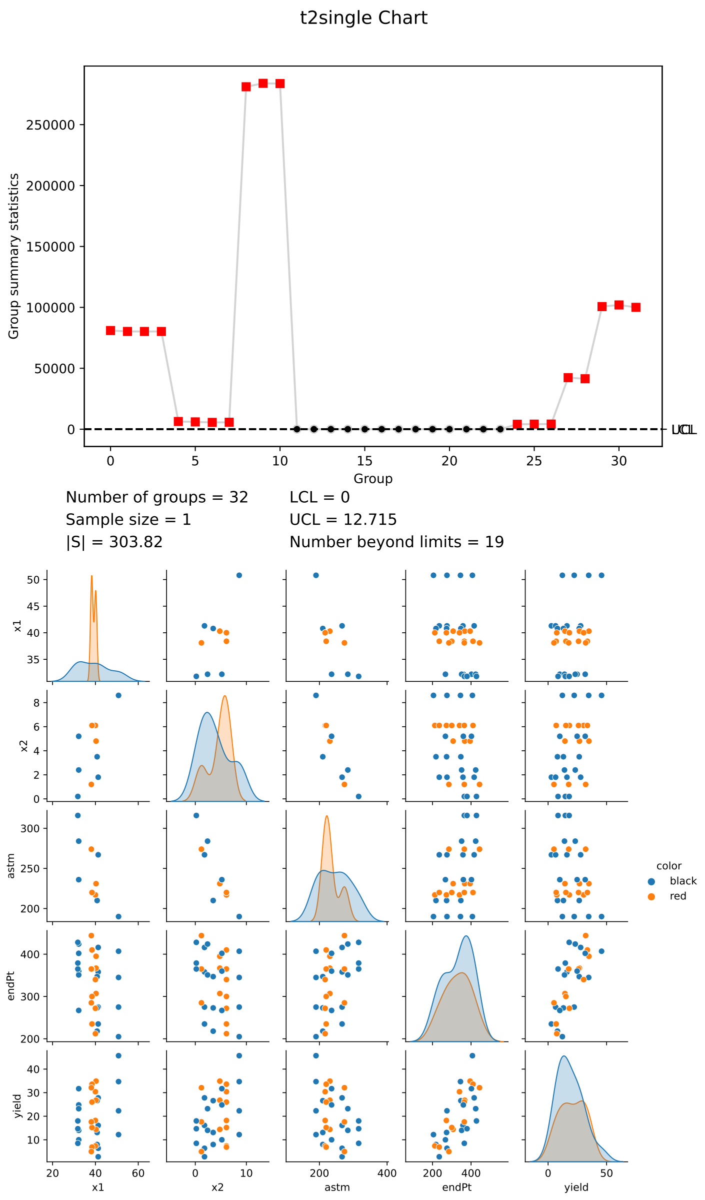


Fig. 4.7: T^2 charts for the **GASOL** dataset using observations 12 to 24 (note the 0-indexing) as an external assigned target. The bottom scatterplot matrix highlights the subset values.

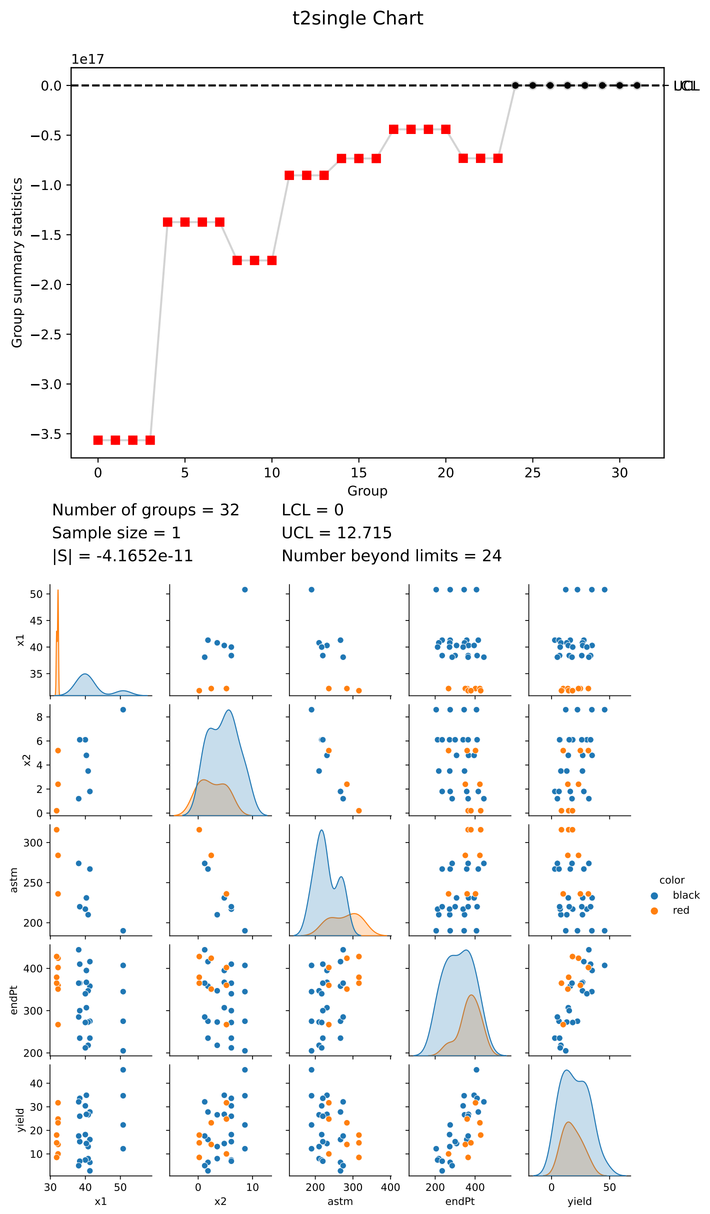
```
           confidence_level=0.99)
mqcc = mistat.MultivariateQualityControlChart(gasol, qcc_type='T2single',
          center=mqcc_base.stats.center, cov=mqcc_base.stats.cov,
          confidence_level=0.99)
ax = mqcc.plot()
plt.show()
gasol['color'] = ['red' if 24 <= i < 32 else 'black'
                         for i in range(len(gasol))]
sns.pairplot(gasol, hue='color', height=1.75)
plt.show()
```

Solution 4.7 [2]

Solution 4.8 [3]

[2] TODO

[3] TODO



```
Number of groups = 32          LCL = 0
Sample size = 1                UCL = 12.715
|S| = -4.1652e-11              Number beyond limits = 24
```

Fig. 4.8: T^2 charts for the **GASOL** dataset using observations 25 to 32 (note the 0-indexing) as an external assigned target. The bottom scatterplot matrix highlights the subset values.

Chapter 5
Classical Design and Analysis of Experiments

Import required modules and define required functions

```
import itertools
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.formula.api as smf
from statsmodels.stats import anova
import matplotlib.pyplot as plt
from pyDOE2 import fracfact

import mistat
```

Solution 5.1 Preparing pancakes is a production process readily available for everyone to experiment with. The steps are as follows:

1. Open readymix box.
2. Measure prespecified amount of readymix into measuring cup.
3. Pour prespecified amount of readymix into container.
4. Measure prespecified amount of water in measuring cup.
5. Add prespecified amount of water to container.
6. Mix material in container with one or two eggs.
7. Warm cooking pan.
8. Add oil to pan in liquid, solid or spray form.
9. Pour material into cooking pan for one or more pancakes.
10. Wait for darkening signs on the pancakes rim.
11. Turn over pancakes.
12. Wait again.
13. Remove pancakes from cooking pan.

Examples of response variables include subjective taste testing by household members with classification on a 1 to 5 scale, drop test for measuring pancake body composition (drop pancake from the table to the floor and count the number of pieces it decomposes into), quantitative tests performed in the laboratory to determine pancake chemical composition and texture.

Noise variables include environmental temperature and humidity, variability in amounts of readymix, water, oil and cooking time, stove heating capacity and differences in quality of raw materials.

Examples of control variables include:

- Amounts of readymix–less and more than present value
- Preheating time of cooking pan–less and more than present value
- Pouring speed of water into readymix container–slower and faster
- Waiting time before turn over–not just ready and overdone.

Solution 5.2 Response variable: Bond strength test. Controllable factors: Adhesive type, Curing pressure. Factor levels: Adhesive type - A, B, C, D, Curing pressure - low, nominal, high. Experimental layout: 4x3 full factorial experiment with 4 replications.

Experimental Run	Adhesive type	Curing pressure	Replication			
1	A	low	1	2	3	4
2	A	nominal	1	2	3	4
3	A	high	1	2	3	4
4	B	lowl	1	2	3	4
5	B	nominal	1	2	3	4
...	...	...	...	...	...	...

Experiment protocol:

1. Prepare 48 (3x4x4) computer cards.
2. Randomly split computer cards into 4 groups, 12 cards per group.
3. Randomly assign adhesive type to each group, 1 type per group.
4. Randomly assign curing pressure level to group of 12 cards, 1 level for 4 cards.
5. Attach to each computer card a sticker indicating experimental run, replication number, adhesive type and curing pressure.
6. Randomize the order of the 48 cards.
7. Run experiment according to randomized order and factorlevels indicated on sticker by keeping factors not participating in experiment fixed (e.g. amount of water).
8. Perform bond strength test and record data.
9. Analyze data with statistical model including main effects and interaction terms.

Solution 5.3 Blocking reduces the variability of the product relative to factors being studied. Examples include:

- Track and field athletes on a college team decided to investigate the effect of sleeping hours on athletic performance. The experiment consisted of sleeping a controlled amount of time prior to competitions. Individual athletes are natural experimental blocks.

- A natural extension of the example in section 5.2 on testing shoe sole materials is an experiment designed to test car tires. A natural block consists of the four wheels of a car, with an additional blocking variable determined by the position of the tires (front or rear).
- Experiments performed on plants are known to be sensitive to environmental conditions. Blocking variables in such experiments consist of neighboring plots of land where soil, humidity and temperature conditions are similar.

Solution 5.4 Main effects: τ_i^A, τ_j^B, τ_k^C each at 3 levels. First order interactions: τij^{AB}, τ_{ik}^{AC}, τ_{jk}^{BC} each at 9 levels. Second order interaction: τ_{ijk}^{ABC} at 27 levels.

Solution 5.5 $Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + e$. If $\beta_{12} = 0$ then the model is additive. The regression of Y on x_1 (x_2) are parallel for different values of x_2 (x_1). On the other hand, if $\beta_{12} \neq 0$ then the regression of Y on x_1, for two values of x_2, say $x_2^{(1)}$ and $x_2^{(2)}$, are $Y = \beta_0 + (\beta_1 + \beta_{12} x_2^{(1)}) x_1 + \beta_2 x_2^{(1)} + e$, and $Y = \beta_0 + (\beta_1 + \beta_{12} x_2^{(2)}) x_1 + \beta_2 x_2^{(2)} + e$ These are regression lines with different slopes. This means that the model is nonadditive and the extent of the interaction depends on β_{12}.

Solution 5.6 [1] Use the function *ttest_rel* from `scipy` to run a paired t-test.

```
ISC = mistat.load_data('SOCELL')
stats.ttest_rel(ISC['t2'], ISC['t1'])
```

```
Ttest_relResult(statistic=11.302222594952937,
pvalue=9.758913823226797e-09)
```

The difference between the two means is significant

Solution 5.7 [2] The permutation test requires the definition of the target statistic.

```
def statistic(x, y):
    return np.mean(x-y) # - np.mean(y)
```

The permutation test is then run as follows. The keyword argument `permutation_type='samples'` corresponds to the running a paired t-test.

```
def statistic(x, y):
    return np.mean(x) - np.mean(y)

res = stats.permutation_test((ISC['t2'], ISC['t3']), statistic,
                             permutation_type='samples', n_resamples=1000)
res.pvalue.round(5)
```

```
0.11588
```

A standard t-test gives;

[1] TODO: is this using a paired t-test or a normal t-test?

[2] TODO: same question - paired t-test or normal t-test? Is the code for RPCOMP.MTB available

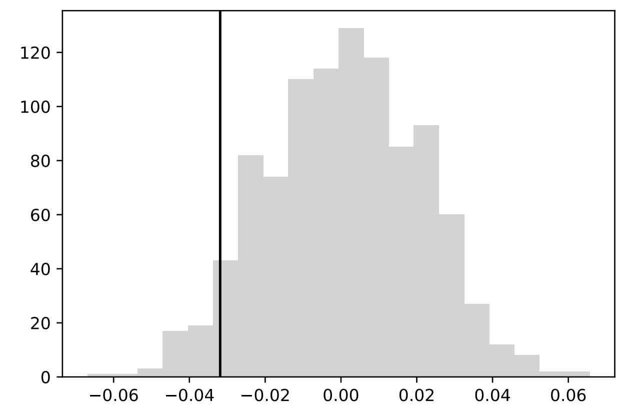


Fig. 5.1: Distribution resulting from the randomization paired comparison

Table 5.1: Result of four treatments in pencillin manufacturing

	Treatments			
blends	*A*	*B*	*C*	*D*
1	89	88	97	94
2	84	77	92	79
3	81	87	87	85
4	87	92	89	84
5	79	81	80	88

```
stats.ttest_rel(ISC['t2'], ISC['t3']).pvalue.round(5)
```

```
0.11567
```

Both results are comparable. We can also visualize the distribution of the statistic; see Fig. 5.1.

```
fig, ax = plt.subplots()
ax.hist(res.null_distribution, bins=20, color='lightgrey')
ax.axvline(statistic(ISC['t2'], ISC['t3']), color='black')
plt.show()
```

Solution 5.8 First create the dataset:

```
df = pd.DataFrame([
  ['B1', 'A', 89], ['B1', 'B', 88], ['B1', 'C', 97], ['B1', 'D', 94],
  ['B2', 'A', 84], ['B2', 'B', 77], ['B2', 'C', 92], ['B2', 'D', 79],
  ['B3', 'A', 81], ['B3', 'B', 87], ['B3', 'C', 87], ['B3', 'D', 85],
  ['B4', 'A', 87], ['B4', 'B', 92], ['B4', 'C', 89], ['B4', 'D', 84],
  ['B5', 'A', 79], ['B5', 'B', 81], ['B5', 'C', 80], ['B5', 'D', 88],
], columns=['blend', 'treatment', 'result'])
```

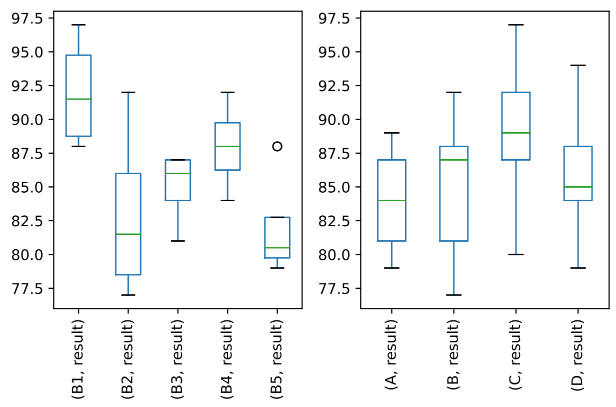


Fig. 5.2: Boxplot representation shows differences of blend and treatment effect.

Then use `statsmodels` to perform an ANOVA.

```
model = smf.ols('result ~ C(blend) + C(treatment)', data=df).fit()
anova.anova_lm(model)
```

```
                  df  sum_sq    mean_sq         F    PR(>F)
C(blend)         4.0   264.0  66.000000  3.504425  0.040746
C(treatment)     3.0    70.0  23.333333  1.238938  0.338658
Residual        12.0   226.0  18.833333       NaN       NaN
```

There are no significant differences between the treatments. There are significant differences between blends, the most extreme difference being between blend 1 and blend 5. See also Fig. 5.2.

```
fig, axes = plt.subplots(ncols=2)
df.groupby('blend').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[0])
df.groupby('treatment').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[1])
plt.tight_layout()
plt.show()
```

Solution 5.9 First create the dataset:

```
df = pd.DataFrame([
    [1, 'A', 38], [1, 'B', 30],   [2, 'C', 50], [2, 'D', 27],
    [3, 'E', 33], [3, 'F', 28],   [4, 'G', 62], [4, 'H', 30],
    [5, 'A', 37], [5, 'C', 25],   [6, 'B', 38], [6, 'H', 52],
    [7, 'D', 89], [7, 'E', 89],   [8, 'F', 27], [8, 'G', 75],
    [9, 'A', 17], [9, 'D', 25],   [10, 'B', 47], [10, 'G', 63],
    [11, 'C', 32], [11, 'F', 39], [12, 'E', 20], [12, 'H', 18],
    [13, 'A', 5], [13, 'E', 15],  [14, 'B', 45], [14, 'C', 38],
    [15, 'D', 11], [15, 'G', 24], [16, 'F', 37], [16, 'H', 39],
    [17, 'A', 23], [17, 'F', 40], [18, 'B', 20], [18, 'D', 14],
    [19, 'C', 18], [19, 'H', 10], [20, 'E', 22], [20, 'G', 52],
```

Table 5.2: Results of treatments $A, B, C, \cdots, H$ for different treatments

block	Treatments				block	Treatments			
1	A	38	B	30	15	D	11	G	24
2	C	50	D	27	16	F	37	H	39
3	E	33	F	28	17	A	23	F	40
4	G	62	H	30	18	B	20	D	14
5	A	37	C	25	19	C	18	H	10
6	B	38	H	52	20	E	22	G	52
7	D	89	E	89	21	A	66	G	67
8	F	27	G	75	22	B	23	F	46
9	A	17	D	25	23	C	22	E	28
10	B	47	G	63	24	D	20	H	40
11	C	32	F	39	25	A	27	H	32
12	E	20	H	18	26	B	10	E	40
13	A	5	E	15	27	C	32	G	33
14	B	45	C	38	28	D	18	F	23

```
    [21, 'A', 66], [21, 'G', 67], [22, 'B', 23], [22, 'F', 46],
    [23, 'C', 22], [23, 'E', 28], [24, 'D', 20], [24, 'H', 40],
    [25, 'A', 27], [25, 'H', 32], [26, 'B', 10], [26, 'E', 40],
    [27, 'C', 32], [27, 'G', 33], [28, 'D', 18], [28, 'F', 23],
], columns=['block', 'treatment', 'result'])
```

The ANOVA gives the following result:

```
model = smf.ols('result ~ C(block) + C(treatment)', data=df).fit()
anova.anova_lm(model)
```

```
                 df        sum_sq     mean_sq         F    PR(>F)
C(block)       27.0  15030.482143  556.684524  5.334113  0.000109
C(treatment)    7.0   1901.875000  271.696429  2.603376  0.042207
Residual       21.0   2191.625000  104.363095       NaN       NaN
```

Both the effects of the treatments and the blocks are significant at the $\alpha = 0.05$ level.

