pyheatintegration

リリース *0.7.0*

tarao1006

2021 年 11 月 08 日

Getting Started

pyheatintegration は プロセス設計 で必要なヒートインテグレーションを支援するパッケージです。グランドコンポジットカーブの作成、TQ 線図の作成、必要な熱交換器数の導出を行うことができます。

第 1 章

Installation

```
pip install pyheatintegration
```

第 2 章

Examples

- simple example

第 3 章

Quick Start

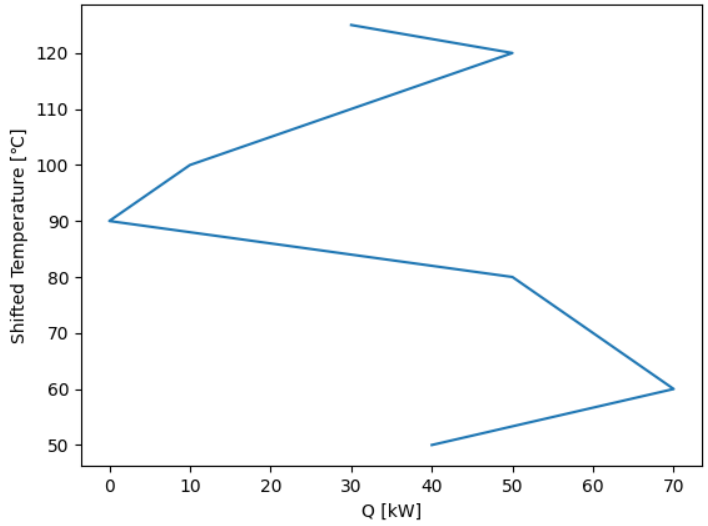
```
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

from pyheatintegration import PinchAnalyzer, Stream, extract_x, y_range

# 熱交換を行う流体を準備
streams = [
    Stream(40.0, 90.0, 150.0),
    Stream(80.0, 110.0, 180.0),
    Stream(125.0, 80.0, 180.0),
    Stream(100.0, 60.0, 160.0)
]

# 最小接近温度差を指定し、作成した流体のリストとともに PinchAnalyzer のインスタンスを得る。
minimum_approach_temperature_difference = 10.0
analyzer = PinchAnalyzer(streams, minimum_approach_temperature_difference)

# グランドコンポジットカーブ
heats, temps = analyzer.create_grand_composite_curve()
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("Shifted Temperature [℃]")
ax.plot(heats, temps)
fig.savefig("path/to/grand_composite_curve.png")

# TQ 線図
hot_lines, cold_lines = analyzer.create_tq()
```

(次のページに続く)

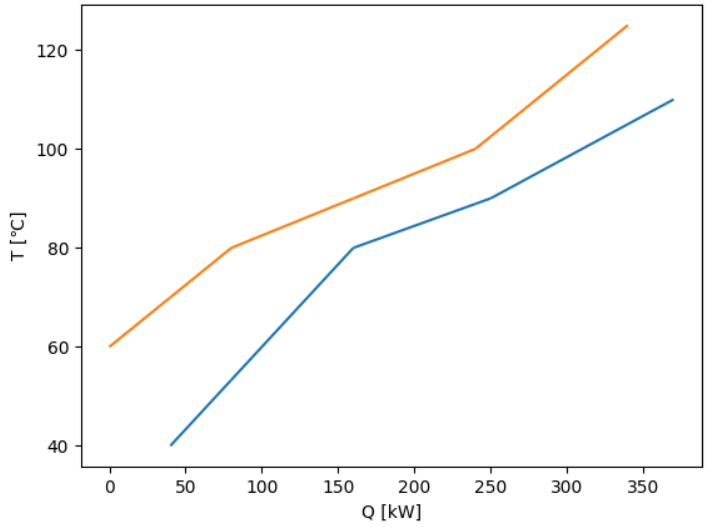
(前のページからの続き)

```
ymin, ymax = y_range(hot_lines + cold_lines)
heats = extract_x(hot_lines + cold_lines)
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines, colors="#1f77b4"))
ax.vlines(heats, ymin=ymin, ymax=ymax, linestyles=':', colors='k')
ax.autoscale()
fig.savefig("path/to/tq_diagram.png")
```

- グランドコンポジットカーブ



- TQ 線図



より詳しい説明は *Getting Started* を確認してください。

3.1 Getting Started

図を描くための手順を説明しています。最低限の Python の知識があることを前提としています。また、実際に動く例は simple example を参照してください。

3.1.1 Step1. Create Stream

熱交換を行いたい流体の情報を元に、 `Stream` のリストを生成します。 `Stream` のコンストラクタは、以下のようになっています。

class Stream(*input_temperature: float*, *output_temperature: float*, *heat_load: float*, *type_: StreamType* = <*StreamType.AUTO: 5*>, *state: StreamState* = <*StreamState.UNKNOWN: 5*>, *cost: float* = *0.0*, *reboiler_or_reactor: bool* = *False*, *id_: str* = *''*)

- `input_temperature (float)`: 入り口温度 [℃]
- `output_temperature (float)`: 出口温度 [℃]
- `heat_load (float)`: 熱量 [W]
- `type_ (StreamType, optional)`: 流体の種類
  - `StreamType.COLD`: 受熱
  - `StreamType.HOT`: 与熱

  - StreamType.EXTERNAL_COLD: 外部受熱 (冷却用ユーティリティ)
  - StreamType.EXTERNAL_HOT: 外部与熱 (加熱用ユーティリティ)
  - StreamType.AUTO: 自動
- state (StreamState, optional): 流体の状態
  - StreamState.GAS: ガス
  - StreamState.LIQUID: 液
  - StreamState.GAS_CONDENSATION: ガス (凝縮)
  - StreamState.LIQUID_EVAPORATION: 液 (蒸発)
  - StreamState.UNKNOWN: 不明/指定しない。
- cost (float): コスト [円/J] (外部流体の場合)
- reboiler_or_reactor (bool, optional): リボイラーまたは反応器の熱交換に用いられるか。熱交換器のコストを計算する際の係数を決定するために必要な情報。
- id_ (str, optional): ID

