第4章：計測データ解析の基礎

KYUSHU
UNIVERSITY

全体構成

Chapter 1
- 第1章：導入

Chapter 2
- 第2章：IoT基礎（センサー）

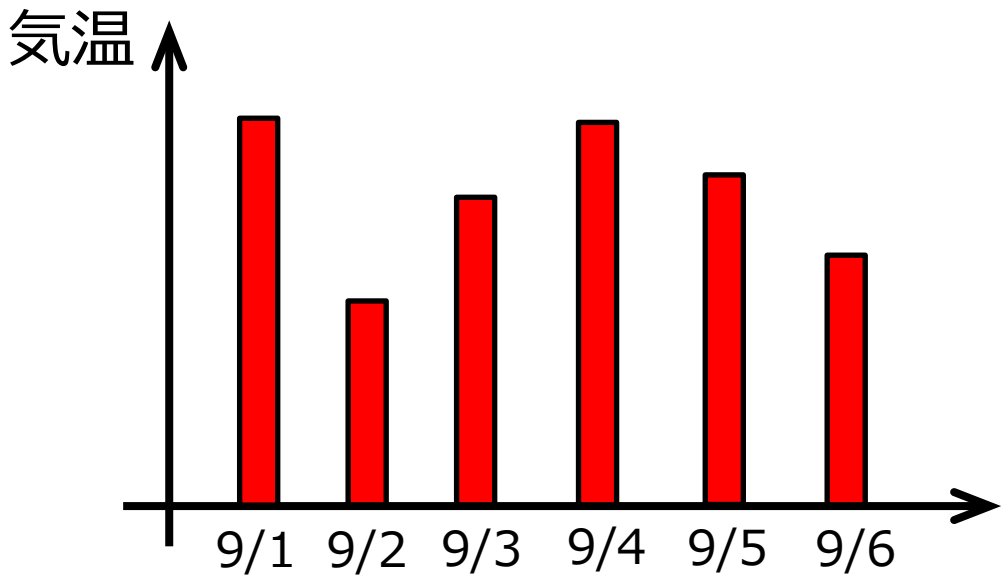
Chapter 3
- 第3章：マイクロコントローラ（Arduino）の基礎

Chapter 4
- 第4章：計測データ解析の基礎

Chapter 5
- 第5章：計測データ処理(IoT)の応用

計測データ処理の目的

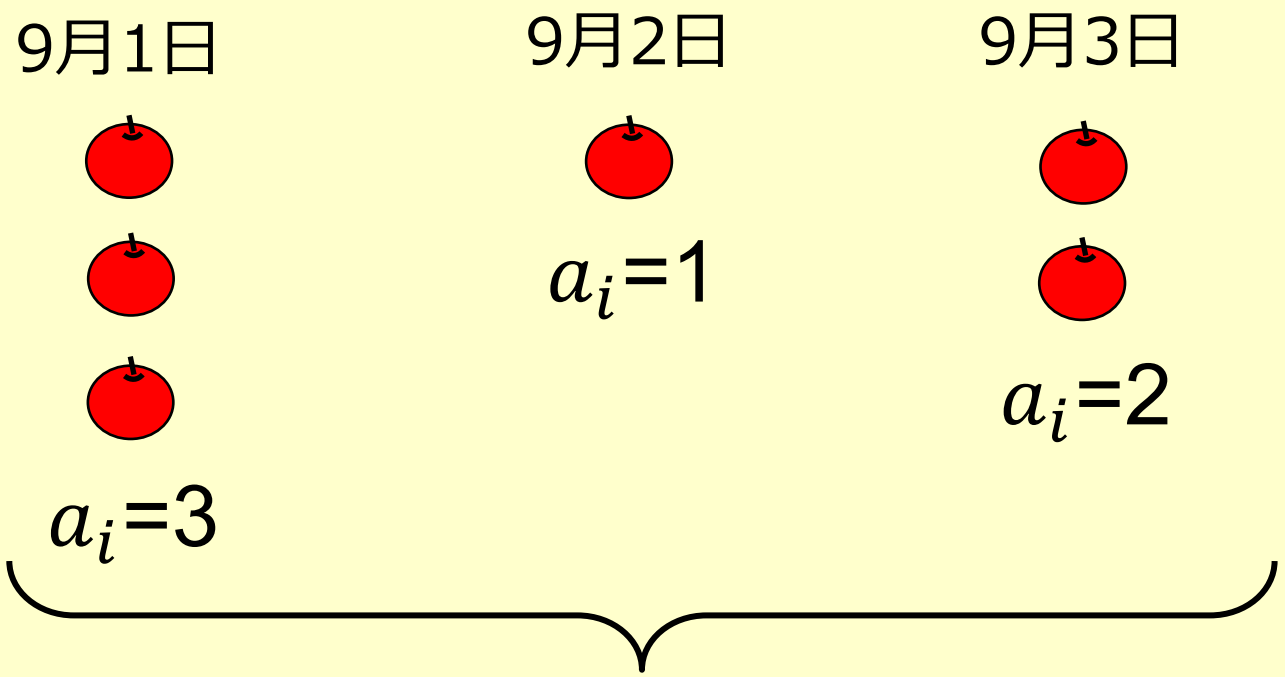
計測したデータにはノイズやバラツキが含まれる。そのため、取得したデータを整理・分析して、その全体的な傾向や統計的な特徴を理解することが重要である。



例えば、毎日の気温や収穫量のようなデータも日によって変動するため、統計的な処理を施すことでその特徴や傾向を理解しやすくなる。

本章では、簡単な例題を用いながら、平均・分散・移動平均・確率分布等の統計的な指標およびその考え方と、それらを求めるプログラムの実装方法について学習する。

例：3日間のリンゴの収穫量



一日あたり平均で「期待できる収穫量」

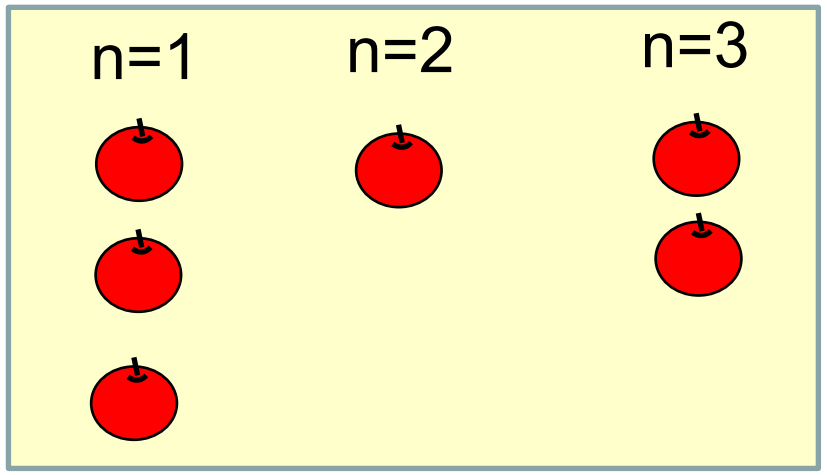
平均値（期待値）

$$A = \frac{1}{N}\sum_{n=1}^{N} a_n$$

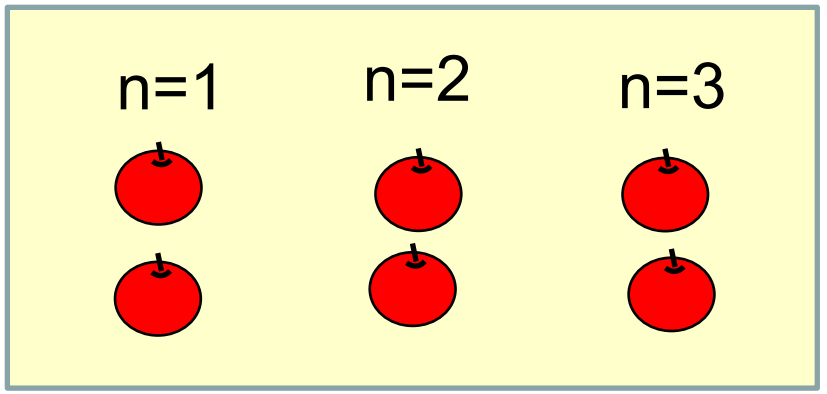
計算例：

$$A = \frac{1}{3}\sum_{n=1}^{3} a_n = \frac{3+1+2}{3} = 2$$

以下の場合、いずれも平均値は「２」、でも、統計的な意味は同じではない。



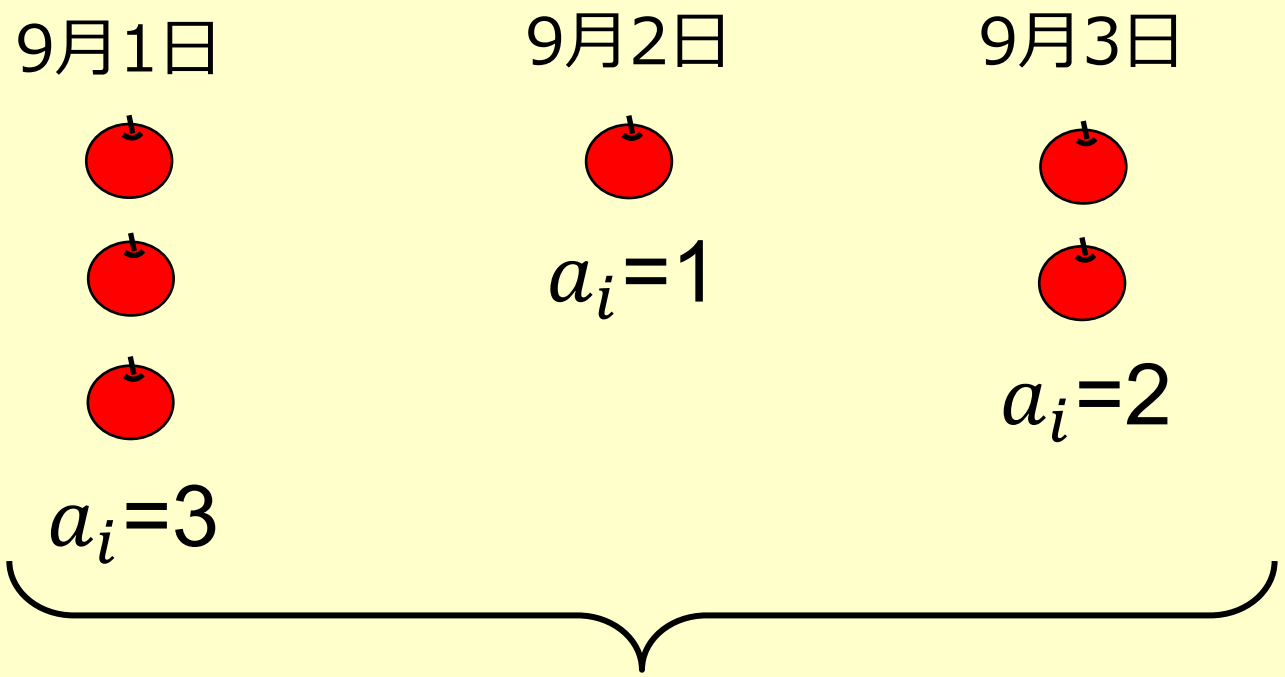
平均は２個だけど、
バラツキが大きい



毎日２個収穫、
でも３個とれることはなさそう

そのようなバラツキの程度を表現するには、
「分散（平均値からのバラツキの程度）」を用いることが有効

例：3日間のリンゴの収穫量



平均値からある程度のバラツキがある
その程度を定量化した指標が必要 ➔

分散

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(a_n - A)^2$$

計算例：

$$\sigma^2 = \frac{1}{3}\sum_{n=1}^{3}(a_n - 2)$$

$$= \frac{1 + (-1)^2 + 0}{3} = \frac{2}{3}$$

補足：母分散と標本分散

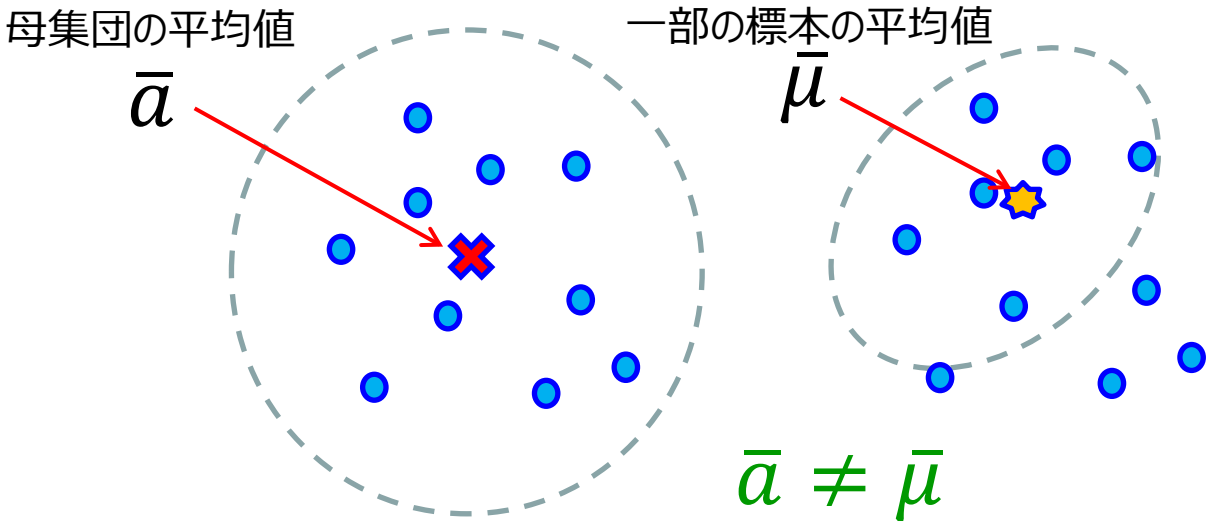
- 前ページで示した式は母分散の定義式であり、観測データ全体が母集団そのものとみなせる場合に用いる。実際には母集団の一部（標本）しか観測できないことが多く、その場合は以下の標本分散を用いる。

$$\sigma_s^2 = \frac{1}{m-1}\sum_{n=1}^{m}(a_n - \bar{\mu})^2$$

ここで

$$\bar{\mu} = \frac{1}{m}\sum_{n=1}^{m} a_n$$

m：標本サイズ



母分散を

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(a_n - \bar{a})^2$$

標本値の分散の推定量を

$$v^2 = \frac{1}{m}\sum_{n=1}^{m}(a_n - \bar{\mu})^2$$ とするとき

v^2の集合平均は

$$\mathrm{E}[v^2] = \frac{\mathrm{m}-1}{\mathrm{m}}\sigma^2$$

したがって、

$$\sigma_s^2 = \frac{m}{m-1}\mathrm{E}[v^2] = \frac{1}{m-1}\sum_{n=1}^{m}(a_n - \bar{\mu})^2$$

プログラム例（平均）

```
apple = [3, 1, 2] #リンゴの収穫量
total = 0
count = 0

#リンゴの総量(total)と収穫日数（count)を計算
for i in apple:
  total = total + i
  count = count + 1

#平均値を計算
average = total / count

#結果を出力
print("平均:",average)
```

結果の例

収穫量	出力
[3, 1, 2]	平均: 2.0
[3, 5, 7, 9]	平均: 6.0

プログラム例（標本分散）

```
apple = [3, 1, 2] #リンゴの収穫量

#リンゴの総量(total)と収穫日数（count)を計算
total = 0
count = 0
for i in apple:
  total = total + i
  count = count + 1

#平均値を計算
average = total / count

#標本分散を計算
sq_diff_total = 0
for i in apple:
  sq_diff_total = sq_diff_total + (i - average) ** 2

if count>1:
  var = sq_diff_total / (count-1)
else:
  var = float('nan')

#結果を出力
print("平均:", average, "標本分散:", var)
```

結果の例（母分散）

収穫量	出力
[3, 1, 2]	平均: 2.0　母分散: 0.6666666666666666
[2, 2, 2]	平均: 2.0　母分散: 0.0
[3, 5, 7, 9]	平均: 6.0　母分散: 5.0

結果の例（標本分散）

収穫量	出力
[3, 1, 2]	平均: 2.0　標本分散: 1.0
[2, 2, 2]	平均: 2.0　標本分散: 0.0
[3, 5, 7, 9]	平均: 6.0　標本分散: 6.6666…

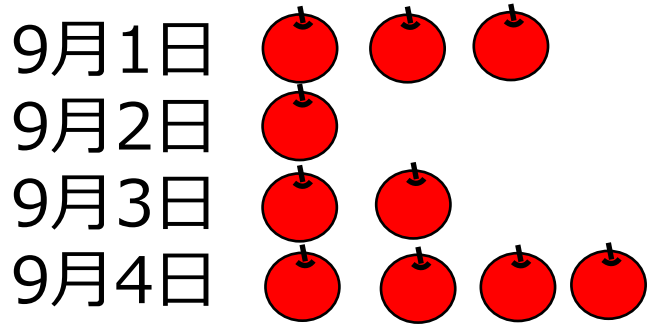
※ プログラム例では標本分散を求めている

- 日ごとの収穫量にバラツキがある。
- 「最近3日間の平均収穫量」を見れば「最近のおおまかな収穫の程度」がわかる。
- 日ごとの細かいバラツキをならして全体的な傾向をみたい（収穫が上昇傾向なのかどうか等）