The Scheffé coefficient for $\alpha = 0.05$ is $S_{.05} = (7 \times F_{.95}[7, 21])^{1/2} = 4.173$ and $\hat{c}_p = \sqrt{104.363} = 10.216$. Thus the treatments can be divided into two homogenous groups, $G_1 = \{A, B, C, D, E, F, H\}$ and $G_2 = \{G\}$.[3] The 0.95 confidence interval for the difference of the group means is 20.413 ± 17.225 which shows that the group means are significantly different. See Fig. 5.3 for a visualization of the different groups.

```
df['group'] = ['G1' if t == 'G' else 'G2' for t in df['treatment']]
fig, axes = plt.subplots(ncols=3)
df.groupby('block').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[0])
df.groupby('treatment').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[1])
```

[3] TODO how is Scheffe calculated? why 7,21 and not 7,27? Is the split into groups subjective or somehow derived?

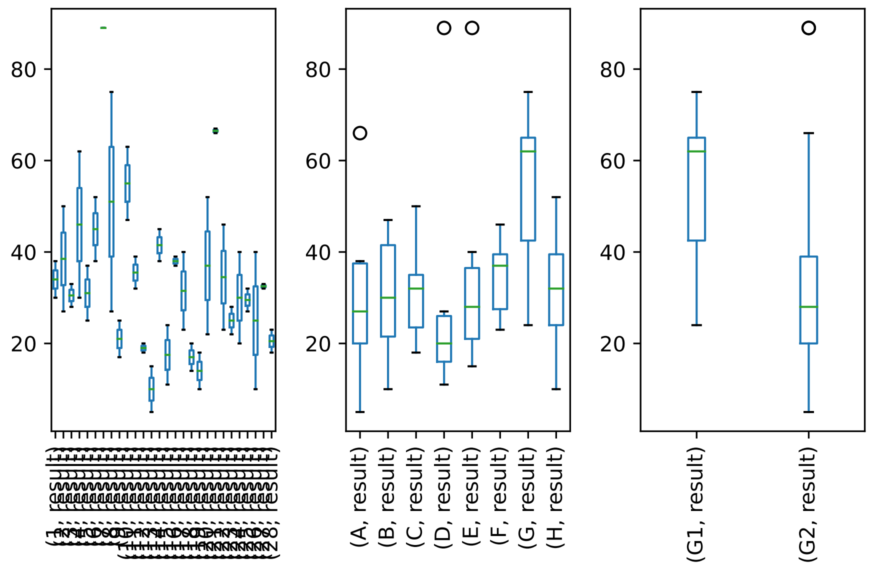


Fig. 5.3: Boxplot representation for Exercise 5.9.

Table 5.3: Compressive strengths of cement for different treatments

	batches			
Days	1	2	3	4
1	*A* 312	*B* 299	*C* 315	*D* 290
2	*C* 295	*A* 317	*D* 313	*B* 300
3	*B* 295	*D* 298	*A* 312	*C* 315
4	*D* 313	*C* 314	*B* 299	*A* 300

```
df.groupby('group').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[2])
plt.tight_layout()
plt.show()
```

Solution 5.10 Prepare the dataset.

```
df = pd.DataFrame([
    [1,1,'A',312], [1,2,'B',299], [1,3,'C',315], [1,4,'D',290],
    [2,1,'C',295], [2,2,'A',317], [2,3,'D',313], [2,4,'B',300],
    [3,1,'B',295], [3,2,'D',298], [3,3,'A',312], [3,4,'C',315],
    [4,1,'D',313], [4,2,'C',314], [4,3,'B',299], [4,4,'A',300],
], columns=['day', 'batch', 'mixture', 'result'])
```

Build a model and perform an ANOVA.

```
model = smf.ols('result ~ C(day) + C(batch) + C(mixture)', data=df).fit()
anova.anova_lm(model)
```

```
              df    sum_sq     mean_sq         F    PR(>F)
C(day)       3.0   16.1875    5.395833  0.048079  0.984704
C(batch)     3.0  165.6875   55.229167  0.492111  0.700652
C(mixture)  3.0  388.6875  129.562500  1.154446  0.401167
Residual     6.0  673.3750  112.229167       NaN       NaN
```

The differences between mixtures, batches or days are not significant.

Solution 5.11 In Python:

```
from mistat.design import doe
np.random.seed(2)

# Build design from factors
FacDesign = doe.full_fact({
    'k': [1500, 3000, 4500],
    's': [0.005, 0.0125, 0.02],
})

# Randomize design
FacDesign = FacDesign.sample(frac=1).reset_index(drop=True)

# Setup and run simulator with five replicates
# for each combination of factors
simulator = mistat.PistonSimulator(n_replicate=5, **FacDesign,
                                   m=30, v0=0.005, p0=90_000, t=290, t0=340)
result = simulator.simulate()

model = smf.ols('seconds ~ C(k) * C(s)', data=result).fit()
print(anova.anova_lm(model).round(4))
```

```
             df  sum_sq  mean_sq        F  PR(>F)
C(k)        2.0  0.0039   0.0019   2.0508  0.1434
C(s)        2.0  0.1030   0.0515  54.8672  0.0000
C(k):C(s)   4.0  0.0058   0.0015   1.5531  0.2078
Residual   36.0  0.0338   0.0009      NaN     NaN
```

Fig. 5.4 shows the marginal interaction plot.

```
_, ax = plt.subplots(figsize=[5, 4])
mistat.marginalInteractionPlot(result[['s', 'k', 'seconds']], 'seconds', ax=ax)
plt.show()
```

```
---------------------------------------------------------------------------KeyError
Traceback (most recent call last)Cell In [1], line 2
      1 _, ax = plt.subplots(figsize=[5, 4])
----> 2 mistat.marginalInteractionPlot(result[['s', 'k', 'seconds']],
'seconds', ax=ax)
      3 plt.show()
File /usr/local/lib/python3.9/site-
packages/mistat/design/graphs.py:175, in marginalInteractionPlot(df,
response, factors, interactions, levels, ax)
    170 for _, row in subdf.iterrows():
    171     mat[level1.index(row['l1']), level2.index(row['l2'])] =
row['mean']
    172     ax.plot(shift_1[row['l1']], row['mean'], zorder=10,
marker='s',
```

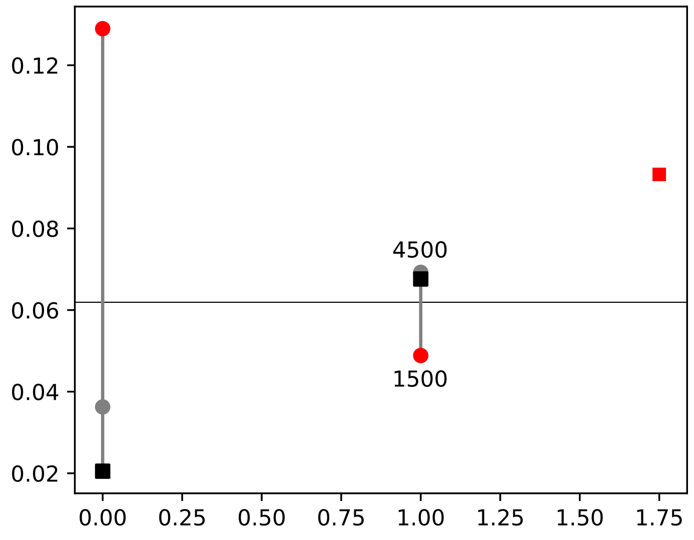


Fig. 5.4: Effect of Spring Coefficient k and Piston surface area s on Cycle Time

```
    173              markeredgewidth=0, fillstyle='right',
    174              markerfacecolor=markercol_1[f'{row["l1"]:.4f}'],
--> 175
markerfacecoloralt=markercol_2[f'{row["l2"]:.4f}'])
    177 for l1, values in zip(level1, mat):
    178     ax.plot([shift_1[l1]] * len(values), values,
color='lightgrey')
KeyError: '3000.0000'
```

Compared to Example 5.7, the cycle times are consistently lower. However, their behavior on changing the spring coefficient or the piston weight is similar.

Solution 5.12 Calculate the means and standard deviations of the cycle time by grouping on s and k.

```
# group and aggregate seconds by group
grouped = result.groupby(['s', 'k']).agg({'seconds': ['mean', 'std']})
# convert the multi-level index back to columns
grouped = grouped.reset_index()
# rename the columns to remove the multi-level index for the column names
grouped.columns = ['s', 'k', 'mean', 'std']
print(grouped)
```

```
        s     k      mean       std
0  0.0050  1500  0.093227  0.026990
1  0.0050  3000  0.148749  0.066597
2  0.0050  4500  0.144906  0.056237
3  0.0125  1500  0.033497  0.003607
4  0.0125  3000  0.037292  0.007923
5  0.0125  4500  0.037975  0.005075
6  0.0200  1500  0.019859  0.001296
7  0.0200  3000  0.021716  0.002626
8  0.0200  4500  0.019957  0.003601
```

Use `statsmodels` for the least squares estimates of `mean`

```
model = smf.ols(f"mean ~ s + k + s:k", data=grouped).fit()
model.params
```

```
Intercept    0.090547
s           -3.791223
k            0.000021
s:k         -0.001146
dtype: float64
```

and `std`.

```
model = smf.ols(f"std ~ s + k + s:k", data=grouped).fit()
model.params
```

```
Intercept    0.025397
s           -1.366082
k            0.000011
s:k         -0.000599
dtype: float64
```

Solution 5.13 Construct a data frame for a 2^4 factorial design in standard order:

```
treatments = []
for combo in itertools.product([0, 1], [0,1], [0,1], [0,1]):
    nu = sum(ij * 2**(j-1) for j, ij in enumerate(combo, 1))
    treatments.append({
        'nu': int(nu),
        'A': combo[0], 'B': combo[1],
        'C': combo[2], 'D': combo[3],
    })
# sort to standard order
df = pd.DataFrame(treatments).sort_values('nu', ignore_index=True)
df = df.set_index('nu')
# change the factors to (-1, 1)
for factor in ('A', 'B', 'C', 'D'):
    df.loc[df[factor] == 0, factor] = -1
```

Combine with the response values.

```
df['response'] = [72, 60, 90, 80, 65, 60, 85, 80, 60, 50,
                  88, 82, 58, 50, 84, 75]
```

(i) «» model = smf.ols("response A + B + C + D", data=df).fit() model.params @ The LSE of the main effects are $\hat{A} = -4.0625$, $\hat{B} = 11.8125$, $\hat{C} = -1.5625$ and $\hat{D} = -2.8125$.

(ii) [4] An estimate of σ^2 with 11 d.f. is $\hat{\sigma}^2 = 9.2898$.

(iii) A 0.99 level confidence interval for σ^2 is (3.819,39.254).

Solution 5.14 Prepare the dataset:

[4] TODO: I don't know how this and the CI are calculated

Table 5.4: Results of 3^2 factorial experiment with $n = 3$ observations

	A_1	A_2	A_3
B_1	18.3	17.9	19.1
	17.9	17.6	19.0
	18.5	16.2	18.9
B_2	20.5	18.2	22.1
	21.1	19.5	23.5
	20.7	18.9	22.9
B_3	21.5	20.1	22.3
	21.7	19.5	23.5
	21.9	18.9	23.3

```
df = pd.DataFrame(
    [['A1', 'B1', v] for v in [18.3, 17.9, 18.5]] +
    [['A2', 'B1', v] for v in [17.9, 17.6, 16.2]] +
    [['A3', 'B1', v] for v in [19.1, 19.0, 18.9]] +
    [['A1', 'B2', v] for v in [20.5, 21.1, 20.7]] +
    [['A2', 'B2', v] for v in [18.2, 19.5, 18.9]] +
    [['A3', 'B2', v] for v in [22.1, 23.5, 22.9]] +
    [['A1', 'B3', v] for v in [21.5, 21.7, 21.9]] +
    [['A2', 'B3', v] for v in [20.1, 19.5, 18.9]] +
    [['A3', 'B3', v] for v in [22.3, 23.5, 23.3]],
    columns=['a', 'b', 'result']
)
```

Build the model and performan an ANOVA.

```
model = smf.ols('result ~ C(a) + C(b) + C(a):C(b)', data=df).fit()
anova.anova_lm(model)
```

```
             df     sum_sq    mean_sq          F        PR(>F)
C(a)        2.0  43.080741  21.540370  70.495758  3.055533e-09
C(b)        2.0  54.169630  27.084815  88.641212  4.802685e-10
C(a):C(b)   4.0   4.345926   1.086481   3.555758  2.633500e-02
Residual   18.0   5.500000   0.305556        NaN           NaN
```

Visualization of the factor and factor interactions are shown in Fig. 5.5

```
df['a:b'] = [f'{a}:{b}' for a, b in zip(df['a'], df['b'])]
fig, axes = plt.subplots(ncols=3)
df.groupby('a').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[0])
df.groupby('b').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[1])
df.groupby('a:b').boxplot(column='result', subplots=False,
                            rot=90, grid=False, ax=axes[2])
plt.tight_layout()
plt.show()
```

Solution 5.15 The subgroup of defining parameters is

```
generators = ['ABCDG', 'ABEFH']
print(mistat.subgroupOfDefining(generators))
```

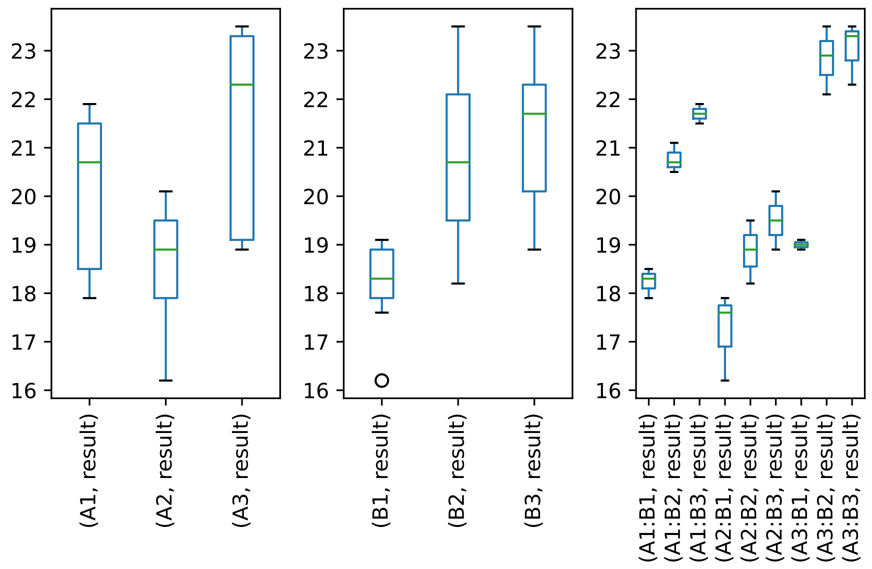


Fig. 5.5: Boxplot representation for Exercise 5.14.

```
['', 'ABCDG', 'ABEFH', 'CDEFGH']
```

The design is therefore of resolution **V**.
The aliases of the design using the generators are:

```
from mistat.design.doeUtilities import aliasesInSubgroup
for main_effect in ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'):
    print(main_effect,
          aliasesInSubgroup(main_effect, generators))
```

```
A ['ACDEFGH', 'BCDG', 'BEFH']
B ['ACDG', 'AEFH', 'BCDEFGH']
C ['ABCEFH', 'ABDG', 'DEFGH']
D ['ABCG', 'ABDEFH', 'CEFGH']
E ['ABCDEG', 'ABFH', 'CDFGH']
F ['ABCDFG', 'ABEH', 'CDEGH']
G ['ABCD', 'ABEFGH', 'CDEFH']
H ['ABCDGH', 'ABEF', 'CDEFG']
```

The shortest alias are of length four. This means our design has resolution **IV**.
The aliases for the first order interaction with the factor *A* are:

```
for interaction in ('B', 'C', 'D', 'E', 'F', 'G', 'H'):
    print(f'A{interaction}',
          aliasesInSubgroup(f'A{interaction}', generators))
```

```
AB ['ABCDEFGH', 'CDG', 'EFH']
AC ['ADEFGH', 'BCEFH', 'BDG']
AD ['ACEFGH', 'BCG', 'BDEFH']
```

Table 5.5: Design matrix and response of 2^2 factorial design

X_1	X_2	Y
−1	−1	55.8
−1	−1	54.4
1	−1	60.3
1	−1	60.9
−1	1	63.9
−1	1	64.4
1	1	67.9
1	1	68.5
0	0	61.5
0	0	62.0
0	0	61.9
0	0	62.4

```
AE ['ACDFGH', 'BCDEG', 'BFH']
AF ['ACDEGH', 'BCDFG', 'BEH']
AG ['ACDEFH', 'BCD', 'BEFGH']
AH ['ACDEFG', 'BCDGH', 'BEF']
```

You can construct a block of the 2^{8-2} design using this Python code.

```
generator = 'A B C D E F G H ABCDG ABEFH'
design = pd.DataFrame(fracfact(generator), columns=generator.split())
fracfact_design = design.query('ABCDG == 1 & ABEFH == 1')
```

Solution 5.16 The subgroup of defining parameters and therefore the parameters that are not estimable for this generator are:

```
print(mistat.subgroupOfDefining(['ACE', 'ABEF', 'ABCD']))
```

```
['', 'ABCD', 'ABEF', 'ACE', 'ADF', 'BCF', 'BDE', 'CDEF']
```

Solution 5.17 **(i)** Prepare the dataset.

```
df = pd.DataFrame([
    [-1, -1, 55.8], [-1, -1, 54.4], [1, -1, 60.3], [1, -1, 60.9],
    [-1, 1, 63.9], [-1, 1, 64.4], [1, 1, 67.9], [1, 1, 68.5],
    [0, 0, 61.5], [0, 0, 62.0], [0, 0, 61.9], [0, 0, 62.4]
], columns=['X1', 'X2', 'Y'])
```

Build the model.

```
formula = ('Y ~ X1 + X2 + X1:X2')
model = smf.ols(formula, data=df).fit()
print(model.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.986
Dependent Variable: Y                AIC:                19.8454
Date:               2022-10-14 21:59 BIC:                21.7851
No. Observations:   12               Log-Likelihood:     -5.9227
Df Model:           3                F-statistic:        262.0
Df Residuals:       8                Prob (F-statistic): 2.52e-08
R-squared:          0.990            Scale:              0.23568
------------------------------------------------------------------
              Coef.   Std.Err.     t      P>|t|    [0.025   0.975]
------------------------------------------------------------------
Intercept    61.9917    0.1401  442.3492  0.0000  61.6685  62.3148
X1            2.3875    0.1716   13.9101  0.0000   1.9917   2.7833
X2            4.1625    0.1716   24.2516  0.0000   3.7667   4.5583
X1:X2        -0.3625    0.1716   -2.1120  0.0677  -0.7583   0.0333
-----------------------------------------------------------------
Omnibus:              0.321        Durbin-Watson:           2.580
Prob(Omnibus):        0.852        Jarque-Bera (JB):        0.447
Skew:                 0.054        Prob(JB):                0.800
Kurtosis:             2.061        Condition No.:           1
=================================================================
```

The response function is $\hat{Y} = 62.0 + 2.39X_1 + 4.16X_2 - 0.363X_1X_2$, with $R^2 = 0.99$. The contour plot is given in Figure 11.1.

```
def plotResponseSurface(model, ncontours=20):
    x1 = np.linspace(-1, 1)
    x2 = np.linspace(-1, 1)
    X1, X2 = np.meshgrid(x1, x2)
    exog = pd.DataFrame({'X1': X1.ravel(), 'X2': X2.ravel()})
    responses = model.predict(exog=exog)
    CS = plt.contour(x1, x2,
                responses.values.reshape(len(x2), len(x1)),
                ncontours, colors='gray')
    ax = plt.gca()
    ax.clabel(CS, inline=True, fontsize=10)
    ax.set_xlabel('X1')
    ax.set_ylabel('X2')
    return ax

plotResponseSurface(model)
plt.show()
```

(ii)

```
# derive variance around the regression using an ANOVA (mean_sq of residuals)
res = anova.anova_lm(model)
res
```

```
           df      sum_sq     mean_sq           F        PR(>F)
X1        1.0   45.601250   45.601250  193.490387  6.905851e-07
X2        1.0  138.611250  138.611250  588.140552  8.915958e-09
X1:X2     1.0    1.051250    1.051250    4.460552  6.766219e-02
Residual  8.0    1.885417    0.235677         NaN           NaN
```

An estimate of the variance is $\hat{\sigma}^2 = 0.136667$[5], 3 d.f. The variance around the regression is $s^2_{y|(x)} = 0.2357$, 8 d.f. Use F-distribution to derive the p-value.