例

```
Stream(40.0, 90.0, 150.0, StreamType(1), StreamState.LIQUID, 0.0, False, 'cold1')
Stream(80.0, 110.0, 180.0, StreamType(1), StreamState.LIQUID, 0.0, False, 'cold2')
Stream(125.0, 80.0, 180.0, StreamType(2), StreamState.GAS, 0.0, False, 'hot1')
Stream(100.0, 60.0, 160.0, StreamType(2), StreamState.GAS, 0.0, False, 'hot2')
Stream(20.0, 20.0, 0.0, StreamType(3), StreamState.LIQUID_EVAPORATION, 100.0, False,
↪'external cold1')
Stream(150.0, 150.0, 0.0, StreamType(4), StreamState.GAS_CONDENSATION, 200.0, False,
↪'external hot1')
```

入り口温度/出口温度/熱量

受熱流体は input_temperature output_temperature 、与熱流体は input_temperature output_temperature である必要があります。等温流体の設定も可能です。外部流体の熱量は、グランドコンポジットカーブ作成時に決定するため、流体作成時には heat_load = 0 を指定してください。

流体の種類

流体の種類 type_ は enum 型である StreamType を用いて指定します。指定可能な種類は、**受熱**、**与熱**、**外部受熱**、**外部与熱** の 4 種類です。また、自動で種類を判断するように指定する **自動** もありますが、明示的に指定する機会はないと思われます。

- StreamType.COLD (= 1): 受熱
- StreamType.HOT (= 2): 与熱
- StreamType.EXTERNAL_COLD (= 3): 外部受熱 (冷却用ユーティリティ)
- StreamType.EXTERNAL_HOT (= 4): 外部与熱 (加熱用ユーティリティ)
- StreamType.AUTO (= 5): 自動

以下の例のように流体を指定します。ただし、オプショナル引数は一部省略しています。

```
# 入り口温度: 40 度 出口温度 90 度 熱量 150.0 W の受熱流体
Stream(40.0, 90.0, 150.0, StreamType(1))

# 入り口温度: 125 度 出口温度 80 度 熱量 180.0 W の与熱流体
Stream(125.0, 80.0, 180.0, StreamType(2))

# 入り口温度: 20 度 出口温度 20 度 の外部受熱流体
Stream(40.0, 90.0, 0.0, StreamType(3))

# 入り口温度: 125 度 出口温度 80 度 の外部与熱流体
Stream(150.0, 150.0, 0.0, StreamType(4))
```

ただし、外部流体でない場合、入り口温度と出口温度の関係から流体の種類を推測することができるため、省略することができます。一方、外部流体は必ず指定する必要があります。

```
# 受熱流体と与熱流体は StreamType を指定しなくても良い。
Stream(40.0, 90.0, 150.0)
Stream(125.0, 80.0, 180.0)

# 外部受熱流体と外部与熱流体は StreamType を指定する必要があります。
Stream(40.0, 90.0, 0.0, StreamType(3))
Stream(150.0, 150.0, 0.0, StreamType(4))
```

流体の状態

流体の状態 state は enum 型である StreamState を用いて指定します。この値を用いて総括伝熱係数の値を指定します。

- StreamState.GAS (= 1): ガス
- StreamState.LIQUID (= 2): 液
- StreamState.GAS_CONDENSATION (= 3): ガス (凝縮)
- StreamState.LIQUID_EVAPORATION (= 4): 液 (蒸発)
- StreamState.UNKNOWN (= 5): 不明/指定しない。

```
# 液体の流体
Stream(40.0, 90.0, 150.0, state=StreamState(2))
```

総括伝熱係数の値は、 プロセスデザインコンテストを参考にして以下のように指定しています。

Hot	Cold	U [$\mathrm{W/m^2 \cdot K}$]
Gas	Gas	150
Liquid	Gas	200
Liquid	Liquid	300
Gas (Condensation)	Liquid (Evaporation)	1,500
Gas	Liquid	200
Gas (Condensation)	Gas	500
Gas (Condensation)	Liquid	1,000
Gas	Liquid (Evaporation)	500
Liquid	Liquid (Evaporation)	1,000

注釈: 与熱流体には StreamState.LIQUID_EVAPORATION を、受熱流体には、StreamState.GAS_CONDENSATION を指定することができません。

ID

id_ は流体を区別するために指定します。複数の流体を作成する場合には、id を重複しないようにする必要があります。

3.1.2 Step2. Analysis

注釈: 流体の指定方法をまた読んでいない方は、 まずは *Step1. Create Stream* を読んでください。

流体を生成した後、 PinchAnalyzer を用いてグランドコンポジットカーブおよび TQ 線図を書きます。PinchAnalyzer のコンストラクタは以下のようになっています。

class PinchAnalyzer(*streams_: list[Stream]*, *minimum_approach_temp_diff: float*, *ignore_maximum: bool = False*)

- streams_ (list[Stream]): 流体のリスト。
- minimum_approach_temp_diff (float): 最小接近温度差 [℃]。
- ignore_maximum (bool, optional): 最小接近温度差を指定可能かを検証する際に、最大値のチェックを無視するかどうか。

注釈: 外部流体を指定せずに解析を行いたい場合などに ignore_maximum を指定すると、エラーが生じずに解析を行うことが可能となる可能性があります。

例

```
streams = [
    Stream(40.0, 90.0, 150.0, StreamType(1), StreamState.LIQUID, 0.0, False, 'cold1')
    Stream(80.0, 110.0, 180.0, StreamType(1), StreamState.LIQUID, 0.0, False, 'cold2')
    Stream(125.0, 80.0, 180.0, StreamType(2), StreamState.GAS, 0.0, False, 'hot1')
    Stream(100.0, 60.0, 160.0, StreamType(2), StreamState.GAS, 0.0, False, 'hot2')
    Stream(20.0, 20.0, 0.0, StreamType(3), StreamState.LIQUID_EVAPORATION, 100.0, False,
↪'external cold1')
    Stream(150.0, 150.0, 0.0, StreamType(4), StreamState.GAS_CONDENSATION, 200.0, False,
↪'external hot1')
]

analyzer = PinchAnalyzer(streams, 10.0)
```

流体の検証

コンストラクタで受け取った流体のリストのバリデーションを行います。以下の場合に不正と判断します。ただし、 `ignore_maximum` を `True` とした場合、3 の条件は無視されます。

1. 流体の id が重複している場合
2. 与熱流体もしくは受熱流体が 1 つもない場合
3. **与熱流体の最高温度** < **受熱流体の最高温度** または **与熱流体の最低温度** > **受熱流体の最低温度** である場合