そのような場合には、「移動平均(Moving Average)」を利用できる

$$f(t) = \frac{1}{k}\sum_{i=0}^{k-1} a_{i-n}$$

※ k の値の設定により
バラツキを抑える程度がかわる



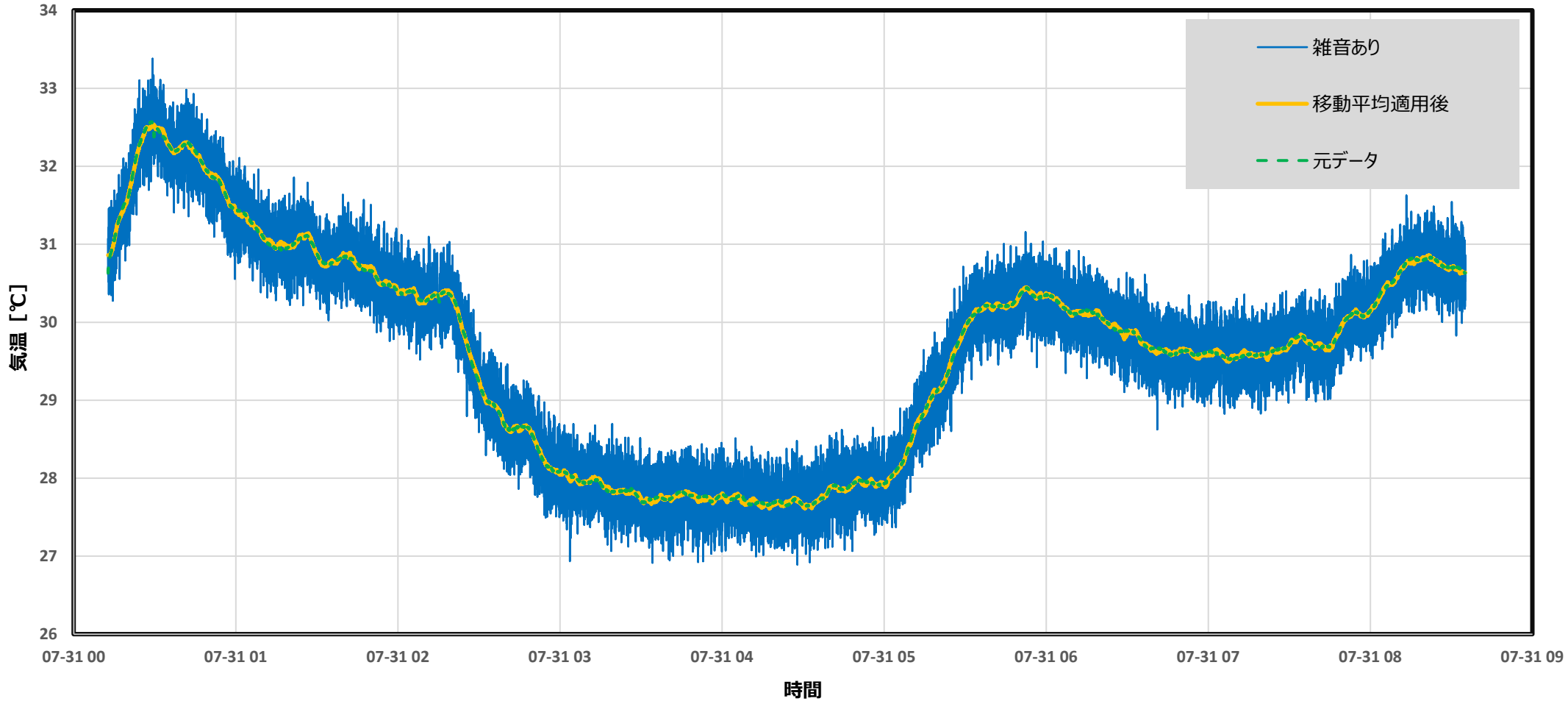
a_1=3
a_2=1
a_3=2
a_4=4
a_5=0

9月3日の移動平均： (3+1+2)/3 = 2
9月4日の移動平均： (1+2+4)/3 = 2.33
9月5日の移動平均： (2+4+0)/3 = 2

直近3日間の収穫量の平均を計算（細かい収穫量の上下をならす）

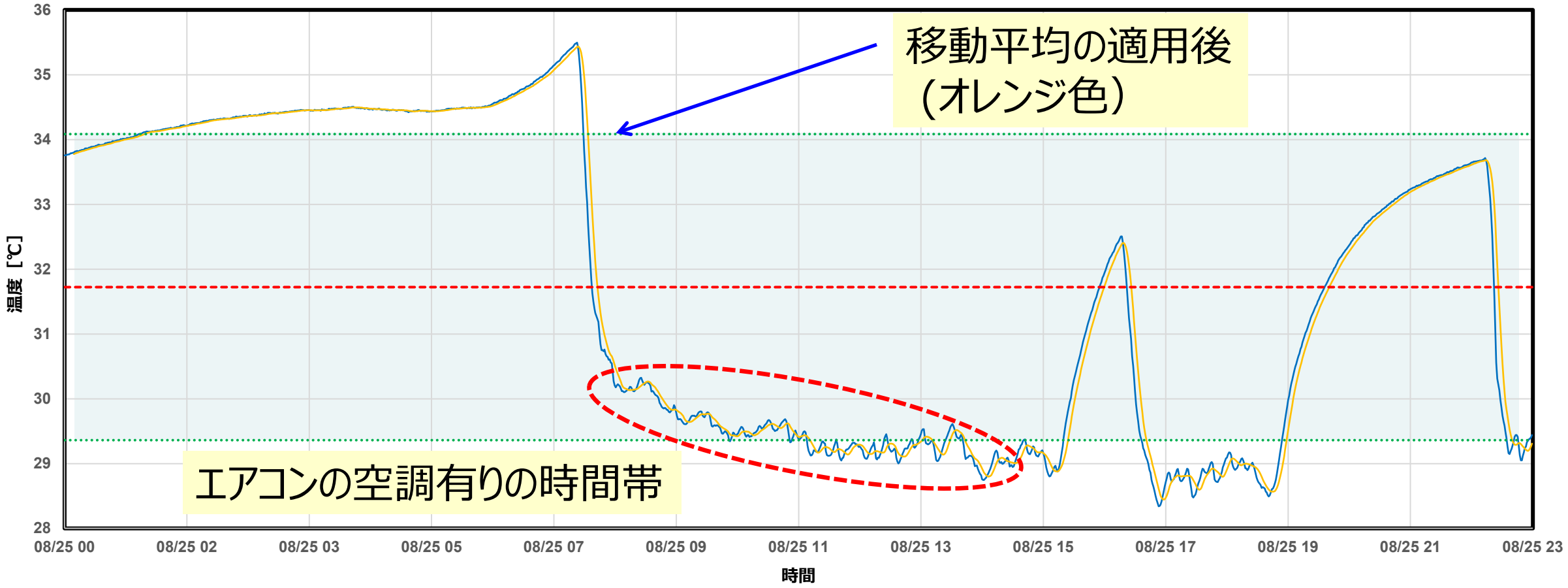
移動平均のシミュレーション例

雑音が加わった観測値に移動平均を適用した例



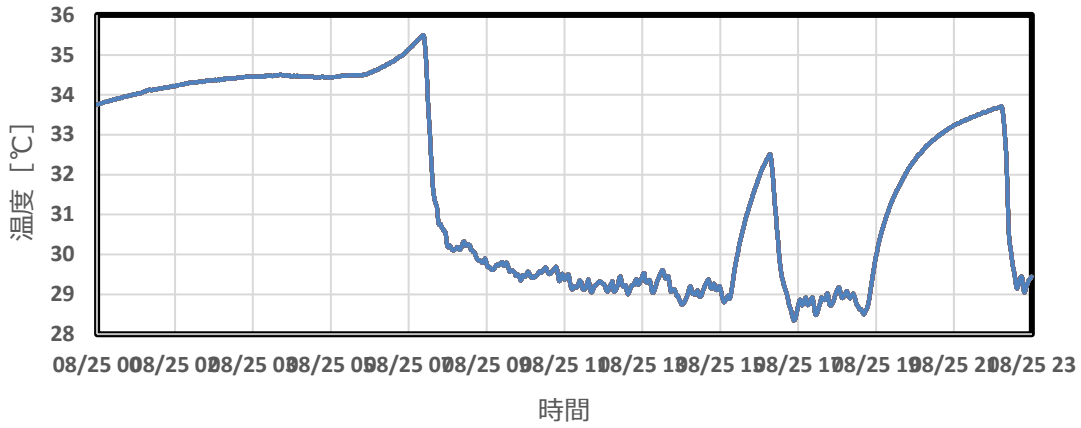
実計測値に対する移動平均例

室内の温度の変化の例（本演習用のデバイスで計測した結果）

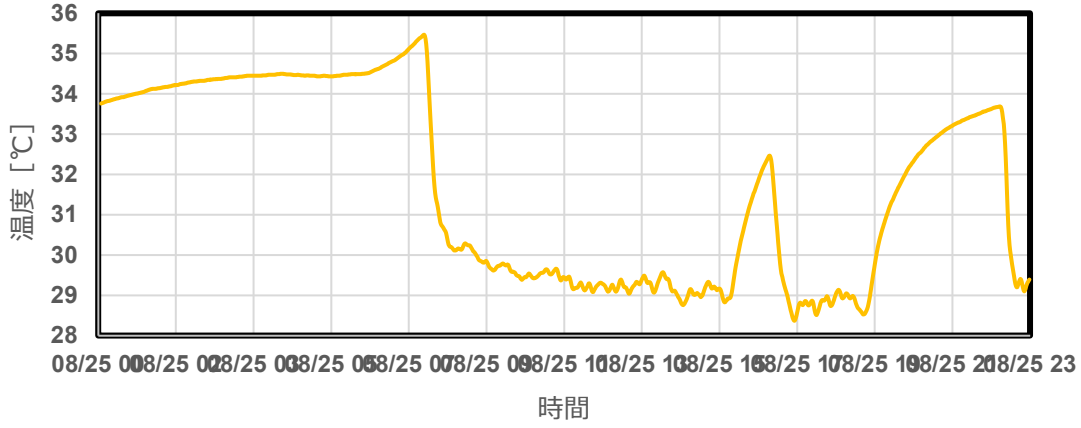


実計測値に対する移動平均例

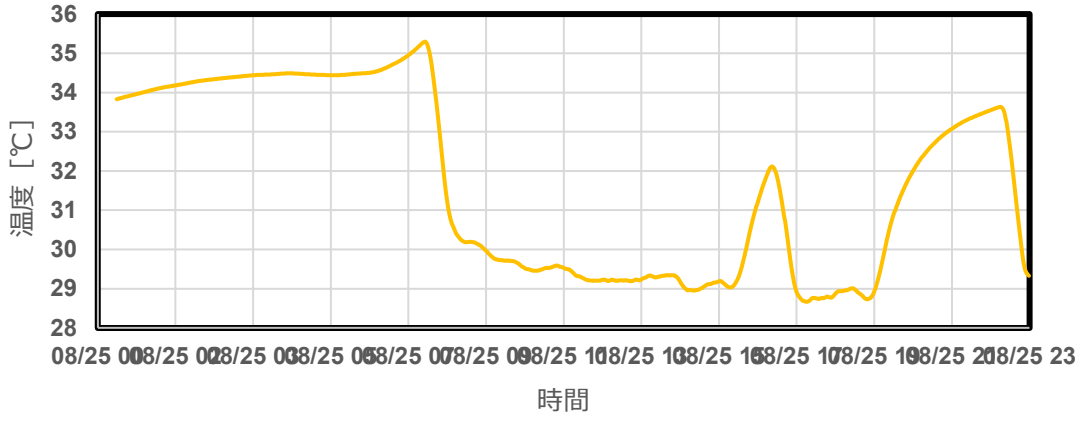
元データ



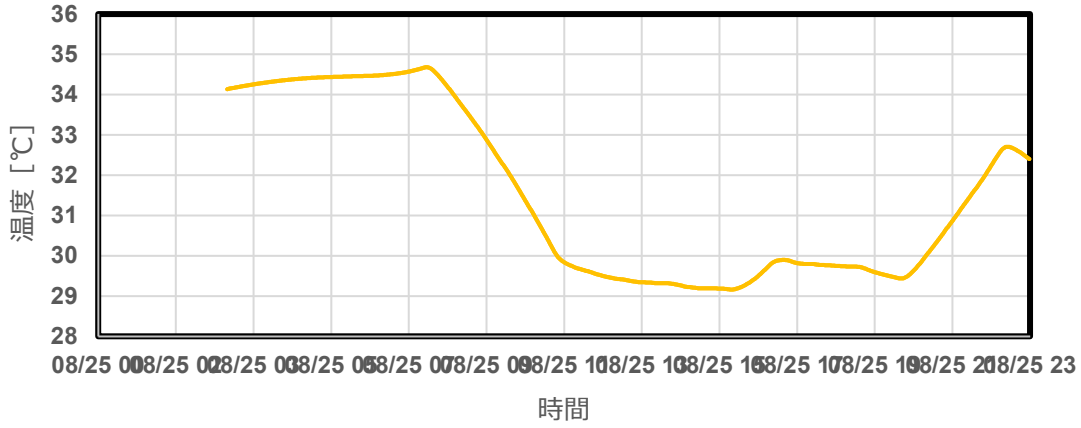
移動平均長＝5



移動平均長＝30

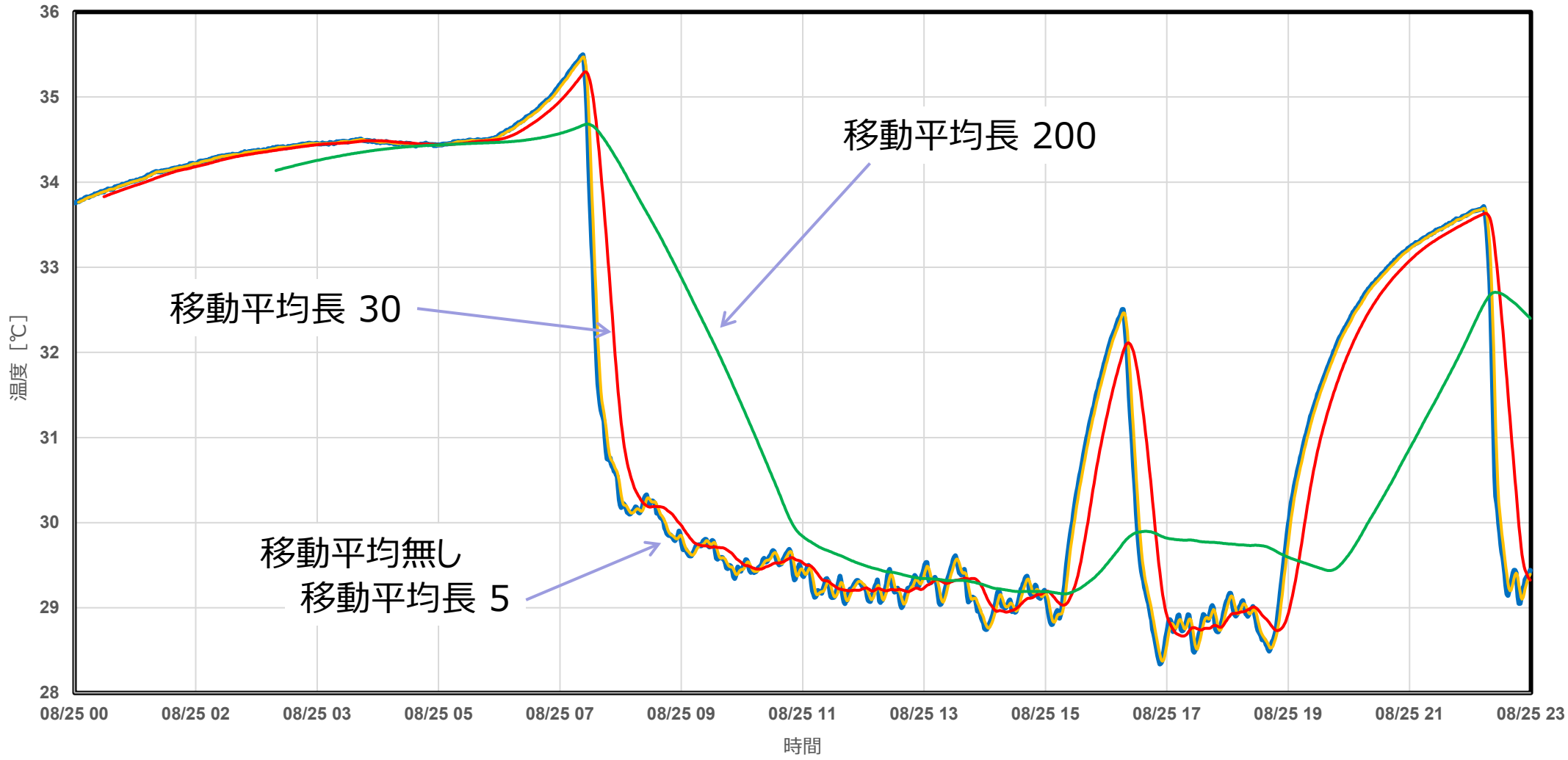


移動平均長＝200



移動平均長を適切に設定することが大切

実計測値に対する移動平均例



移動分散

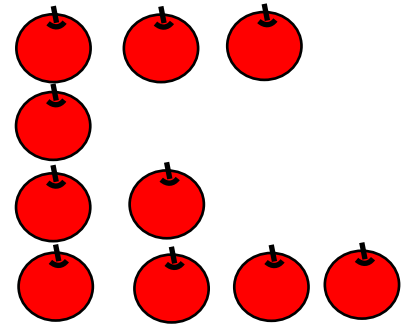
移動平均と同様に、収穫量のバラツキ(安定度)の大まかな傾向をみたい。

移動分散（Moving Variance）を利用できる。

$$s(t) = \frac{1}{k}\sum_{i=0}^{k-1}\left(a_{t-i} - f(t)\right)^2$$

ここで、

$$f(t) = \frac{1}{k}\sum_{i=0}^{k-1} a_{i-n}$$



a_1=3
a_2=1
a_3=2
a_4=4
a_5=0

9月3日の移動分散：$\frac{(3-2)^2+(1-2)^2+(2-2)^2}{3} = 0.67$

9月4日の移動分散： 1.56

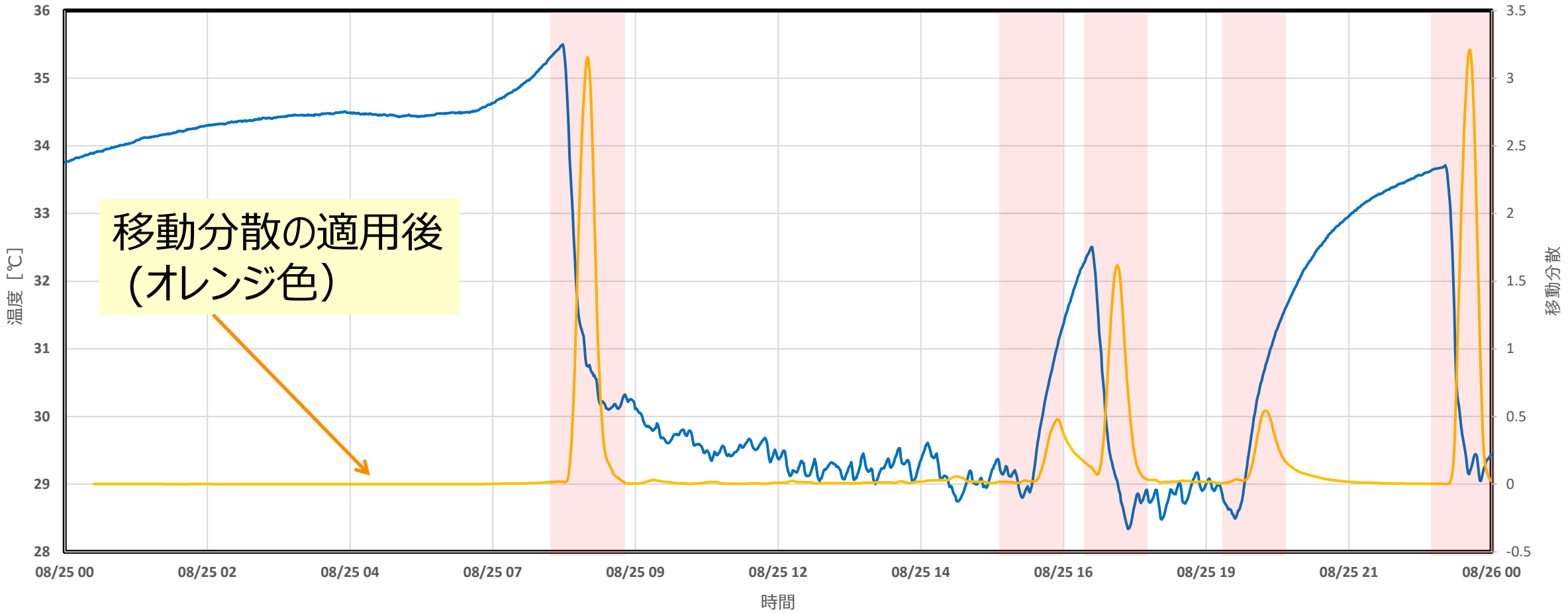
9月5日の移動分散： 2.67

平均は２程度であるが、
分散は大きく異なる。

移動分散：平均だけではみえない「バラツキの変化」を可視化できる

実計測値に対する移動分散の例

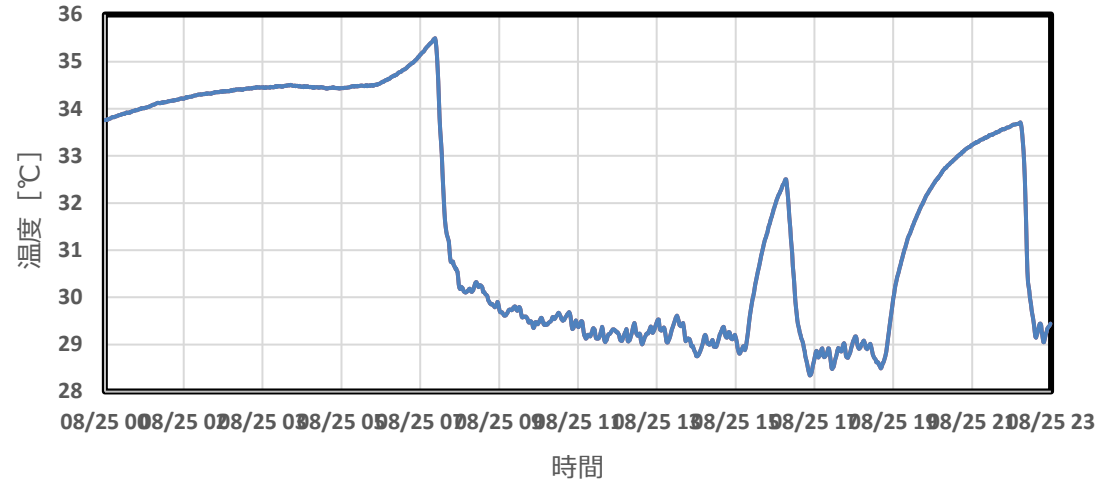
室内の温度の変化の例（本演習用のデバイスで計測した結果）



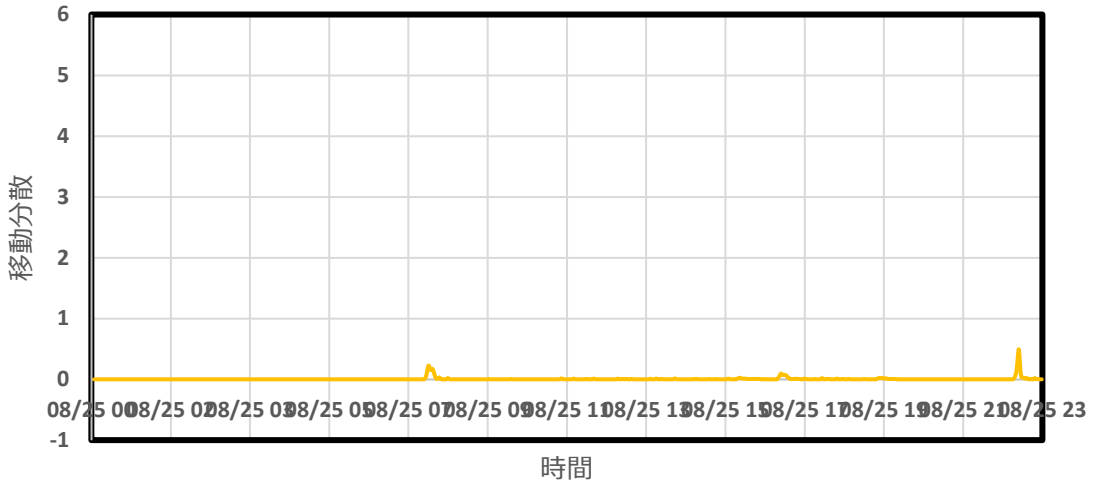
エアコンのONとOFFを可視化

実計測値に対する移動分散の例

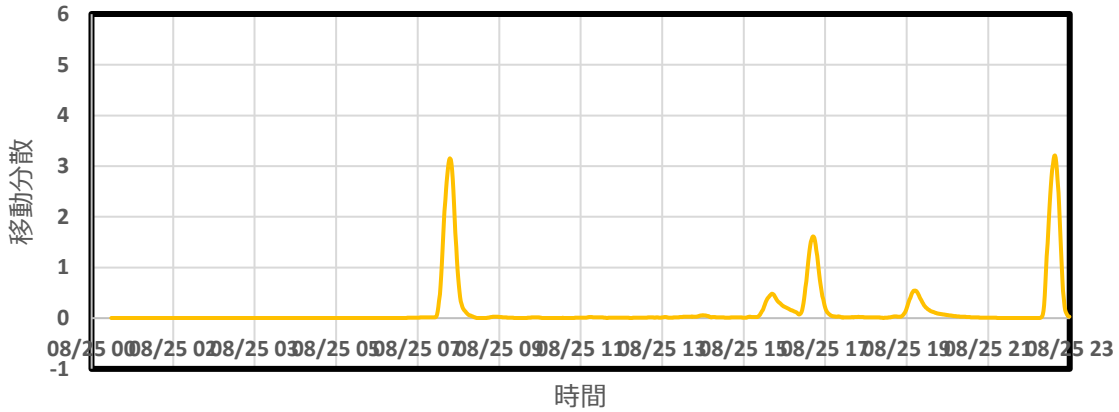
元データ



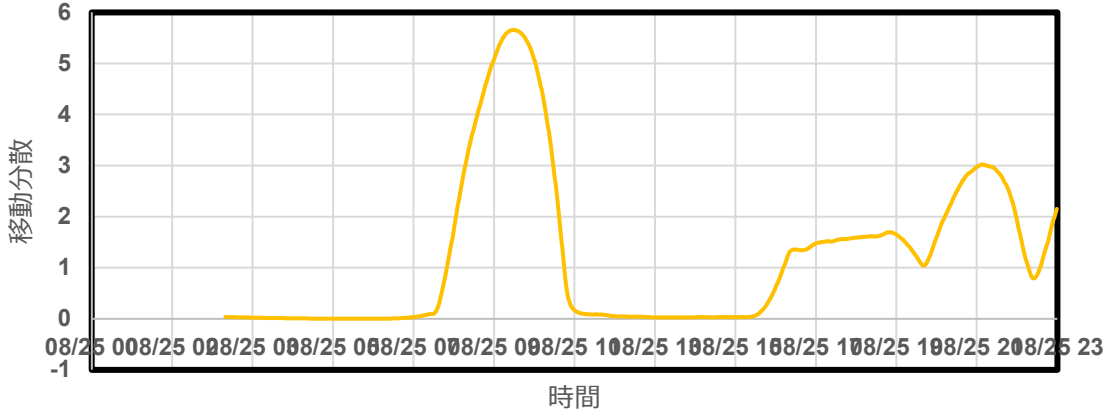
移動分散長＝5



移動平均長＝30



移動平均長＝200



移動分散長を適切に設定することが大切

移動平均のプログラム例

プログラム例（移動平均）

```
apple = [3, 1, 2, 4, 0]    #リンゴの収穫量
window = 3                 #移動平均長
moving_averages = []       #結果を入れるリスト

#移動平均を計算
for i in range (len(apple) - window + 1):
  #平均を計算
  total = 0
  for j in range (window):
    total = total + apple[i+j]
  average = total / window
  #結果を追加
  moving_averages.append(average)

print("移動平均", moving_averages)
```

結果の例

```
#収穫量と移動平均長
apple = [3, 1, 2, 4, 0]
window = 3

#結果
移動平均 [2.0, 2.3333333333333335, 2.0]
```

```
#収穫量と移動平均長
apple = [3, 1, 2, 4, 0, 3, 3, 5, 5, 6]
window = 3

#結果
移動平均 [2.0, 2.3333333333333335, 2.0, 2.0, 2.0, 3.0,
4.0, 5.0]
```

移動分散のプログラム例

プログラム例（移動分散）

```
apple = [3, 1, 2, 4, 0]    #リンゴの収穫量
window = 3                 #移動分散長
moving_variances = []      #結果を入れるリスト


# 移動分散を計算
for i in range(len(apple) - window + 1):

  #平均を計算
  total = 0
  for j in range (window):
    total = total + apple[i+j]
  average = total / window

  #分散を計算
  sq_diff_total = 0
  for j in range (window):
    sq_diff_total = sq_diff_total + (apple[i+j] - average) ** 2
  variance = sq_diff_total / window

  #結果を追加
  moving_variances.append(variance)


print("移動分散", moving_variances)
```

結果の例

```
#収穫量と移動平均長
apple = [3, 1, 2, 4, 0]
window = 3

#結果
移動分散 [0.6666666666666666,
1.5555555555555554, 2.6666666666666665]
```

```
#収穫量と移動平均長
apple = [3, 1, 2, 4, 0, 3, 3, 5, 5, 6]
window = 3

#結果
移動分散 [0.6666666666666666,
1.5555555555555554, 2.6666666666666665,
2.8888888888888893, 2.0, 0.8888888888888888,
0.888888888888889, 0.2222222222222224]
```

確率質量関数・累積分布関数

確率質量関数 PMF (Probability Mass Function) :

$a \leq x \leq b$である確率

$$P(a \leq x \leq b) = \sum_{x=a}^{b} p(x)$$

$p(x)$ ← 確率質量関数

ここで $\sum_{x} p(x) = 1$ $p(x)$の総和は 1

$p(x) \geq 0$ $p(x)$が負になることはない

※ p(x)においてxの取り得る値が連続値の場合、確率密度関数と呼ぶ

確率質量関数・累積分布関数

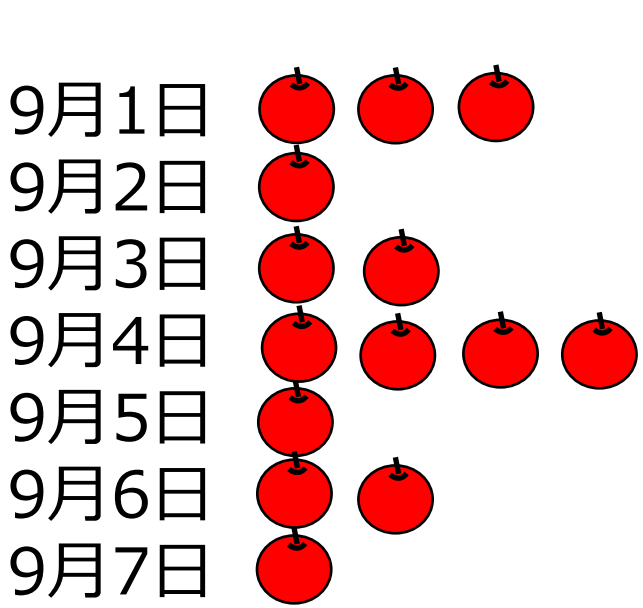
累積分布関数（CDF: Cumulative Distribution Function）

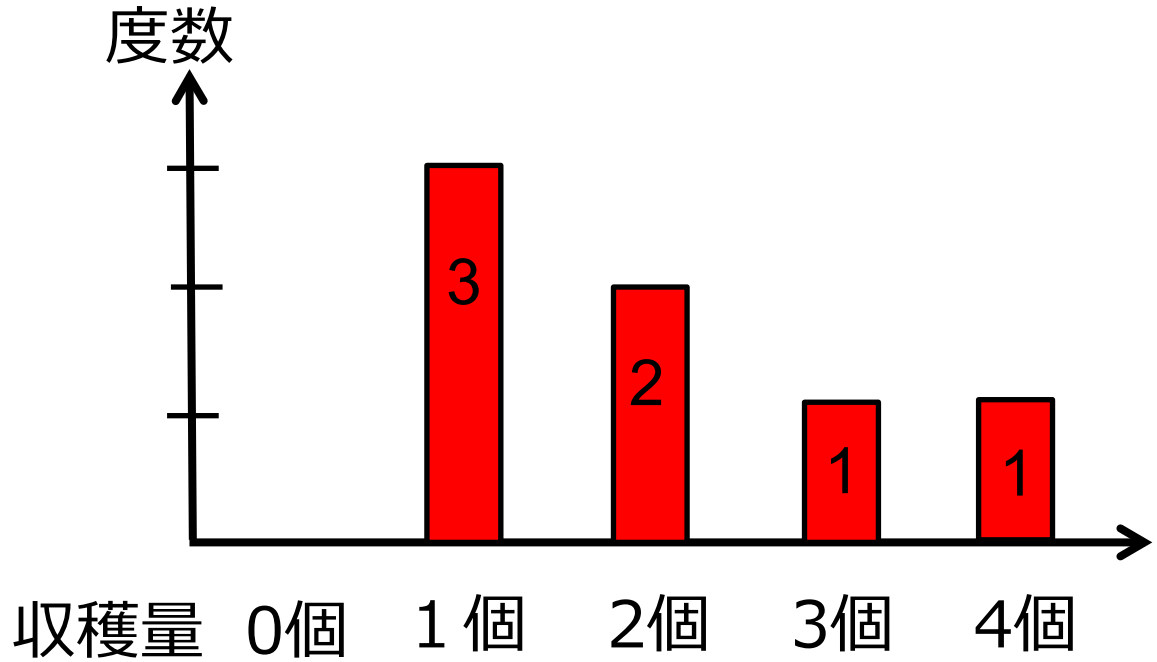
$$F(x) = P(X \leq x) = \sum_{t \leq x} p(t)$$

$X \leq x$となる確率（PMFを$X \leq x$の範囲で足し合わせたもの）

確率質量関数・累積分布関数

例：観測データから確率質量関数と累積分布関数の経験分布を求める

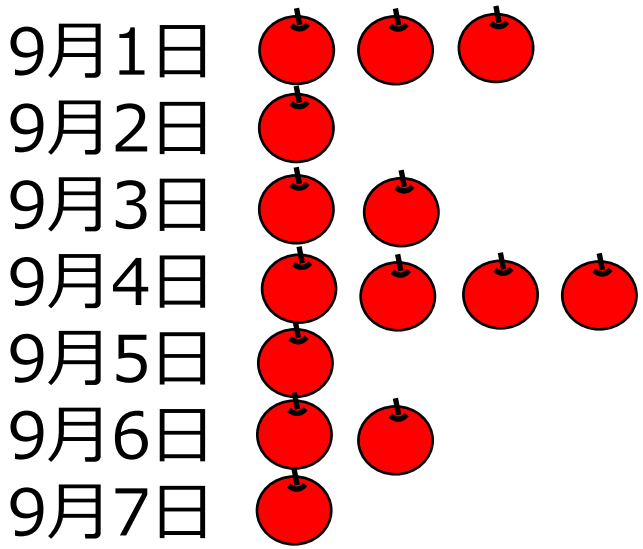




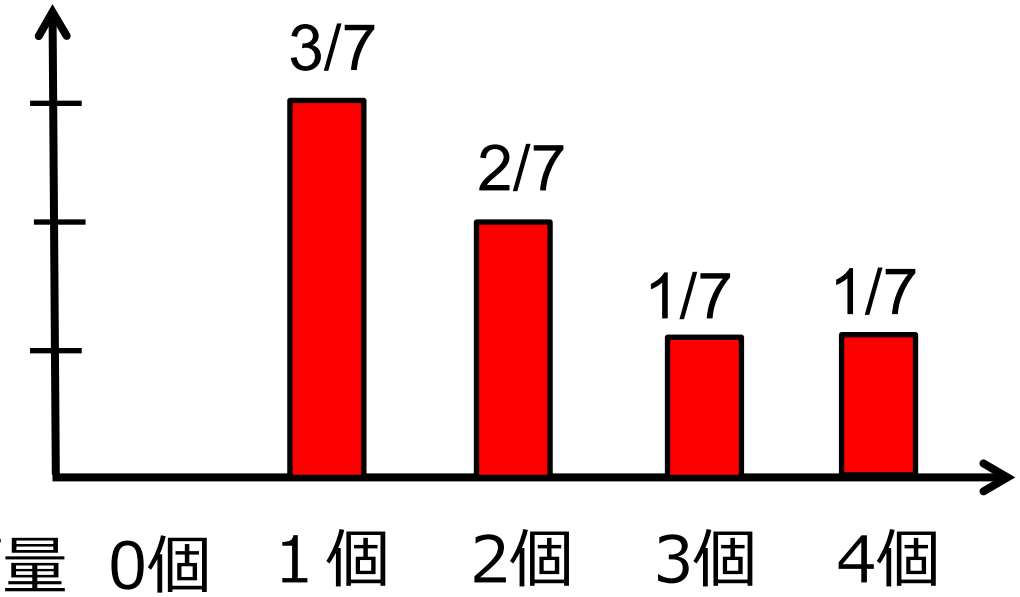
収穫量の合計 ： $1 \times 3 + 2 \times 2 + 3 \times 1 + 4 \times 1 = 14$