[5] TODO: how is this calculated? eqn (5.7.41)? I get a different result with it; about double the value in the solution 0.26785

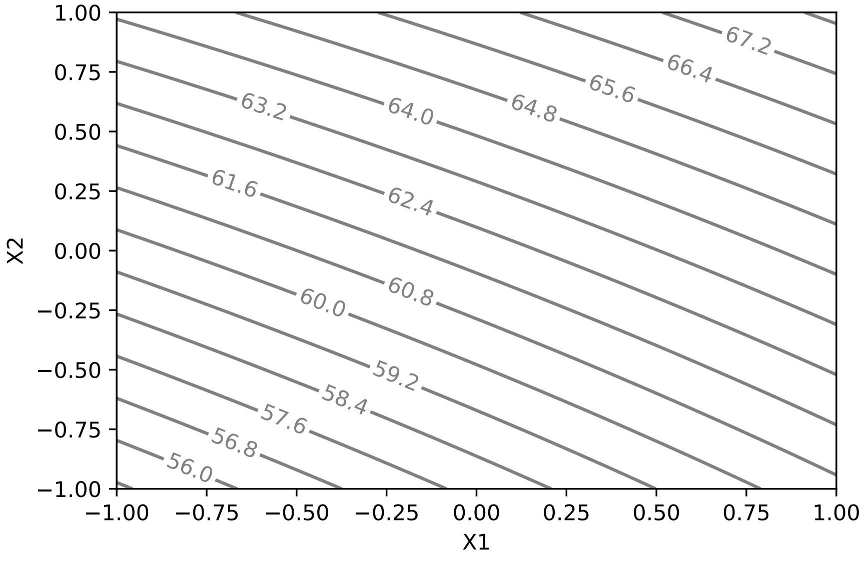


Fig. 5.6: Contour plot of equal responses (Exercise 5.17)

```
# Estimate of variance?
# ['Residual', 'mean_sq'] gives variance around regression
sigma2 =  0.13667
var_residuals = 0.235677

F = var_residuals / sigma2
p = 1 - stats.f(8, 3).cdf(F)
print(f'F-ratio: {F:.4f}; p-value: {p:.2f}')
```

```
F-ratio: 1.7244; p-value: 0.36
```

The ratio $F = \frac{s^2_{y|(x)}}{\hat{\sigma}^2} = 1.7246$ gives a p-value of $P = 0.36$. This shows there is no significant differences between the variances.

Solution 5.18 Prepare the dataset and build a regression model.

```
df = pd.DataFrame([
    [1, 0, 95.6], [0.5, 0.866, 77.9], [-0.5, 0.866, 76.2],
    [-1, 0, 54.5], [-0.5, -0.866, 63.9], [0.5, -0.866, 79.1],
    [0, 0, 96.8], [0, 0, 94.8], [0, 0, 94.4],
], columns=['X1', 'X2', 'Y'])

formula = ('Y ~ X1 + X2 + X1*X2 + I(X1**2) + I(X2**2)')
model = smf.ols(formula, data=df).fit()
print(model.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
```

Table 5.6: Design matrix and the response for a control composite design of Exercise 5.18

X_1	X_2	Y
1.0	0.000	95.6
0.5	0.866	77.9
−0.5	0.866	76.2
−1.0	0	54.5
−0.5	−0.866	63.9
0.5	−0.866	79.1
0	0	96.8
0	0	94.8
0	0	94.4

```
Model:              OLS              Adj. R-squared:     0.855
Dependent Variable: Y                AIC:                59.2942
Date:               2022-10-14 21:59 BIC:                60.4776
No. Observations:   9                Log-Likelihood:     -23.647
Df Model:           5                F-statistic:        10.47
Df Residuals:       3                Prob (F-statistic): 0.0408
R-squared:          0.946            Scale:              33.638
-----------------------------------------------------------------
               Coef.   Std.Err.    t    P>|t|   [0.025   0.975]
-----------------------------------------------------------------
Intercept      95.3333   3.3485 28.4703 0.0001  84.6768 105.9898
X1             16.5167   3.3485  4.9325 0.0160   5.8602  27.1732
X2              3.2044   3.3486  0.9569 0.4092  -7.4524  13.8612
X1:X2          -7.7945   6.6972 -1.1638 0.3286 -29.1081  13.5191
I(X1 ** 2)    -20.2833   5.2945 -3.8310 0.0313 -37.1327  -3.4339
I(X2 ** 2)    -21.3179   5.2948 -4.0262 0.0275 -38.1683  -4.4675
-----------------------------------------------------------------
Omnibus:              3.213       Durbin-Watson:          3.565
Prob(Omnibus):        0.201       Jarque-Bera (JB):       0.950
Skew:                 0.006       Prob(JB):               0.622
Kurtosis:             1.408       Condition No.:          4
=================================================================
```

The response function is $Y = 95.3 + 16.5X_1 + 3.20X_2 - 20.3X_1^2 - 21.3X_2^2 - 7.79X_1X_2$.

We can use *ResponseSurfaceMethod* from the `mistat` package to identify the stationary point.

```
rsm = mistat.ResponseSurfaceMethod(model, ['X1', 'X2'])
stationary = rsm.stationary_point()
stationary
```

```
X1    0.407004
X2    0.000751
dtype: float64
```

The stationary point is at (0.4, 0).

(ii) To plot the response surface, we reuse the function defined in Exercise 5.17; see Fig. 5.7.

```
ax = plotResponseSurface(model)
ax.scatter(*stationary, color='black')
plt.show()
```

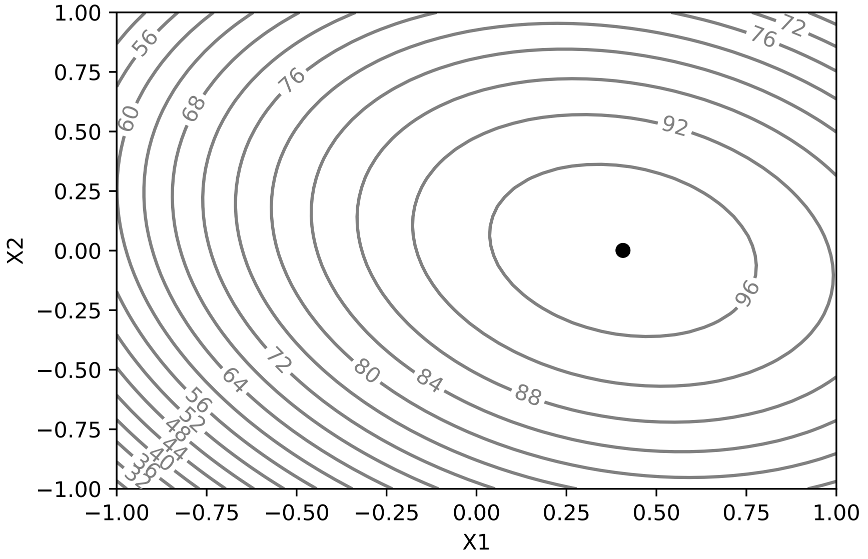


Fig. 5.7: Contour plot of equal responses; the stationary point is shown as a black dot (Exercise 5.18)

(iii) Perform an ANOVA.

```
anova.anova_lm(model)
```

```
                df      sum_sq     mean_sq          F    PR(>F)
X1             1.0  818.400833  818.400833  24.329813  0.015976
X2             1.0   30.802500   30.802500   0.915712  0.409198
X1:X2          1.0   45.562500   45.562500   1.354504  0.328640
I(X1 ** 2)     1.0  320.800500  320.800500   9.536911  0.053790
I(X2 ** 2)     1.0  545.280333  545.280333  16.210355  0.027533
Residual       3.0  100.913333   33.637778        NaN       NaN
```

The factor X_1 and the squares X_1, X_2 are significant. X_2 and the interaction $X1 : X2$ are not significant.

Chapter 6
Quality by Design

Import required modules and define required functions

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
from pyDOE2 import fracfact

import mistat
```

Solution 6.1 Create a factorial design of the five factors using the highest and lowest levels.

```
from mistat.design import doe
np.random.seed(1)

# Build design from factors
FacDesign = doe.full_fact({
    'm': [30, 60],
    's': [0.005, 0.02],
    'v0': [0.002, 0.01],
    'k': [1500, 4500],
    't': [290, 296],
})

# Randomize design
FacDesign = FacDesign.sample(frac=1).reset_index(drop=True)
```

Run the Piston simulator using 100 replicates for each factor combination.

```
# Setup and run simulator with 100 replicates
# for each combination of factors
simulator = mistat.PistonSimulator(n_replicate=100, **FacDesign,
                                   p0=100_000, t0=350)
result = simulator.simulate()
```

Group the results by the five factor levels and calculate mean and standard deviation of the cycle time.

```
factors = ['m', 's', 'v0', 'k', 't']
result = result.groupby(factors, as_index=False).agg({'seconds': ['mean', 'std']})
result.columns = [*factors, 'mean', 'std']
```

(i) Add the MSE column and sort by its value to determine the best factor combinations.

```
result['MSE'] = (result['mean'] - 0.02)**2 + result['std']**2
result = result.sort_values('MSE')
best_MSE = result.iloc[0,:]
result.head()
```

```
     m      s     v0     k    t      mean       std       MSE
26  60  0.020  0.002  4500  290  0.009580  0.002223  0.000114
27  60  0.020  0.002  4500  296  0.009659  0.002783  0.000115
24  60  0.020  0.002  1500  290  0.009201  0.002364  0.000122
25  60  0.020  0.002  1500  296  0.008888  0.002322  0.000129
16  60  0.005  0.002  1500  290  0.023836  0.010829  0.000132
```

```
best_MSE
```

```
m         60.000000
s          0.020000
v0         0.002000
k       4500.000000
t        290.000000
mean       0.009580
std        0.002223
MSE        0.000114
Name: 26, dtype: float64
```

(ii) Calculate the SN ratio and determine the factor combinations with the largest SN ratio.

```
result['SN'] = 10 * np.log10(result['mean']**2 / result['std']**2 - 1/100)
result = result.sort_values('SN', ascending=False).head()
best_SN = result.iloc[0,:]
result.head()
```

```
     m     s    v0     k    t      mean       std       MSE         SN
29  60  0.02  0.01  1500  296  0.049291  0.003361  0.000869  23.325647
13  30  0.02  0.01  1500  296  0.042370  0.003010  0.000509  22.970572
30  60  0.02  0.01  4500  290  0.056468  0.004261  0.001348  22.445850
28  60  0.02  0.01  1500  290  0.049085  0.003820  0.000861  22.177197
31  60  0.02  0.01  4500  296  0.057546  0.004480  0.001430  22.173743
```

```
best_SN
```

```
m         60.000000
s          0.020000
v0         0.010000
k       1500.000000
t        296.000000
mean       0.049291
std        0.003361
MSE        0.000869
SN        23.325647
Name: 29, dtype: float64
```

This treatment combination has an MSE of 0.00087, which is 7.7 times bigger than the minimal MSE. This exercise demonstrates the need for a full analysis of the effects and the dangers of relying on a simplistic observation of the experiment's outcomes.

Solution 6.2 Create a 2^7 full factorial design and run the piston simulator with 100 replicates for each factor combination. After grouping by factors, determine mean and standard deviation of the cycle time. Finally, calculate the signal noise ratio SN.

```
np.random.seed(1)

# Build design from factors
FacDesign = doe.full_fact({
    'm': [30, 60],
    's': [0.005, 0.02],
    'v0': [0.002, 0.01],
    'k': [1500, 4500],
    't': [290, 296],
    'p0': [90_000, 110_000],
    't0': [340, 360],
})

# Randomize design
FacDesign = FacDesign.sample(frac=1).reset_index(drop=True)

# Setup and run simulator with five replicates
# for each combination of factors
simulator = mistat.PistonSimulator(n_replicate=100, **FacDesign)
result = simulator.simulate()
factors = ['m', 's', 'v0', 'k', 't', 'p0', 't0']
result = result.groupby(factors, as_index=False).agg({'seconds': ['mean', 'std']})
result.columns = [*factors, 'mean', 'std']
result['SN'] = 10 * np.log10(result['mean']**2 / result['std']**2 - 1/100)
```

The regression analysis of SN on the seven factors (only main effects) results in.

```
model = smf.ols('SN ~ m + s + k + t + v0 + p0 + t0', data=result).fit()
print(model.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.899
Dependent Variable: SN               AIC:                522.9904
Date:               2022-10-15 13:04 BIC:                545.8067
No. Observations:   128              Log-Likelihood:     -253.50
Df Model:           7                F-statistic:        163.3
Df Residuals:       120              Prob (F-statistic): 3.28e-58
R-squared:          0.905            Scale:              3.2790
-----------------------------------------------------------------
               Coef.   Std.Err.    t    P>|t|   [0.025    0.975]
-----------------------------------------------------------------
Intercept       5.0686  16.6968  0.3036 0.7620 -27.9899   38.1271
m              -0.0094   0.0107 -0.8845 0.3782  -0.0306    0.0117
s             484.4714  21.3403 22.7021 0.0000 442.2191  526.7238
k               0.0001   0.0001  0.5709 0.5691  -0.0002    0.0003
t               0.0012   0.0534  0.0217 0.9827  -0.1045    0.1068
v0           1001.1879  40.0131 25.0215 0.0000 921.9647 1080.4112
p0             -0.0000   0.0000 -0.2122 0.8323  -0.0000    0.0000
t0             -0.0095   0.0160 -0.5949 0.5530  -0.0412    0.0222
-----------------------------------------------------------------
Omnibus:              10.591      Durbin-Watson:         0.520
```

```
Prob(Omnibus):        0.005      Jarque-Bera (JB):      4.418
Skew:                 -0.164     Prob(JB):              0.110
Kurtosis:             2.151      Condition No.:         25136579
=================================================================
* The condition number is large (3e+07). This might indicate
strong multicollinearity or other numerical problems.
```

We see that only factors s and $v0$ (piston surface area, initial gas volume) have a significant effect. The R^2 is only 90.5%. Adding to the regression equation the first order interactions between s and $v0$, we get

```
model = smf.ols('SN ~ s + v0 + s*v0', data=result).fit()
print(model.summary2())
```