最小接近温度差の検証

最小接近温度差は、指定可能範囲が存在する。指定値がその範囲にない場合、不正と判断します。指定可能な最大値を、 `min(`**与熱流体の最高温度** - **受熱流体の最高温度**, **与熱流体の最低温度** - **受熱流体の最低温度**`)` とします。ただし、`ignore_maximum` を `True` とした場合には、指定可能な最大値を**与熱流体の最高温度** - **受熱流体の最低温度** とします。

グラフ描画

`PinchAnalyzer` のインスタンス生成時に、グランドコンポジットカーブおよび TQ 線図の描画に必要な計算が行われます。計算した結果は以下の `create_*` を呼ぶことで得ることができます。実際のグラフ描画については *Step3. Draw Graph* を参照して下さい。

- `create_grand_composite_curve`: グランドコンポジットカーブ
- `create_tq`: TQ 線図
- `create_tq_separated`: 流体ごとに分割した TQ 線図
- `create_tq_split`: 流体ごとに分割し、最初接近温度差を満たすように分割した TQ 線図
- `create_tq_merged`: 結合可能な熱交換器を結合した TQ 線図

3.1.3 Step3. Draw Graph

注釈: 流体の指定方法および `PinchAnalyzer` の説明をまた読んでいない方は、 まずは *Step1. Create Stream* および *Step2. Analysis* を読んでください。

`PinchAnalyzer` を用いた解析によって、グラフを描画するための情報を得ることができます。以下の `create_*` を呼ぶことで得ることができます。

- `create_grand_composite_curve`: グランドコンポジットカーブ

- create_tq: TQ 線図
- create_tq_separated: 流体ごとに分割した TQ 線図
- create_tq_split: 流体ごとに分割し、最初接近温度差を満たすように分割した TQ 線図
- create_tq_merged: 結合可能な熱交換器を結合した TQ 線図

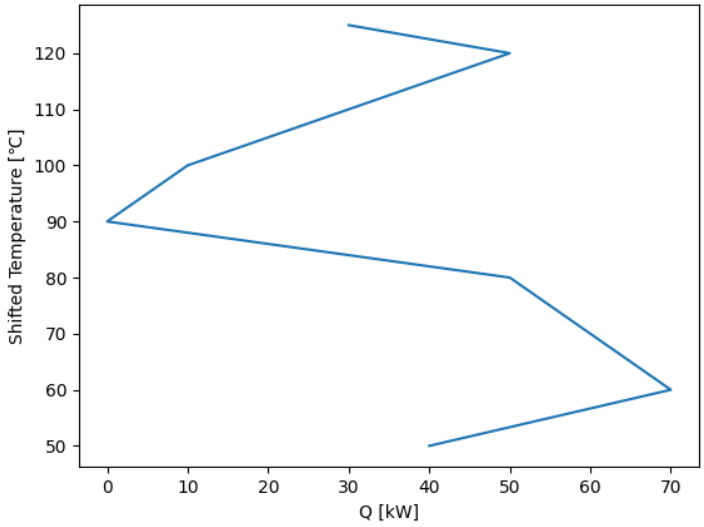
例で用いる analyzer は以下のコードによって作成されたと仮定しています。

```
from pyheatintegration import PinchAnalyzer, Stream

streams = [
    Stream(40.0, 90.0, 150.0),
    Stream(80.0, 110.0, 180.0),
    Stream(125.0, 80.0, 180.0),
    Stream(100.0, 60.0, 160.0)
]

minimum_approach_temperature_difference = 10.0
analyzer = PinchAnalyzer(streams, minimum_approach_temperature_difference)
```

グランドコンポジットカーブ

analyzer.create_grand_composite_curve() を用いて、熱量と温度のリストを取得することができます。

```
heats, temps = analyzer.create_grand_composite_curve()
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("Shifted Temperature [℃]")
ax.plot(heats, temps)
fig.savefig("path/to/grand_composite_curve.png")
```



TQ 線図

PinchAnalyzer は以下の 4 種類の TQ 線図を描くためのデータを提供します。

- **通常の *TQ* 線図**
- **流体ごとに分割した *TQ* 線図**
- **最小接近温度差を満たすように流体を分割した *TQ* 線図**
- **分割後の流体のうち、結合可能な熱交換器を結合した *TQ* 線図**

create_tq() create_tq_separated() create_tq_split() create_tq_merged() は、プロットに必要な直線を以下のような形式で返します。

```
# [((始点の座標), (終点の座標)), ((始点の座標), (終点の座標)), ...]
lines = [((0, 0), (1, 1)), ((1, 2), (2, 3))]
```

タプルの第一成分が直線の始点の座標、第二成分が終点の座標を表します。また、与熱複合線と受熱複合線をタプルで返します。それぞれを matplotlib.collections.LineCollection に変換後、ax.add_collection を行うことで直線をプロットすることができます。

```
# 複合線を表示
fig, ax = plt.subplots()
hot_lines, cold_lines = analyzer.create_tq()
ax.add_collection(LineCollection(hot_lines))
ax.add_collection(LineCollection(cold_lines))
```

さらに、複合線において、直線が折れ曲がっている点を通る熱量の線をプロットしたい場合、 y_range と extract_x を呼ぶことで、必要な情報を得ることができます。

```
# たて線を表示
ymin, ymax = y_range(hot_lines + cold_lines)
heats = extract_x(hot_lines + cold_lines)
ax.vlines(heats, ymin=ymin, ymax=ymax, linestyles=':', colors='k')
```

extract_x(*lines: list[Line]*) → list[float]

例

```
>>> extract_x([
        ((0, 0), (1, 1)),
        ((1, 1), (2, 2)),
        ((2, 2), (3, 5)),
        ((3, 3), (5, 8))
    ])
>>> [0, 1, 2, 3, 5]
```

y_range(*hot_lines: list[Line]*, *cold_lines: list[Line]*) → tuple[float, float]

例

```
>>> y_range([
        ((0, 0), (1, 1)),
        ((1, 1), (2, 2)),
        ((2, 2), (3, 5)),
        ((3, 3), (5, 8))
    ])
>>> (0, 8)
```