度数和 $= 3 + 2 + 1 + 1 = 7$

確率質量関数・累積分布関数



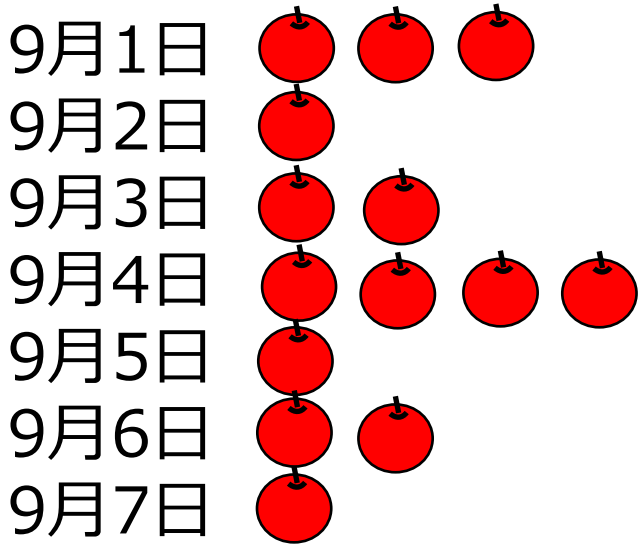
確率質量関数 $p(x)$



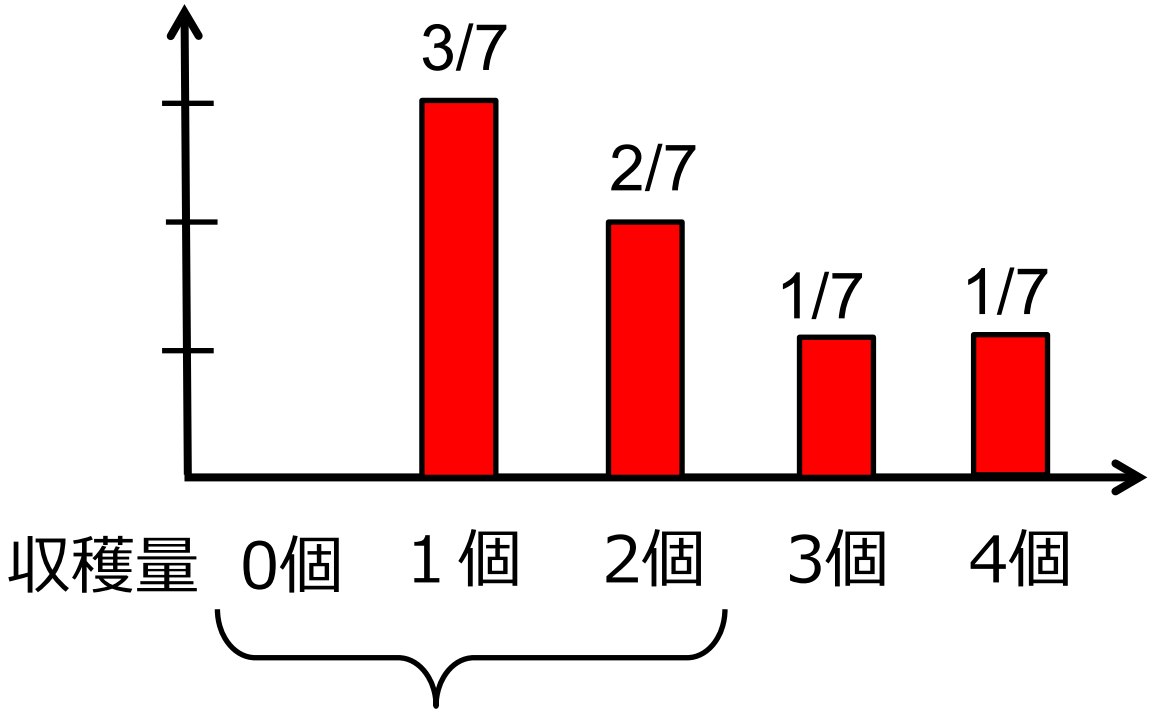
度数和が 1 になるように正規化したもの

$$\sum_x p(x) = 1 \qquad p(x) \geq 0$$

確率質量関数・累積分布関数



確率質量関数 $p(x)$　※経験分布

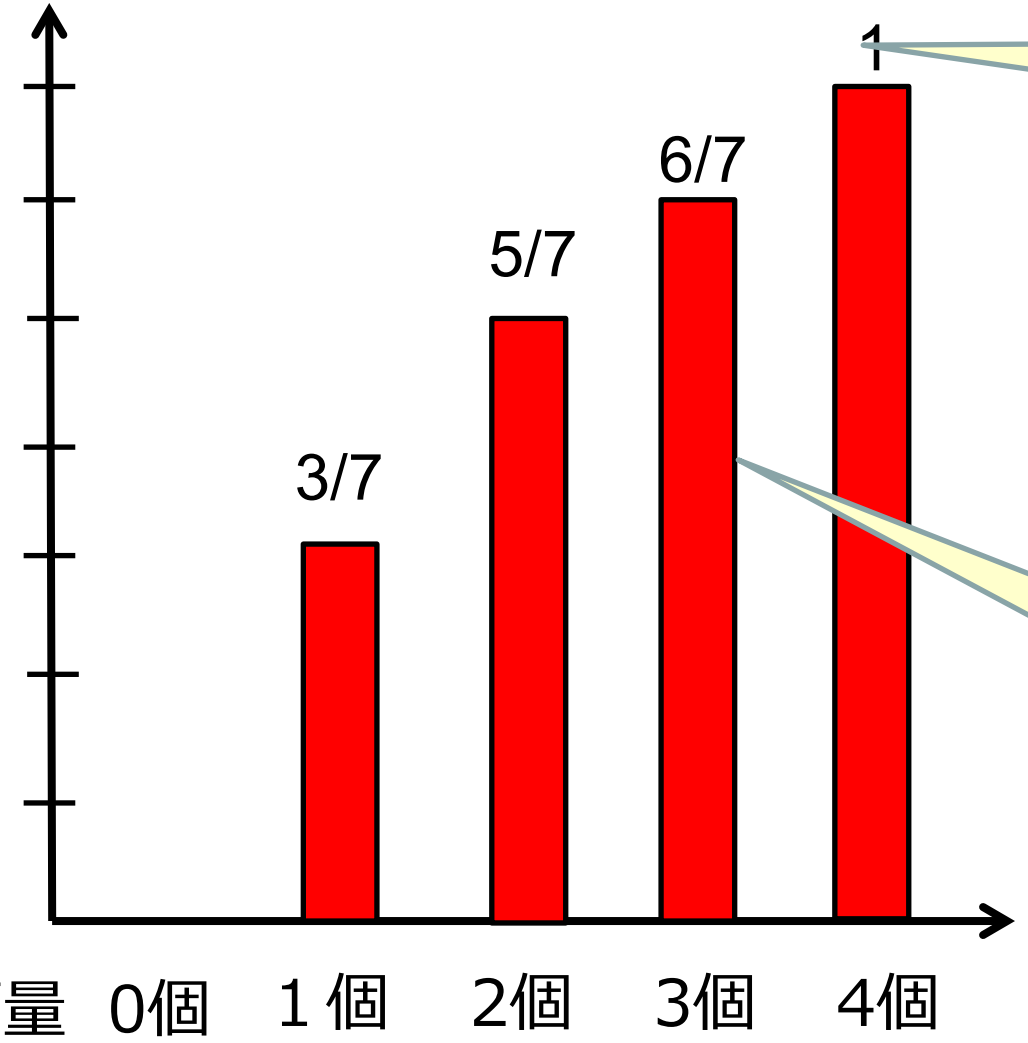


$$F(2) = P(X \leq 2) = p(X = 0) + p(X = 1) + p(X = 2)$$

リンゴの収穫量が２個以下となる確率（累積分布関数値 F(X=2)）

確率質量関数・累積分布関数

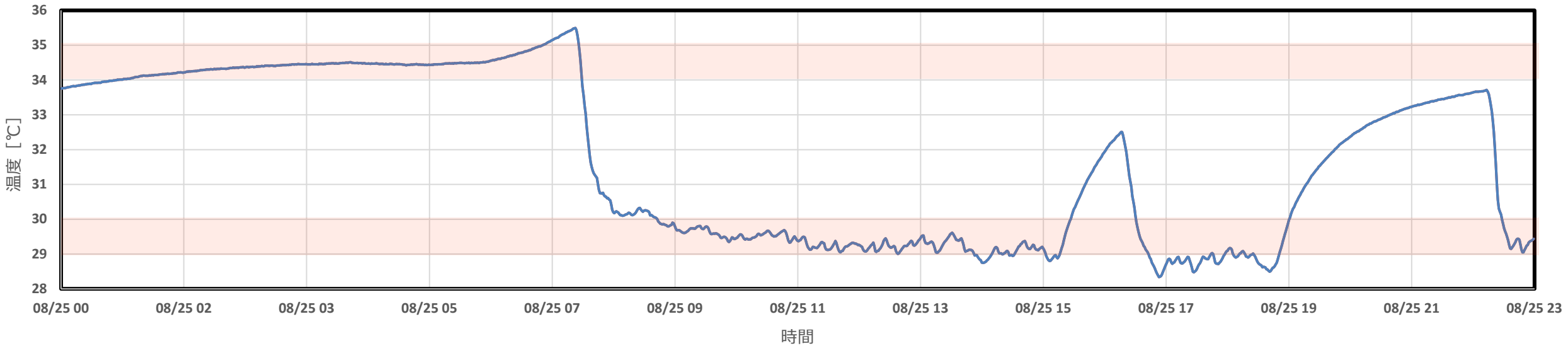
累積分布関数 $F(x)$　※経験分布

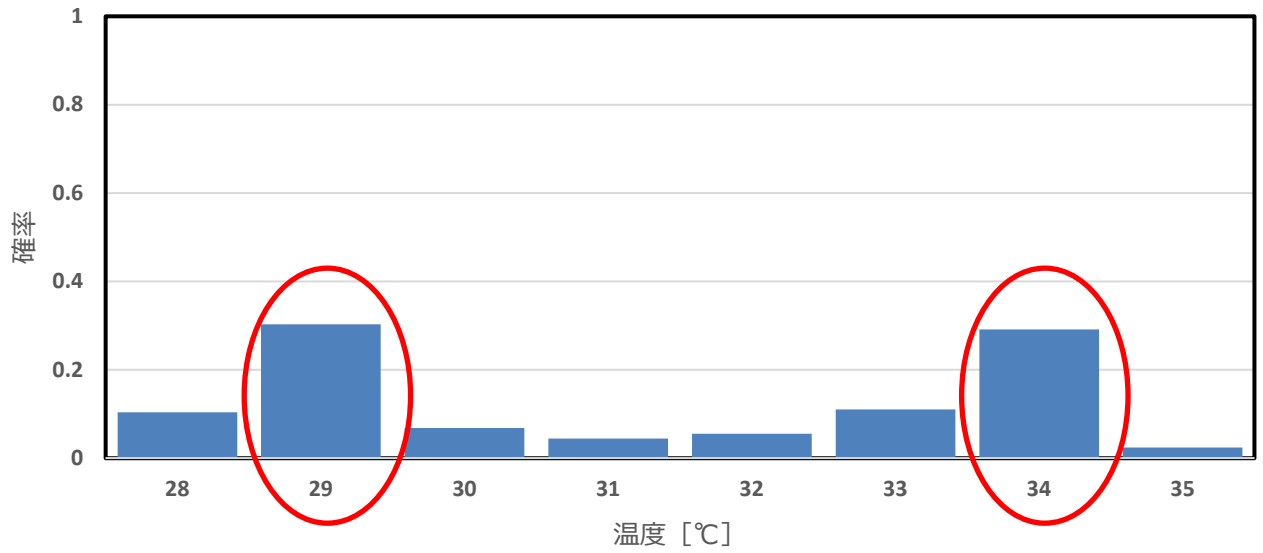


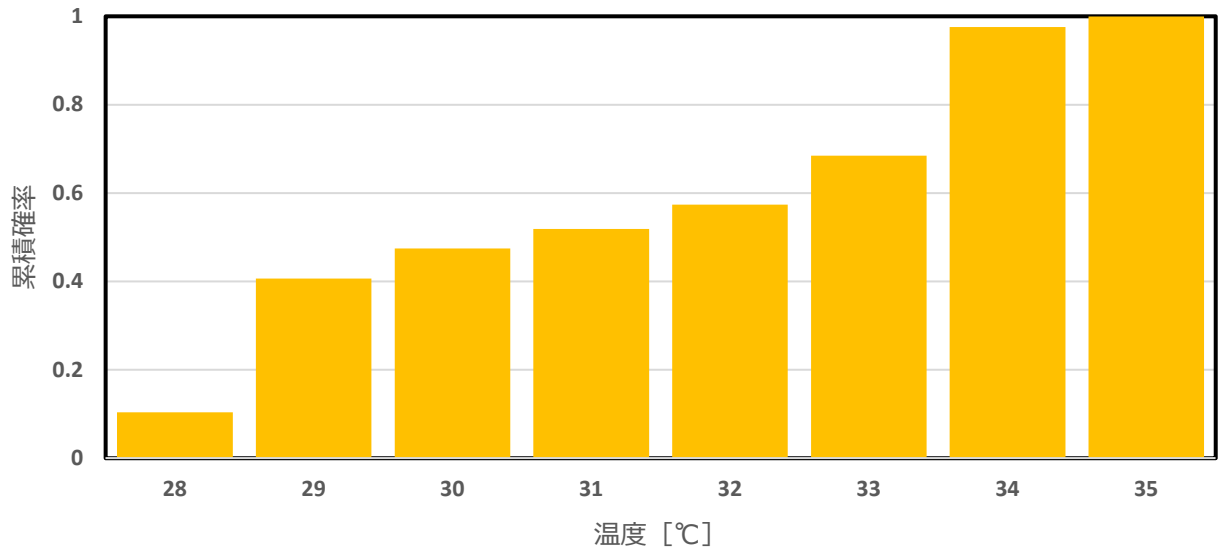
CDFは必ず1に近づく
（PMFの総和は１）

リンゴが３個以下
の確率は6/7

実測値を用いた例


温度［℃］
36
35
34
33
32
31
30
29
28
08/25 00
08/25 02
08/25 03
08/25 05
08/25 07
08/25 09
08/25 11
08/25 13
08/25 15
08/25 17
08/25 19
08/25 21
08/25 23
時間


確率
1
0.8
0.6
0.4
0.2
0
28
29
30
31
32
33
34
35
温度［℃］


累積確率
1
0.8
0.6
0.4
0.2
0
28
29
30
31
32
33
34
35
温度［℃］

確率質量関数のプログラム例

プログラム例（確率質量関数）

```
apple = [3, 1, 2, 4, 1, 2, 1]    #リンゴの収穫量

#出現回数を数える（辞書でカウント）
counts = {}
for x in apple:
    if x in counts:
        counts[x] = counts[x] + 1
    else:
        counts[x] = 1

#総和を１にする（出現回数 / 全体数）
n = 0
for _ in apple:
    n = n + 1

pmf = {}
for key in counts:
    pmf[key] = counts[key] / n

#見やすいようにキーで並べて表示
sorted_keys = sorted(pmf.keys())
print("PMF:")
for k in sorted_keys:
    print(k, ":", pmf[k])
```

結果の例

```
#収穫量
apple = [3, 1, 2, 4, 1, 2, 1]

#結果
PMF:
1 : 0.42857142857142855
2 : 0.2857142857142857
3 : 0.14285714285714285
4 : 0.14285714285714285
```

```
#収穫量
apple = [3, 1, 2, 4, 1, 2, 1, 7, 1, 6]

#結果
PMF:
1 : 0.4
2 : 0.2
3 : 0.1
4 : 0.1
6 : 0.1
7 : 0.1
```

累積分布関数のプログラム例

プログラム例（累積分布関数）

```
apple = [3, 1, 2, 4, 1, 2, 1, 7]    # リンゴの収穫量

# 出現回数を数える（辞書でカウント）
counts = {}
for x in apple:
    if x in counts:
        counts[x] = counts[x] + 1
    else:
        counts[x] = 1

# 総和を１にする（出現回数 / 全体数）
n = 0
for _ in apple:
    n = n + 1

pmf = {}
for key in counts:
    pmf[key] = counts[key] / n

# 累積分布関数（CDF）
cdf = {}
cumulative = 0
for k in sorted(pmf.keys()):
    cumulative = cumulative + pmf[k]
    cdf[k] = cumulative

# 表示
print("CDF:")
for k in sorted(cdf.keys()):
    print(k, ":", cdf[k])
```

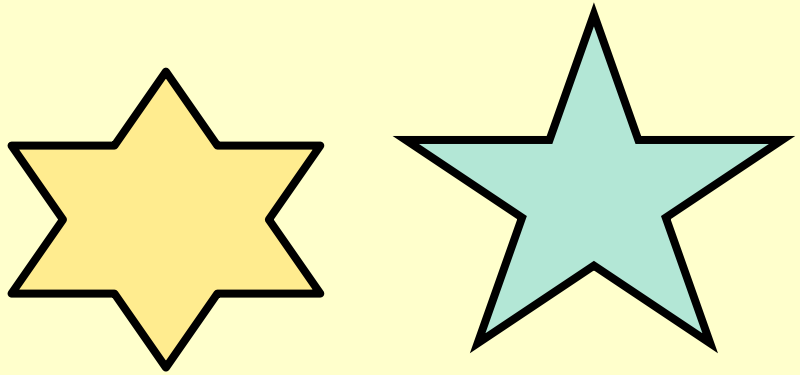
結果の例

```
#収穫量
apple = [3, 1, 2, 4, 1, 2, 1]

#結果
CDF:
1 : 0.42857142857142855
2 : 0.7142857142857142
3 : 0.857142857142857
4 : 0.9999999999999998
```

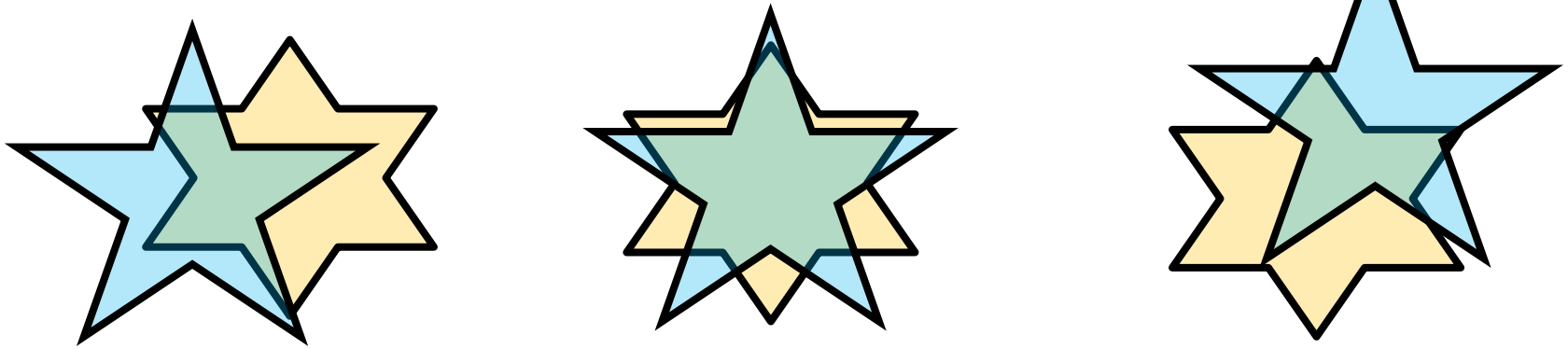
```
#収穫量
apple = [3, 1, 2, 4, 1, 2, 1, 7, 1, 6]

#結果
CDF:
1 : 0.4
2 : 0.6000000000000001
3 : 0.7000000000000001
4 : 0.8
6 : 0.9
7 : 1.0
```

相関関数

問題：以下の２つの類似度（どのくらい似ているか）を定量的に表したい。



例えば、２つを重ね合わせて、その重なり具合をみたらどうだろう。



両者の位置関係によって、重なり具合はかなり違うみたい。
このような、重なり具合を類似度として数値化できれば大変便利！

相互相関関数

$x[n]$と$y[n]$をそれぞれ離散の実信号※1（データ列）とする場合、

$x[n]$と$y[n]$の(正規化)相互相関関数※2は以下の式で与えられる。

$$R_{xy}[k] = \frac{1}{\underbrace{\sqrt{\sum_n x^2[n]}\sqrt{\sum_n y^2[n]}}_{\text{正規化項}}} \sum_{n\in\Omega} x[n]y[n+k]$$

N=|Ω|

$x[n] = y[n], k = 0$ の場合,

$$\frac{1}{\sqrt{\sum_n x^{2[n]}}\sqrt{\sum_n x^{2[n]}}} \sum_n x[n]x[x] = 1$$

つまり、 $x[n]$と$y[n+k]$の内積を求めればよい。

→ 前のページの例に置き換えると、片方の図形を固定して、もう片方を動かしながら図形の重なり具合を計算していることに相当。

※1 $x[n]$と$y[n]$が複素信号の場合は$y[n+k]$の複素共役との内積を求める。

※2 $x[n] = y[n]$の場合、自己相関関数と呼ばれる。信号の周期性や繰り返し構造等を調べるために用いられる。

共分散

以下の正規化相互相関関数では、x[n]とy[n]の平均値の影響を含む

$$R_{xy}[k] = \frac{1}{\sqrt{\sum_{n\in\Omega} x^2[n]}\sqrt{\sum_{n\in\Omega} y^2[n+k]}} \sum_{n\in\Omega} x[n]y[n+k]$$

N=|Ω|

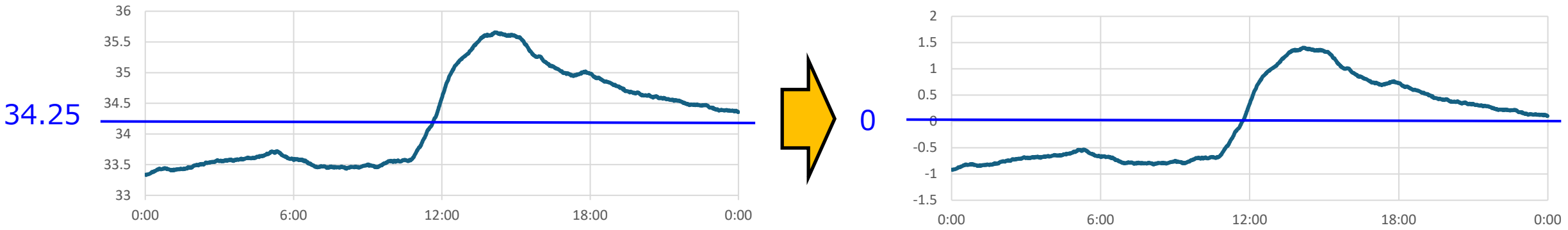
そのため、x[n]とy[n]の平均値の影響を除いた指標として、以下の共分散が用いられる。

x[n]とy[n+k]の共分散

$$C_{xy}[k] = \frac{1}{N}\sum_{n\in\Omega}(x[n]-\bar{x})(y[n+k]-\overline{y_k})$$

$$\bar{x} = \frac{1}{N}\sum_{n\in\Omega} x[n]$$

$$\overline{y_k} = \frac{1}{N}\sum_{n\in\Omega} y[n+k]$$



相関係数

共分散の正規化項の分母をx[n]とy[n+k]の標準偏差として与えた場合、
相関係数※が与えられる。

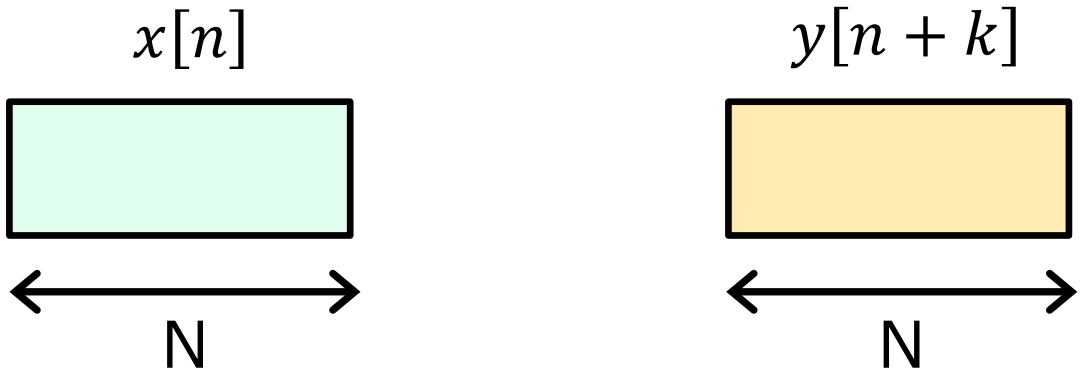
$$\rho_{xy}[k] = \underbrace{\frac{1}{\sqrt{\sum_{n\in\Omega}(x[n]-\bar{x})^2}\sqrt{\sum_{n\in\Omega}(y[n+k]-\overline{y_k})^2}}}_{\text{正規化項}} \sum_{n\in\Omega}(x[n]-\bar{x})(y[n+k]-\overline{y_k})$$

$N=|\Omega|$

※ピアソンの積率相関係数とも呼ばれる

相互相関関数

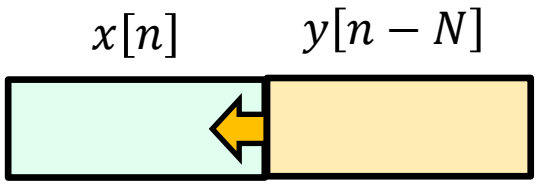
試しに計算してみよう。例えば、$x[n]$と$y[n+k]$を以下のような図形とする。