```
                  Results: Ordinary least squares
==================================================================
Model:                OLS              Adj. R-squared:     0.964
Dependent Variable:   SN               AIC:                388.4606
Date:                 2022-10-15 13:04 BIC:                399.8687
No. Observations:     128              Log-Likelihood:     -190.23
Df Model:             3                F-statistic:        1128.
Df Residuals:         124              Prob (F-statistic): 8.90e-90
R-squared:            0.965            Scale:              1.1808
------------------------------------------------------------------
            Coef.     Std.Err.    t    P>|t|    [0.025     0.975]
------------------------------------------------------------------
Intercept     5.0013    0.3365 14.8604 0.0000     4.3351     5.6674
s           203.8500   23.0871  8.8296 0.0000   158.1541   249.5459
v0          416.5600   46.6710  8.9254 0.0000   324.1849   508.9350
s:v0      46770.2389 3201.6068 14.6084 0.0000 40433.3622 53107.1157
------------------------------------------------------------------
Omnibus:               0.674        Durbin-Watson:           1.053
Prob(Omnibus):         0.714        Jarque-Bera (JB):        0.303
Skew:                  -0.032       Prob(JB):                0.859
Kurtosis:              3.229        Condition No.:           33340
==================================================================
* The condition number is large (3e+04). This might indicate
strong multicollinearity or other numerical problems.
```

We see that the added interaction $s*v0$ is very significant. The regression equation with the interaction terms predicts the SN better, $R^2 = 96.5\%$.

Solution 6.3 The approximations to the expected values and variances are as follows:

(i)

$$E\left\{\frac{X_1}{X_2}\right\} \approx \frac{\xi_1}{\xi_2} - \sigma_{12}\frac{1}{\xi_2^2} + \sigma_2^2\frac{\xi_1}{\xi_2^3}$$

$$V\left\{\frac{X_1}{X_2}\right\} \approx \frac{\sigma_1^2}{\xi_2^2} + \sigma_2^2\frac{\xi_1^2}{\xi_2^4} - 2\sigma_{12}\frac{\xi_1}{\xi_2^3}$$

(ii)

$$E\left\{\log \frac{X_1^2}{X_2^2}\right\} \approx \log\left(\frac{\xi_1}{\xi_2}\right)^2 - \frac{\sigma_1^2}{\xi_1^2} + \frac{\sigma_2^2}{\xi_2^2}$$
$$V\left\{\log\left(\frac{X_1}{X_2}\right)^2\right\} \approx \frac{4\sigma_1^2}{\xi_1^2} + \frac{4\sigma_2^2}{\xi_2^2} - 8\frac{\sigma_{12}}{\xi_1\xi_2}.$$

(iii)

$$E\{(X_1^2+X_2^2)^{1/2}\} \approx (\xi_1^2+\xi_2^2)^{1/2} + \frac{\sigma_1^2}{2(\xi_1^2+\xi_2^2)^{1/2}}\left(1-\frac{\xi_1^2}{\xi_1^2+\xi_2^2}\right)$$
$$+ \frac{\sigma_2^2}{2(\xi_1^2+\xi_2^2)^{1/2}}\left(1-\frac{\xi_2^2}{\xi_1^2+\xi_2^2}\right) - \frac{\sigma_{12}\xi_1\xi_2}{(\xi_1^2+\xi_2^2)^{3/2}};$$
$$V\{(X_1^2+X_2^2)^{1/2}\} \approx \frac{\sigma_1^2\xi_1^2}{(\xi_1^2+\xi_2^2)} + \frac{\sigma_2^2\xi_2^2}{(\xi_1^2+\xi_2^2)} + 2\frac{\sigma_{12}\xi_1\xi_2}{(\xi_1^2+\xi_2^2)}.$$

Solution 6.4 Approximation formulas yield: $E\{Y\} \approx 0.5159$ and $V\{Y\} \approx 0.06762$. Simulation with 5,000 runs yields the estimates $E\{Y\} \approx 0.6089$, $V\{Y\} \approx 0.05159$. The first approximation of $E\{Y\}$ is significantly lower than the simulation estimate.

Solution 6.5 We have that $E\{\bar{X}_n\} = \mu$, $V\{\bar{X}_n\} = \frac{\sigma^2}{n}$, $E\{S_n^2\} = \sigma^2$ and $V\{S_n^2\} = \frac{2\sigma^4}{n-1}$. The first and second order partial derivatives of $f(\mu,\sigma^2) = 2\log(\mu) - \log(\sigma^2)$ are

$$\frac{\partial}{\partial\mu} f = \frac{2}{\mu}, \qquad \frac{\partial}{\partial\sigma^2} f = -\frac{1}{\sigma^2},$$
$$\frac{\partial^2}{\partial\mu\partial\sigma^2} f = 0, \qquad \frac{\partial^2}{\partial\mu^2} f = -\frac{2}{\mu^2} \quad \text{and} \quad \frac{\partial^2}{\partial(\sigma^2)^2} f = \frac{1}{\sigma^4}.$$

Thus the approximations to the expected value and variance of $Y = \log\left(\frac{\bar{X}_n^2}{S_n^2}\right)$ are

$$E\left\{\log\left(\frac{\bar{X}_n^2}{S_n^2}\right)\right\} \approx \log\left(\frac{\mu^2}{\sigma^2}\right) - \frac{\sigma^2}{n\mu^2} + \frac{1}{n-1} \quad \text{and}$$
$$V\left\{\log\left(\frac{\bar{X}_n^2}{S_n^2}\right)\right\} \approx \frac{4\sigma^2}{n\mu^2} + \frac{2}{n-1}.$$

Solution 6.6 First, create the data frame.

```
df = pd.DataFrame([
  [1, 1, 1, 1, 1, 1, 1, 1, 2.5, 0.0827],
  [1, 1, 2, 2, 2, 2, 2, 2, 2.684, 0.1196],
  [1, 1, 3, 3, 3, 3, 3, 3, 2.66, 0.1722],
  [1, 2, 1, 1, 2, 2, 3, 3, 1.962, 0.1696],
  [1, 2, 2, 2, 3, 3, 1, 1, 1.87, 0.1168],
  [1, 2, 3, 3, 1, 1, 2, 2, 2.584, 0.1106],
  [1, 3, 1, 2, 1, 3, 2, 3, 2.032, 0.0718],
```

Table 6.1: Results of experiment based on L_{18} orthogonal array

	Factors									
run	1	2	3	4	5	6	7	8	$\bar{X}$	S
1	1	1	1	1	1	1	1	1	2.500	0.0827
2	1	1	2	2	2	2	2	2	2.684	0.1196
3	1	1	3	3	3	3	3	3	2.660	0.1722
4	1	2	1	1	2	2	3	3	1.962	0.1696
5	1	2	2	2	3	3	1	1	1.870	0.1168
6	1	2	3	3	1	1	2	2	2.584	0.1106
7	1	3	1	2	1	3	2	3	2.032	0.0718
8	1	3	2	3	2	1	3	1	3.267	0.2101
9	1	3	3	1	3	2	1	2	2.829	0.1516
10	2	1	1	3	3	2	2	1	2.660	0.1912
11	2	1	2	1	1	3	3	2	3.166	0.0674
12	2	1	3	2	2	1	1	3	3.323	0.1274
13	2	2	1	2	3	1	3	2	2.576	0.0850
14	2	2	2	3	1	2	1	3	2.308	0.0964
15	2	2	3	1	2	3	2	1	2.464	0.0385
16	2	3	1	3	2	3	1	2	2.667	0.0706
17	2	3	2	1	3	1	2	3	3.156	0.1569
18	2	3	3	2	1	2	3	1	3.494	0.0473

```
  [1, 3, 2, 3, 2, 1, 3, 1, 3.267, 0.2101],
  [1, 3, 3, 1, 3, 2, 1, 2, 2.829, 0.1516],
  [2, 1, 1, 3, 3, 2, 2, 1, 2.66, 0.1912],
  [2, 1, 2, 1, 1, 3, 3, 2, 3.166, 0.0674],
  [2, 1, 3, 2, 2, 1, 1, 3, 3.323, 0.1274],
  [2, 2, 1, 2, 3, 1, 3, 2, 2.576, 0.085],
  [2, 2, 2, 3, 1, 2, 1, 3, 2.308, 0.0964],
  [2, 2, 3, 1, 2, 3, 2, 1, 2.464, 0.0385],
  [2, 3, 1, 3, 2, 3, 1, 2, 2.667, 0.0706],
  [2, 3, 2, 1, 3, 1, 2, 3, 3.156, 0.1569],
  [2, 3, 3, 2, 1, 2, 3, 1, 3.494, 0.0473],
], columns=['F1', 'F2', 'F3', 'F4', 'F5',
            'F6', 'F7', 'F8', 'Xbar', 'S'])
```

Transform the factors to [-1, 1] for factor 1 and to [-1, 0, 1] for the remaining factors. Calculate the ratio.

```
df['F1'] = (df['F1']-1)*2-1
for column in ['F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8']:
    df[column] = df[column] - 2
df['SNR'] = np.log(df['Xbar'] / df['S'])
```

Perform a regression analysis taking only the linear effects into account.

```
model = smf.ols('SNR ~ F1 + F2 + F3 + F4 + F5 + F6 + F7 + F8', data=df).fit()
```

Regression analysis yields the following:

```
print(model.summary2().tables[1].round(5))
```

```
             Coef.  Std.Err.         t    P>|t|   [0.025   0.975]
Intercept  3.22601   0.07518  42.90997  0.00000  3.05594  3.39608
F1         0.26565   0.07518   3.53348  0.00638  0.09558  0.43572
F2         0.07900   0.09208   0.85797  0.41317 -0.12929  0.28729
F3         0.13848   0.09208   1.50392  0.16686 -0.06982  0.34677
F4        -0.14339   0.09208  -1.55733  0.15382 -0.35169  0.06490
F5        -0.31231   0.09208  -3.39185  0.00798 -0.52061 -0.10402
F6         0.12630   0.09208   1.37164  0.20340 -0.08200  0.33459
F7         0.02632   0.09208   0.28587  0.78145 -0.18197  0.23462
F8        -0.17109   0.09208  -1.85810  0.09610 -0.37938  0.03720
```

The linear effects of F1, F5, and F8 are significant. There might be significant interactions or quadratic effects.

Solution 6.7 Use the generators from Table 5.29 to create the different designs.

```
generators = {
    '2_7': 'A B C D E F G',
    '2_7_1': 'A B C D E F G ABCDEFG',
    '2_7_2': 'A B C D E F G ABCDF ABDEG',
    '2_7_3': 'A B C D E F G ABCE BCDF ACDG',
}
```

Create the designs using the `pyDOE2` function *fracfact*

```
designs = {}
for name, generator in generators.items():
  designs[name] = pd.DataFrame(fracfact(generator), columns=generator.split())

# reduce the fractional factorial designs to a single block
designs['2_7_1'] = designs['2_7_1'].query('ABCDEFG == 1')
designs['2_7_2'] = designs['2_7_2'].query('ABCDF == 1 & ABDEG == 1')
designs['2_7_3'] = designs['2_7_3'].query('ABCE == 1 & BCDF == 1 & ACDG == 1')
```

The function *fracfact* returns a design matrix with values $(-1, 1)$. We need to map these to the actual factor levels.

```
FacLevels = {
    'm': [30, 60],
    's': [0.005, 0.02],
    'v0': [0.002, 0.01],
    'k': [1500, 4500],
    't': [290, 296],
    'p0': [90_000, 110_000],
    't0': [340, 360],
}
FacMap = {'A': 'm', 'B': 's', 'C': 'v0', 'D': 'k',
          'E': 't', 'F': 'p0', 'G': 't0'}

for name, design in designs.items():
    # replace (-1, 1) with factor levels
    facDesign = {}
    for colname in design:
        if colname not in FacMap: # skip generators
            continue
        factor = FacMap[colname]
        levels = FacLevels[factor]
        facDesign[factor] = [levels[max(0, int(v))] for v in design[colname]]
    designs[name] = pd.DataFrame(facDesign)
```

For each design execute the *PistonSimulation* with 5 replicates and determine the SN ratio.

```
results = {}
for name, design in designs.items():
    np.random.seed(1)
    # Setup and run simulator
    simulator = mistat.PistonSimulator(n_replicate=5, **design)
    result = simulator.simulate()
    factors = list(FacLevels)
    result = result.groupby(factors, as_index=False).agg({'seconds': ['mean', 'std']})
    result.columns = [*factors, 'mean', 'std']
    result['SN'] = np.log10(result['mean']**2 / result['std']**2)
    results[name] = result
```

Build linear regression models to estimate the main effects.

```
models = {}
for name, result in results.items():
    model = smf.ols('mean ~ m + s + k + t + v0 + p0 + t0', data=result).fit()
    models[name] = model
```

We can now look at the effect of the design on model performance metrics:

```
for name, model in models.items():
    print(f'{name:10s}: r2={model.rsquared:.3f}, r2_adj={model.rsquared_adj:.3f}')
```

```
2_7       : r2=0.769, r2_adj=0.755
2_7_1     : r2=0.777, r2_adj=0.749
2_7_2     : r2=0.765, r2_adj=0.697
2_7_3     : r2=0.788, r2_adj=0.602
```

While the r^2 metric stays basically the same for all four designs, the adjusted r^2 drops with the size of the design from the full to the 1/8 factorial.

We see a similar effect on the signficance levels and the confidence intervals of the parameter estimates.

```
for name, model in models.items():
    print(name)
    print(model.summary2().tables[1].round(4))
```

```
2_7
             Coef.  Std.Err.        t   P>|t|   [0.025   0.975]
Intercept   0.2183    0.4116   0.5304  0.5968  -0.5967   1.0333
m           0.0005    0.0003   2.0359  0.0440   0.0000   0.0011
s          -6.8075    0.5261 -12.9393  0.0000  -7.8492  -5.7658
k           0.0000    0.0000   1.6946  0.0927  -0.0000   0.0000
t          -0.0004    0.0013  -0.3135  0.7544  -0.0030   0.0022
v0         14.7097    0.9865  14.9116  0.0000  12.7566  16.6628
p0         -0.0000    0.0000  -1.3802  0.1701  -0.0000   0.0000
t0         -0.0000    0.0004  -0.0287  0.9771  -0.0008   0.0008
2_7_1
             Coef.  Std.Err.        t   P>|t|   [0.025   0.975]
Intercept  -0.1876    0.5760  -0.3257  0.7458  -1.3416   0.9663
m           0.0006    0.0004   1.6551  0.1035  -0.0001   0.0013
s          -6.5822    0.7362  -8.9401  0.0000  -8.0571  -5.1073
k           0.0000    0.0000   1.0515  0.2975  -0.0000   0.0000
t           0.0006    0.0018   0.3090  0.7585  -0.0031   0.0043
```

```
v0         14.4861    1.3805  10.4936  0.0000  11.7207  17.2515
p0         -0.0000    0.0000  -0.7496  0.4566  -0.0000   0.0000
t0          0.0003    0.0006   0.5019  0.6177  -0.0008   0.0014
2_7_2
             Coef.  Std.Err.        t   P>|t|   [0.025   0.975]
Intercept   0.3676    0.9370  0.3923  0.6983  -1.5663   2.3014
m           0.0005    0.0006  0.7528  0.4589  -0.0008   0.0017
s          -6.8111    1.1976 -5.6874  0.0000  -9.2828  -4.3394
k           0.0000    0.0000  0.4662  0.6453  -0.0000   0.0000
t          -0.0014    0.0030 -0.4713  0.6417  -0.0076   0.0048
v0         14.9546    2.2454  6.6600  0.0000  10.3202  19.5890
p0         -0.0000    0.0000 -0.5629  0.5787  -0.0000   0.0000
t0          0.0004    0.0009  0.4529  0.6547  -0.0014   0.0023
2_7_3
             Coef.  Std.Err.        t   P>|t|   [0.025   0.975]
Intercept   0.1662    1.4378  0.1156  0.9108  -3.1494   3.4818
m           0.0003    0.0009  0.3306  0.7494  -0.0018   0.0024
s          -6.3784    1.8377 -3.4709  0.0084 -10.6161  -2.1407
k           0.0000    0.0000  0.4130  0.6905  -0.0000   0.0000
t          -0.0005    0.0046 -0.1158  0.9107  -0.0111   0.0101
v0         14.3326    3.4457  4.1596  0.0032   6.3869  22.2784
p0         -0.0000    0.0000 -0.1167  0.9100  -0.0000   0.0000
t0          0.0001    0.0014  0.1067  0.9177  -0.0030   0.0033
/usr/local/lib/python3.9/site-packages/scipy/stats/_stats_py.py:1769:
UserWarning: kurtosistest only valid for n>=20 ... continuing anyway,
n=16
  warnings.warn("kurtosistest only valid for n>=20 ... continuing "
```

The p-values and the width of the confidence intervals increase with decreasing size of the design.

Solution 6.8 [1]

Solution 6.9 Repeat the simulation from Example 6.5 with tolerances of 1% and 2%.

```
tolerances = [f'tl{c}' for c in 'ABCDEFGHIJKLM']
factors = {tl: [1, 2] for tl in tolerances}
Design = doe.frac_fact_res(factors, 4)

# Randomize and create replicates
nrepeat = 100
Design = Design.sample(frac=1).reset_index(drop=True)
Design = Design.loc[Design.index.repeat(nrepeat)].reset_index(drop=True)

# Run simulation
simulator = mistat.PowerCircuitSimulation(**{k: list(Design[k]) for k in Design})
result = simulator.simulate()
result = mistat.simulationGroup(result, nrepeat)

# Combine results with the Design matrix
Design['response'] = result['volts']
Design['group'] = result['group']

# calculate mean, standard deviation, and MSE
def groupAggregation(g):
    return {
        'mean': g['response'].mean(),
        'std': g['response'].std(),
        'MSE': g['response'].var(ddof=0),
```

[1] TODO: I'm not clear what this refers to. From reading the solution, it looks like it refers to something in the chapter and not the previous exercise.

```
    }
results = pd.DataFrame(list(Design.groupby('group').apply(groupAggregation)))
results
```

```
          mean       std       MSE
0   229.952660  1.239031  1.519846
1   230.061268  1.247408  1.540467
2   229.956442  1.175165  1.367202
3   230.002957  1.071569  1.136778
4   230.189922  1.238505  1.518555
5   229.811038  1.378896  1.882342
6   230.020092  1.276255  1.612537
7   230.069781  1.341914  1.782726
8   229.896648  1.132489  1.269705
9   230.078890  1.231510  1.501450
10  229.999059  0.801640  0.636201
11  230.039024  1.351770  1.809010
12  230.078307  1.212124  1.454551
13  230.063413  1.126659  1.256667
14  230.199067  1.209140  1.447400
15  230.169439  1.241263  1.525326
16  229.847311  1.430404  2.025596
17  230.082858  1.196681  1.417725
18  229.863564  1.135535  1.276545
19  229.940238  0.990011  0.970320
20  229.903679  1.308186  1.694237
21  230.086787  1.140915  1.288671
22  230.080309  1.081863  1.158724
23  230.037336  1.203702  1.434409
24  230.060425  0.678781  0.456137
25  229.795996  1.037095  1.064811
26  229.931210  0.895302  0.793549
27  229.949907  1.150458  1.310319
28  229.958535  1.345003  1.790943
29  229.970812  1.046762  1.084753
30  229.874653  1.285508  1.636005
31  229.972208  0.968239  0.928112
```

Comparing this table to Table 6.13 in the text, we see that by reducing the tolerances to 1% and 2%, the MSE is reduced by a factor of 25. This, however, increases the cost of the product.

Chapter 7
Computer Experiments

Import required modules and define required functions

```
import random
import numpy as np
import pandas as pd
from scipy import stats
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
from pyKriging.krige import kriging
import mistat
```

Solution 7.1 In Python:

```
nrepeat = 10000
days = 365
for size in (22, 23): # range(1, 365+1):
  same_birthday = 0
  for _ in range(nrepeat):
    birthdays = stats.randint.rvs(1, 365+1, size=size)
    if len(birthdays) != len(set(birthdays)):
        same_birthday += 1
  print(f'size of party: {size}, ',
        f'p(same birthday) {same_birthday / nrepeat:.2f}')
```

```
size of party: 22,  p(same birthday) 0.48
size of party: 23,  p(same birthday) 0.51
```

Solution 7.2 We first define a function that simulates the experiment. The marbel will drop close to the drop position with a deviation that has a bivariate normal distribution. As the scale is not defined, we set the standard deviation to 1. The target is positioned at (0, 0).

```
def funnel_drop(position):
  ''' based on funnel position, returns marble drop position'''
  x_new = position[0] + stats.norm().rvs()
  y_new = position[1] + stats.norm().rvs()
  return np.array([x_new, y_new])
```

```
def result(strategy, dropped):
  ''' returns result information '''
  return {
    'strategy': strategy,
    'x': dropped[0],
    'y': dropped[1],
    'distance': np.sqrt(dropped[0]**2 + dropped[1]**2),
  }
```

Next we simulate the funnel drop experiment for different strategies. See Fig. 7.1 for a visualization of the simulation results.