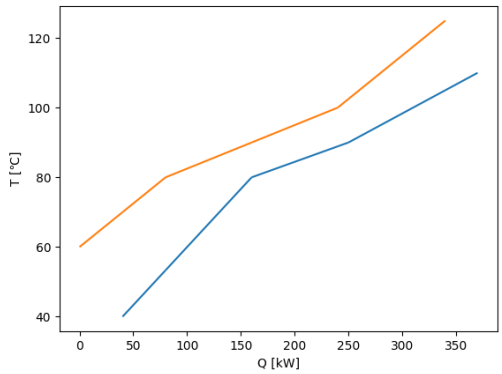
通常の TQ 線図

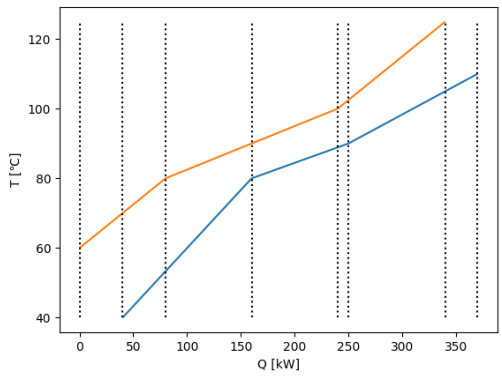
```
hot_lines, cold_lines = analyzer.create_tq()

# 与熱複合線と受熱複合線
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines, colors="#ff7f0e"))
```

(次のページに続く)

(前のページからの続き)

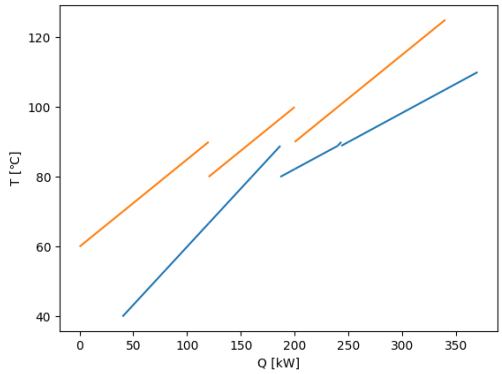
```
ax.add_collection(LineCollection(cold_lines, colors="#1f77b4"))
ax.autoscale()
fig.savefig("path/to/tq_diagram.png")

# 熱量の区間ごとのたて線も表示
ymin, ymax = y_range(hot_lines + cold_lines)
heats = extract_x(hot_lines + cold_lines)
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines, colors="#1f77b4"))
ax.vlines(heats, ymin=ymin, ymax=ymax, linestyles=':', colors='k')
ax.autoscale()
fig.savefig("path/to/tq_diagram_with_vlines.png")
```

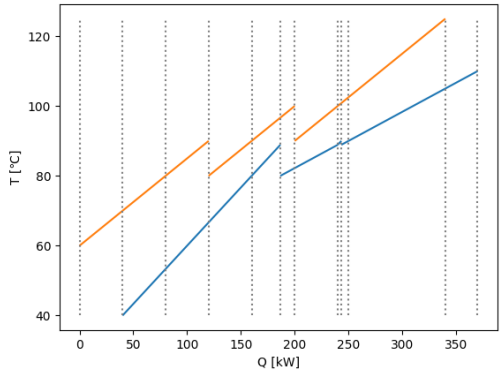




流体ごとに分割した TQ 線図

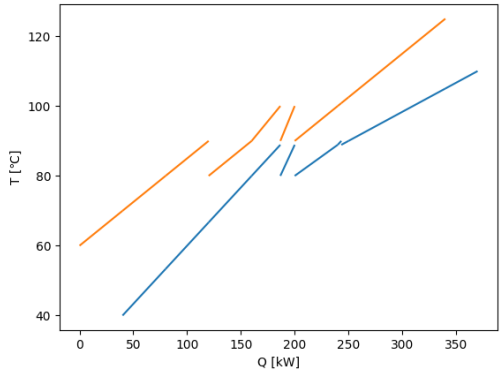
```
hot_lines_separated, cold_lines_separated = analyzer.create_tq_separated()

# 与熱複合線と受熱複合線
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_separated, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_separated, colors="#1f77b4"))
ax.autoscale()
fig.savefig("path/to/tq_diagram_separated.png")

# 熱量の区間ごとのたて線も表示
ymin, ymax = y_range(hot_lines_separated + cold_lines_separated)
heats_separated = extract_x(hot_lines_separated + cold_lines_separated)
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_separated, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_separated, colors="#1f77b4"))
ax.vlines(heats_separated, ymin=ymin, ymax=ymax, linestyles=':', colors='gray')
ax.autoscale()
fig.savefig("path/to/tq_diagram_separated_with_vlines.png")
```

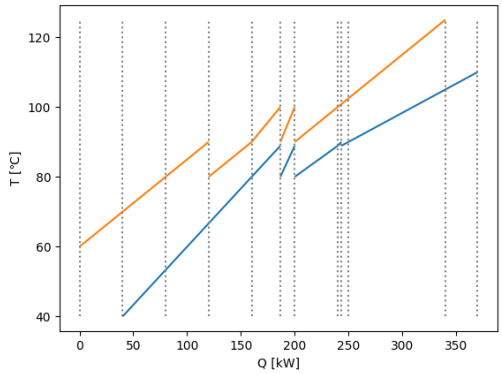




最小接近温度差を満たすように流体を分割した TQ 線図

```
hot_lines_split, cold_lines_split = analyzer.create_tq_split()

# 与熱複合線と受熱複合線
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_split, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_split, colors="#1f77b4"))
ax.autoscale()
fig.savefig("path/to/tq_diagram_split.png")

# 熱量の区間ごとのたて線も表示
ymin, ymax = y_range(hot_lines_split + cold_lines_split)
heats_split = extract_x(hot_lines_separated + cold_lines_separated)
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_split, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_split, colors="#1f77b4"))
ax.vlines(heats_split, ymin=ymin, ymax=ymax, linestyles=':', colors='gray')
ax.autoscale()
fig.savefig("path/to/tq_diagram_split_with_vlines.png")
```