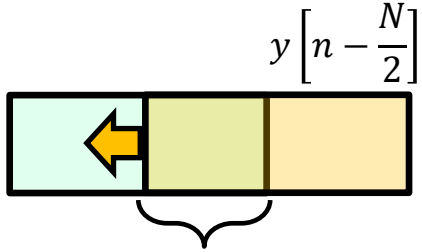
この２つの類似度を計算したい。片方の図形を固定して、もう片方を移動させながら重なり具合を求めるとよい。

相互相関関数

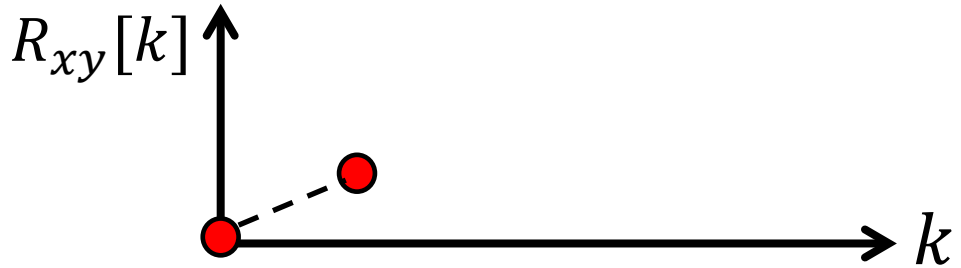
$k = -N$



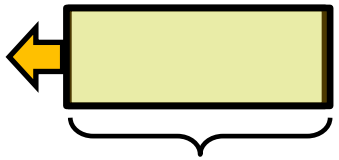
$k = -\frac{N}{2}$



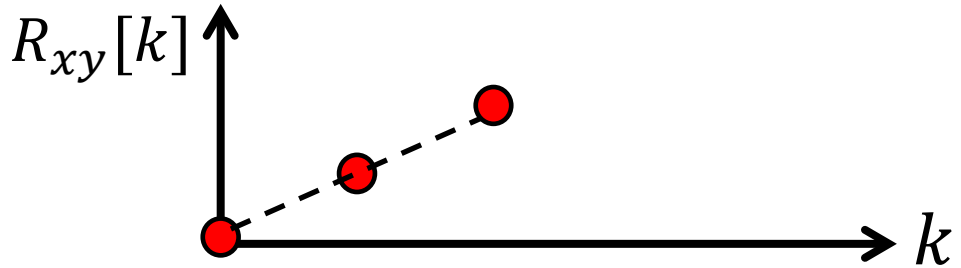
この重なっている部分が$R_{xy}[\frac{N}{2}]$



$k = 0$

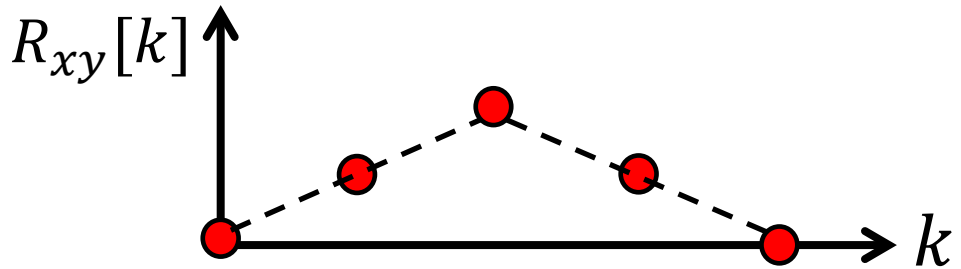


完全一致（最大値をとる）



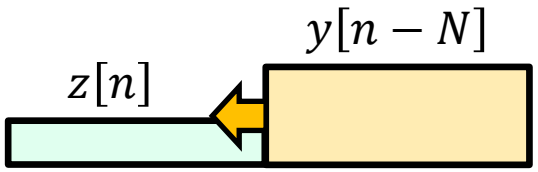
$k = N$

重なっている部分がない（$R_{xy}[N] = 0$）

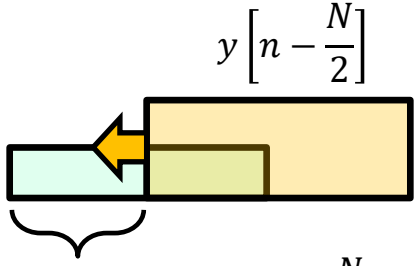


相互相関関数

$k = -N$

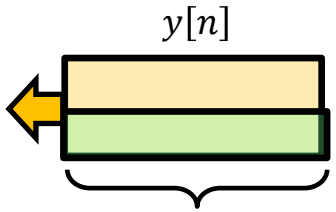


$k = -\frac{N}{2}$



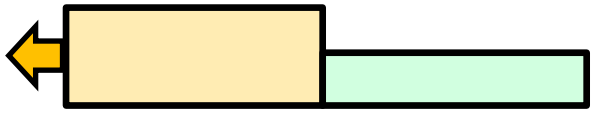
この重なっている部分が$R_{yz}[\frac{N}{2}]$

$k = 0$

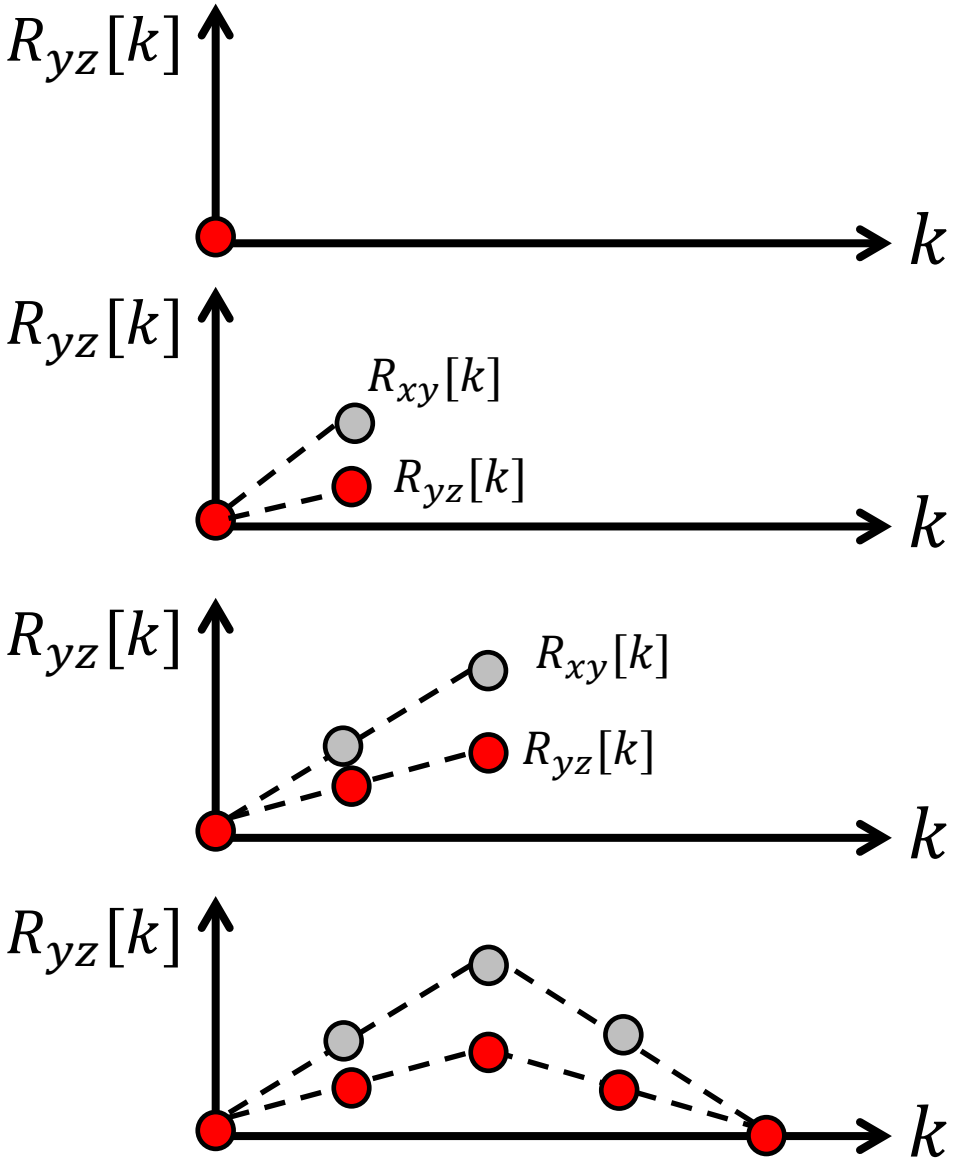


$R_{yz}[0]$で最大値をとる、でも$R_{xy}[0]$よりも小さい

$k = N$



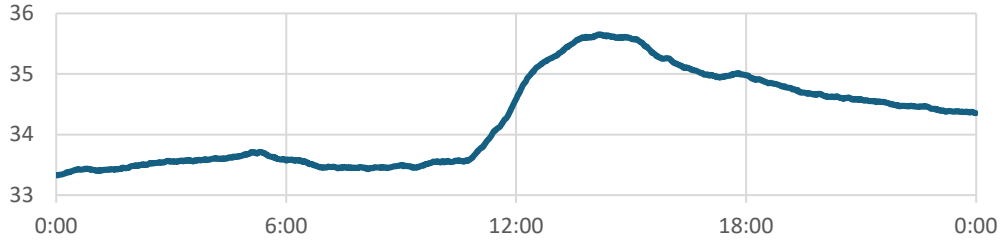
重なっている部分がない（$R_{yz}[N] = 0$）



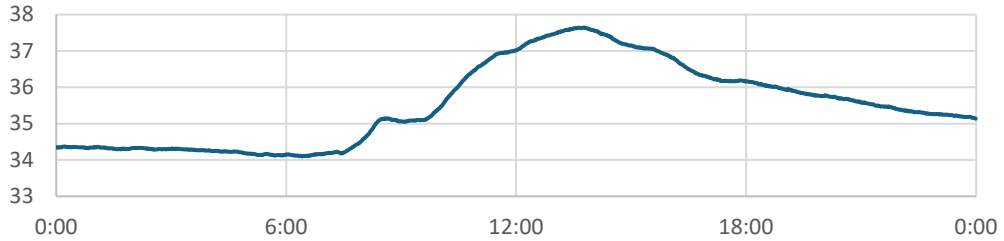
実測値を用いた相互相関関数の計算例

室内の温度　(本演習のハードウェアで計測)

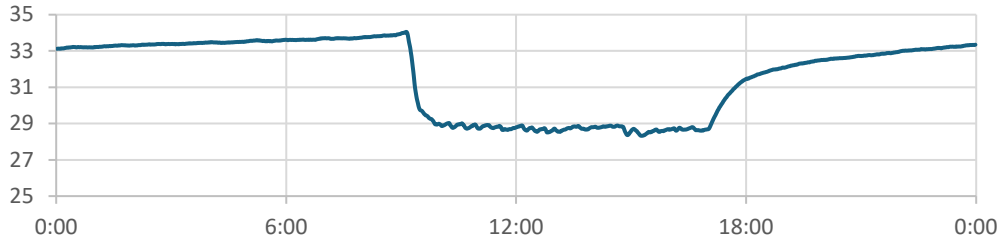
- 9月14日(土曜日)の温度変化 $x[n]$



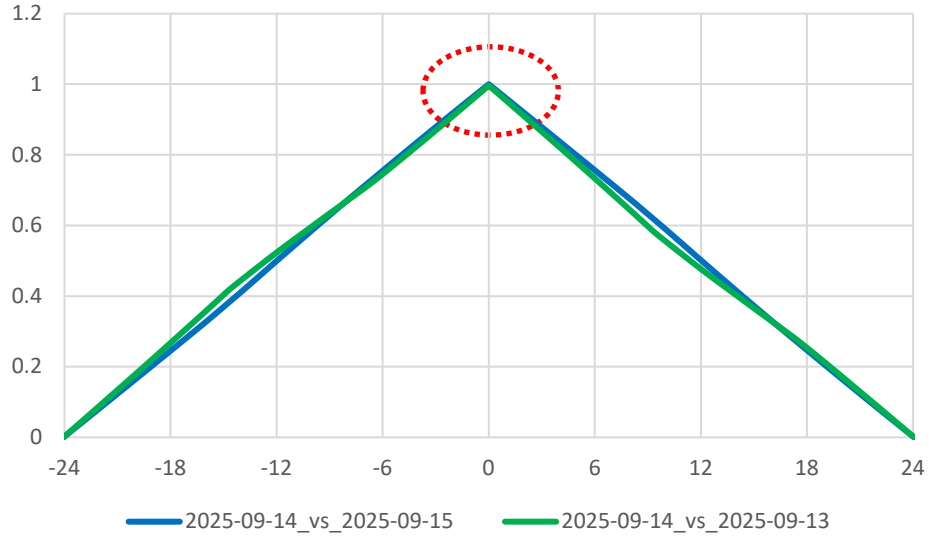
- 9月15日(日曜日)の温度変化 $y[n]$



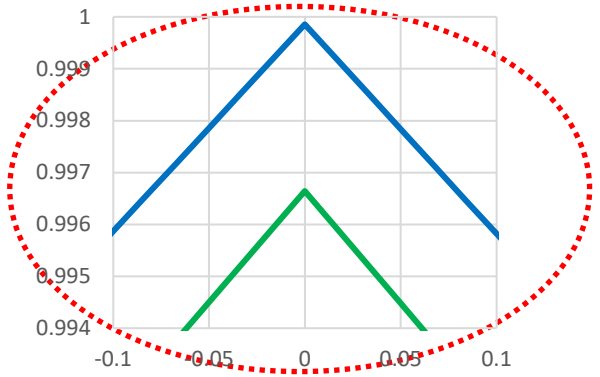
- 9月13日(金曜日)の温度変化 $z[n]$



- 相関関数の計算結果($R_{xy}[k]$, $R_{xz}[k]$)



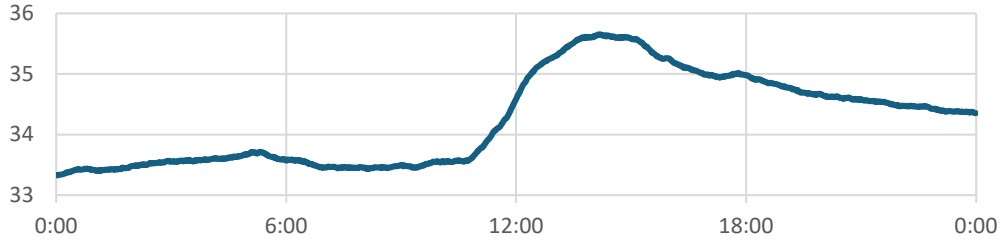
拡大図



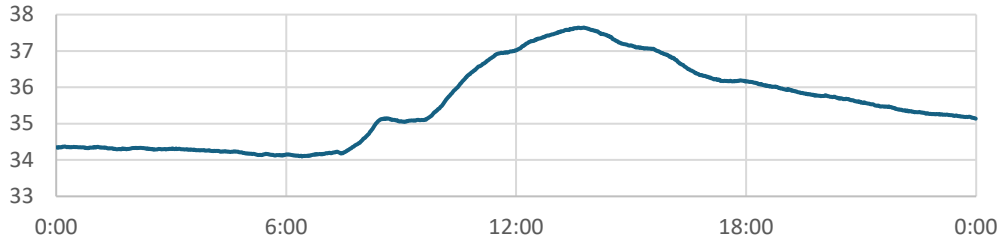
実測値を用いた相関係数の計算例

室内の温度　(本演習のハードウェアで計測)

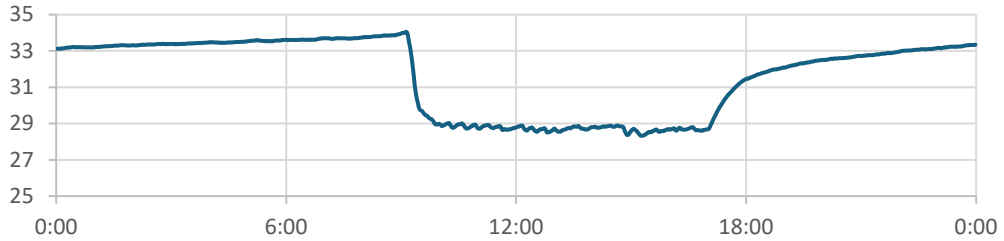
- 9月14日(土曜日)の温度変化 $x[n]$



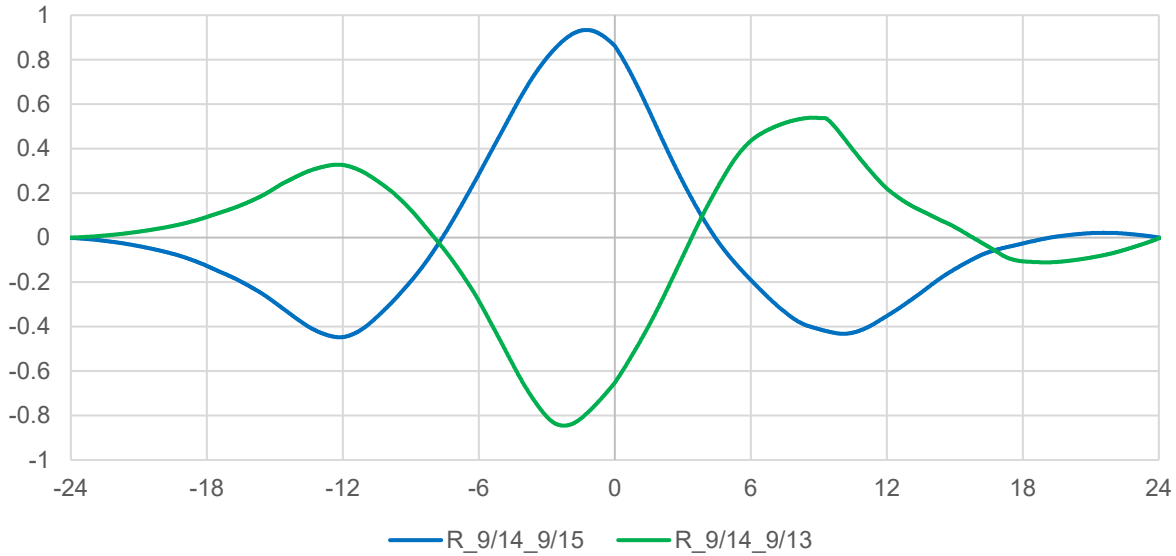
- 9月15日(日曜日)の温度変化 $y[n]$



- 9月13日(金曜日)の温度変化 $z[n]$



- 相関関数の計算結果($R_{xy}[k]$, $R_{xz}[k]$)

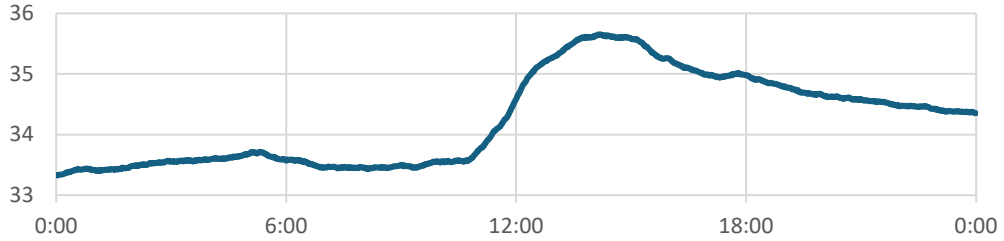


- $x[n]$と$y[n]$の平均値がそれぞれ0となるように前処理を行った場合：

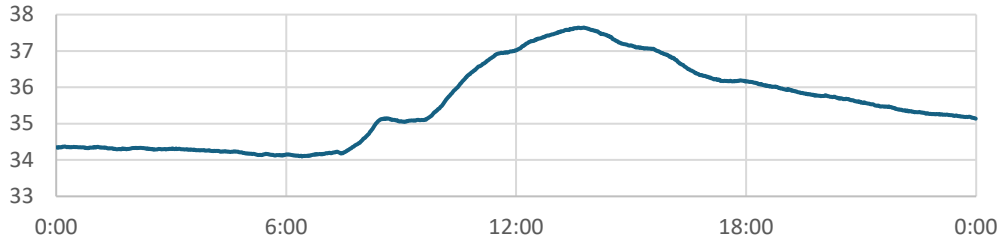
実測値を用いた相関関数の計算例

室内の温度　(本演習のハードウェアで計測)

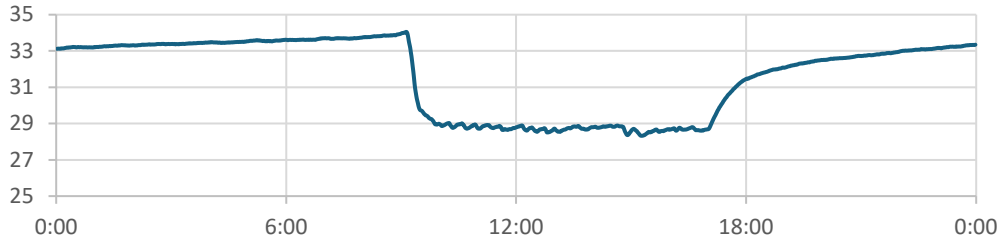
- 9月14日(土曜日)の温度変化 $x[n]$



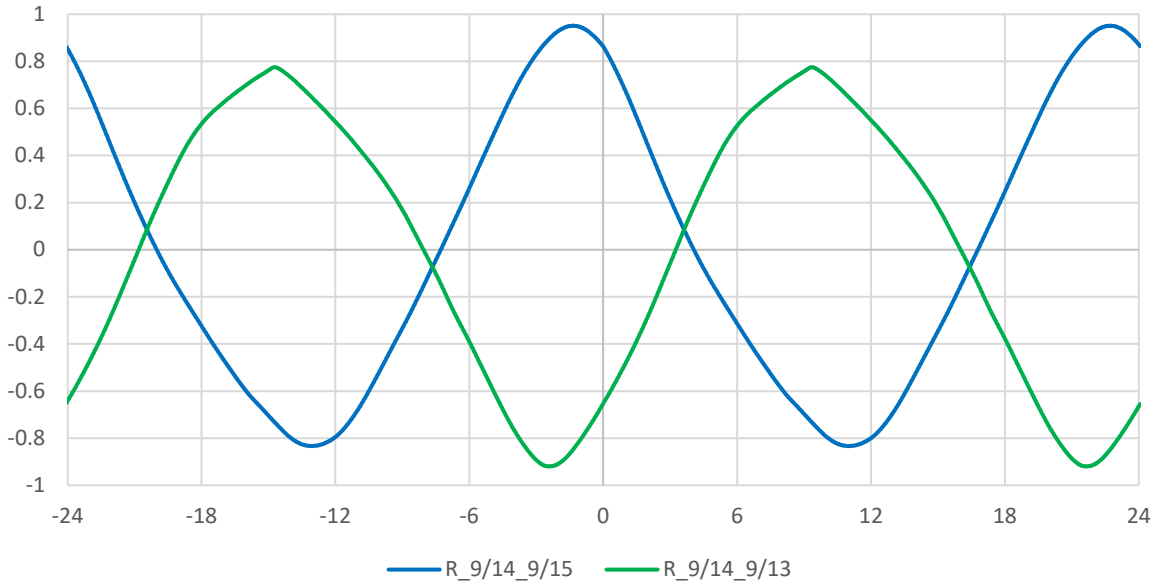
- 9月15日(日曜日)の温度変化 $y[n]$



- 9月13日(金曜日)の温度変化 $z[n]$



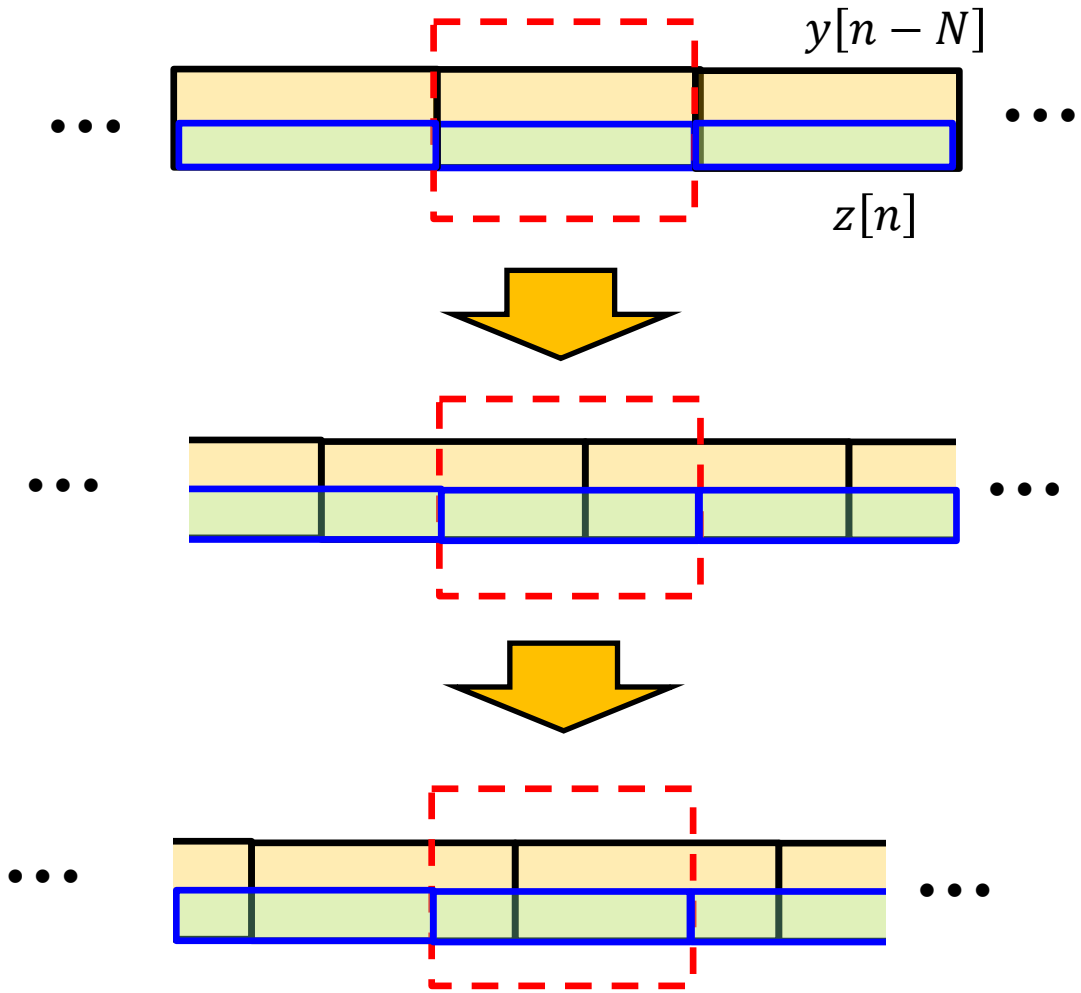
- 相関関数の計算結果($R_{xy}[k]$, $R_{xz}[k]$)