```
np.random.seed(1)
results = []
nrepeat = 100

# strategy 1
# funnel fixed
position = (0, 0)
for _ in range(nrepeat):
  dropped = funnel_drop(position)
  results.append(result(1, dropped))

# strategy 2
# position funnel to compensate for error relative to funnel position
position = np.array([0, 0])
for _ in range(nrepeat):
  dropped = funnel_drop(position)
  results.append(result(2, dropped))
  position = position - dropped

# strategy 3
# position funnel to compensate for error relative to target
position = np.array([0, 0])
for _ in range(nrepeat):
  dropped = funnel_drop(position)
  results.append(result(3, dropped))
  position = - dropped

# strategy 4
# position funnel to compensate for error relative to target
position = np.array([0, 0])
for _ in range(nrepeat):
  dropped = funnel_drop(position)
  results.append(result(4, dropped))
  position = dropped

results = pd.DataFrame(results)
g = sns.FacetGrid(results, col='strategy', col_wrap=2)
g.map(sns.scatterplot, 'x', 'y')
plt.show()
sns.boxplot(x='distance', y='strategy', data=results, orient='h')
#g = sns.FacetGrid(results, col='strategy')
#g.map(sns.boxplot, 'distance', order='strategy', orient='v')
plt.show()
```

The result of the simulation shows that strategy 1, no interference, leads to results closest to the target. Strategy 2 still creates results close to the target, but with a larger deviation. Strategies 3 and 4 lead to results far away from the target.

Solution 7.3 We first create a series of design using the different methods

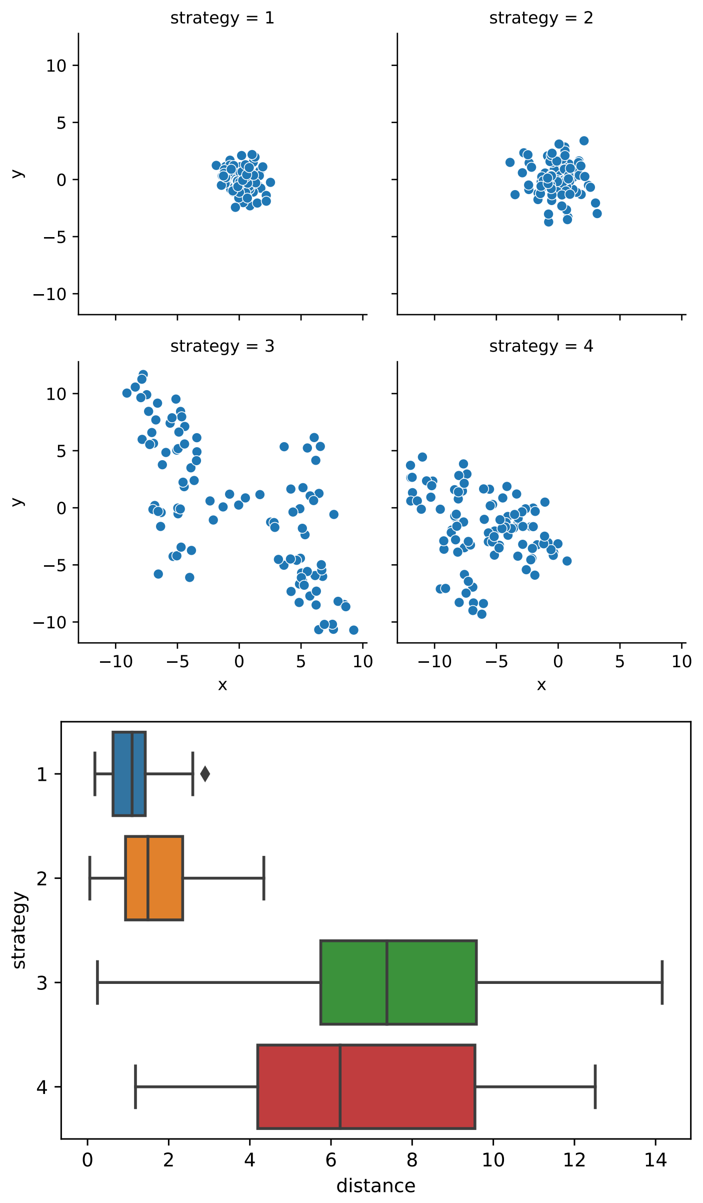


Fig. 7.1: Result of funnel drop experiment for the different strategies

```
from mistat.design import doe

np.random.seed(1) # set random seed for reproducibility
Factors = {
    'm': [30, 60],
    's': [0.005, 0.02],
    'v0': [0.002, 0.01],
    'k': [1_000, 5_000],
    'p0': [90_000, 110_000],
    't': [290, 296],
    't0': [340, 360],
}
Designs = {
    'Latin hypercube': doe.lhs(Factors, num_samples=50),
    'Latin hypercube (space-filling)': doe.space_filling_lhs(Factors, num_samples=50),
    'Random k-means cluster': doe.random_k_means(Factors, num_samples=50),
    'Maximin reconstruction': doe.maximin(Factors, num_samples=50),
    'Halton sequence based': doe.halton(Factors, num_samples=50),
    'Uniform random matrix': doe.uniform_random(Factors, num_samples=50),
}
```

Next we can visualize each design using the pandas scatterplot matrix method. We hide axis ticks for clarity.

```
for method, design in Designs.items():
    sm = pd.plotting.scatter_matrix(design, figsize=[4, 4])
    # hide all axis labels in visualization
    for subaxis in sm:
        for ax in subaxis:
            ax.xaxis.set_ticks([])
            ax.yaxis.set_ticks([])
            ax.set_ylabel("")
    plt.suptitle(method)
    plt.show()
```

The result is shown in Fig. 7.2. The methods lead to cleary distinct distributions of the design. The distributions for both latin hypercube designs, the maximin reconstruction design, and the uniform random matrix design lead to random coverage of the design space. Looking a the distribution for each factor, both latin hypercube designs show the most uniform distributions, while the maximin reconstruction and uniform random matrix methods lead to a design where factors are not uniformly explored.

The random k-means cluster and the Halton sequence based designs are clearly distinct. The random k-means cluster method leads to a design that favors the extreme values for each factor. The Halton sequence based design shows patterns in the pair-wise scatter plots.

Solution 7.4 We run a 5-fold cross-validation using the *kriging* model from the `pyKriging` package. The model training will take some time.

```
random.seed(1)
np.random.seed(1)

outcome = 'seconds'
predictors = ['m', 's', 'v0', 'k', 'p0', 't', 't0']
```

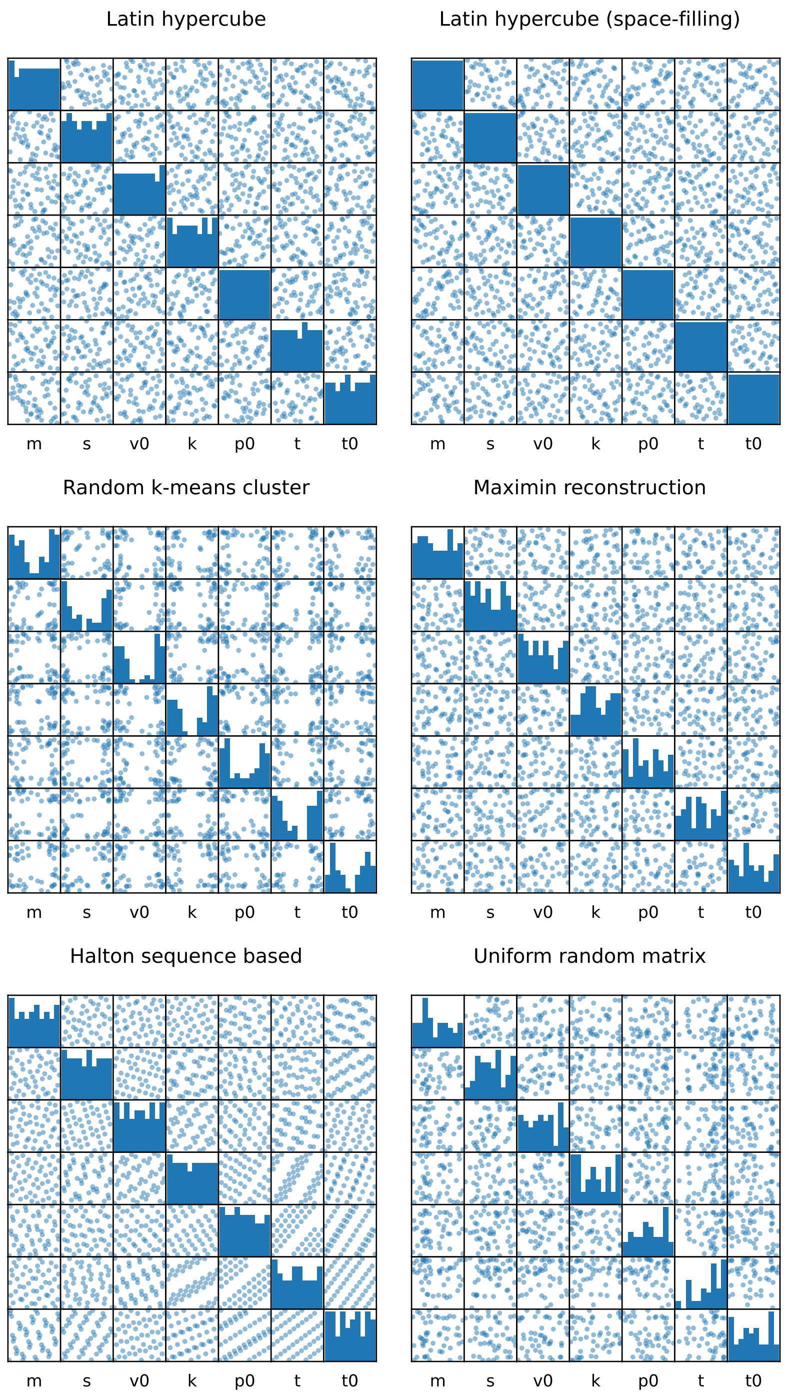


Fig. 7.2: Visualizations of different designs

```
performance = []
for method, Design in Designs.items():
    # randomize the design and run the piston simulator
    DesignRandomized = Design.sample(frac=1, random_state=1).reset_index(drop=True)
    simulator = mistat.PistonSimulator(parameter=DesignRandomized)
    result = simulator.simulate()

    # convert to X and y matrics to use with the
    X = result[predictors].values
    y = result[outcome].values

    # run 5-fold cross-validation to determine MSE values
    # we collect the out-of-fold predictions to determine
    # the MSE
    predicted = []
    actual = []
    for train_idx, test_idx in KFold(n_splits=5).split(result):
        model = kriging(X[train_idx], y[train_idx])
        model.train()

        # predict using the test set
        ypred = [model.predict(row) for row in X[test_idx]]
        # keep prediction results for performance metric calculation
        actual.extend(y[test_idx])
        predicted.extend(ypred)

    mse = mean_squared_error(actual, predicted)
    performance.append({
        'design': method,
        'mse': mse,
        'rmse': np.sqrt(mse),
    })
    print(performance)
    # skip the remaining designs
    break
```

The performance results are:

design	mse	rmse
Latin hypercube	0.00048	0.02183
Latin hypercube (space-filling)	0.00033	0.01820
Random k-means cluster	0.00015	0.01232
Maximin reconstruction	0.00029	0.01700
Halton sequence based	0.00022	0.01478
Uniform random matrix	0.00019	0.01360

In all cases, we observe good predictive performance. Due to randomness in the calculations, the relative order can be different between runs.

Solution 7.5 We first generate the uniform random design

```
np.random.seed(1)

num_samples = 1000
Design = doe.uniform_random(Factors, num_samples=num_samples)
```

Run the piston simulator using replicates of each parameter set. As the next exercise requires the same process, we wrap the required steps into a custom function

```
def evaluateDesign(Design, nrepeat=20):
  #Design = Design.loc[np.repeat(Design.index.values, nrepeat), :]
  #settings = {c: list(Design[c]) for c in Design.columns}
  #simulator = mistat.PistonSimulator(seed=1, **settings)
  simulator = mistat.PistonSimulator(parameter=Design, seed=1, n_replicate=nrepeat)
  result = simulator.simulate()
  result = mistat.simulationGroup(result, nrepeat)

  # next we aggregate the replicates and determine mean and
  # standard deviation of each group.
  options = {p: 'mean' for p in ['m', 's', 'v0', 'k', 'p0', 't', 't0']}
  options['seconds'] = ['mean', 'std']
  result = result.groupby('group').aggregate(options)
  # convert multi-index to single index
  result.columns = [' '.join(col) if col[0] == 'seconds' else col[0]
                    for col in result.columns]
  return result

result = evaluateDesign(Design, nrepeat=30)
```

In order to find the parameter set that gives a cycle time around 0.02 with minimal variability, we filter the results by the mean values and sort by the standard deviation.

```
# determine rows with seconds around 0.02 and
# identify row with smallest standard deviation
rows = [0.0195 < v < 0.0205 for v in result['seconds mean'].values]
target = result.loc[rows, :]
target = target.sort_values('seconds std')
target.head(3)
```

```
               m         s        v0            k             p0
t  \
group
871    59.694778  0.018177  0.004087  1152.373331  109480.515301
293.342719
331    46.278492  0.018261  0.004200  1432.269941   93825.441035
292.322277
947    37.548930  0.012126  0.003148  1280.072660  105859.307614
291.076845

               t0  seconds mean  seconds std
group
871    350.794200      0.020356     0.002469
331    357.081196      0.020009     0.002751
947    342.524145      0.019582     0.003092
```

We can see that all three possible settings lead to a cycle time close to 0.02 seconds with low variability. The visualization of the results in Fig. 7.3 shows that the settings with a cycle time around 0.02 can be achieved with different values of the piston surface area *s* and the initial gas volume *v*0. The two variables show correlation. Only a small number of the design parameters match our criteria. It would be useful to repeat the similuation with a larger number of samples.

```
ax = target.plot.scatter(x='v0', y='s')
target.head(3).plot.scatter(x='v0', y='s', ax=ax, color='red')
ax = target.plot.scatter(x='seconds mean', y='seconds std')
target.head(3).plot.scatter(x='seconds mean', y='seconds std', ax=ax, color='red')
plt.show()
```

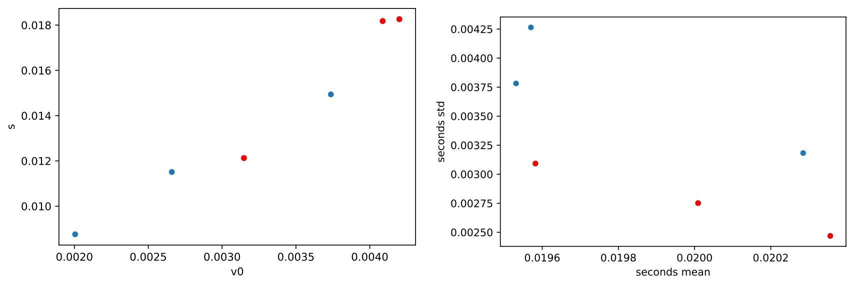


Fig. 7.3: Visualization of results from stochastic emulator based on random uniform design.

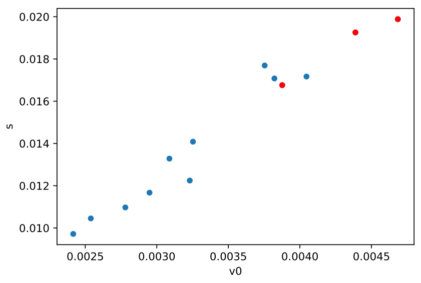
Solution 7.6 We can make use of the function defined in the previous exercise to evaluate a latin hyper cube design

```
np.random.seed(1)

num_samples = 1000
Design = doe.lhs(Factors, num_samples=num_samples)
result = evaluateDesign(Design, nrepeat=30)

# determine rows with seconds around 0.02 and
# identify row with smallest standard deviation
rows = [0.0195 < v < 0.0205 for v in result['seconds mean'].values]
target = result.loc[rows, :]
target = target.sort_values('seconds std')
target.head(3)
```

```
                m          s         v0            k            p0
t  \
group
368    33.166208  0.016760  0.003876  2998.808445  96800.530567
291.237264
940    39.401808  0.019256  0.004388  3024.658005  97630.295509
295.393884
150    33.434638  0.019884  0.004685  1401.296554  92707.973181
292.463943

               t0  seconds mean  seconds std
group
368    341.684282      0.020127     0.001956
940    351.093709      0.020001     0.002313
150    353.536655      0.020030     0.002467
```

The results derived using this design are similar to what was obtained in Exercise 7.5. We can see more points matching our criteria. This could be due to a better coverage of the design space. However, as already mentioned in Exercise 7.5, we could improve the analysis by increasing the number of design points.