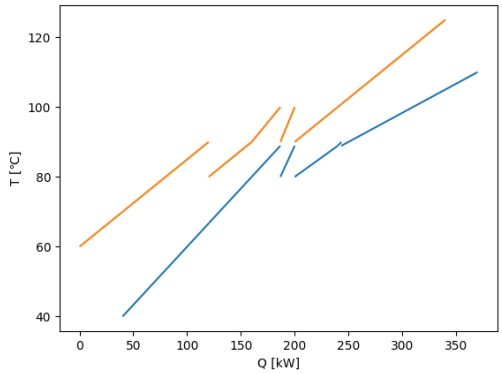
分割後の流体のうち、結合可能な熱交換器を結合した TQ 線図

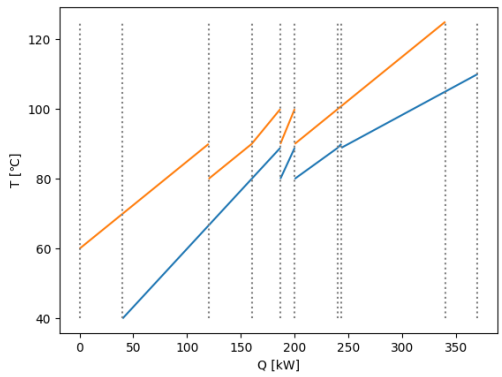
```
hot_lines_merged, cold_lines_merged = analyzer.create_tq_merged()

# 与熱複合線と受熱複合線
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_merged, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_merged, colors="#1f77b4"))
ax.autoscale()
fig.savefig("path/to/tq_diagram_merged.png")

# 熱量の区間ごとのたて線も表示
ymin, ymax = y_range(hot_lines_merged + cold_lines_merged)
heats_merged = extract_x(hot_lines_merged + cold_lines_merged)
fig, ax = plt.subplots(1, 1)
ax.set_xlabel("Q [kW]")
```

(次のページに続く)

(前のページからの続き)

```
ax.set_ylabel("T [℃]")
ax.add_collection(LineCollection(hot_lines_merged, colors="#ff7f0e"))
ax.add_collection(LineCollection(cold_lines_merged, colors="#1f77b4"))
ax.vlines(heats_merged, ymin=ymin, ymax=ymax, linestyles=':', colors='gray')
ax.autoscale()
fig.savefig("path/to/tq_diagram_merged_with_vlines.png")
```





3.1.4 Step4. Export to Excel

注釈: Matplotlib ではなく、Excel を用いてグラフを作成したい方向けの説明です。

グラフを作成する際に matplotlib ではなく Excel を用いたい場合、プロットに必要なデータを加工する必要があります。例で用いる `analyzer` は以下のコードによって作成されたと仮定しています。

```
from pyheatintegration import PinchAnalyzer, Stream
```

(次のページに続く)

(前のページからの続き)

```
streams = [
    Stream(40.0, 90.0, 150.0),
    Stream(80.0, 110.0, 180.0),
    Stream(125.0, 80.0, 180.0),
    Stream(100.0, 60.0, 160.0)
]

minimum_approach_temperature_difference = 10.0
analyzer = PinchAnalyzer(streams, minimum_approach_temperature_difference)
```

convert_to_excel_data を用いることで、直線のリストを x 座標のリストと y 座標のリストに変換することができます。その後、例えば numpy などを用いて CSV ファイルに書き込んだ後に、Excel で処理してください。

```
from pyheatintegration import convert_to_excel_data
import numpy as np

hot_lines, cold_lines = analyzer.create_tq()
hot_lines_excel = convert_to_excel_data(hot_lines)
cold_lines_excel = convert_to_excel_data(cold_lines)

print(hot_lines_excel)
# ([0.0, 40.0, 80.0, 160.0, 240.0, 250.0, 340.0], [60.0, 70.0, 80.0, 90.0, 100.0, 102.5,␣
↪125.0])

print(cold_lines_excel)
# ([40.0, 80.0, 160.0, 240.0, 250.0, 340.0, 370.0], [40.0, 53.33333333333333, 80.0, 88.
↪88888888888889, 90.0, 105.0, 110.0])

np.savetxt("./tq_diagram_hot.csv", np.array(hot_lines_excel).T)
np.savetxt("./tq_diagram_cold.csv", np.array(cold_lines_excel).T)
```

3.2 Heat Exchanger Cost

注釈: グランドコンポジットカーブや TQ 線図の作成方法は *Getting Started* を読んでください。

熱交換器のコストを、以下の式によって推算しています。

$$\mathrm{Cost}[\mathrm{yen}] = 1,500,000 \times A[\mathrm{m}^2]^{0.65} \times k$$

ここで、 A は熱交換器の面積、 k は係数で、熱交換器がリボイラーまたは反応器を流れる流体に対して用いる場合は `2.0` とし、それ以外の場合は `1.0` としています。

熱交換器の面積は、以下の式を用いて求めています。

$$Q = U A \Delta T_{\mathrm{LMTD}}$$

- Q [W]: 伝熱量
- U [$\mathrm{W/m^2 \cdot K}$]: 総括伝熱係数
- A [$\mathrm{m^2}$]: 熱交換器面積
- T_{LMTD} [K]: 対数平均温度差

総括伝熱係数の値は、 プロセスデザインコンテストを参考にして指定しています (**総括伝熱係数の表**)。また、対数平均温度差は向流を仮定して求めています。係数 k は `Stream` のコンストラクタの引数 `reboiler_or_reactor` の値を用いて決定しています。

熱交換器コストの合計は、 `PinchAnalyzer` 経由で取得することが可能です。

```
analyzer.get_heat_exchanger_cost(ignore_unknown=True)
# or
analyzer.get_heat_exchanger_cost(ignore_unknown=False)
```

コスト計算の際に、流体の状態が `StreamState.UNKNOWN` に設定されている場合、総括伝熱係数が設定されていないため、熱交換器の面積を求めることができず、コストを求めることができません。その際に、エラーを生じさせるかどうかを `ignore_unknown` によって設定することが可能です。`True` の場合、エラーは生じず、 `False` の場合、エラーが生じ、プログラムが終了します。デフォルトは `True` に設定されています。

3.3 pinch_analyzer

class pyheatintegration.pinch_analyzer.**PinchAnalyzer**(*streams_: list[pyheatintegration.streams.streams.Stream]*, *minimum_approach_temp_diff: float*, *force_validation: bool = True*)

流体のリストと最小接近温度差を設定し、グランドコンポジットカーブおよび TQ 線図を作成します。

解析を行う場合はこのクラス経由で扱います。このクラスを経由することで、流体の id が重複していないことや、最小接近温度差が指定可能な値であるかを検証したのちに図を作成するため、予想外のエラーが生じることを回避することができます。