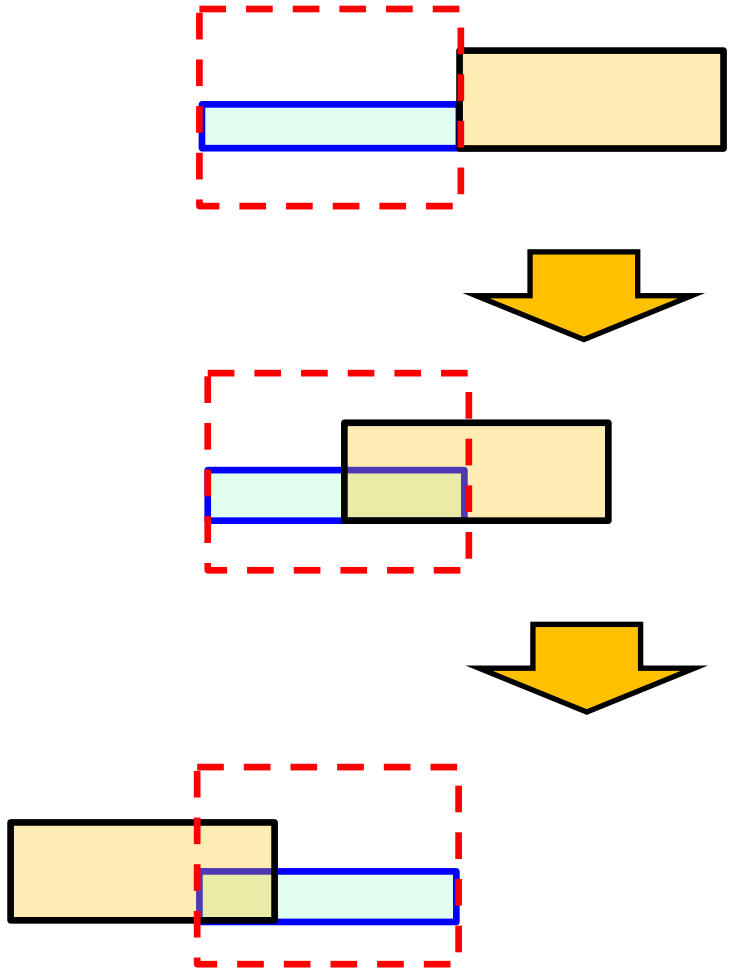
- $x[n]$と$y[n]$の平均値がそれぞれ0となるように前処理
- $x[n]$と$y[n]$を周期関数と見なして巡回処理を行った場合

補足：巡回処理

巡回シフト有り
（信号の周期性を仮定する場合）



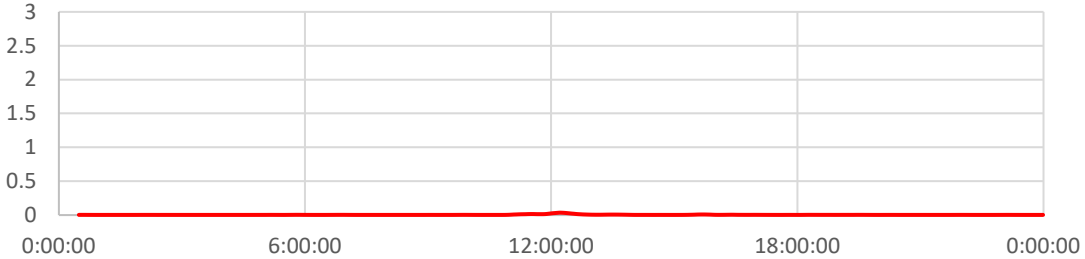
巡回シフト無し
（孤立パルスを仮定する場合）



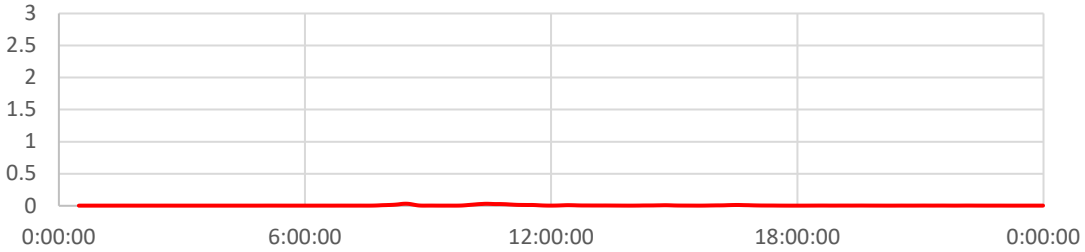
実測値を用いた相互相関関数の計算例

室内の温度の移動分散 (本演習のハードウェアで計測) (移動平均長＝30分)

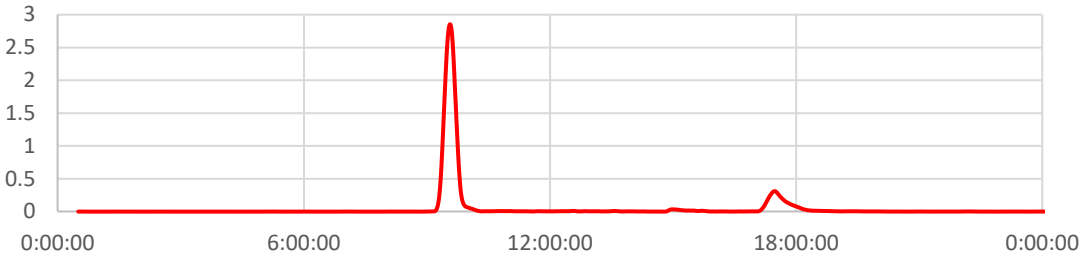
- 9月14日(土曜日)の移動分散$x[n]$



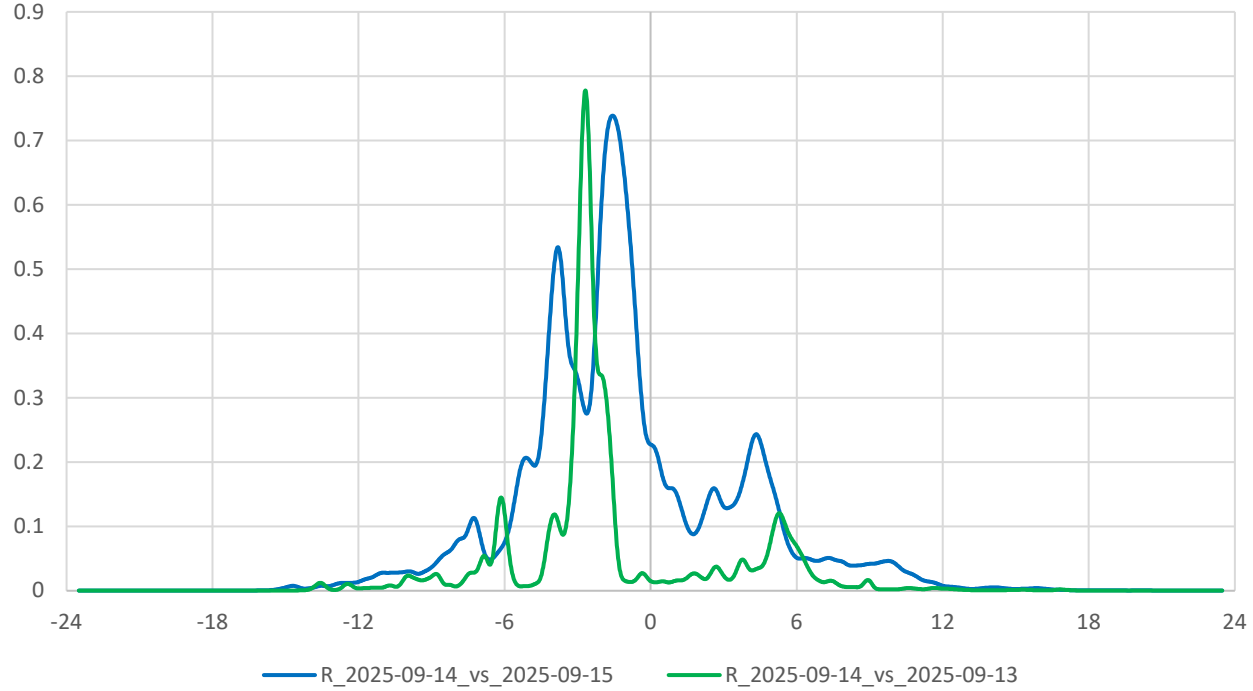
- 9月15日(日曜日)の移動分散y$[n]$



- 9月13日(金曜日)の移動分散z$[n]$



- 相関関数の計算結果($R_{xy}[k]$, $R_{xz}[k]$)



※ 信号の巡回処理は行わない

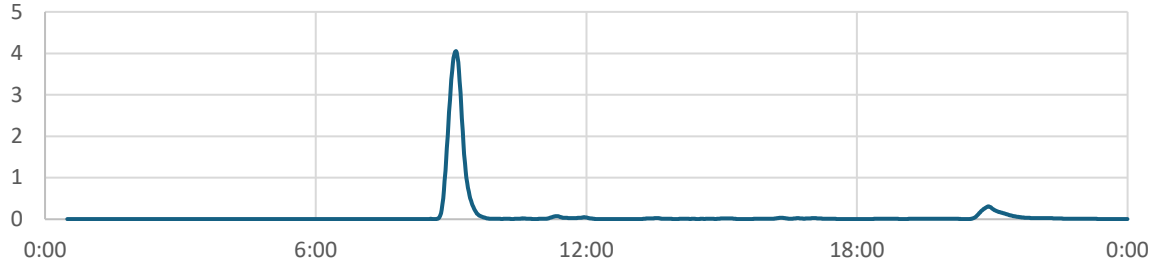
実測値を用いた相関関数の計算例

$$R_{xy}[k] = \frac{1}{\sqrt{\sum_n x^2[n]}\sqrt{\sum_n y^2[n]}} \sum_n x[n]y[n+k]$$

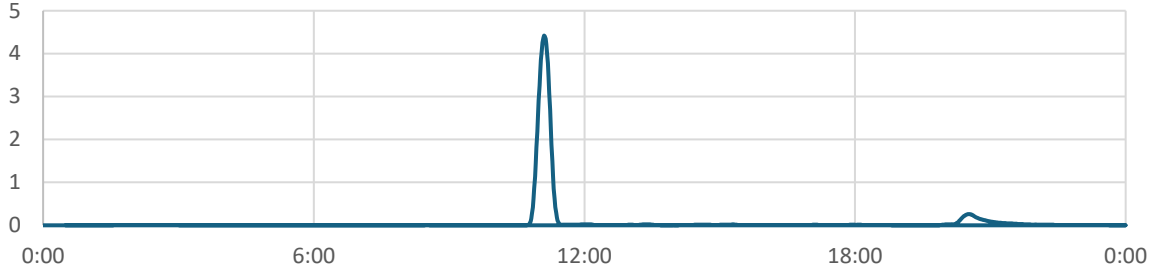
・データ循環無し

室内の温度の移動分散 (本演習のハードウェアで計測) (移動平均長＝30分)

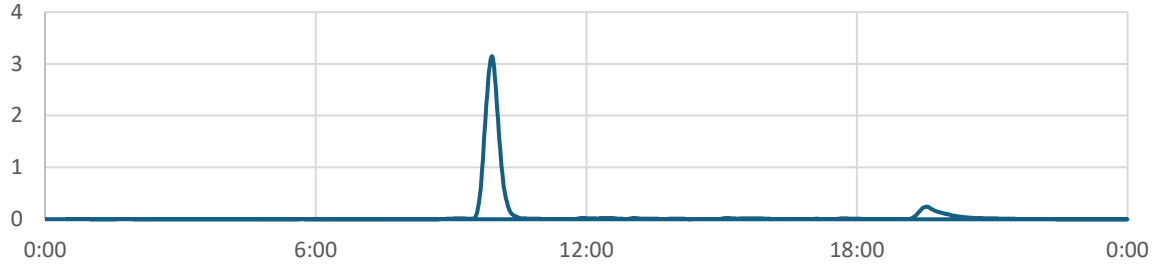
- 9月 8日(月曜日)の移動分散$x[n]$



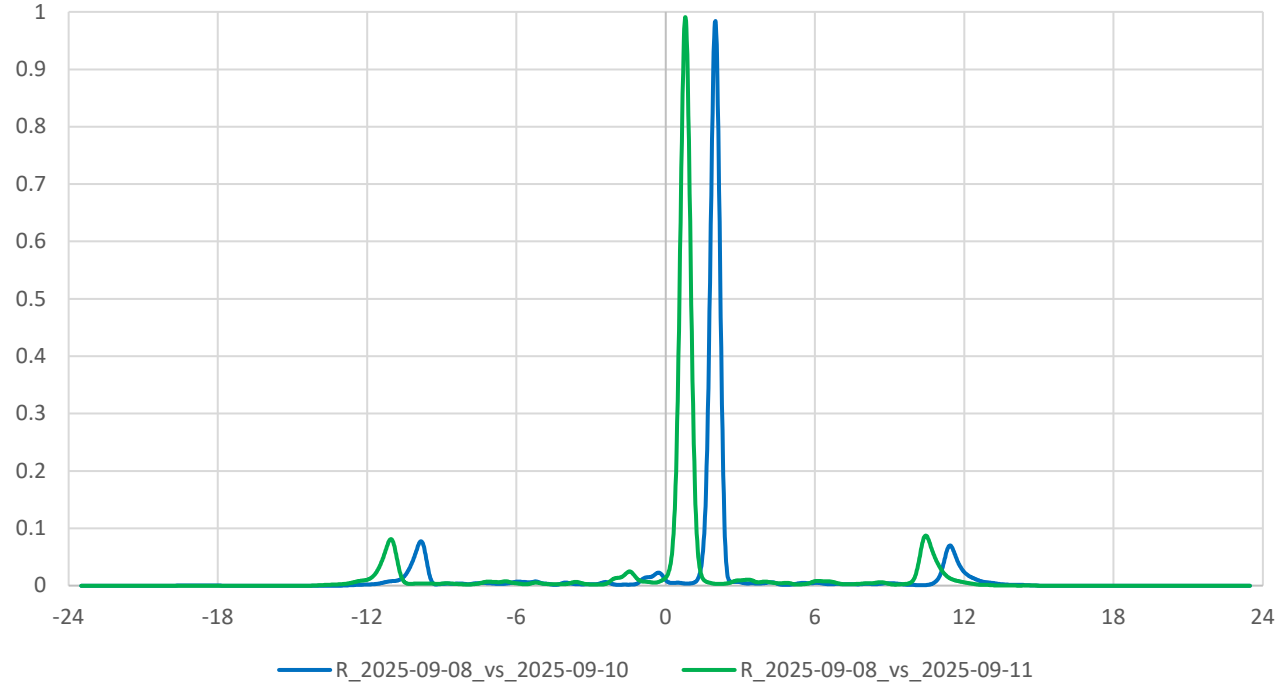
- 9月10日(水曜日)の移動分散y$[n]$



- 9月11日(木曜日)の移動分散z$[n]$



- 相関関数の計算結果($R_{xy}[k]$, $R_{xz}[k]$)



空調の動作開始時間（OFF→ONの時刻)を移動分散の変化として観測できる。

相互相関関数計算のプログラム例

プログラム例（相互相関関数）

```
# データ
x = [1, 0, 0]
y = [0, 1, 0]

#データの2乗和を計算
square_sum_x = 0
square_sum_y = 0
for i in range (len(x)):
  square_sum_x = square_sum_x + x[i]**2
  square_sum_y = square_sum_y + y[i]**2
#データを正規化
for i in range (len(x)):
  x[i] = x[i] / (square_sum_x ** 0.5)
  y[i] = y[i] / (square_sum_y ** 0.5)

#xとyのリストの長さを計算
n = len(x)
m = len(y)

result = [] #結果を入れるリスト

# シフト量 k を -n+1 からm まで変更
for k in range(-n+1, m):
    s = 0
    for i in range(n):
        j = i + k
        if 0 <= j < m:    # b の値が存在するときだけ掛け算
            s += x[i] * y[j]
    result.append(s)

#結果を四捨五入して、小数点以下3桁にする
rounded = [round(x, 3) for x in result]
print(rounded)
```

結果の例

```
# データ
x = [1, 0, 0]
y = [0, 1, 0]
# 結果
[0.0, 0.0, 0.0, 1.0, 0.0]
```

```
# データ
x = [1, 2, 3, 1]
y = [2, 3, 1, 1]
# 結果
[0.133, 0.6, 0.933, 0.8, 0.533, 0.2, 0.067]
```

$$R_{xy}[k] = \frac{1}{\sqrt{\sum_n x^2[n]}\sqrt{\sum_n y^2[n]}} \sum_n x[n]y[n+k]$$

相関係数計算のプログラム例

プログラム例

```
# データ
x = [1, 0, 0]
y = [0, 1, 0]

#平均を計算
ave_x = sum(x) / len(x)
ave_y = sum(y) / len(y)
#元データから平均を引いて、平均値を0にする
for i in range (len(x)):
  x[i] = x[i] - ave_x
  y[i] = y[i] - ave_y

#データの2乗和を計算
square_sum_x = 0
square_sum_y = 0
for i in range (len(x)):
  square_sum_x = square_sum_x + x[i]**2
  square_sum_y = square_sum_y + y[i]**2
#データを正規化
for i in range (len(x)):
  x[i] = x[i] / (square_sum_x ** 0.5)
  y[i] = y[i] / (square_sum_y ** 0.5)

#xとyのリストの長さを計算
n = len(x)
m = len(y)

result = [] #結果を入れるリスト

# シフト量 k を -n+1 からm まで動かす()
for k in range(-n+1, m):
    s = 0
    for i in range(n):
        j = i + k
        if 0 <= j < m:    # b の値が存在するときだけ掛け算
            s += x[i] * y[j]
    result.append(s)

#結果を四捨五入して、小数点以下3桁にする
rounded = [round(x, 3) for x in result]
print(rounded)
```

結果の例

```
# データ
x = [1, 0, 0]
y = [0, 1, 0]
# 結果
[0.167, -0.167, -0.5, 0.833, -0.333]
```

```
# データ
X = [1, 2, 3, 1]
y = [2, 3, 1, 1]
# 結果
[-0.068, -0.227, 0.795, -0.091, -0.75, 0.136, 0.205]
```

相関係数計算のプログラム例

プログラム例　巡回シフトを仮定した場合

```
x = [1, 0, 0]
y = [0, 1, 0]

#平均を計算
ave_x = sum(x) / len(x)
ave_y = sum(y) / len(y)
#元データから平均を引いて、平均値を0にする
for i in range (len(x)):
  x[i] = x[i] - ave_x
  y[i] = y[i] - ave_y

#データの2乗和を計算
square_sum_x = 0
square_sum_y = 0
for i in range (len(x)):
  square_sum_x = square_sum_x + x[i]**2
  square_sum_y = square_sum_y + y[i]**2
#データを正規化
for i in range (len(x)):
  x[i] = x[i] / (square_sum_x ** 0.5)
  y[i] = y[i] / (square_sum_y ** 0.5)

#aとbのリストの長さを計算
n = len(x)
m = len(y)

result = [] #結果を入れるリスト

# シフト量 k を -n+1 からm まで動かす()
for k in range(-n+1, m):
    s = 0
    for i in range(n):
        j = (i + k) % n
        s += x[i] * y[j]
    result.append(s)

#結果を四捨五入して、小数点以下3桁にする
rounded = [round(x, 3) for x in result]
print(rounded)
```

結果の例

```
# データ
x = [1, 0, 0]
y = [0, 1, 0]
# 結果
[1.0, -0.5, -0.5, 1.0, -0.5]
```

```
# データ
x = [1, 2, 3, 1]
y = [2, 3, 1, 1]
# 結果
[-0.818, -0.091, 1.0, -0.091, -0.818, -0.091, 1.0]
```

課題

- 任意に数種類のデータ系列（例えば、乱数列、サイコロの出目、点数など）を仮定し、それらの平均と分散を計算して比較するプログラムを作成してください。
- さらに、それぞれのデータ系列について以下を求めてください：
  - ✓ ヒストグラムを正規化して近似的な PDF（確率密度関数）を描画する。
  - ✓ CDF（累積分布関数）を計算して描画する。
  - ✓ 移動平均・移動分散を求めてプロットし、データの変化を確認する。

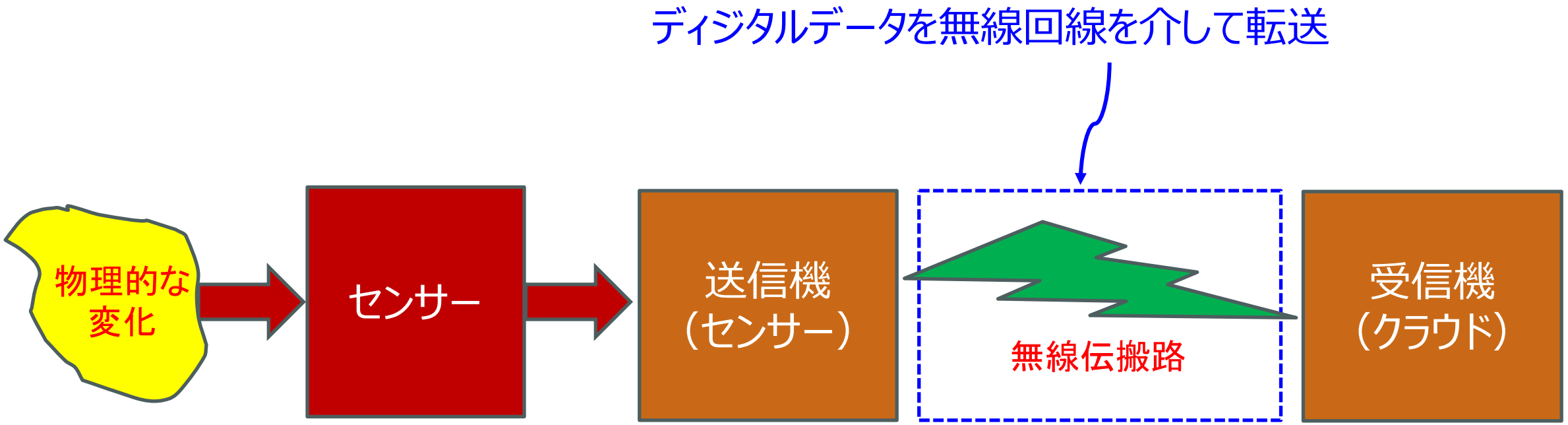
無線通信を用いたデータ伝送の例

IoTシステムにおける
通信シミュレーションのプログラミング例
(上級者向け)