```
ax = target.plot.scatter(x='v0', y='s')
target.head(3).plot.scatter(x='v0', y='s', ax=ax, color='red')
ax = target.plot.scatter(x='seconds mean', y='seconds std')
target.head(3).plot.scatter(x='seconds mean', y='seconds std', ax=ax, color='red')
plt.show()
```



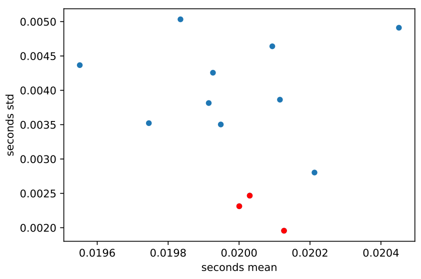


Fig. 7.4: Visualization of results from stochastic emulator based on a latin hypersquare design.

Chapter 8
Cybermanufacturing and digital twins

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.formula.api as smf
import lifelines
import pingouin as pg
import seaborn as sns
import matplotlib.pyplot as plt
import mistat
```

Solution 8.1 [1] [2]

Solution 8.2 [3] [4]

Solution 8.3 [5] [6]

Solution 8.4 [7] [8]

[1] TODO
[2] TODO: add files to mistat
[3] TODO
[4] TODO: add files to mistat
[5] TODO
[6] TODO: add files to mistat
[7] TODO
[8] TODO: add files to mistat

Chapter 9
Reliability Analysis

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.formula.api as smf
import lifelines
import pingouin as pg
import seaborn as sns
import matplotlib.pyplot as plt
import mistat
```

Solution 9.1 $MTTF = 5.1$ [hr]; $MTTR = 6.6$ [min]; Intrinsic Availability = 0.979.

Solution 9.2 [1]

(i) Since 4 yrs = 35,040 hr, the reliability estimate is $R(4) = 0.30$.
(ii) Of the solar cells that survive 20,000 hours, the proportion expected to survive 40,000 hours is $\frac{0.25}{0.60} = 0.42$.

Solution 9.3 [2]

(i) The hazard function is

$$h(t) = \begin{cases} \frac{4t^3}{20736\cdot\left(1-\frac{t^4}{20736}\right)}, & 0 \leq t < 12 \\ \infty, & 12 \leq t. \end{cases}$$

(ii) $MTTF = \int_0^{12}\left(1 - \frac{t^4}{20736}\right)\,\mathrm{d}t = 9.6$ [Months].
(iii) $R(4) = 1 - F(4) = 0.9877$.

Solution 9.4 [3] **(i)** $h(t) = 2 + 6t,\ \ h(3) = 20.$ **(ii)** $R(5)/R(3) = 0.$

[1] TODO
[2] TODO
[3] TODO

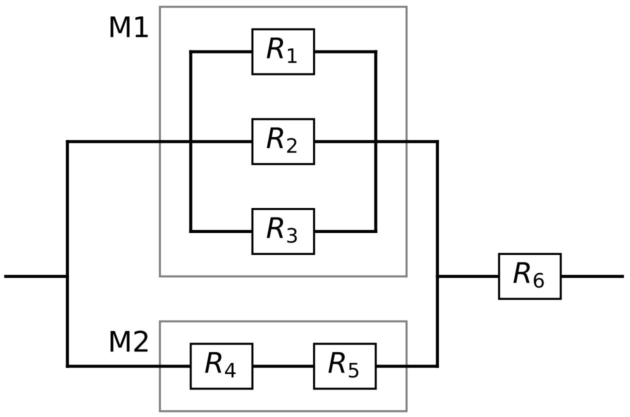


Fig. 9.1: System schema

Solution 9.5 [4] **(i)** $1-B(1;4,0.95) = 0.9995$ **(ii)** $(1-B(0;2,0.95))^2 = (1-0.05^2)^2 = 0.9950$.

Solution 9.6 **(a)** The block diagram of the system is shown in Fig. 9.1.
(b) $R_{M_1} = 1-(1-R_1)(1-R_2)(1-R_3)$ and $R_{M_2} = R_4R_5$. Thus,

$$\begin{aligned} R_{sys} &= [1-(1-R_1)(1-R_2)(1-R_3)(1-R_4R_5)]R_6 \\ &= R_1R_6 + R_2R_6 + R_3R_6 - R_1R_2R_6 + R_4R_5R_6 \\ &\quad - R_1R_3R_6 - R_2R_3R_6 + R_1R_2R_3R_6 \\ &\quad - R_1R_4R_5R_6 - R_2R_4R_5R_6 - R_3R_4R_5R_6 \\ &\quad + R_2R_3R_4R_5R_6 + R_1R_3R_4R_5R_6 + R_1R_2R_4R_5R_6 \\ &\quad - R_1R_2R_3R_4R_5R_6. \end{aligned}$$

If all values of $R = 0.8$ then $R_{sys} = 0.7977$.

Solution 9.7 Extending Bonferroni's inequality,

$$\begin{aligned} \Pr\{C_1 = 1, \ldots, C_n = 1\} &= 1 - \Pr\left\{\bigcup_{i=1}^{n}\{C_i = 0\}\right\} \\ &\geq 1 - \sum_{i=1}^{n} \Pr\{C_i = 0\} \\ &= 1 - \sum_{i=1}^{n}(1-R_i); \end{aligned}$$

[4] TODO

where $\{C_i = 1\}$ is the event that the i-th component functions and $\{C_i = 0\}$ is the event that it fails. Since $R_{sys} = \Pr\{C_1 = 1, \ldots, C_n = 1\}$ for a series structure, the inequality follows.

Solution 9.8 The reliability of a component is $\theta(5) = \Pr\left\{W\left(\frac{1}{2}, 100\right) > 5\right\} = 0.79963$.

Therefore, $R_{sys} = 1 - B(3; 8, 0.7996) = 0.9895$.

Solution 9.9 The life length of the system is the sum of 3 independent $E(100)$, that is $G(3, 100)$ (gamma distribution). Hence the $MTTF = 300$ hr. The reliability function of the system is

$$R_{sys}(t) = 1 - G(t; 3, 100).$$

Solution 9.10 Let C be the length of the renewal cycle.

$$E\{C\} = \beta\Gamma\left(1 + \frac{1}{\alpha}\right) + e^{\mu+\sigma^2/2}.$$

$$\sigma(C) = \left[\beta^2\left(\Gamma\left(1 + \frac{2}{\alpha}\right) - \Gamma^2\left(1 + \frac{1}{\alpha}\right)\right) + e^{2\mu+\sigma^2}(e^{\sigma^2} - 1)\right]^{1/2}.$$

Solution 9.11 Let $C_1, C_2, \ldots$ denote the renewal cycle times, $C_i \sim N(100, 10)$. $N_R(200)$ is a random variable representing the number of repairs that occur during the time interval $(0, 200]$.

$$\Pr\{N_R(200) = 0\} = \Pr\{C_1 > 200\} = 1 - \Phi\left(\frac{200 - 100}{10}\right) = 1 - \Phi(10) = 0.$$

For $k = 1, 2, 3, \ldots$, $C_1 + \cdots + C_k \sim N(100k, 10\sqrt{k})$ and so

$$\begin{aligned}
\Pr\{N_R(200) = k\} &= \Pr\{N_R(200) \geq k\} - \Pr(N_R(200) \geq k + 1\} \\
&= \Pr\{C_1 + \ldots + C_k \leq 200\} - \Pr\{C_1 + \ldots + C_{k+1} \leq 200\} \\
&= \Phi\left(\frac{200 - 100k}{10\sqrt{k}}\right) - \Phi\left(\frac{200 - 100(k + 1)}{10\sqrt{k + 1}}\right) \\
&= \Phi\left(\frac{20}{\sqrt{k}} - 10\sqrt{k}\right) - \Phi\left(\frac{20}{\sqrt{k + 1}} - 10\sqrt{k + 1}\right),
\end{aligned}$$

where $\Phi(\cdot)$ denotes the c.d.f. of a $N(0, 1)$ distribution. Thus the p.d.f. of $N_R(200)$ is given by the following table

k	$\Pr\{N_R(200) = k\}$
0	0
1	0.5
2	0.5
3	0
4	0
$\vdots$	$\vdots$

Solution 9.12 $V(1000) = \sum_{n=1}^{\infty} \Phi\left(\frac{100-10n}{\sqrt{n}}\right) \approx 9.501$.

Solution 9.13 $v(t) = \frac{1}{10\sqrt{n}} \sum_{n=1}^{\infty} \phi\left(\frac{t-100n}{10\sqrt{n}}\right)$, where $\phi(Z)$ is the p.d.f. of $N(0,1)$.

Solution 9.14 Since $TTF \sim \max(E_1(\beta), E_2(\beta))$, $F(t) = (1 - e^{-t/\beta})^2$ and $R(t) = 1 - (1 - e^{-t/\beta})^2$.

Using Laplace transforms we have

$$R^*(s) = \frac{\beta(3 + s\beta)}{(1 + s\beta)(2 + s\beta)}, \quad f^*(s) = \frac{2}{(1 + \beta s)(2 + \beta s)}, \quad g^*(s) = \frac{1}{(1 + s\gamma)^2} \quad \text{and}$$

$$A^*(s) = \frac{\beta(3 + s\beta)(1 + \gamma s)^2}{s[3\beta + 4\gamma + (\beta^2 + 6\beta\gamma + 2\gamma^2)s + (2\beta^2\gamma + 3\beta\gamma^2)s^2 + \beta^2\gamma^2 s^3]}.$$

Solution 9.15 For our samples, with $M = 500$ runs, we get the following estimates:

$$E\{N_R(2000)\} = V(2000) = 16.574 \quad \text{and} \quad \mathrm{Var}\{N_R(2000)\} = 7.59.$$

Solution 9.16 The expected length of the test is 2069.8 [hr].

Solution 9.17 Let $T_{n,0} \equiv 0$. Then $T_{n,r} = \sum_{j=1}^{r}(T_{n,j} - T_{n,j-1}) = \sum_{j=1}^{r} \Delta_{n,j-1}$. Thus,

$$V\{T_{n,r}\} = \beta^2 \sum_{j=1}^{r} \left(\frac{1}{n - j + 1}\right)^2.$$

In Exercise 9.16, $\beta = 2,000$ hr, $n = 15$ and $r = 10$. This yields

$$V\{T_{15,10}\} = 4 \times 10^6 \times 0.116829 = 467,316 \text{ (hr)}^2.$$

Solution 9.18 Let $S_{n,r} = \sum_{i=1}^{r} T_{n,i} + (n - r)T_{n,r}$. An unbiased estimator of β is

$$\hat{\beta} = \frac{S_{n,r}}{r}, \qquad V\{\hat{\beta}\} = \frac{\beta^2}{r}.$$

Solution 9.19 The Weibull QQ plotting is based on the regression of $\ln T_{(i)}$ on

$$W_i = \ln\left(-\ln\left(1 - \frac{i}{n+1}\right)\right).$$

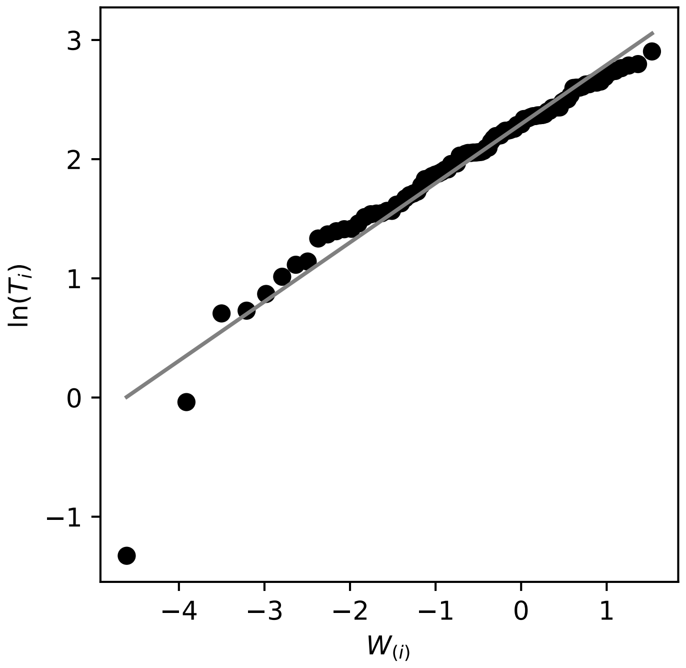


Fig. 9.2: Weibull probability plot

The linear relationship is $\ln T_{(i)} = \frac{1}{\nu} W_i + \ln\beta$. The slope of the straight line estimates $\frac{1}{\nu}$ and the intercept estimates $\log\beta$.

```
np.random.seed(1)
n = 100
lnTi = np.log(sorted(stats.weibull_min(2.5, scale=10).rvs(n)))
Wi = [np.log(-np.log(1 - i / (n+1))) for i in range(1, n + 1)]
df = pd.DataFrame({'Wi': Wi, 'lnTi': lnTi})
model = smf.ols('lnTi ~ Wi + 1', data=df).fit()
print(model.params)
intercept, slope = model.params
fig, ax = plt.subplots(figsize=(4, 4))
ax.plot((Wi[0], Wi[-1]),
        (slope * Wi[0] + intercept, slope * Wi[-1] + intercept),
        color='grey')
ax.scatter(Wi, lnTi, color='black')
ax.set_xlabel(r'$W_{(i)}$')
ax.set_ylabel(r'$\ln(T_i)$')
plt.show()
```

```
Intercept    2.294805
Wi           0.496692
dtype: float64
```

For our sample of $n = 100$ from $W(2.5, 10)$ the regression of $\ln T_{(i)}$ on W_i is

$$\ln(\text{TTF}) = 2.295 + 0.497W$$

```
Predictor     Coef    Stdev    t-ratio       p
Constant    2.2948    0.018    130.703   0.000
W           0.4967    0.013     37.629   0.000
  R-sq = 0.935   R-sq(adj) = 0.935.
```

```
beta = np.exp(intercept)
nu = 1 / slope
print(f'beta {beta:.3f}, nu {nu:.3f}')
```

```
beta 9.922, nu 2.013
```

The graphical estimate of β is $\hat{\beta} = \exp(2.295) = 9.922$. The graphical estimate of ν is $\hat{\nu} = \frac{1}{0.4643} = 2.013$. Both estimates are close to the nominal values. In Fig. 9.2 we see the Weibull probability plot. This graph puts the sorted observations $W_{(i)}$ on the x-axis and $\ln T_i$ on the y-axis where T_i is the calculated probability of occurrence for each observation $W_{(i)}$ assuming a Weibull distribution. As shown above, the $\hat{\beta}$ and $\hat{\nu}$ can be calculated from the slope and intercept of the liner regression line.

Solution 9.20 The regression of $Y_i = \ln X_{(i)}, i = 1, \ldots, n$ on the normal scores $Z_i = \Phi^{-1}\left(\frac{i-3/8}{n+1/4}\right)$ is

```
Xi = [94.9, 106.9, 229.7, 275.7, 144.5, 112.8, 159.3, 153.1,
        270.6, 322.0, 216.4, 544.6, 266.2, 263.6, 138.5, 79.0,
        114.6, 66.1, 131.2, 91.1]
n = len(Xi)
Zi = [stats.norm().ppf((i - 3/8)/(n + 1/4)) for i in range(1, n+1)]

df = pd.DataFrame({'Zi': Zi, 'lnXi': np.log(sorted(Xi))})
model = smf.ols('lnXi ~ Zi + 1', data=df).fit()

intercept, slope = model.params
fig, ax = plt.subplots(figsize=(4, 4))
ax.plot((Zi[0], Zi[-1]),
        (slope * Zi[0] + intercept, slope * Zi[-1] + intercept),
        color='grey')
ax.scatter(Zi, df['lnXi'], color='black')
ax.set_xlabel(r'$Z_{(i)}$')
ax.set_ylabel(r'$\ln(X_(i))$')
plt.show()
```

$$\ln(X) = 5.10 + 0.556Z.$$

```
Predictor      Coef      Stdev    t-ratio       p
Constant    5.09571    0.02201     232.23   0.000
Z           0.55589    0.02344      23.79   0.000
  R-sq = 0.969   R-sq(adj) = 0.967.
```

```
mu = intercept
sigma = slope
print(f'mu {mu:.3f}, sigma {sigma:.3f}')
```

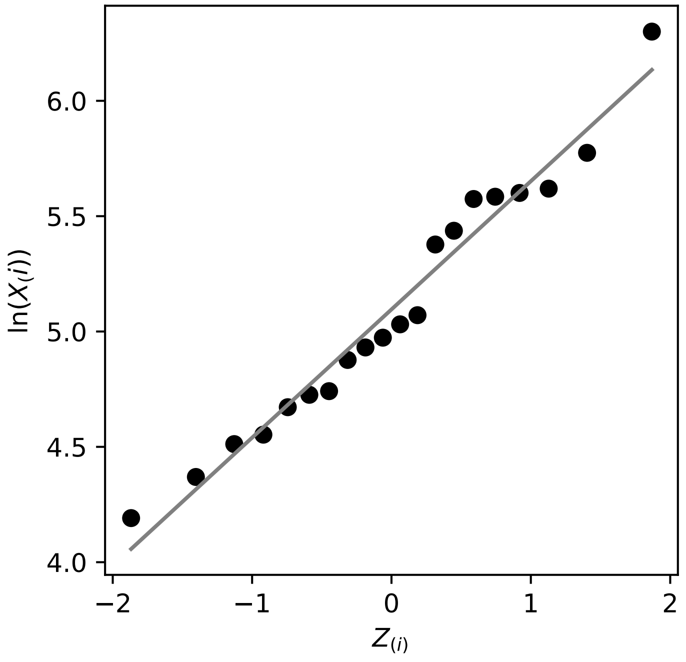


Fig. 9.3: Normal Probability Plot of ln X

```
mu 5.096, sigma 0.556
```

The intercept $\hat{a} = 5.09571$ is an estimate of μ, and the slope $\hat{b} = 0.5559$ is an estimate of σ. The mean of LN(ξ, σ^2) is $\xi = e^{\mu+\sigma^2/2}$ and its variance is $D^2 = e^{2\mu+\sigma^2}(e^{\sigma^2}-1)$. Thus, the graphical estimates are $\hat{\xi} = 190.608$ and $\hat{D} = 13154.858$. The mean $\bar{X}$ and variance S^2 of the given sample are 189.04 and 12948.2. In Fig. 9.3 we see the normal probability plotting of the data.

Solution 9.21 The Q-Q plot is given in Fig. 9.4.