パラメータ

- **streams** (*list[Stream]*) -- 流体のリスト。
- **minimum_approach_temp_diff** (*float*) -- 最小接近温度差 [℃]。
- **force_validation** (*bool*) -- 与熱流体と受熱流体の最高温度と最低温度の関係の検証を強制するか。

gcc

グランドコンポジットカーブ。

Type *GrandCompositeCurve*

tq

TQ 線図。

Type *TQDiagram*

streams

流体のリスト。

Type list[*Stream*]

minimum_approach_temp_diff_range

最小接近温度差の指定可能範囲。

Type TemperatureRange

pinch_point_temp

ピンチポイントの温度 [℃]。

Type float

heat_exchangers

熱交換器のリスト。

Type list[*HeatExchanger*]

external_heating_demand
必要加熱量 [W]。

Type float

external_cooling_demand
必要冷却熱量 [W]。

Type float

例外

- **ValueError** -- 流体の id が重複している場合。また、最小接近温度差の値が不正な場合。
- **RuntimeError** -- 受熱流体、与熱流体が一つも指定されていない場合。

create_grand_composite_curve() → tuple[list[float], list[float]]
グランドコンポジットカーブを描くために必要な熱量と温度を返します。

create_tq() → tuple[list[tuple[tuple[float, float], tuple[float, float]]], list[tuple[tuple[float, float], tuple[float, float]]]]
tq 線図をを描くために必要な与熱複合線および受熱複合線を返します。

create_tq_merged() → tuple[list[tuple[tuple[float, float], tuple[float, float]]], list[tuple[tuple[float, float], tuple[float, float]]]]
結合可能な熱交換器を結合した tq 線図をを描くために必要な与熱複合線および受熱複合線を返します。

create_tq_separated() → tuple[list[tuple[tuple[float, float], tuple[float, float]]], list[tuple[tuple[float, float], tuple[float, float]]]]
流体ごとに分割した tq 線図をを描くために必要な与熱複合線および受熱複合線を返します。

create_tq_split() → tuple[list[tuple[tuple[float, float], tuple[float, float]]], list[tuple[tuple[float, float], tuple[float, float]]]]
流体ごとに分割し、最小接近温度差の条件を満たした tq 線図をを描くために必要な与熱複合線および受熱複合線を返します。

get_heat_exchanger_cost(*ignore_unknown: bool = True*) → float

static validate_streams(*streams: list[*pyheatintegration.streams.streams.Stream*]*, *ignore_validation: bool = False*) → str

pyheatintegration.pinch_analyzer.**calculate_heat_exchanger_cost**(*area: float*, *reboiler_or_reactor: bool = False*) → float
熱交換器にかかるコストを返します。

パラメータ

- **area** (*float*) -- 熱交換器の面積。
- **k** (*float*) -- 係数。リボイラーまたは反応器の場合は 2

戻り値 コスト [円]。

戻り値の型 float

3.4 grand_composite_curve

class pyheatintegration.grand_composite_curve.**GrandCompositeCurve**(*streams_: list[pyheatintegration.streams.streams.Stream]*, *minimum_approach_temp_diff: float*)

グランドコンポジットカーブを作成するために必要な情報を得るためのクラス。

パラメータ

- **streams** (*list[Stream]*) -- 熱交換を行いたい流体。
- **minimum_approach_temp_diff** (*float*) -- 最小接近温度差 [℃]。

extarnal_streams

外部流体。

Type list[*Stream*]

minimum_approach_temp_diff

最小接近温度差 [℃]。

Type float

temps

温度のリスト [℃]。

Type list[float]

heats

熱量のリスト [W]。

Type list[float]

pinch_points

ピンチポイントとなるインデックスと温度の tuple のリスト。

Type list[tuple[int, float]]

pinch_point_temp() → float
ピンチポイントの温度を返します。

ピンチポイントが複数ある場合、最小値を返します。

例外 **RuntimeError** -- ピンチポイントが求まっていないかピンチポイントが存在しない場合。

solve_external_heat() → dict[str, float]
外部流体による熱交換量を求めます.

戻り値 流体の id ごとの交換熱量。

戻り値の型 dict[int, float]

例外 **RuntimeError** -- ピンチポイントを求める前に呼び出した場合。

3.5 tq_diagram

class pyheatintegration.tq_diagram.**TQDiagram**(*streams: list[*pyheatintegration.streams.streams.Stream*]*, *minimum_approach_temp_diff: float*, *pinch_point_temp: float*)
TQ 線図を描くために必要な情報を得るためのクラス。

パラメータ

- **streams** (*list[*Stream*]*) -- 熱交換を行いたい流体。
- **minimum_approach_temp_diff** (*float*) -- 最小接近温度差 [℃]。
- **pinch_point_temp** (*float*) -- ピンチポイントの温度 [℃]。

hot_lines
TQ 線図の与熱複合線 (プロット用の直線のリスト)。

Type list[Line]

cold_linse
TQ 線図の受熱複合線 (プロット用の直線のリスト)。

Type list[Line]

hot_lines_separated
流体ごとに分割した与熱複合線 (プロット用の直線のリスト)。

Type list[Line]

cold_lines_separated
流体ごとに分割した受熱複合線 (プロット用の直線のリスト)。

Type list[Line]

hot_lines_split
流体ごとに分割し、最小接近温度差を満たした与熱複合線 (プロット用の直線のリスト)。

Type list[Line]

cold_lines_split
流体ごとに分割し、最小接近温度差を満たした受熱複合線 (プロット用の直線のリスト)。

Type list[Line]

hot_lines_merged
熱交換器を結合した与熱複合線 (プロット用の直線のリスト)。

Type list[Line]

cold_lines_merged
熱交換器を結合した受熱複合線 (プロット用の直線のリスト)。

Type list[Line]

hcc_merged
熱交換器を結合した与熱複合線。

Type list[PlotSegment]

ccc_merged
熱交換器を結合した受熱複合線。

Type list[PlotSegment]

pyheatintegration.tq_diagram.**get_possible_minimum_temp_diff_range**(*streams: list[pyheatintegration.streams.streams.Stream]*, *ignore_validation: bool = False*) → pyheatintegration.ranges.temperature_range.Temp

設定可能な最小接近温度差を返します。

パラメータ

- **streams** (*list[Stream]*) -- 流体のリスト。
- **ignore_validation** (*bool*) -- 最大値のチェックを無視するか。