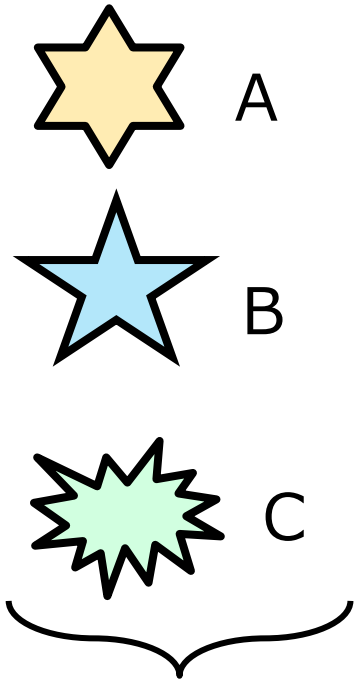
無線通信を用いたデータ伝送の例

センサーを用いて計測したデータをクラウドに集約する場合を考える

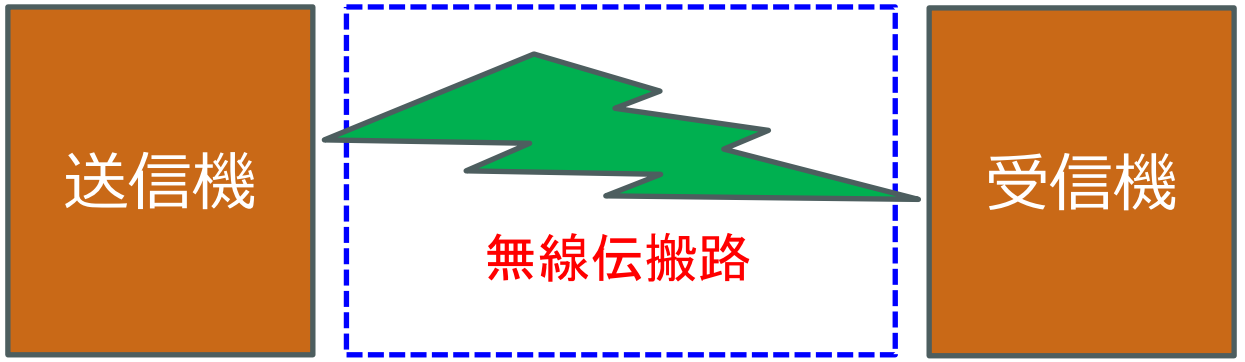
- 本演習におけるIoTキットでは、無線通信（WiFi等）を用いてデータを転送



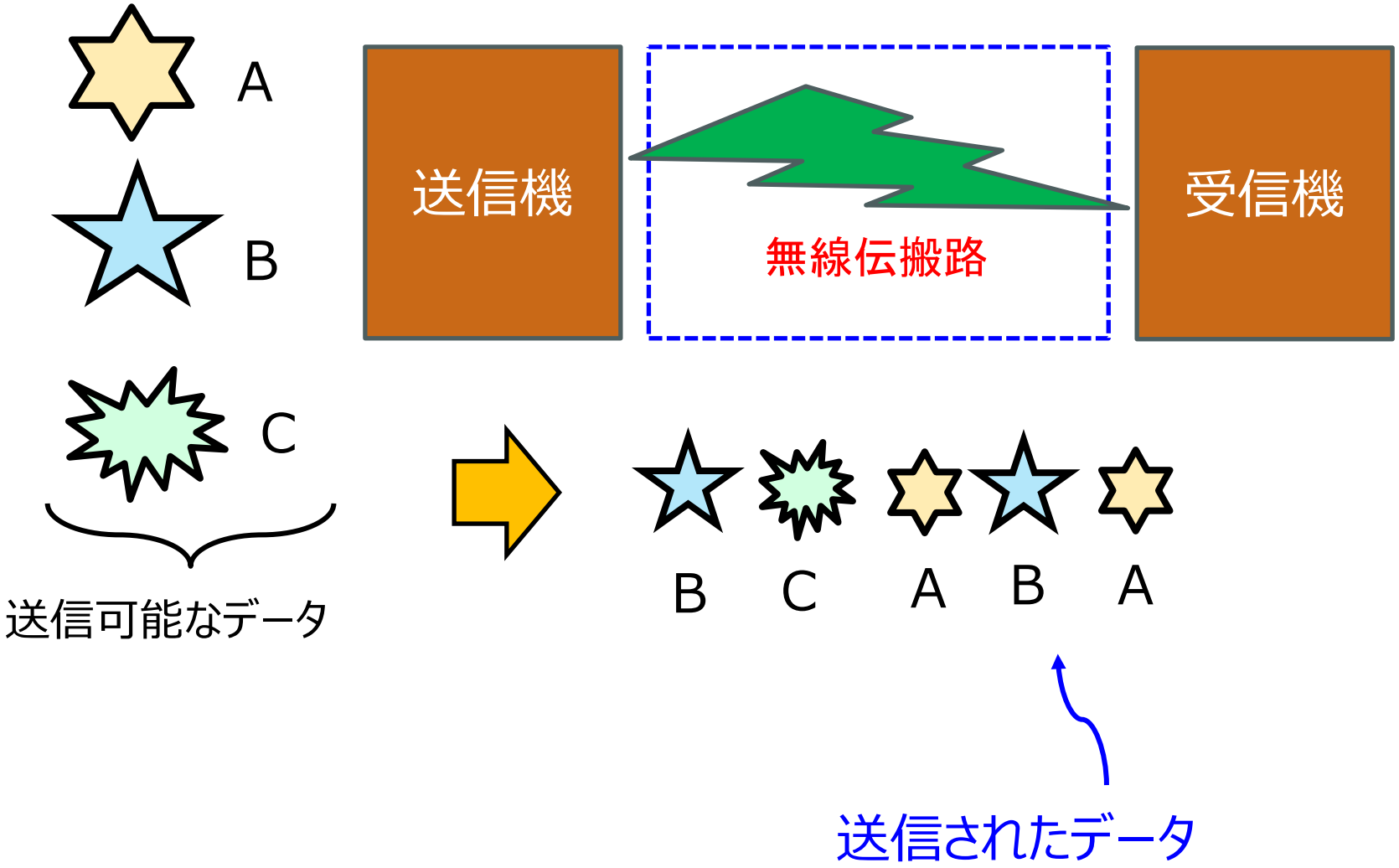
無線通信を用いたデータ伝送の例



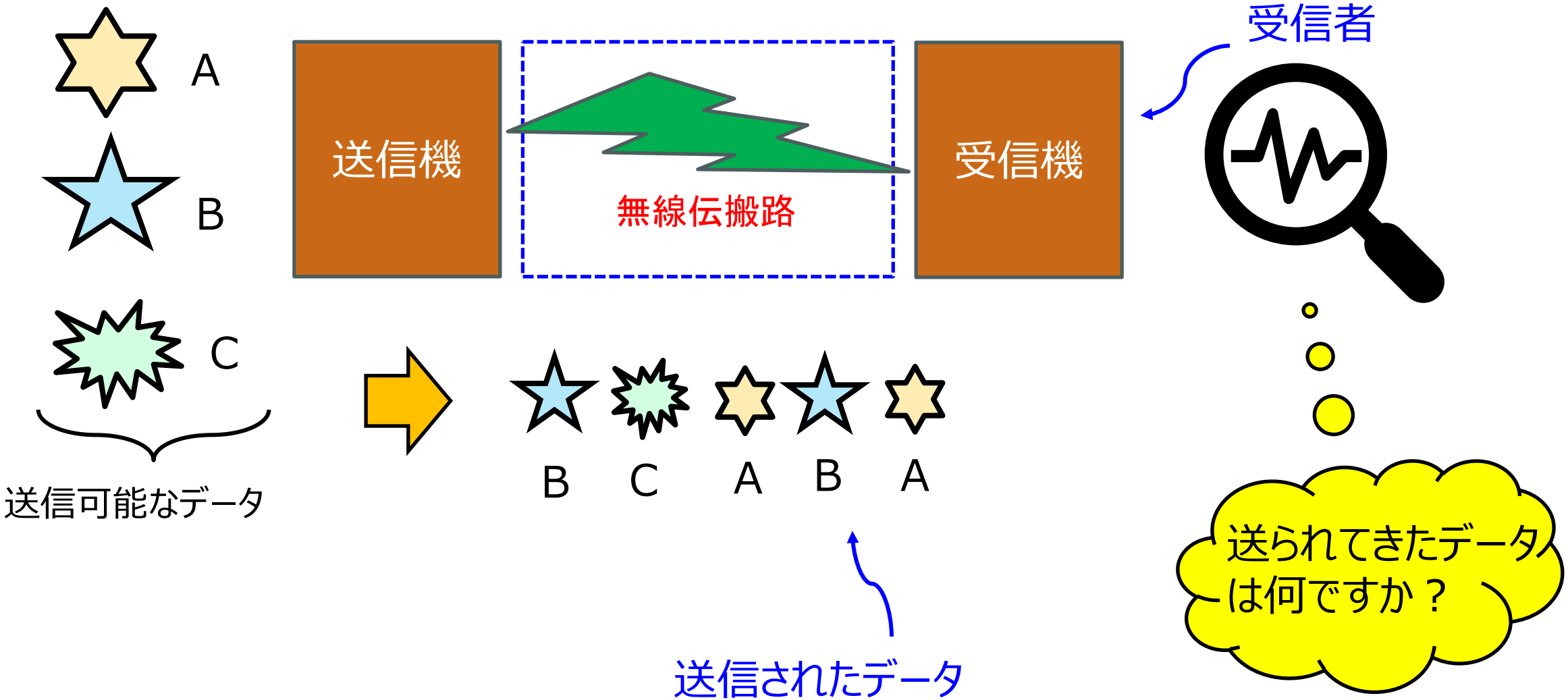
送信可能なデータ



無線通信を用いたデータ伝送の例

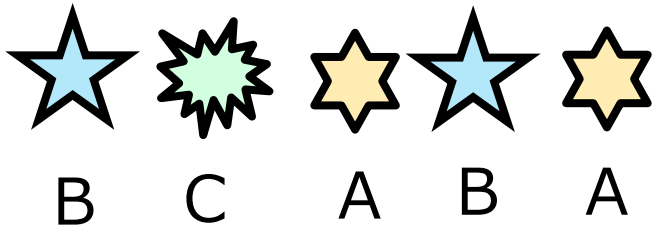


無線通信を用いたデータ伝送の例

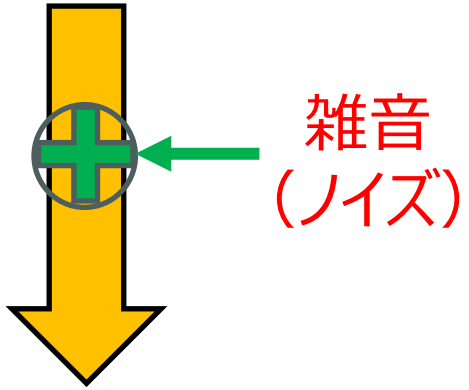


無線通信を用いたデータ伝送の例

送信したデータ

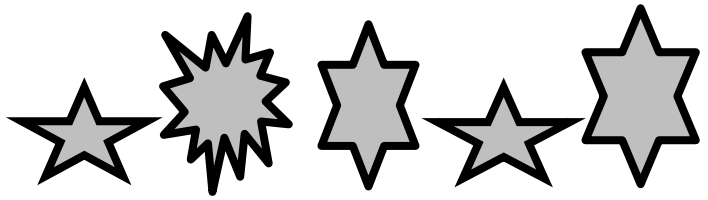


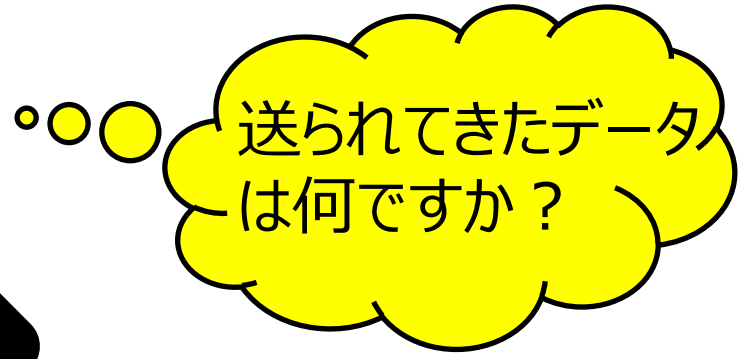
B C A B A



雑音
（ノイズ）

雑音を含む
観測データ

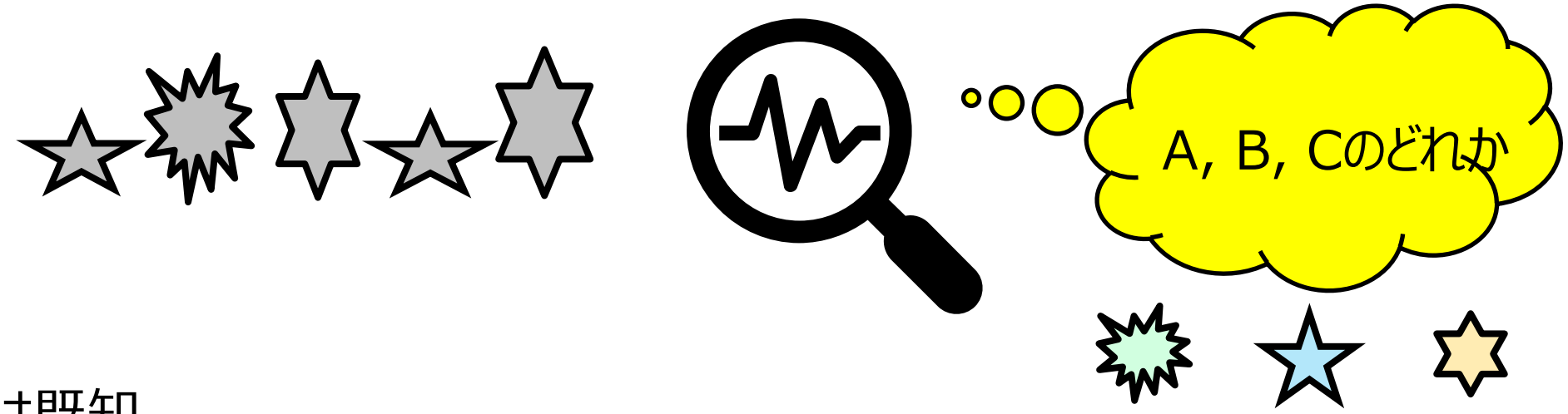




送られてきたデータは何ですか？

無線通信を用いたデータ伝送の例

雑音を含む
観測データ
（受信波形）



A, B, C の波形は既知。

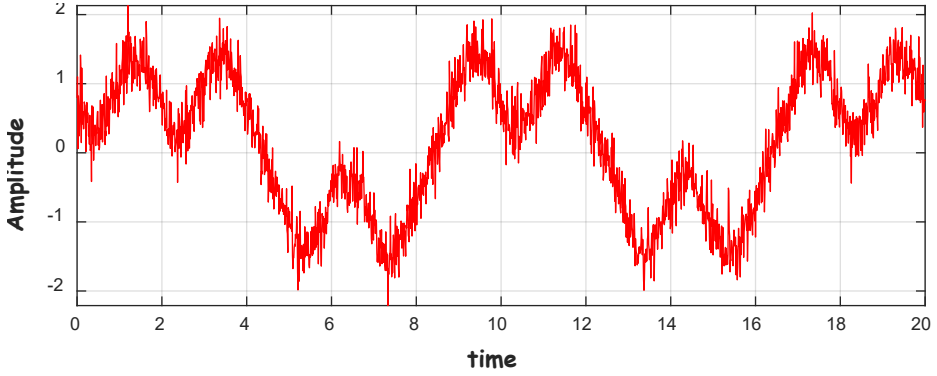
したがって、A, B, Cのうち一番似ているものとみなせばよい。
➔ A, B, Cの波形と受信波形の相関を計算し、その値が最大のものとみなせばよい。

通信システムの受信機の演算 ➔ 相関演算といってもよい
（現在のWiFiや携帯電話システムにおいても基本原理は同じ）

無線通信を用いたデータ伝送の例

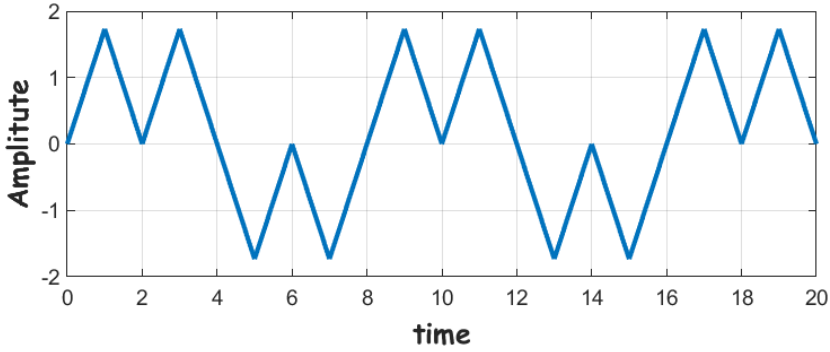
もう少し具体的な例をあげてみよう。

雑音を含む受信データ

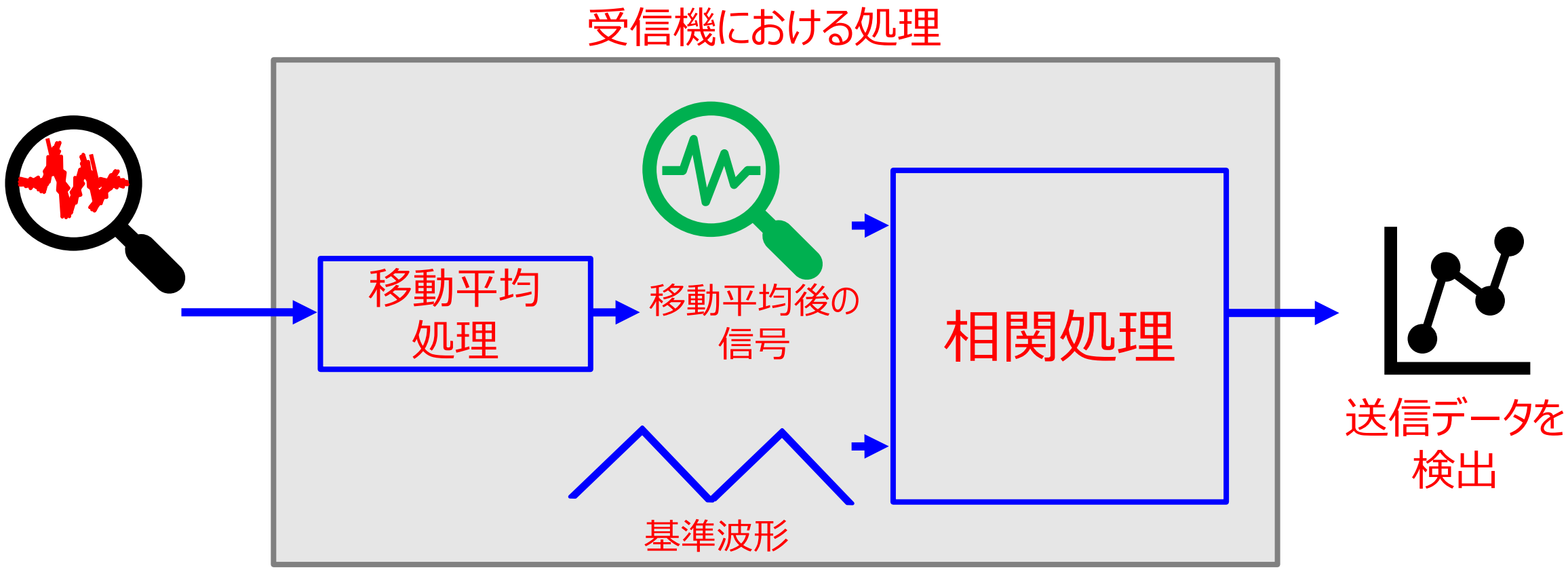


受信処理

受信機



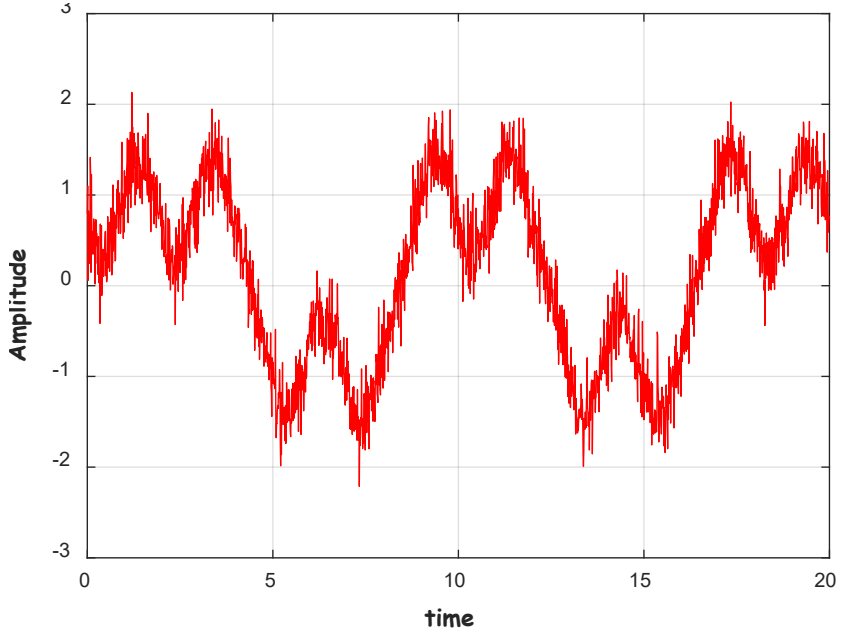
無線通信を用いたデータ伝送の例



※ 相関演算は雑音抑圧や平滑化の効果を有しますが、ここではプログラミング演習の観点からあえて個別に実装しています。
実際の受信機における演算処理については専門書を参照ください。