```
np.random.seed(1)
data = [13, 157, 172, 176, 249, 303, 350, 400, 400]
n = len(data)
lnTi = np.log(data)
Wi = [np.log(-np.log(1 - i / (n+1))) for i in range(1, n + 1)]
# exclude the censored data for the regression analysis
df = pd.DataFrame({'Wi': Wi[:7], 'lnTi': lnTi[:7]})
model = smf.ols('lnTi ~ Wi + 1', data=df).fit()
print(model.params)
intercept, slope = model.params
fig, ax = plt.subplots(figsize=(4, 4))
ax.plot((Wi[0], Wi[-1]),
        (slope * Wi[0] + intercept, slope * Wi[-1] + intercept),
        color='grey')
ax.scatter(Wi[:-2], lnTi[:-2], color='black')
```

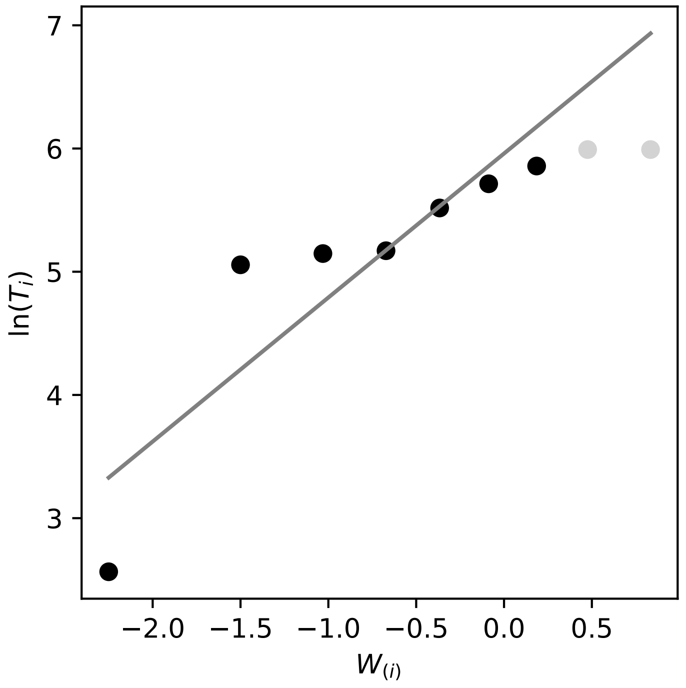


Fig. 9.4: Weibull probability plot

```
ax.scatter(Wi[-2:], lnTi[-2:], color='lightgrey')
ax.set_xlabel(r'$W_{(i)}$')
ax.set_ylabel(r'$\ln(T_i)$')
plt.show()
```

```
Intercept    5.959374
Wi           1.168853
dtype: float64
```

The regression of $\ln T_{(i)}$ on $W_i = \ln\left(-\ln\left(1 - \frac{i}{10}\right)\right)$ for $i = 1, \dots, 7$ is

$$\ln(T) = 5.96 + 1.17W \tag{9.0.1}$$

```
Predictor     Coef     Stdev    t-ratio       p
Constant    5.9594    0.3068      19.43   0.000
W           1.1689    0.2705       4.32   0.008
  s = 0.5627   R-sq = 0.789   R-sq(adj) = 0.747.
```

According to this regression line, the estimates of β and ν are $\hat{\beta} = \exp(5.9594) = 387.378$ and $\hat{\nu} = 1/1.1689 = 0.855$. The estimate of the median of the distribution is

$$\hat{M}e = \hat{\beta}\left(-\ln\left(\frac{1}{2}\right)\right)^{1/\hat{\nu}} = 252.33.$$

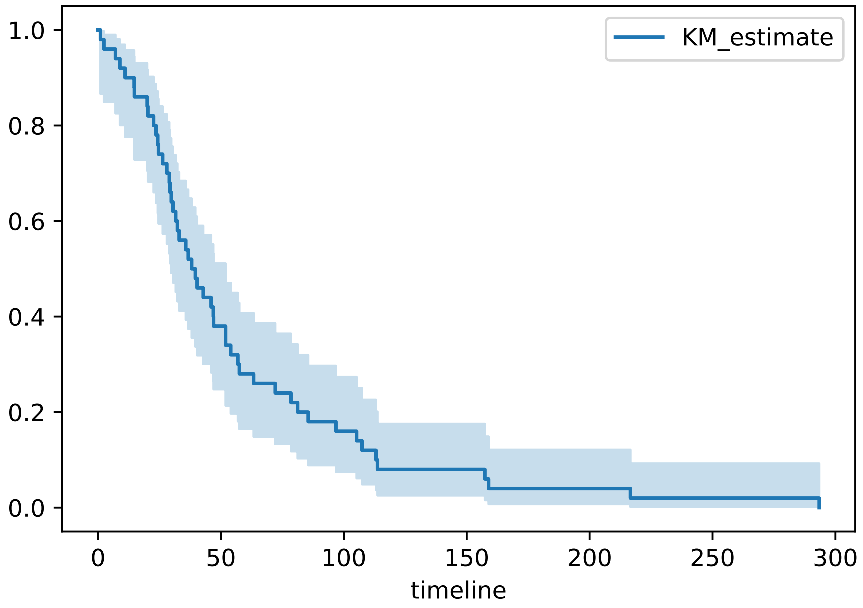


Fig. 9.5: Kaplan-Meier Estimator of The Reliability of An Electric Device

Solution 9.22 Use the `lifelines` package to fit a Kaplan-Meier estimator. The reliability function is shown in Fig. 9.5.

```
elecfail = mistat.load_data('ELECFAIL.csv')

kmf = lifelines.KaplanMeierFitter()
kmf.fit(elecfail)
kmf.plot_survival_function()
plt.show()
```

Solution 9.23 Use the `lifelines` package to fit an exponential model.

```
elecfail = mistat.load_data('ELECFAIL.csv')

kmf = lifelines.ExponentialFitter()
kmf.fit(elecfail)
```

```
<lifelines.ExponentialFitter:"Exponential_estimate", fitted with 50
total observations, 0 right-censored observations>
```

```
kmf.print_summary()
```

	coef	se(coef)	coef lower 95%	coef upper 95%	cmp to	z	p	-log2(p)
lambda_	57.07	8.07	41.25	72.89	0.00	7.07	0.00	39.24

The MLE of β is $\hat{\beta} = \bar{X} = 57.07$. Since $R(t) = \exp(-t/\beta)$, the MLE of R(50) is $\hat{R}(50) = 0.4164$. Note that $\bar{X} \sim \frac{\beta}{2n}\chi^2[2n]$. Hence, confidence limits for β, with $1-\alpha = 0.95$ are (44.048, 76.891). Substituting these limits into $R(t)$ gives $(0.3214, 0.5219)$ as a 0.95 confidence interval for $R(50)$.[5]

Solution 9.24 Use the `lifelines` package to fit an exponential model.

```
T = [96.88, 154.24, 67.44, 191.72, 173.36, 200, 140.81, 200,
     154.71, 120.73, 24.29, 10.95, 2.36, 186.93, 57.61, 99.13,
     32.74, 200, 39.77, 39.52]
E = [ti < 200 for ti in T]

kmf = lifelines.ExponentialFitter()
kmf.fit(T, E)
```

```
<lifelines.ExponentialFitter:"Exponential_estimate", fitted with 20
total observations, 3 right-censored observations>
```

```
kmf.print_summary()
```

	coef	se(coef)	coef lower 95%	coef upper 95%	cmp to	z	p	-log2(p)
lambda_	129.01	31.29	67.68	190.34	0.00	4.12	0.00	14.71

We have $n = 20$, $K_n = 17$ and $\hat{\beta} = 129.011$. With $M = 500$ runs, the estimated STD of $\hat{\beta}$ is 34.912, with confidence interval $(78.268, 211.312)$. [6]

Solution 9.25 For $\beta = 100$, $S.E.\{\hat{\beta}_n\} = 0.1\beta = 10 = \frac{100}{\sqrt{r}}$. Hence $r = 100$ and

$$n^0 \approx \frac{100}{2}\left(1 + \left(1 + 4\frac{100}{1000}\right)^{1/2}\right) = 109.16 \approx 110.$$

$E\{T_{n,r}\} = 235.327$ and the expected cost of testing is $E\{C\} = 10\times n + 1\times E\{T_{n,r}\} =$ \$1335.33.

Solution 9.26 (a)

```
T = mistat.load_data('WEIBUL.csv')

kmf = lifelines.WeibullFitter()
kmf.fit(T)
```

```
<lifelines.WeibullFitter:"Weibull_estimate", fitted with 50 total
observations, 0 right-censored observations>
```

```
kmf.print_summary()
```

[5] TODO: the Python code prints CI based on normal distribution

[6] TODO: where in the book is this calculation described?

	coef	se(coef)	coef lower 95%	coef upper 95%	cmp to	z	p	-log2(p)
lambda_	27.08	2.94	21.31	32.85	1.00	8.86	0.00	60.17
rho_	1.37	0.15	1.09	1.66	1.00	2.57	0.01	6.63

For the **WEIBUL.csv** dataset we obtained $\hat{\beta} = 27.0789$ and $\hat{\nu} = 1.374$.
(b)[7]

```
n_boot=5
idx = list(range(len(T)))
def stat_func(x):
    epf = lifelines.WeibullFitter().fit(T[x])
    return epf.params_['lambda_']

ci, dist = pg.compute_bootci(idx, func=stat_func, n_boot=n_boot, confidence=0.95,
                             method='per', seed=1, return_dist=True)
print(f'std(beta): {np.std(dist)}')

def stat_func(x):
    epf = lifelines.WeibullFitter().fit(T[x])
    return epf.params_['rho_']
ci, dist = pg.compute_bootci(idx, func=stat_func, n_boot=n_boot, confidence=0.95,
                             method='per', seed=1, return_dist=True)
print(f'nu(std): {np.std(dist)}')
```

```
std(beta): 2.8735759059232406
nu(std): 0.07224217322172025
```

The EBD estimates are $SE\{\hat{\beta}_{50}\} = 2.971$[8] and $SE\{\hat{\nu}_{50}\} = 0.1822$. The asymptotic estimates are $SE\{\hat{\beta}_{50}\} = 2.94$[9] and $SE\{\hat{\nu}_{50}\} = 0.15$. The EBD estimates and the asymptotic estimates of the standard deviations of $\hat{\beta}$ and $\hat{\nu}$ are very similar.

Solution 9.27 To discriminate between $R_0 = 0.99$ and $R_1 = 0.90$ with $\alpha = \beta = 0.01$, $n \approx 107$.

Solution 9.28 [10] For $R_0 = 0.99$, $R_1 = 0.90$, and $\alpha = \gamma = 0.01$ we get $s = 0.9603$, $h_1 = 1.9163$, $h_2 = 1.9163$ and ASN(0.9) = 31.17.

Solution 9.29 Without loss of generality, assume that $t = 1$. Thus, $R_0 = e^{-1/\beta_0} = 0.99$, or $\beta_0 = 99.5$. Also, $R_1 = 0.9$, or $\beta_1 = 9.49$. The parameters of the SPRT are: $h_1 = 48.205$, $h_2 = 48.205$ and $s = 24.652$. $ASN(0.9) \approx 3$.

Solution 9.30 Using $\alpha = \gamma = 0.05$ we have $n = 20$, $H_0 : \beta = 2,000$, $H_1 : \beta = 1,500$. Thus, $\lambda_0 = 0.0005$ and $\lambda_1 = 0.00067$. The parameters of the SPRT are:

$$h_1 = \frac{\log(\frac{95}{5})}{\log(\frac{\lambda_1}{\lambda_0})} = 10.061, \qquad h_2 = 10.061$$

[7] TODO: set n_boot to 500 once chapter is done!!!
[8] TODO: should this be $SE\{\hat{\beta}_{500}\}$
[9] TODO: should this be $SE\{\hat{\beta}\}$
[10] TODO

and

$$\begin{aligned} s &= \frac{\lambda_1 - \lambda_0}{\log(\frac{\lambda_1}{\lambda_0})} = 0.000581. \\ E_{\beta_0}\{\tau\} &= \frac{2000}{20} E_\beta\{X_{20}(\tau)\} \\ &\approx 100 \left(\frac{10.061 - 0.95 \times 20.122}{1 - 2000 \times 0.000581} \right) \\ &= 5589.44 \text{ [hr]}. \end{aligned}$$

Solution 9.31 With $\beta = 1,000$, $p = 0.01$, $\gamma = 500$ and $t^* = 300$, we get

$$F^*(t) = 1 - \frac{0.01 e^{-t/500} + 0.99 e^{-t/1,000}}{0.01 e^{-300/500} + 0.99 e^{-300/1,000}}, \quad t \geq 300.$$

[11] The expected life of units surviving the burn-in is thus,

```
factor = 0.01*np.exp(-300/500) + 0.99*np.exp(-300/1000)
integral = 0.01*500*np.exp(-300/500) + 0.99*1000*np.exp(-300/1000)
with_burn_in = 300 + integral/factor
no_burn_in = 0.01 * 500 + 0.99 * 1000
with_burn_in - no_burn_in
```

```
301.2862836203999
```

$$\begin{aligned} \beta^* &= 300 + \frac{1}{0.7389} \int_{300}^{\infty} \left(0.01 \exp\left\{-\frac{t}{500}\right\} + 0.99 \exp\left\{-\frac{t}{1,000}\right\} \right) \mathrm{d}t \\ &= 300 + \frac{5}{0.7389} \exp\left\{-\frac{300}{500}\right\} + \frac{990}{0.7389} \exp\left\{-\frac{300}{1,000}\right\} \\ &= 1,296.29 \text{ [hr]}. \end{aligned}$$

Without burn in, the expected life of these units is $0.01 * 500 + 0.99 * 1000 = 995$ [hr]. The expected life of the units in the field increases by 300 [hr].

[11] TODO: the calculation of expected life of units is missing in the current solution. Please check what I added

Chapter 10
Bayesian Reliability Estimation and Prediction

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.formula.api as smf
import lifelines
import pingouin as pg
import seaborn as sns
import matplotlib.pyplot as plt
import mistat
```

Solution 10.1 [1] **(i)** Following Example 10.3, the posterior distribution of λ is

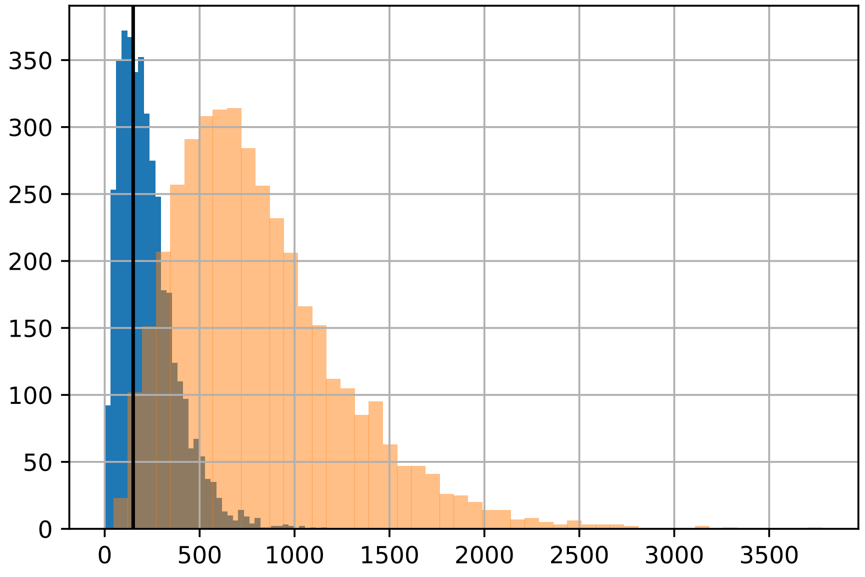
$$G(\nu + 1, \frac{\tau}{1 + x\tau}) = G(3.25, \frac{0.01}{1 + 150 \times 0.01}) = G(3.25, 0.04)$$

The following figure shows the prior distribution of λ in blue and the posterior distribution in orange. Is it reasonable that we see such a large shift or is something wrong with the code?

```
ttf = pd.Series(stats.gamma(2.25, scale=1/0.01).rvs(4000))
ax = ttf.hist(bins=40)
ax.axvline(150, color='black')

post_ttf = pd.Series(stats.gamma(3.25, scale=1/0.004).rvs(4000))
post_ttf.hist(bins=50, alpha=0.5)
```

```
<AxesSubplot: >
```

[1] TODO



(ii) [2] **(iii)** [3]

Solution 10.2 [4]

Solution 10.3 **(i)** The uniform prior distribution on $(0, 1)$ can be expressed as the beta distribution $B(1, 1)$. With the $K_{10} = 3$ defectives, the posterior distribution is $B(1 + 3, 1 + 10 - 3) = B(4, 8)$.

(ii) The Bayes estimator of θ can be calculated using equation (10.2.2).

$$\hat{\theta} = \frac{\nu_1 F_{0.5}[2\nu_1, 2\nu_2]}{\nu_2 + \nu_1 F_{0.5}[2\nu_1, 2\nu_2]} = \frac{4F_{0.5}[8, 16]}{8 + 4F_{0.5}[8, 16]} \approx \frac{4 \times 0.94422}{8 + 4 \times 0.94422} \approx 0.3207$$

Visualization of results:

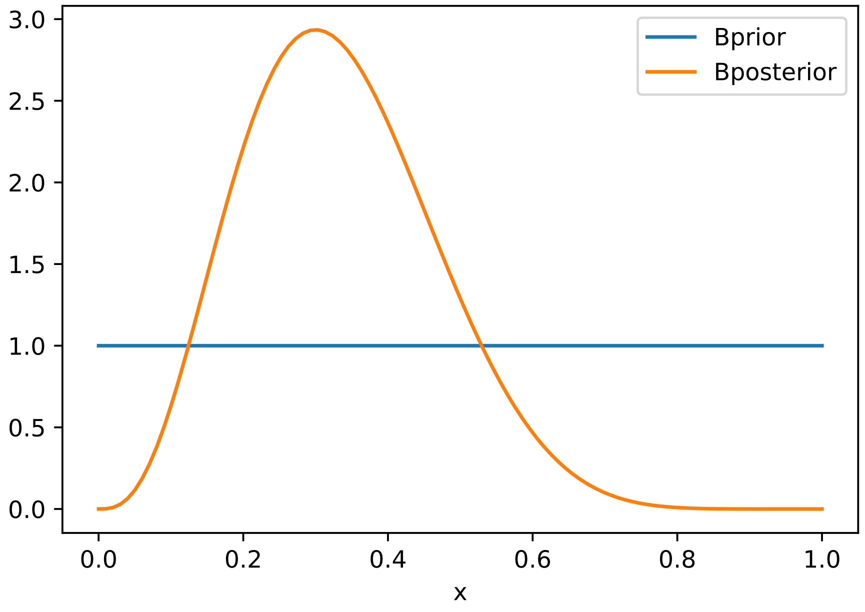
```
x = np.linspace(0, 1, 100)
df = pd.DataFrame({'x': x, 'Bprior': stats.beta(1,1).pdf(x)})
df['Bposterior'] = stats.beta(1+3,1 + 10 - 3).pdf(x)

ax = df.plot(x='x', y='Bprior')
df.plot(x='x', y='Bposterior', ax=ax)
```

```
<AxesSubplot: xlabel='x'>
```

[2] TODO: equation(10.2.5)?

[3] TODO: ?

[4] TODO: ?