戻り値 可能な最小接近温度差 [℃]。

戻り値の型 float

3.6 streams

class pyheatintegration.streams.streams.**Stream**(*input_temperature: float*, *output_temperature: float*, *heat_load: float*, *type_: pyheatintegration.enums.StreamType = StreamType.AUTO*, *state: pyheatintegration.enums.StreamState = StreamState.UNKNOWN*, *cost: float = 0.0*, *reboiler_or_reactor: bool = False*, *id_: str = ''*)

熱交換を行う流体を表すクラス。

パラメータ

- **input_temperature** (*float*) -- 入り口温度 [℃]。
- **output_temperature** (*float*) -- 出口温度 [℃]。
- **heat_load** (*float*) -- 熱量 [W]。
- **type_** (StreamType) -- 流体の種類。
- **state** (StreamState) -- 流体の状態。
- **cost** (*float*) -- 流体のコスト [円/J]。外部流体の場合のみ設定可能。
- **reboiler_or_reactor** (*bool*) -- 流体がリボイラーまたは反応器で用いられるかどうか。
- **id_** (*str*) -- 流体を区別する識別子。

id_

流体を区別する識別子。

Type str

temperature_range

温度範囲。

Type TemperatureRange

heat_load

熱量 [W]。

Type float

cost

コスト [円/J]。

Type float

type_
流体の種類。

Type *StreamType*

state
流体の状態。

Type *StreamState*

reboiler_or_reactor
流体がリボイラーまたは反応器で用いられるかどうか。

Type bool

例外 ***InvalidStreamError*** -- 入り口温度と出口温度の大小関係と流体種の関係が不正である場合。また、外部流体の熱量が 0 以外の場合および外部流体以外の流体の熱量が 0 である場合。外部流体以外にコストを設定した場合。

サンプル

```
>>> Stream(0, 20, 300)
Stream(0, 20, 300, type_=StreamType.COLD, state=StreamState.UNKNOWN, cost=0.0,␣
↪reboiler_or_reactor=False, id_="e0c1facb-538b-417f-862c-5cf8043ec075")
>>> Stream(20, 0, 300)
Stream(20, 0, 300, type_=StreamType.HOT, state=StreamState.UNKNOWN, cost=0.0,␣
↪reboiler_or_reactor=False, id_="1a193fc7-9f34-4e6a-8e99-615d40be1b20")
>>> Stream(0, 0, 0)
Traceback (most recent call last):
...
InvalidStreamError: 入り口温度と出口温度が同じ流体の種類は明示的に指定する必要があります。
```

contain_temperature(*temperature: float*) → bool
与えられた温度をとるかを返します。

パラメータ **temperature** (*float*) -- 検証したい温度。

戻り値 与えられた温度をとるかどうか。

戻り値の型 bool

contain_temperatures(*temperatures: collections.abc.Iterable[float]*) → bool
与えられた複数の温度をとるかを返します。

全ての温度をとる場合のみ True を返します。

パラメータ **temperatures** (*Iterable[float]*) -- 検証したい温度のリスト。

戻り値 与えられた複数の温度をとるかどうか。

戻り値の型 bool

heat() → float
熱量を返します。

戻り値 熱量 [W]。

戻り値の型 float

input_temperature() → float
入り口温度を返します。

戻り値 入り口温度。

戻り値の型 float

is_cold() → bool
受熱流体であるかを返します。

戻り値 受熱流体であるかどうか。

戻り値の型 bool

is_external() → bool
外部流体であるかを返します。

戻り値 外部流体であるかどうか。

戻り値の型 bool

is_hot() → bool
与熱流体であるかを返します。

戻り値 与熱流体であるかどうか。

戻り値の型 bool

is_internal() → bool
外部流体以外であるかを返します。

戻り値 外部流体以外であるかどうか。

戻り値の型 bool

is_isothermal() → bool
等温流体かを返します。

戻り値 等温流体であるかどうか。

戻り値の型 bool

output_temperature() → float
出口温度を返します。

戻り値 出口温度。

戻り値の型 float

shift_temperature(*delta: float*) → None
入り口温度と出口温度をずらします。

パラメータ **delta** (*float*) -- ずらす値。

sort_key() → float
ソートの際に用いるキーを返します。

与熱流体は出口温度、受熱温度は入口温度を返します。

戻り値 ソート時にキーとなる値。

戻り値の型 float

temperature() → float
温度変化を返します。

戻り値 温度変化 [℃]。

戻り値の型 float

temperatures() → tuple[float, float]
入り口温度と出口温度を返します。

戻り値 温度範囲。

戻り値の型 tuple[float, float]

update_heat(*heat_load: float*) → None
熱量を更新します。

入り口温度と出口温度が異なる流体に対しては呼び出せません。

パラメータ **heat_load** (*float*) -- 更新する熱量。

例外 **ValueError** -- 等温流体以外に対して熱量を更新しようとした場合。

update_temperature(*input_temperature: float*, *output_temperature: float*) → None
入り口温度と出口温度を更新します。

等温流体に対しては呼び出せません。温度の値に加えて、熱量の値を元々の温度変化と新たな温度変化の比に従って更新します。

パラメータ

- **input_temperature** (*float*) -- 更新する入り口温度 [℃]。
- **output_temperature** (*float*) -- 更新する出口温度 [℃]。

例外 **ValueError** -- 等温流体に対して温度を更新しようとした場合。

pyheatintegration.streams.streams.**get_temperature_range_heats**(*streams: list[*pyheatintegration.streams.streams.Stream*]*) → tuple[list[pyheatintegration.ranges.temperature_range.Te dict[pyheatintegration.ranges.temperature_range.Temp float]]

温度領域ごとに必要な熱量を返します。

パラメータ **streams** (*list[*Stream*]*) -- 流体のリスト。

戻り値 温度領域、温度領域ごとの必要熱量。

戻り値の型 tuple[list[TemperatureRange], dict[TemperatureRange, float]]

pyheatintegration.streams.streams.**get_temperature_range_streams**(*streams: list[*pyheatintegration.streams.streams.Stream*]*) → tuple[list[pyheatintegration.ranges.temperature_range collections.defaultdict[pyheatintegration.ranges.temperatu set[*pyheatintegration.streams.streams.Stream*]]]