無線通信を用いたデータ伝送の例

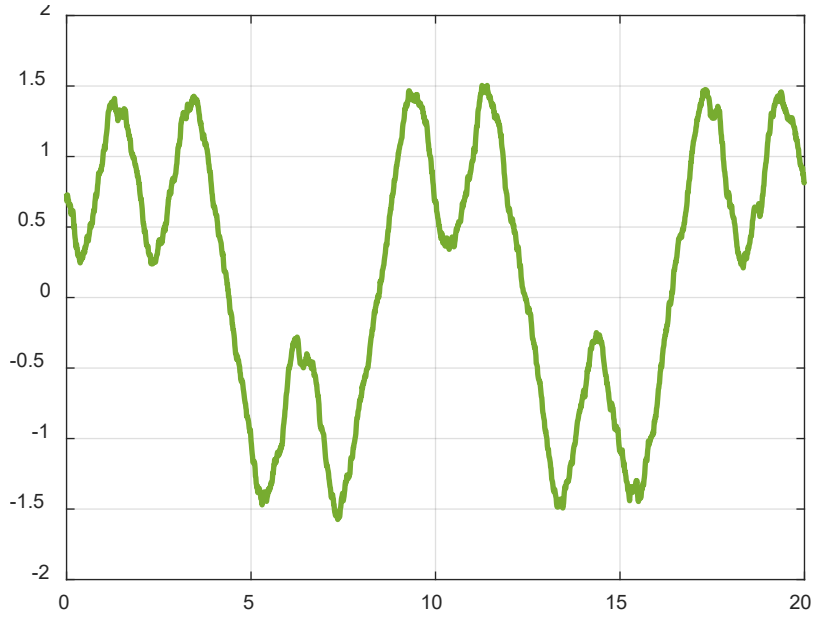
雑音を含む受信データ



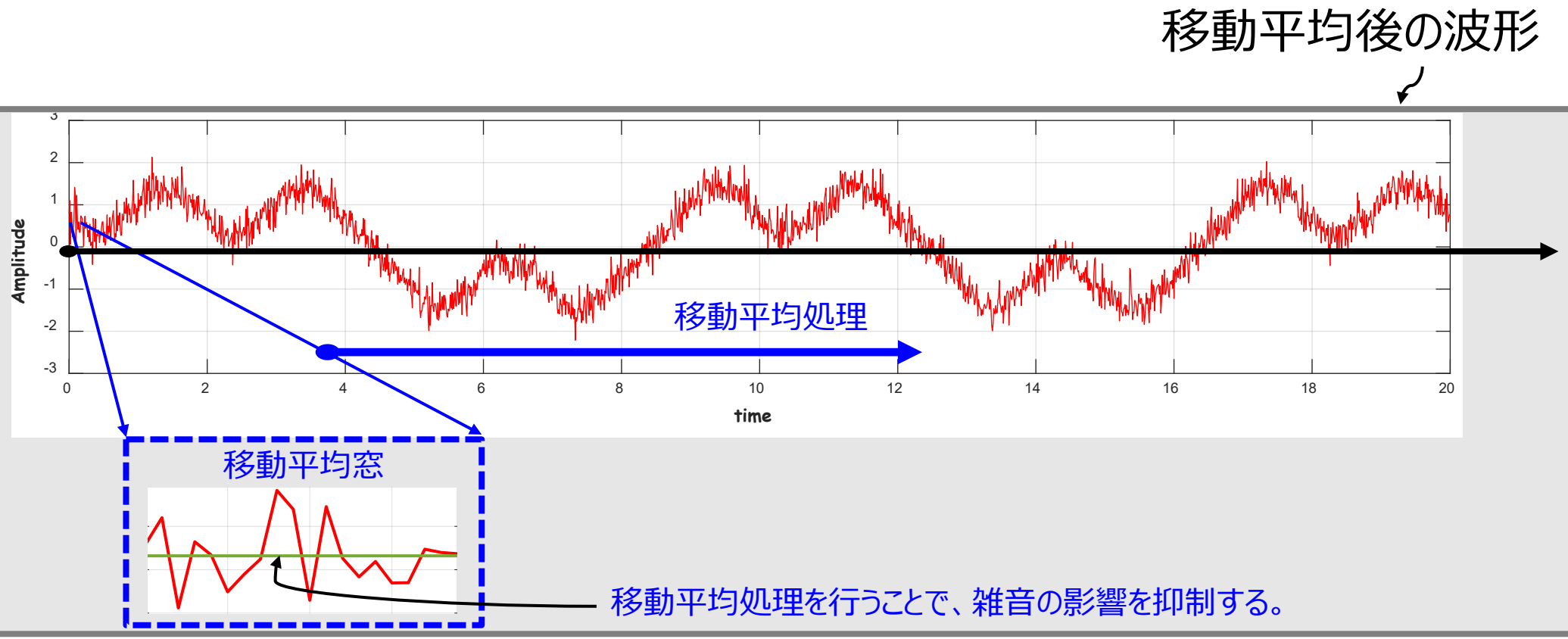
移動平均

どのような動作をしているのでしょうか？

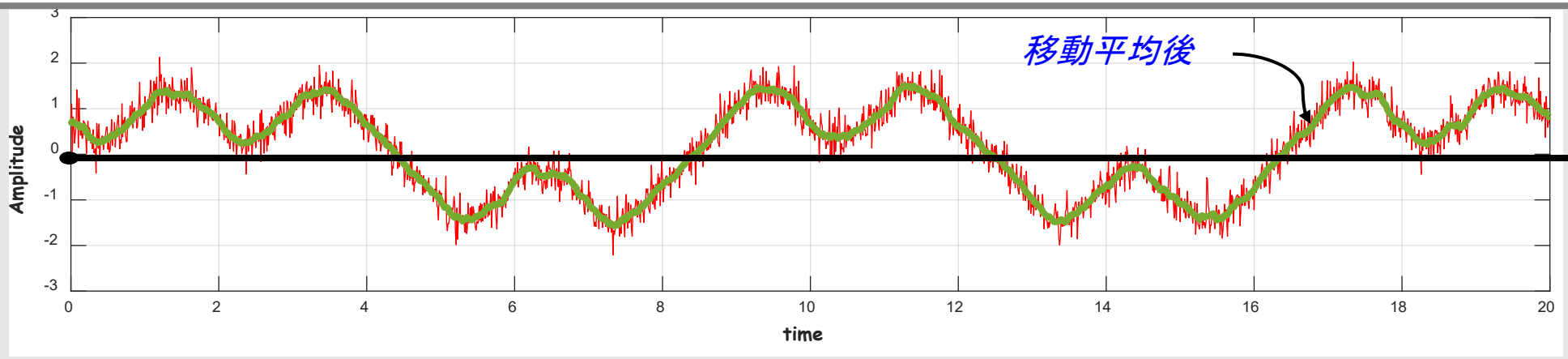
移動平均後



無線通信を用いたデータ伝送の例



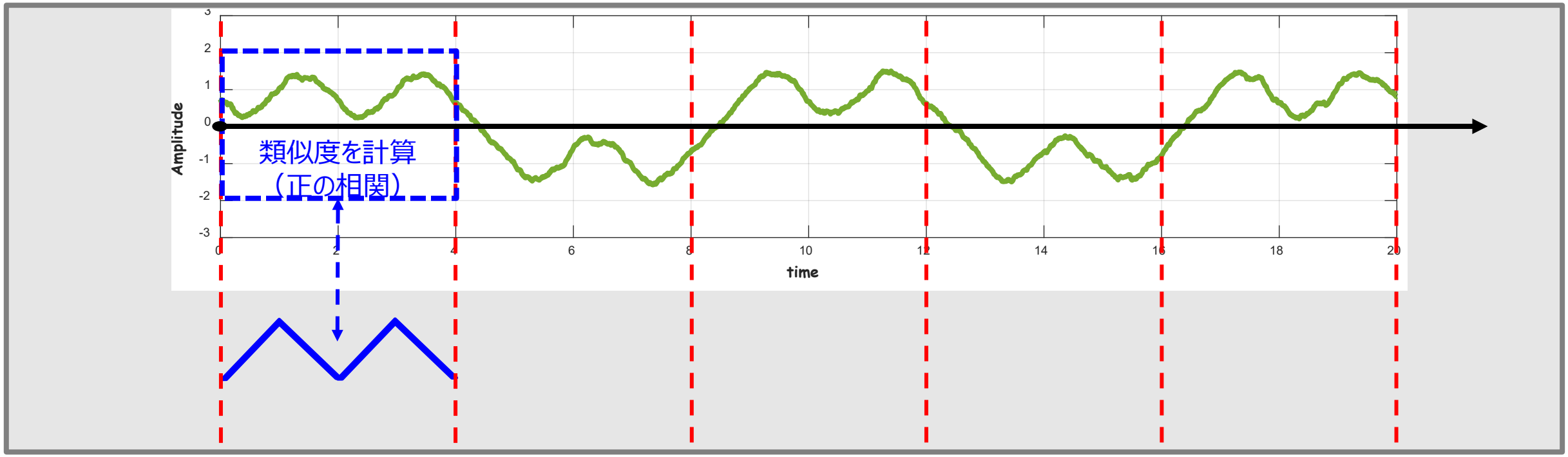
無線通信を用いたデータ伝送の例



移動平均（低域通過フィルタ）処理の後、
相関演算によるデータ検出を行う

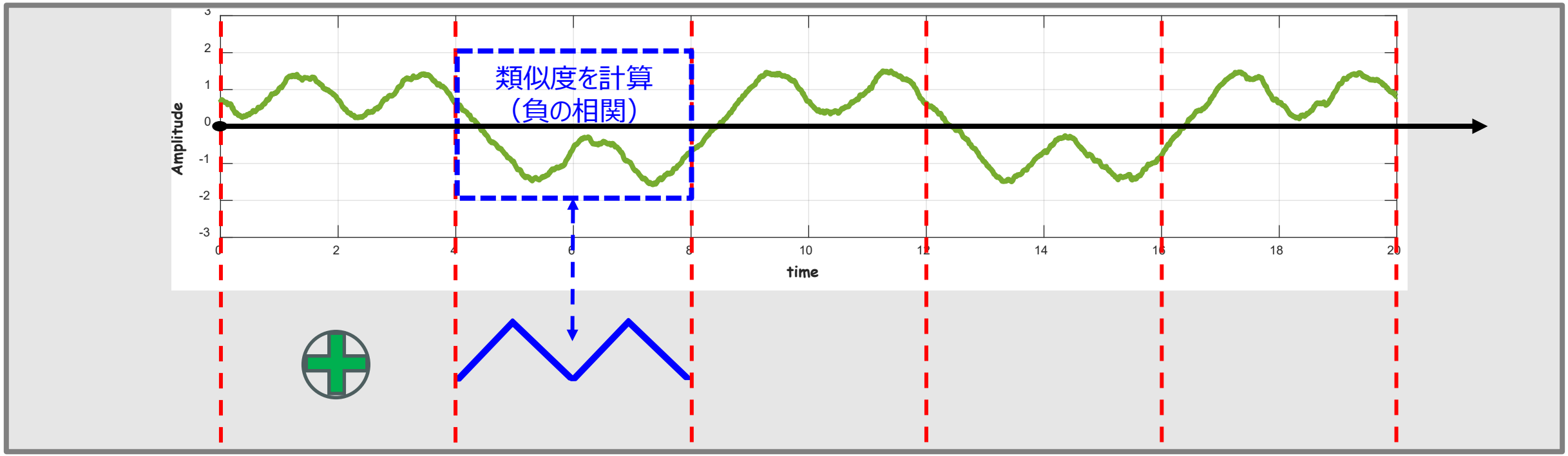
無線通信を用いたデータ伝送の例

相関演算を行い、送信データを推測（データ検出）

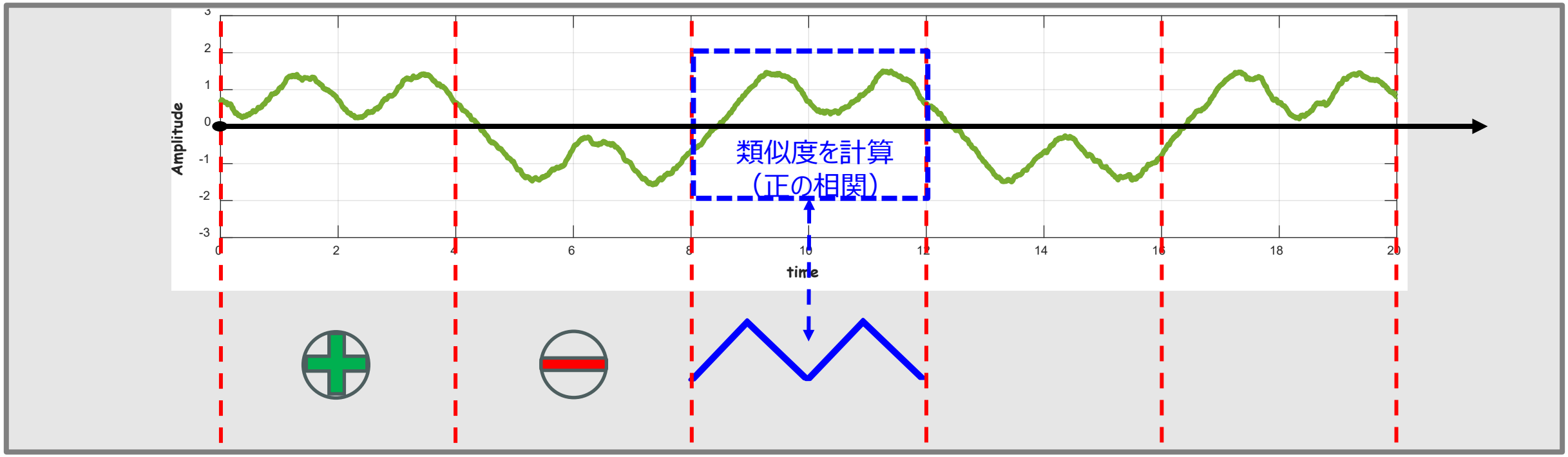


無線通信を用いたデータ伝送の例

相関演算を行い、送信データを推測（データ検出）

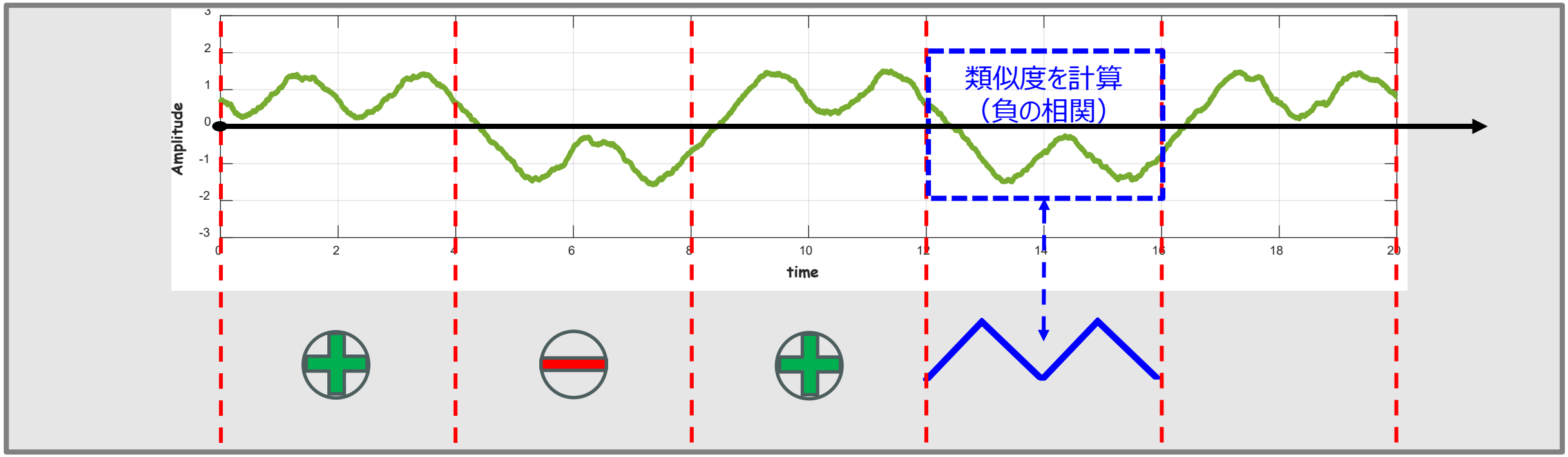


相関演算を行い、送信データを推測（データ検出）



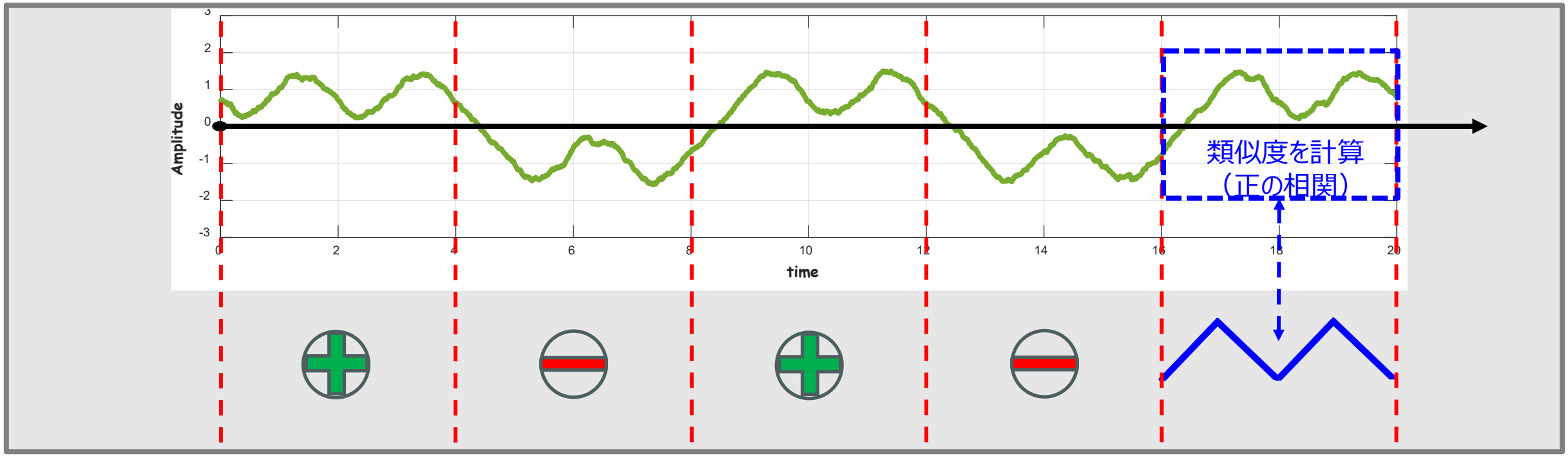
無線通信を用いたデータ伝送の例

相関演算を行い、送信データを推測（データ検出）



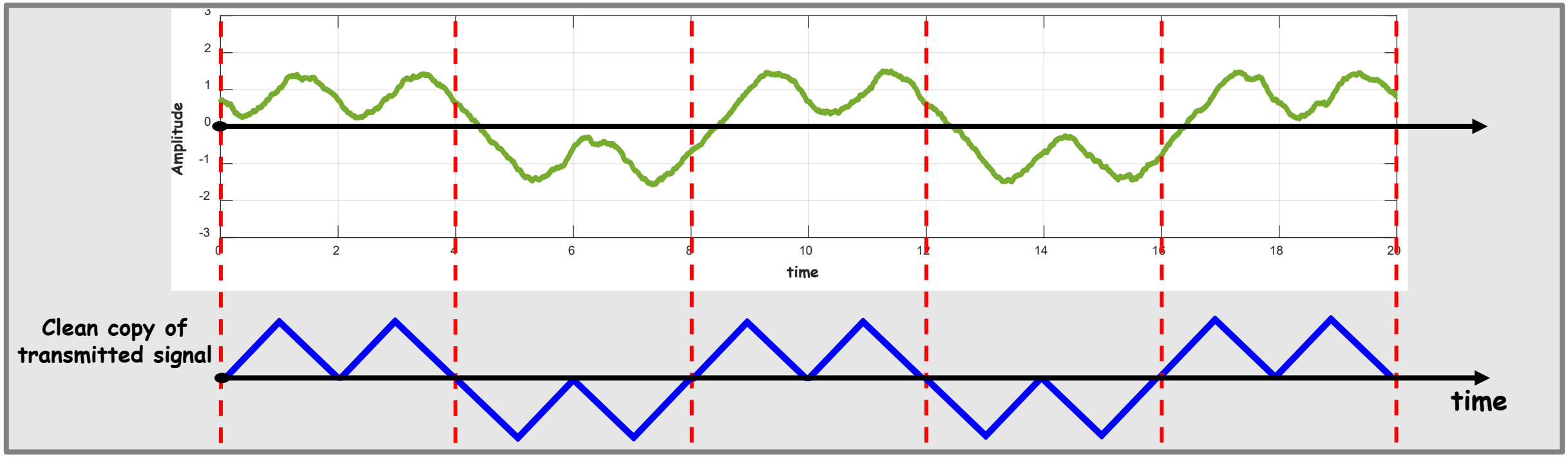
無線通信を用いたデータ伝送の例

相関演算を行い、送信データを推測（データ検出）



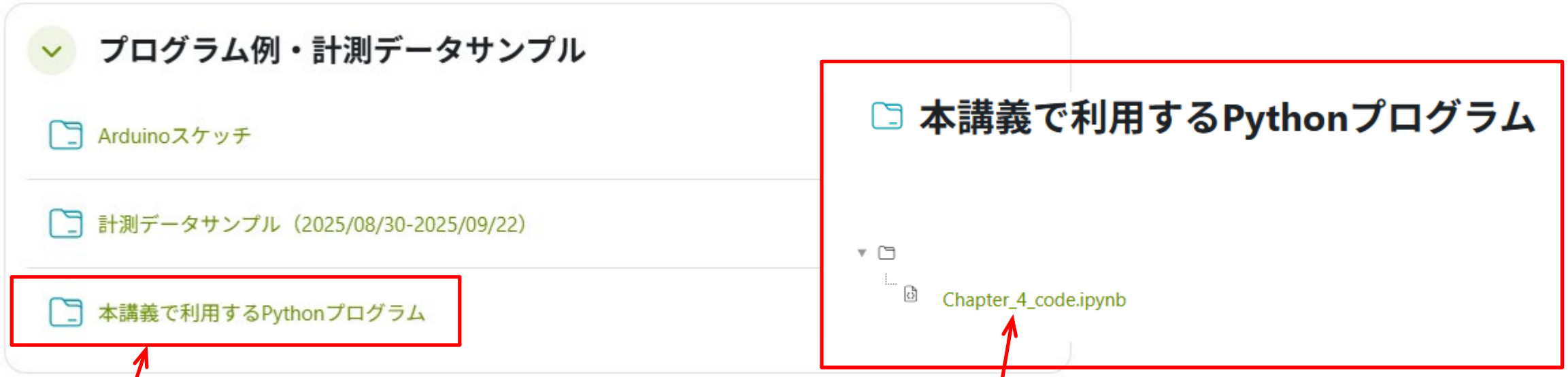
無線通信を用いたデータ伝送の例

相関演算を行い、送信データを推測（データ検出）



プログラム例

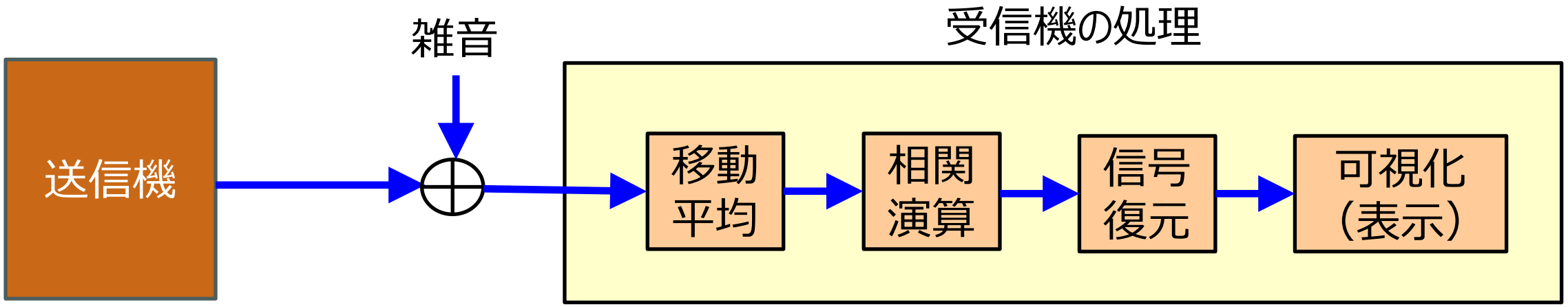
Moodleからサンプルプログラムをダウンロードできます。
それを用いて演習を行います。



このプログラムコードをダウンロード

Pythonのプログラムコードをここからダウンロードできます。

プログラム全体の流れ



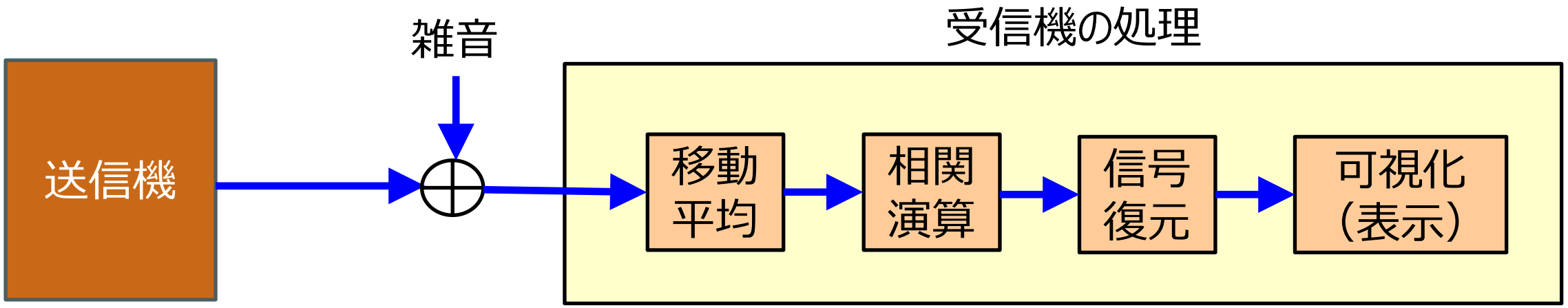
① 送信信号波形（三角波パルス系列）の生成

② 受信信号波形の生成（送信信号波形＋雑音）

③ 受信機の処理部（移動平均→相関演算→信号復元→可視化（表示）

※ 補足注：相関演算は雑音抑圧や平滑化の効果を有しますが、ここではプログラミング演習の観点からあえて個別に実装しています。

プログラム全体の流れ



① 送信信号波形（三角波パルス系列）の生成

② 受信信号波形の生成（送信信号波形＋雑音）

③ 受信機の処理部（移動平均→相関演算→信号復元→可視化（表示）

※ 補足注：相関演算は雑音抑圧や平滑化の効果を有しますが、ここではプログラミング演習の観点からあえて個別に実装しています。

プログラム例（1/4）


```
# 必要なライブラリをインポート
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0) #再現性のために乱数シードを設定
Fs = 100 # サンプリング周波数
dt = 1 / Fs # 時間間隔はサンプリング周波数の逆数


# Create a time vector from 0 to 1, inclusive.
# np.arange is exclusive of the stop value, so we add dt.
value = np.arange(0, 1 + dt, dt)

# Create a triangular pulse of duration T = 2s
x_cycle = np.concatenate([value, value[-2:0:-1]])

# Construct the full signal by concatenating positive and negative
pulses
x = np.concatenate([
x_cycle, x_cycle, -x_cycle, -x_cycle, x_cycle,
x_cycle, -x_cycle, -x_cycle, x_cycle, x_cycle, [0]
])

x = x / np.sqrt(np.mean(np.abs(x)**2))

# Create the full time and frequency vectors
t = np.arange(0, len(x)) * dt
df = 1 / t[-1]
f = np.arange(0, len(x)) * df
Xf = np.fft.fft(x) / len(x)
```

ライブラリの読み込み
NumPy：数値計算用（FFT等）
Matplotlib：可視化用（波形表示）

送信波形（三角パルス）の生成

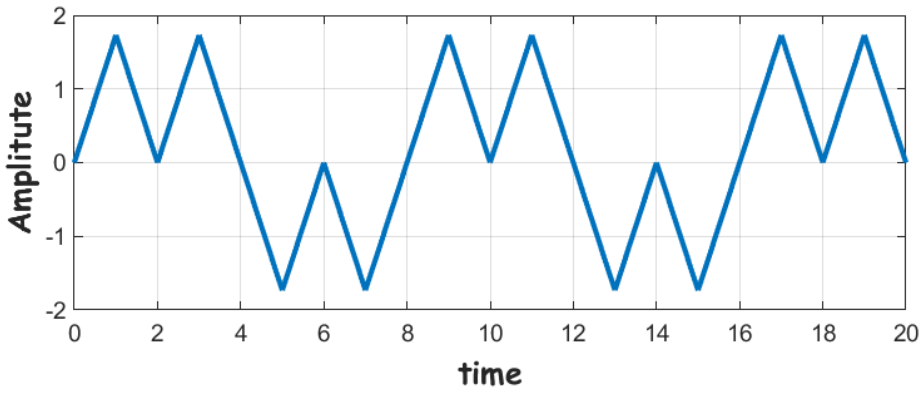
`value = np.arange(0, 1 + dt, dt)`
0, dt, 2dt, …, 1 までの等間隔の数列を生成（三角波の前半部）
dt=0.1ならば 0, 0.1, 0.2, ⋯, 1 の数列となる

`value[-2:0:-1]`
配列valueの「末尾から2番目の要素」から「最初の要素(0)の1つ手前」までを1つずつ逆順に取り出す（つまり、valueを反転させたものから先頭要素(0)を除去した形状となる）

`np.concatenate([value, value[-2:0:-1]])`
三角波の前半部がvalue，後半部がvalue[-2:0:-1]であり、それらをつなげて0→1→0 の形状の三角波を生成する。

正または負の三角波を10個並べたものを送信信号 x とする。

送信信号 x の平均電力を正規化



補足

例えば、dt=0.2 の場合、value = np.arange(0, 1 + dt, dt)は

value = [0, 0.2, 0.4, 0.6, 0.8, 1.0]

となる。

array[start : stop : step]
start: 開始位置（含む）
stop : 終了位置（含まない）
step : 増分（正なら順方向、負なら逆方向）

value[-2:0:-1]は、配列valueの「末尾から2番目の要素」から「最初の要素(0)の1つ手前」までを1つずつ逆順に取り出すことを表す。つまり、

value[-2:0:-1]=[0.8, 0.6, 0.4, 0.2]

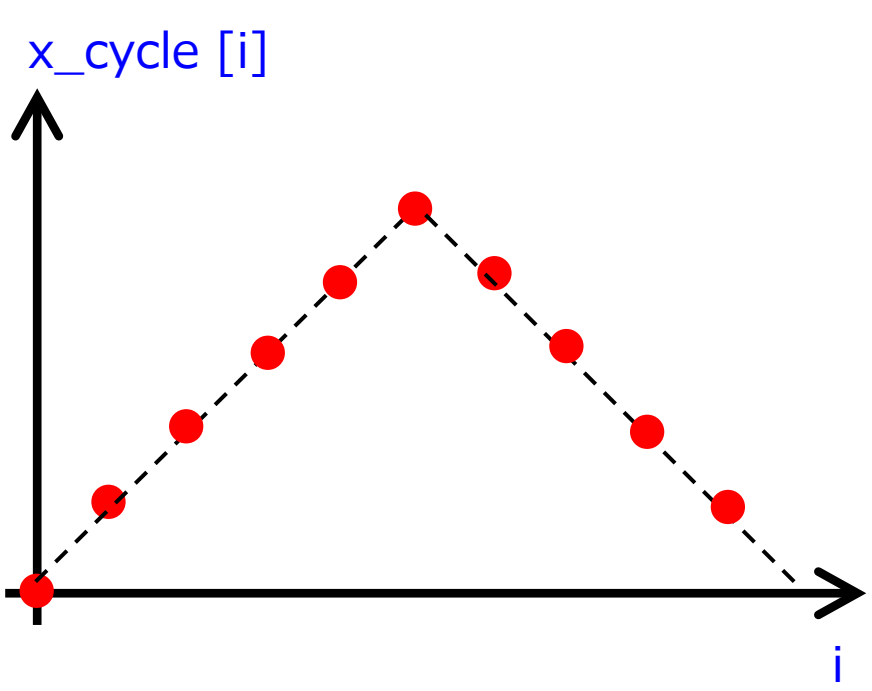
となる。

「NumPyライブラリを参照する」ことを表す

したがって、x_cycle = np.concatenate([value, value[-2:0:-1]])
は上記の２つの配列をつなげたものなので、

x_cycle = [0, 0.2, 0.4, 0.6, 0.8, 1.0, 0.8, 0.6, 0.4, 0.2]

となり、三角波パルスが生成される。



プログラム例（2/4）

雑音の混じった受信波から元の三角パルス列を復元する関数

```
def sigDetection(corruptedSig, refPulse):
    sigLen = len(corruptedSig)
    pulseLen = len(refPulse)
    # The length of corruptedSig must be a multiple integer of the length of refPulse
    if sigLen % pulseLen != 0:
        raise ValueError("Signal length must be an integer multiple of the reference pulse length.")
    numSegments = sigLen // pulseLen
    # Reshape the signal into a 2D array (column-wise).
    corruptedSig_reshaped = corruptedSig.reshape((pulseLen, numSegments), order='F')
    # Pre-allocate the output array
    cleanSig_reshaped = np.zeros((pulseLen, numSegments))
    # Ensure refPulse is a 1D array for correlation
    refPulse_flat = refPulse.flatten()
    for seg in range(numSegments):
        # Get the current segment
        current_segment = corruptedSig_reshaped[:, seg]
        # Calculate the Pearson correlation coefficient between the reference
        # and the current segment. np.corrcoef returns a 2x2 matrix.
        # The value we need is at [0, 1] or [1, 0].
        corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
        corrCoeff = corrCoeff_matrix[0, 1]
        # Make the decision based on the sign of the correlation
        if corrCoeff >= 0:
            cleanSig_reshaped[:, seg] = refPulse_flat
        else:
            cleanSig_reshaped[:, seg] = -refPulse_flat
    # Flatten the result back into a 1D array, preserving column-wise order.
    cleanSig = cleanSig_reshaped.flatten(order='F')
    return cleanSig
```

入力信号と参照信号の長さを抽出

len( )は配列の長さを返す関数
numbers = [10, 20, 30, 40]の場合
len(numbers)は4となる

プログラム例（2/4）

雑音の混じった信号から元の信号を復元する関数

```
def sigDetection(corruptedSig, refPulse):
    sigLen = len(corruptedSig)
    pulseLen = len(refPulse)
    # The length of corruptedSig must be a multiple integer of the length of refPulse
    if sigLen % pulseLen != 0:
        raise ValueError("Signal length must be an integer multiple of the reference pulse length.")
    numSegments = sigLen // pulseLen
    # Reshape the signal into a 2D array (column-wise).
    corruptedSig_reshaped = corruptedSig.reshape((pulseLen, numSegments), order='F')
    # Pre-allocate the output array
    cleanSig_reshaped = np.zeros((pulseLen, numSegments))
    # Ensure refPulse is a 1D array for correlation
    refPulse_flat = refPulse.flatten()
    for seg in range(numSegments):
        # Get the current segment
        current_segment = corruptedSig_reshaped[:, seg]
        # Calculate the Pearson correlation coefficient between the reference
        # and the current segment. np.corrcoef returns a 2x2 matrix.
        # The value we need is at [0, 1] or [1, 0].
        corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
        corrCoeff = corrCoeff_matrix[0, 1]
        # Make the decision based on the sign of the correlation
        if corrCoeff >= 0:
            cleanSig_reshaped[:, seg] = refPulse_flat
        else:
            cleanSig_reshaped[:, seg] = -refPulse_flat
    # Flatten the result back into a 1D array, preserving column-wise order.
    cleanSig = cleanSig_reshaped.flatten(order='F')
    return cleanSig
```