Solution 10.4 [5]

Solution 10.5 Using formula (10.2.3), we get:

```
n = 50; Kn = 49
medFdist = stats.f.ppf(0.5, 2*Kn+2, 2*n+2-2*Kn)
R_t = (Kn+1) * medFdist / (n+1 -Kn+(Kn+1)*medFdist)
R_t
```

```
0.9673091500837799
```

Solution 10.6 [6] Using equation (**??**) we get for R_{sys}:

```
v1 = 2.5; v2 = 2.75; v3 = 3; n=r=10
tau = 1/1000
T1 = 4565; T2 = 5720; T3 = 7505
def R(t, Ti, tau, r, v):
    return ((1 + Ti*tau)/(1+(Ti + t)*tau))**(r+v)
t = np.linspace(0, 3000, 100)
df = pd.DataFrame({
    't': t,
    'R1': R(t, T1, tau, r, v1),
    'R2': R(t, T2, tau, r, v2),
    'R3': R(t, T3, tau, r, v3),
})
df['Rsys'] = df['R3'] + df['R1']*df['R2'] - df['R1']*df['R2']*df['R3']
```

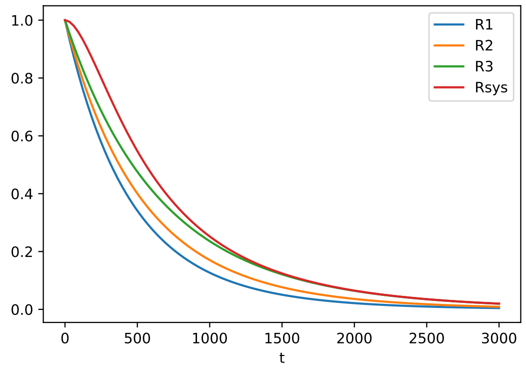
[5] TODO

[6] TODO

Visualization of R_{sys} over time:

```
df.plot(x='t')
```

```
<AxesSubplot: xlabel='t'>
```



Is this declining too quickly?

Solution 10.7 Using equation (10.3.1), we get:

```
n = 30
Kn = 28
gamma = 0.95

v1 = Kn + 1; v2 = n - Kn + 1
eps1 = (1-gamma)/2; eps2 = (1+gamma)/2

F_ll = stats.f.ppf(eps2, 2*n+2-2*Kn, 2*Kn+2)
F_ul = stats.f.ppf(eps2, 2*Kn+2, 2*n+2-2*Kn)
print(f'F-distribution median: ({F_ll:.3f}, {F_ul:.3f})')

LL = (Kn+1) / ((Kn+1) + (n-Kn+1)*F_ll)
UL = (Kn+1)*F_ul / ((n-Kn+1)+(Kn+1)*F_ul)
print(f'Credibility limits: ({LL:.3f}, {UL:.3f})')
```

```
F-distribution median: (2.635, 4.963)
Credibility limits: (0.786, 0.980)
```

Solution 10.8 Using the results of Section 10.3.2, we get:

```
gamma = 0.95
t = 25
nu = 3
tau = 0.01
r = 27
Tnr = 3500

eps1 = (1-gamma)/2; eps2 = (1+gamma)/2

beta_L = (Tnr + 1/tau) / stats.gamma.ppf(eps2, nu+r, 1)
beta_U = (Tnr + 1/tau) / stats.gamma.ppf(eps1, nu+r, 1)
print(f'Credibility limits beta: ({beta_L:.2f}, {beta_U:.2f})')
```

```
RL = np.exp(-t / beta_L)
RU = np.exp(-t / beta_U)
RL, RU
print(f'Credibility limits R(50): ({RL:.3f}, {RU:.3f})')
```

```
Credibility limits beta: (84.41, 169.48)
Credibility limits R(50): (0.744, 0.863)
```

Solution 10.9 [7]

Solution 10.10 [8]

Solution 10.11 [9]

[7] TODO: ??

[8] TODO: ?

[9] TODO: ?

Chapter 11
Sampling Plans for Batch and Sequential Inspection

Import required modules and define required functions

```
import numpy as np
import pandas as pd
from scipy import stats, optimize
import statsmodels.formula.api as smf
import lifelines
import pingouin as pg
import seaborn as sns
import matplotlib.pyplot as plt
import mistat
from mistat import acceptanceSampling
```

Solution 11.1 The single-sampling plans for attributes with $N = 2500$, $\alpha = \beta = 0.01$, and AQL and LQL as specified are (i) $n = 1878$, $c = 13$; (ii) $n = 702$, $c = 12$; (iii) $n = 305$, $c = 7$.[1]

Solution 11.2 For $\alpha = \beta = 0.05$, $AQL = 0.01$, and $LQL = 0.03$ the single-sampling plans for attributes are

N	n	c
100	87	1
500	254	4
1000	355	6
2000	453	8

We see that as the lot size, N, increases then the required sample size increases, but n/N decreases from 87% to 22.6%. The acceptance number c increases very slowly.

Solution 11.3 For the sampling plan in Exercise 11.1(iii), $OC(p) = H(7; 2500, M_p, 305)$. When $p = 0.025$, we get $M_p = 62$ and $OC(0.025) = 0.5091$.

[1] TODO: example Python code?

Solution 11.4 The large sample approximation yields $n^* = 292$, $c^* = 11$. The “exact" plan is $n = 311$, $c = 12$. Notice that the actual risks of the large sample approximation plan (n^*, c^*) are $\alpha^* = 0.0443$ and $\beta^* = 0.0543$. The actual risks of the “exact" plan are $\alpha = 0.037$ and $\beta = 0.0494$.

Solution 11.5 The large sample approximation is $n^* = 73$, $c^* = 1$ with actual risks of $\alpha^* = 0.16$ and $\beta^* = 0.06$. The exact plan is $n = 87$, $c = 2$ with actual risks of $\alpha = 0.054$ and $\beta = 0.097$.

Solution 11.6 (i) The attained α and β are $\alpha = 0.021$, $\beta = 0.0532$. (ii) ASN = 228.7. (iii) The single-sampling plan is $n = 253$, $c = 7$ with $\alpha^* = 0.0283$, $\beta^* = 0.0486$. If $p = AQL$ we expect to save 24 observations.

Solution 11.7 The double-sampling plan with $n_1 = 150$, $n_2 = 200$, $c_1 = 5$, $c_2 = c_3 = 10$, and $N = 2000$ yields

p	OC(p)	ASN(p)	p	OC(p)	ASN(p)
0	1.000	150	0.06	0.099	247.7
0.01	0.999	150.5	0.07	0.039	218.9
0.02	0.966	164.7	0.08	0.014	191.5
0.03	0.755	206.0	0.09	0.005	171.9
0.04	0.454	248.4	0.10	0.001	160.4
0.05	0.227	262.8			

The single sampling plan for $\alpha = 0.02$, $\beta = 0.10$, AQL= 0.02, LQL= 0.06 is $n = 210$, $c = 8$ with actual $\alpha = 0.02$, $\beta = 0.10$.

Solution 11.8 For the sequential plan, $OC(0.02) = 0.95 = 1 - \alpha$, $OC(0.06) = 0.05 = \beta$. ASN(0.02) = 140, ASN(0.06) = 99 and ASN(0.035) = 191.

Solution 11.9 The single-sampling plan for $N = 10,000$, $\alpha = \beta = 0.01$, $AQL = 0.01$ and $LQL = 0.05$ is $n = 341$, $c = 8$. The actual risks are $\alpha = 0.007$, $\beta = 0.0097$. The corresponding sequential plan, for $p = AQL = 0.01$ has $ASN(0.01) = 182$. On the average, the sequential plan saves, under $p = AQL$, 159 observations. This is an average savings of $159 per inspection.

Solution 11.10 [2] In Python:

```
np.random.seed(1)

N=75; lambda_=0.6; k0=15; gamma=0.95; Ns=1000

results = []
for p in (0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8):
  r = acceptanceSampling.simulateOAB(N, p, lambda_, k0, gamma, Ns)
  results.append({
      'p': p,
      'Mgamma_mean': r.mgamma.mean,
      'Mgamma_std': r.mgamma.std,
```

[2] TODO: are these reasonable results?

```
      'Reward_mean': r.reward.mean,
      'Reward_std': r.reward.std,
  })
pd.DataFrame(results)
```

```
      p  Mgamma_mean  Mgamma_std  Reward_mean  Reward_std
0  0.40       22.639   11.931080      40.4016    0.904609
1  0.45       31.382   19.866758      40.3588    1.222515
2  0.50       42.608   24.492822      40.8492    2.075587
3  0.55       57.779   23.582794      42.6366    3.416241
4  0.60       66.750   18.822526      45.4380    4.143640
5  0.65       71.662   12.721154      49.1878    4.508895
6  0.70       73.924    7.576953      52.9706    4.282127
7  0.75       74.762    3.755710      57.0788    3.971714
8  0.80       74.945    1.738383      60.7050    3.518803
```

Solution 11.11 [3] In Python:

```
result = acceptanceSampling.optimalOAB(75, 0.6)
print(f'Case (75, 0.6): {result.max_reward:.3f}')
```

```
Case (75, 0.6): 62.447
```

Solution 11.12 [4]

Solution 11.13 For a continuous variable-size sampling plan when (p_0) = AQL = 0.01, (p_t) = LQL = 0.05, and $\alpha = \beta = 0.05$, we obtain $n = 70$ and $k = 1986$.

Solution 11.14 From the data we get $\bar{X} = 14.9846$ and $S = 0.019011$. For $p_0 = 0.01$ and $\alpha = 0.05$ we obtain $k = 2.742375$. Since $\bar{X} - kS = 14.9325 > \zeta = 14.9$, the lot is *accepted*. $OC(0.03) \approx 0.4812$.

Solution 11.15 For $AQL = 0.02$, $LQL = 0.04$ and $\alpha = \beta = 0.10$, $n = 201$, $k = 1.9022$.

Solution 11.16 For a single-sampling plan for attributes where $n = 139$, $c = 3$ and $N = 500$ we obtain $OC(p) = H(3; 500, M_p, 139)$, and $R^* = H(2; 499, M_p - 1, 138)/H(3; 500, M_p, 139)$.

p	AOQ	ATI
0.01	0.0072	147.2
0.02	0.0112	244.2
0.03	0.0091	369.3
0.05	0.0022	482.0

Solution 11.17 A switch to a tightened plan, when all 5 consecutive lots have $p = AQL$, with $\alpha = 0.05$, is $\sum_{j=2}^{5} b(j; 5, 0.05) = 0.0226$. The probability of switching to a tightened plan if $\alpha = 0.3$ is 0.4718.

[3] TODO: are these reasonable results?

[4] TODO: how is this done?

Solution 11.18 The total sample size from 10 consecutive lots is 1000. Thus, from Table 11.15[5] S_{10} should be less than 5. The last 2 samples should each have less than 2 defective items. Hence, the probability for qualification is

$$\begin{aligned} QP &= b^2(1; 100, 0.01)B(2; 800, 0.01) \\ &\quad + 2b(1; 100, 0.01)b(0; 100, 0.01)B(3; 800, 0.01) \\ &\quad + b^2(0; 100, 0.01)B(4; 800, 0.01) \\ &= 0.0263. \end{aligned}$$

Solution 11.19 Define function for the four models

```
def modelGoelOkumoto(t, a, b):
    return a * (1 - np.exp(-b * t))

def modelMusaOkumoto(t, phi, lam):
    return (1/phi) * np.log(lam*phi* t + 1)

def modelYamada(t, a, b):
    return a * (1 - (1+b*t)*np.exp(-b*t))

def modelInflectedSshaped(t, a, b, c):
    return a * (1 - np.exp(-b * t)) / (1 + c * np.exp(-b * t))
```

Fit the four functions to the cumulative failure count data `CFC`

```
def optimizeModelFit(model, data, x, y):
    fit = optimize.curve_fit(model, data[x], data[y], method='lm')
    popt = fit[0]
    # add the fit curve to the data
    data[model.__name__] = [model(t, *popt) for t in data[x]]
    return popt
data = mistat.load_data('FAILURE_J2')
optimizeModelFit(modelGoelOkumoto, data, 'T', 'CFC')
optimizeModelFit(modelMusaOkumoto, data, 'T', 'CFC')
optimizeModelFit(modelYamada, data, 'T', 'CFC')
optimizeModelFit(modelInflectedSshaped, data, 'T', 'CFC')
```

```
array([2.26933527e+02, 5.40121092e-02, 5.96246327e+01])
```

The fits are shown in Fig. 11.1. The inflected S-shaped model describes the data best.

Solution 11.20 Modify the *optimizeModelFit* function from the previous exercise to restrict the fit to a subset of the data. The

```
def optimizeModelFit(model, data, x, y, subset):
    # create the subset
    subsetX = data[x][:subset]
    subsetY = data[y][:subset]
    # fit curve to subset - a increase of maxfev is required
    fit = optimize.curve_fit(model, subsetX, subsetY,
                             method='lm', maxfev=2000)
    popt = fit[0]
```

[5] TODO: is this the correct table?

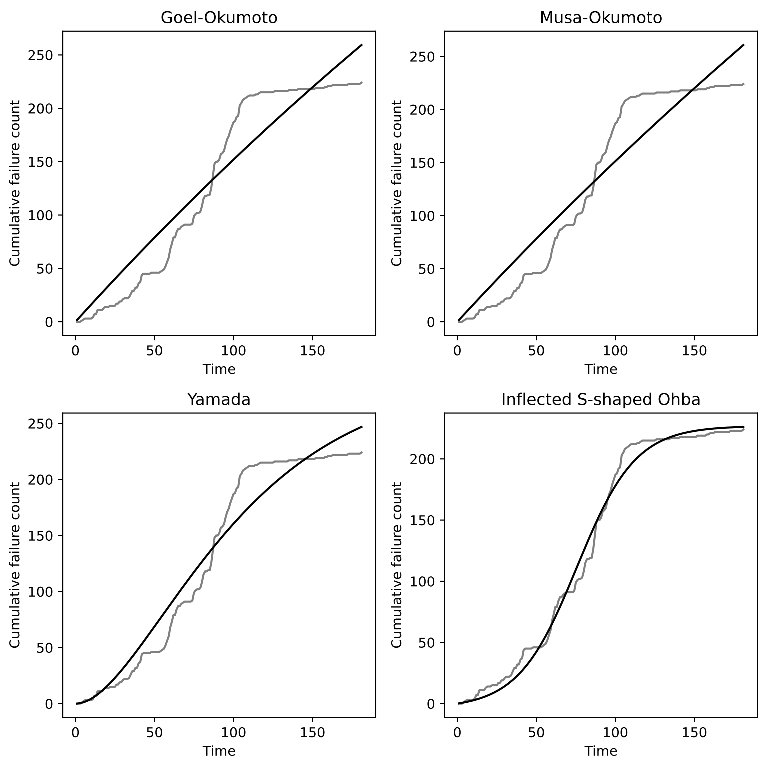


Fig. 11.1: NHPP models fitted to cumulative failure count data **FAILURE_J2**

```
    data[f'{model.__name__} {subset}'] = [model(t, *popt) for t in data[x]]
    return popt
for subset in [25, 50, 75, 100, 125, 150]:
    optimizeModelFit(modelGoelOkumoto, data, 'T', 'CFC', subset)
    optimizeModelFit(modelInflectedSshaped, data, 'T', 'CFC', subset)
```

The result of this simulation is shown in Fig. 11.2. As in the previous exercise, the inflected S-shaped model performs best. However, the fit using only 25 weeks of data is insufficient to extrapolate into the future. With 50 or 75 weeks of data, the inflected S-shaped model predicts the following 20-30 weeks well. Shortly after 100 weeks, the cumulative failure count curve flattens. This is not predicted by the model fit. Even with 125 weeks of data, this is not well described. Only at 150 weeks, the inflected S-shaped model starts to shows flattening.

Solution 11.21 Load the data and fit the model

```
data = mistat.load_data('FAILURE_DS2')
fit = optimize.curve_fit(modelInflectedSshaped, data['T'], data['CFC'])
popt = fit[0]
```

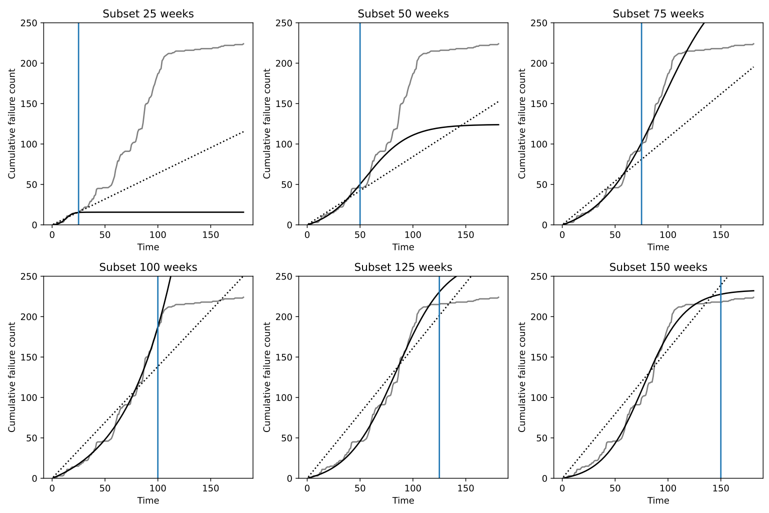


Fig. 11.2: NHPP models fitted to cumulative failure count data **FAILURE_J2** using subsets. The actual data are shown in grey, the Goel-Okumoto fits as dotted black lines, and the inflected S-shaped models as solid black lines. The vertical line shows the cutoff used for subsetting the data for the fits.

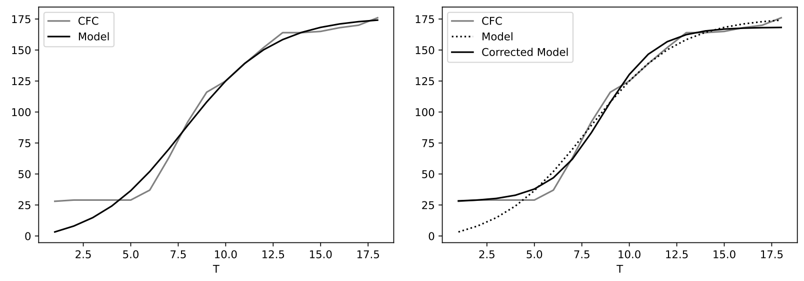


Fig. 11.3: NHPP models fitted to cumulative failure count data **FAILURE_DS2**. Left: initial model, right: after correction

```
# add the fit curve to the data and visualize
data['Model'] = [modelInflectedSshaped(t, *popt) for t in data['T']]

ax = data.plot(x='T', y='CFC', color='grey')
data.plot(x='T', y='Model', color='black', ax=ax)
plt.show()
```

The result of this simulation is shown in Fig. 11.3. It is clear that the fit could be better if the model ignores the initial failure count. Define a new model to correct for this problem.

```
initial = data['CFC'][0]
def correctedModel(t, a, b, c):
    return modelInflectedSshaped(t, a, b, c) + initial

fit = optimize.curve_fit(correctedModel, data['T'], data['CFC'])
popt = fit[0]
# add the fit curve to the data and visualize
data['Corrected Model'] = [correctedModel(t, *popt) for t in data['T']]

ax = data.plot(x='T', y='CFC', color='grey')
data.plot(x='T', y='Model', color='black', ax=ax, linestyle=':')
data.plot(x='T', y='Corrected Model', color='black', ax=ax)
plt.show()
```

Including the offset in the model, leads to a better fit of the data.