信号長が参照パルスの整数倍であることを確認し、その個数を numSegments として出力する

A // B　A/Bを行い、余りを切り捨て
例：4.1/2=2.05
　　4.1//2=2

プログラム例（2/4）

雑音の混じった信号から元の信号を復元する関数

```
def sigDetection(corruptedSig, refPulse):
    sigLen = len(corruptedSig)
    pulseLen = len(refPulse)
    # The length of corruptedSig must be a multiple integer of the length of refPulse
    if sigLen % pulseLen != 0:
        raise ValueError("Signal length must be an integer multiple of the reference pulse length.")
    numSegments = sigLen // pulseLen
    # Reshape the signal into a 2D array (column-wise).
    corruptedSig_reshaped = corruptedSig.reshape((pulseLen, numSegments), order='F')
    # Pre-allocate the output array
    cleanSig_reshaped = np.zeros((pulseLen, numSegments))
    # Ensure refPulse is a 1D array for correlation
    refPulse_flat = refPulse.flatten()
    for seg in range(numSegments):
        # Get the current segment
        current_segment = corruptedSig_reshaped[:, seg]
        # Calculate the Pearson correlation coefficient between the reference
        # and the current segment. np.corrcoef returns a 2x2 matrix.
        # The value we need is at [0, 1] or [1, 0].
        corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
        corrCoeff = corrCoeff_matrix[0, 1]
        # Make the decision based on the sign of the correlation
        if corrCoeff >= 0:
            cleanSig_reshaped[:, seg] = refPulse_flat
        else:
            cleanSig_reshaped[:, seg] = -refPulse_flat
    # Flatten the result back into a 1D array, preserving column-wise order.
    cleanSig = cleanSig_reshaped.flatten(order='F')
    return cleanSig
```

- 信号波形をパルス長単位で区切り、それを相関演算用の2次元配列に格納する。
- 相関演算の結果格納用の配列(`cleanSig_reshaped`)を初期化する。
- 相関演算用に参照パルス波形を1次元配列化する。

flatten()：配列を一次元配列に変換する関数
例えば、a = np.array([[1, 2],
[3, 4]])
のとき、a.flatten() は [1, 2, 3, 4] を表す

プログラム例（2/4）

雑音の混じった信号から元の信号を復元する関数

```
def sigDetection(corruptedSig, refPulse):
    sigLen = len(corruptedSig)
    pulseLen = len(refPulse)
    # The length of corruptedSig must be a multiple integer of the length of refPulse
    if sigLen % pulseLen != 0:
        raise ValueError("Signal length must be an integer multiple of the reference pulse length.")
    numSegments = sigLen // pulseLen
    # Reshape the signal into a 2D array (column-wise).
    corruptedSig_reshaped = corruptedSig.reshape((pulseLen, numSegments), order='F')
    # Pre-allocate the output array
    cleanSig_reshaped = np.zeros((pulseLen, numSegments))
    # Ensure refPulse is a 1D array for correlation
    refPulse_flat = refPulse.flatten()
    for seg in range(numSegments):
        # Get the current segment
        current_segment = corruptedSig_reshaped[:, seg]
        # Calculate the Pearson correlation coefficient between the reference
        # and the current segment. np.corrcoef returns a 2x2 matrix.
        # The value we need is at [0, 1] or [1, 0].
        corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
        corrCoeff = corrCoeff_matrix[0, 1]
        # Make the decision based on the sign of the correlation
        if corrCoeff >= 0:
            cleanSig_reshaped[:, seg] = refPulse_flat
        else:
            cleanSig_reshaped[:, seg] = -refPulse_flat
    # Flatten the result back into a 1D array, preserving column-wise order.
    cleanSig = cleanSig_reshaped.flatten(order='F')
    return cleanSig
```

np.corrcoef ：参照パルス refPulse_flat と現在のセグメント current_segment の相関係数 を計算。

np.corrcoef（2x2相関行列）の [0,1]要素（相関係数）をcorrCoeffに代入する

補足：相関係数の計算部について


```
corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
corrCoeff = corrCoeff_matrix[0, 1]
```

このプログラムでは、NumPyのライブラリ関数の一つであるnp.corrcoefを用いて、相関係数を求めている。

np.corrcoef(a, b)では、以下の2x2の行列(共分散行列を正規化したもの)を返す。
[[corr(a,a), corr(a,b)],
 [corr(b,a), corr(b,b)]]
ここで、corr(a,b)はaとbの相関係数を表す。

ここで、 corr(a,a)=corr(b,b)=1なので、以下となる。

[[1, corr(a,b)],
 [corr(b,a), 1]]

corrcoef(a, b) は対称行列なので、corr(a,b)=corr(b,a)となる。
したがって、corrCoeff_matrix[1, 0]を用いてもよい。
※共分散行列は対称行列であり(定義式から自明)、それを正規化した相関係数行列も対称行列である。

プログラム例（2/4）

雑音の混じった信号から元の信号を復元する関数

```
def sigDetection(corruptedSig, refPulse):
    sigLen = len(corruptedSig)
    pulseLen = len(refPulse)
    # The length of corruptedSig must be a multiple integer of the length of refPulse
    if sigLen % pulseLen != 0:
        raise ValueError("Signal length must be an integer multiple of the reference pulse length.")
    numSegments = sigLen // pulseLen
    # Reshape the signal into a 2D array (column-wise).
    corruptedSig_reshaped = corruptedSig.reshape((pulseLen, numSegments), order='F')
    # Pre-allocate the output array
    cleanSig_reshaped = np.zeros((pulseLen, numSegments))
    # Ensure refPulse is a 1D array for correlation
    refPulse_flat = refPulse.flatten()
    for seg in range(numSegments):
        # Get the current segment
        current_segment = corruptedSig_reshaped[:, seg]
        # Calculate the Pearson correlation coefficient between the reference
        # and the current segment. np.corrcoef returns a 2x2 matrix.
        # The value we need is at [0, 1] or [1, 0].
        corrCoeff_matrix = np.corrcoef(refPulse_flat, current_segment)
        corrCoeff = corrCoeff_matrix[0, 1]
        # Make the decision based on the sign of the correlation
        if corrCoeff >= 0:
            cleanSig_reshaped[:, seg] = refPulse_flat
        else:
            cleanSig_reshaped[:, seg] = -refPulse_flat
    # Flatten the result back into a 1D array, preserving column-wise order.
    cleanSig = cleanSig_reshaped.flatten(order='F')
    return cleanSig
```

相関係数が正であれば、参照パルスをそのままコピー
負であれば、参照パルスの符号（正または負）を反転して格納。

`cleanSig_reshaped.flatten`を列優先(`order='F'`)で一次元配列に並べ直して、`cleanSig`に代入

プログラム例（3/4）

移動平均処理

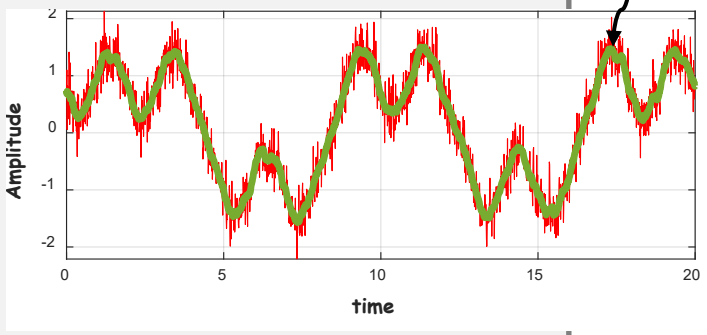
受信信号に雑音を加え、その雑音を移動平均フィルタで平滑化

```
# Retrieve a clean copy of the transmitted signal from corrupted signal
rxnoisy = x + 0.25 * np.random.randn(len(x)) # corrupted
window = 20 # filtering window size
# Implement moving average (using convolution with a flat window) to smooth the corrupted signal
rx_mean = np.convolve(rxnoisy, np.ones(window)/window, mode='same')
# 'same' mode handles boundaries.
```

雑音が混ざった受信信号 rxnoisy を生成

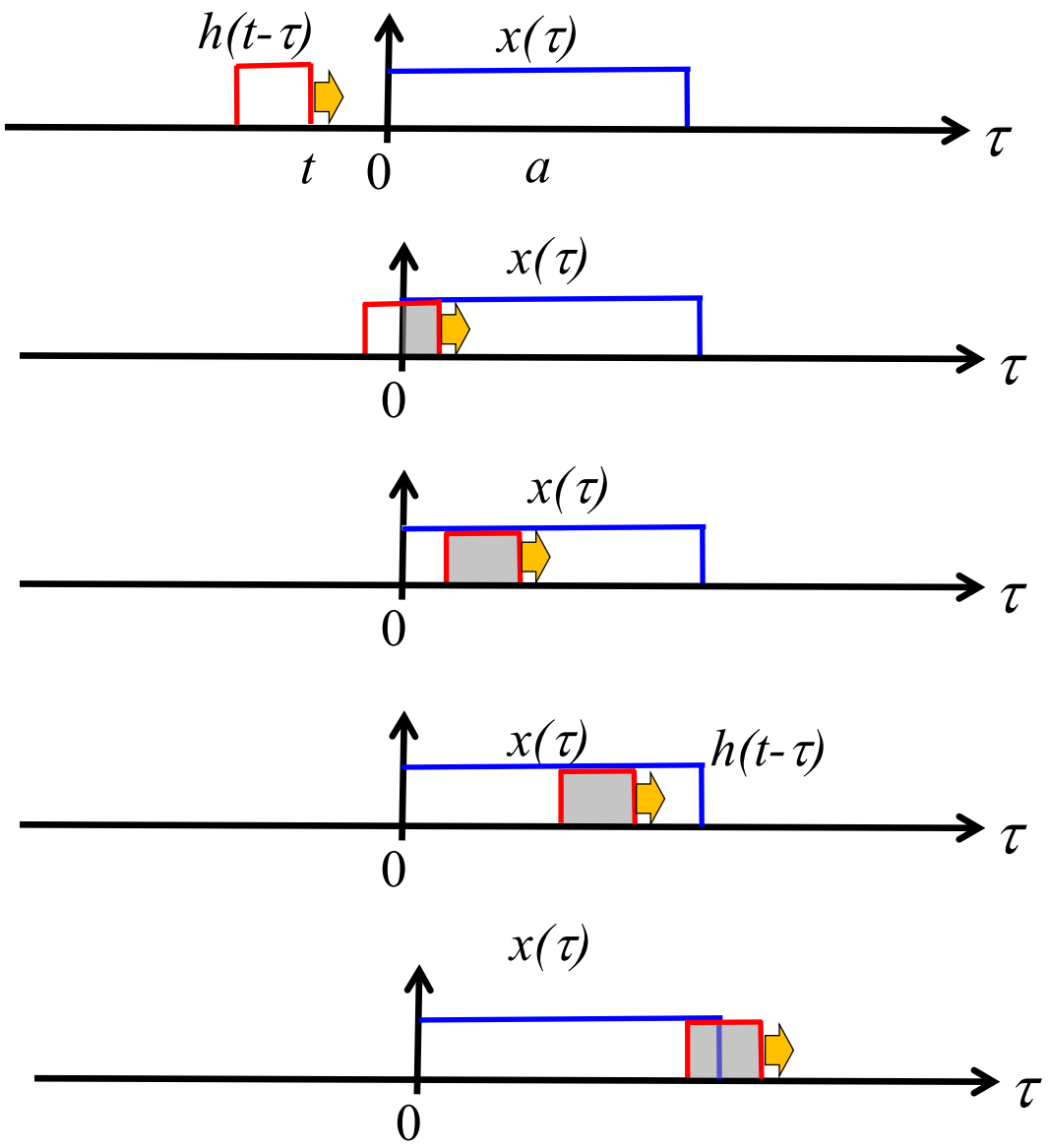
np.convolve：畳み込み関数
np.ones(window)/window：長さwindow、要素値1/windowのベクトル
rxnoisyとnp.ones(window)/windowとの畳み込み演算=移動平均処理

移動平均後

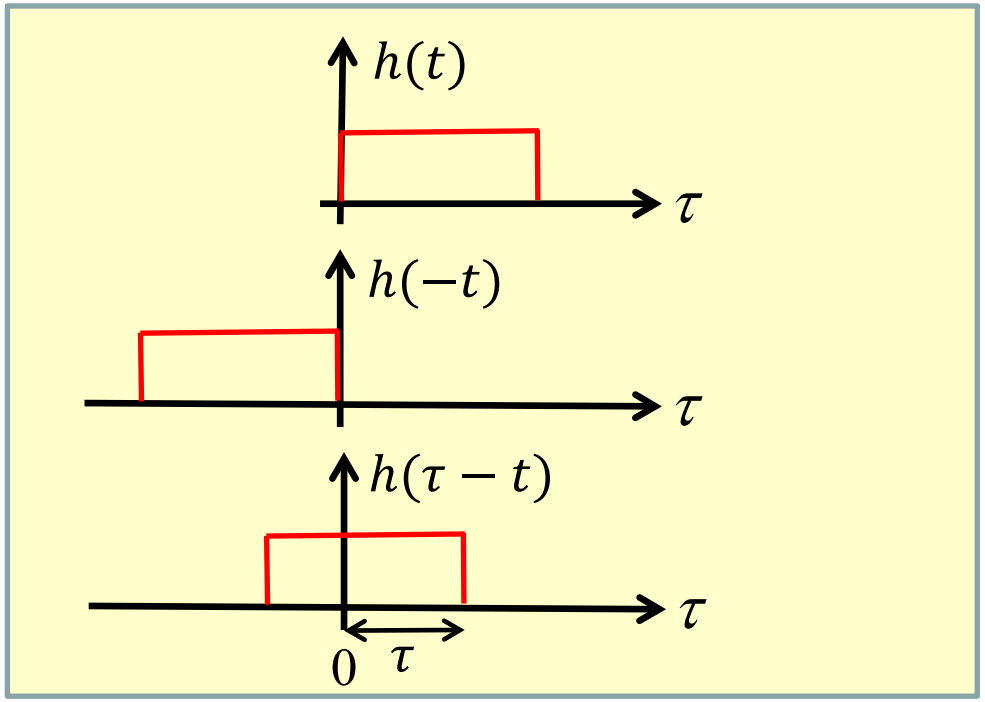



```
# Reference pulse for detection
refPulse = np.concatenate([x_cycle, x_cycle])
refPulse = refPulse / np.sqrt(np.mean(np.abs(refPulse)**2)) # normalizing pulse power
# Cleaning the smoothed signal
rx_mean_sliced = rx_mean[:-1] # Exclude the last sample for ease of calculation in sigDetection
rx_clean_sliced, correlationPolar = sigDetection(rx_mean_sliced, refPulse)
rx_clean = np.append(rx_clean_sliced, 0)
# Append the final zero to match the original signal length
```

補足：たたみ込み積分演算と移動平均



$$y(t) = \int_{-\infty}^{\infty} f_1(\tau) f_2(t-\tau) d\tau = \int_{-\infty}^{\infty} f_1(t-\tau) f_2(\tau) d\tau$$



赤枠の範囲の積分を実行している。
（＝移動平均処理）

※ここでは、連続時間関数のたたみ込み積分の式を用いて説明した。
プログラム中では離散時間系列（サンプル列）に対する畳み込み演算となる。その場合も考え方は同様である。

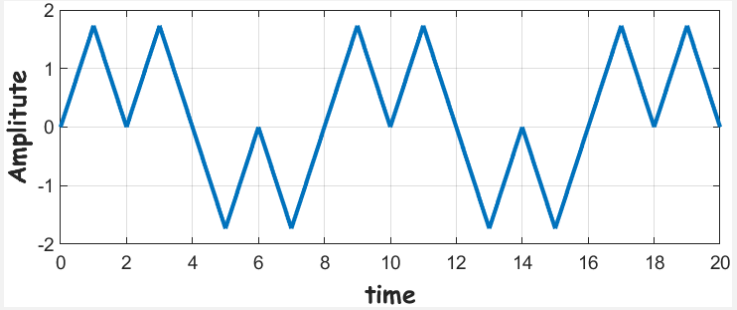
プログラム例（3/4）


```
# Retrieve a clean copy of the transmitted signal from corrupted signal
rxnoisy = x + 0.25 * np.random.randn(len(x)) # corrupted
window = 20 # filtering window size
# Implement moving average (using convolution with a flat window) to smooth the corrupted signal
rx_mean = np.convolve(rxnoisy, np.ones(window)/window, mode='same')
# 'same' mode handles boundaries.
```

相関演算

```
# Reference pulse for detection
refPulse = np.concatenate([x_cycle, x_cycle])
refPulse = refPulse / np.sqrt(np.mean(np.abs(refPulse)**2))   # パルス電力を1に正規化
# Cleaning the smoothed signal
rx_mean_sliced = rx_mean[:-1] # 移動平均後の信号 rx_mean の最後の1サンプルを除外
rx_clean_sliced, correlationPolar = sigDetection(rx_mean_sliced, refPulse)
rx_clean = np.append(rx_clean_sliced, 0) # 元の信号長と一致させるために、末尾に 0 を追加
```

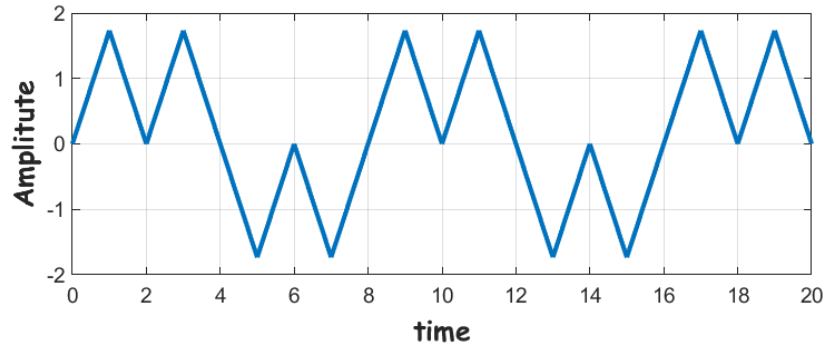
`np.concatenate([x_cycle, x_cycle])`
三角波 x_cycle を 2 個つなげて参照パルスを生成する。

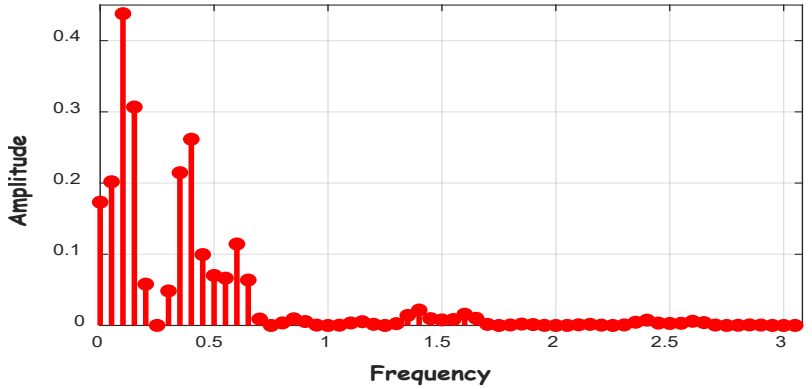


プログラム例（4/4）

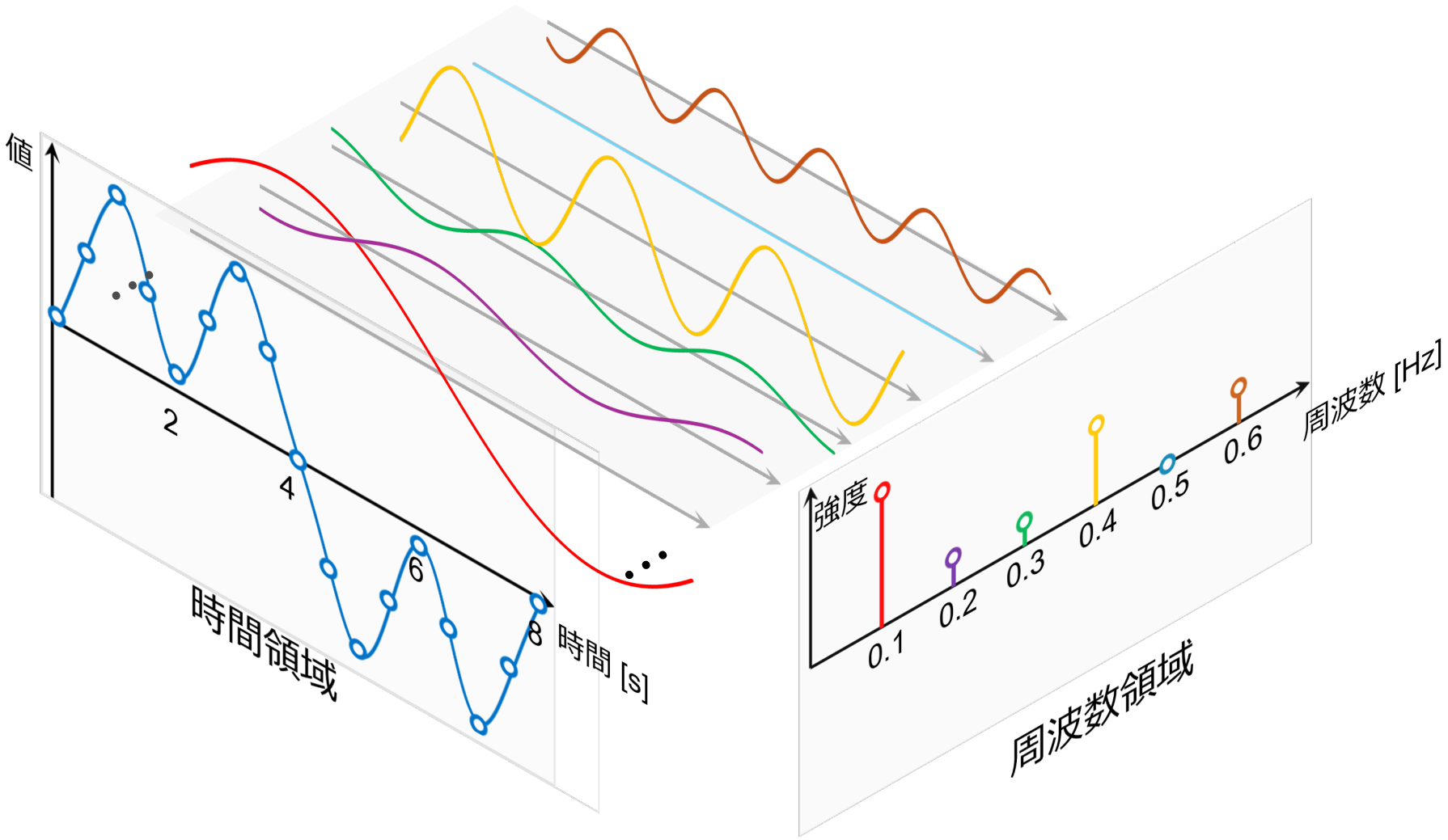
受信信号の時間領域波形と周波数領域スペクトル

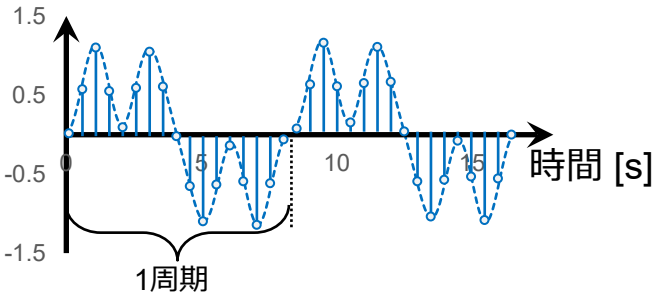
```
# Signal plot in time
plt.figure()
plt.plot(t, x)
plt.title('Original Signal')
plt.xlabel('time')
plt.ylabel('Amplitude')
plt.xlim(0, t[-1])
plt.grid(True)


# Signal plot in Frequency
plt.figure()
plt.stem(f, np.abs(Xf), linefmt='r', markerfmt='r')
plt.title('Frequency Spectrum of Original Signal')
plt.xlabel('Frequency')
plt.ylabel('Amplitude')
plt.xlim(0, 3) # Sets the f-axis limits for better visualization
plt.grid(True)
```





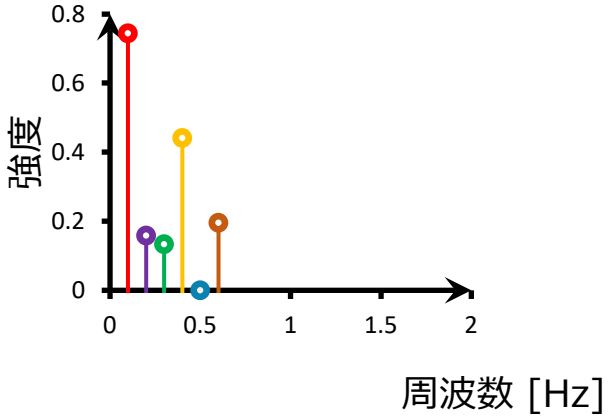
補足：時間領域と周波数領域における信号表現の例





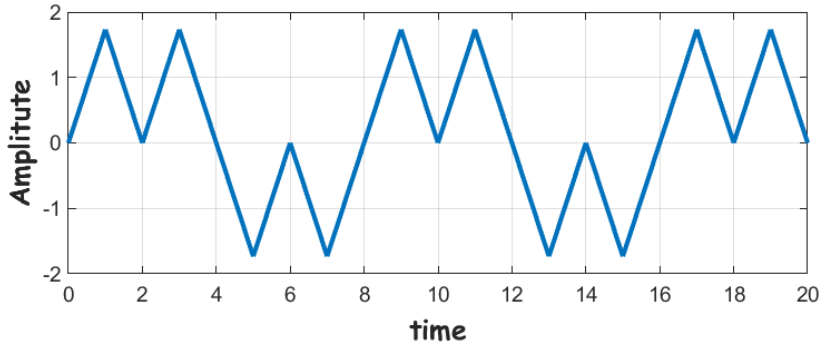
サンプリング間隔：0.5 s

最高周波数：0.6 Hz
サンプリング周波数： 2Hz



課題：実際に使ってみよう

先のサンプルプログラムでは、以下の波形を送信信号として用いた。
（データ系列 「＋＋ーー＋＋－－＋＋」）



以下を試して観察結果をまとめてください。

- 三角波以外に波形を変更する（例えば、矩形波） 。
- データ系列を「＋＋ーー＋＋－－＋＋」以外に変更する。（例えば、「＋－＋－＋－＋－」）
- 移動平均長を変えて結果を比較する。

まとめ

- 計測データ処理の基礎として、平均、分散、移動平均、移動分散、確率質量関数、確率分布関数、相関関数といった基本的な演算や統計処理の手法、およびそれらのプログラムによる実装方法について概説した。

- また、IoTシステムの基盤となる無線通信システムにおけるデータ伝送の仕組みとプログラム例を紹介し、これらの基本演算が通信処理においても応用されていることを示した。