温度領域に属する流体を返します。

パラメータ **streams** (*list[*Stream*]*) -- 流体のリスト。

戻り値 温度領域、温度領域に属する流体。

戻り値の型 list[list[TemperatureRange], defaultdict[TemperatureRange, set[*Stream*]]

3.7 heat_exchanger

class pyheatintegration.heat_exchanger.**HeatExchanger**(*heat_range: pyheatintegration.ranges.heat_range.HeatRange*, *hot_plot_segment: pyheatintegration.plots.plot_segment.PlotSegment*, *cold_plot_segment: pyheatintegration.plots.plot_segment.PlotSegment*, *counterflow: bool = True*)

熱交換器を表すクラス。

パラメータ

- **heat_range** (*HeatRange*) -- 熱量領域。
- **hot_plot_segment** (*PlotSegment*) -- 与熱流体のプロットセグメント。
- **cold_plot_segment** (*PlotSegment*) -- 受熱流体のプロットセグメント。

heat_range

熱交換の範囲。

Type HeatRange

hot_stream_uuid

与熱流体の id。

Type str

cold_stream_uuid

受熱流体の id。

Type str

hot_stream_state

与熱流体の状態。

Type *StreamState*

cold_stream_state

受熱流体の状態。

Type *StreamState*

hot_temperature_range

与熱流体の温度領域。

Type TemperatureRange

cold_temperature_range
受熱流体の温度領域。

Type TemperatureRange

lmtd
対数平均温度差。

Type float

hot_plot_segment
与熱流体のプロットセグメント。

Type PlotSegment

cold_plot_segment
受熱流体のプロットセグメント。

Type PlotSegment

reboiler_or_reactor
リボイラーもしくは反応器で用いるか。

Type bool

get_area(*ignore_unknown: bool = True*) → float

init_lmtd_counterflow() → float
向流の場合の対数平均温度差を返します。

戻り値 向流の場合の対数平均温度差。

戻り値の型 float

init_lmtd_pararell_flow() → float
並流の場合の対数平均温度差を返します。

並流が不可能な場合は None を返します。

戻り値 並流の場合の対数平均温度差。並流が不可能な場合は None。

戻り値の型 Optional[float]

pyheatintegration.heat_exchanger.**get_overall_heat_transfer_coefficient**(*hot_stream_state:* pyheatintegration.enums.StreamState, *cold_stream_state:* pyheatintegration.enums.StreamState) → float

対応する総括伝熱係数を返します。

パラメータ

- **hot_stream_state** (StreamState) -- 与熱流体の状態。
- **cold_stream_state** (StreamState) -- 受熱流体の状態。

戻り値 総括伝熱係数 [W/m2.K]

戻り値の型 float

Exapmles:

```
>>> get_overall_heat_transfer_coefficient(StreamState.LIQUID, StreamState.LIQUID)
300.0
```

pyheatintegration.heat_exchanger.**merge_heat_exchangers**(*heat_exchanger:* pyheatintegration.heat_exchanger.HeatExchanger, *other:* pyheatintegration.heat_exchanger.HeatExchanger) → *pyheatintegration.heat_exchanger.HeatExchanger*

熱交換器を結合します。

パラメータ

- **heat_exchanger** (*HeatExchange*) -- 熱交換器。
- **other** (HeatExchanger) -- 熱交換器 (結合対象)。

戻り値 結合後の熱交換器。

戻り値の型 *HeatExchanger*

3.8 enums

class pyheatintegration.enums.**StreamState**(*value*)

流体の状態

GAS = 2

GAS_CONDENSATION = 4

LIQUID = 1

LIQUID_EVAPORATION = 3

UNKNOWN = 5

describe() → str

class pyheatintegration.enums.**StreamType**(*value*)

流体の種類

AUTO = 5

COLD = 1

EXTERNAL_COLD = 3

EXTERNAL_HOT = 4

HOT = 2

describe() → str

3.9 base_range

3.10 errors

exception pyheatintegration.errors.**InvalidMinimumApproachTempDiffError**

Error related to minimum approach temperature difference.

exception pyheatintegration.errors.**InvalidStreamError**

流体に関するエラー

exception pyheatintegration.errors.**PyHeatIntegrationError**

ヒートインテグレーションに由来するエラー

3.11 heat_range

3.12 line

pyheatintegration.line.**convert_to_excel_data**(*lines_: list[tuple[tuple[float, float], tuple[float, float]]]*) → tuple[list[float], list[float]]

直線のリストを x 座標のリストと y 座標のリストに変換します。

パラメータ **lines** (*list[Line]*) -- 直線のリスト

戻り値 x 座標のリストと y 座標のリスト

戻り値の型 typle[list[float], list[float]]

サンプル

```
>>> convert_to_excel_data([((0, 0), (1, 2)), ((1, 2), (3, 3)), ((3, 3), (4, 5))])
([0, 1, 3, 4], [0, 2, 3, 5])
>>> convert_to_excel_data([((0, 0), (1, 2)), ((1, 0), (2, 2))])
([0, 1, 1, 2], [0, 2, 0, 2])
```

pyheatintegration.line.**extract_x**(*lines: list[tuple[tuple[float, float], tuple[float, float]]]*) → list[float]

xy 座標系における複数の直線から重複のない x の値を返します。

パラメータ **lines** (*list[Line]*) -- 直線。

戻り値 x 座標の値。

戻り値の型 list[float]

サンプル

```
>>> extract_x([((0, 0), (1, 1)), ((1, 1), (2, 2)), ((2, 2), (3, 5)), ((3, 3), (5,␣
↪8))])
[0, 1, 2, 3, 5]
```

pyheatintegration.line.**y_range**(*lines: list[tuple[tuple[float, float], tuple[float, float]]]*) → tuple[float, float]

xy 座標系における複数の直線から y の最小値と最大値を返します。

直線は広義単調増加であることを期待しています。

パラメータ **list[Line]** (*lines*) -- 直線。

戻り値 最小値と最大値。

戻り値の型 tuple[float, float]

サンプル

```
>>> y_range([((0, 0), (1, 1)), ((1, 1), (2, 2)), ((2, 2), (3, 5)), ((3, 3), (5,␣
↪8))])
(0, 8)
```

3.13 plot_segment

3.14 segment

class pyheatintegration.segment.**Segment**(*heats: tuple[float, float]*, *hot_temperatures: Optional[tuple[float, float]] = None*, *cold_temperatures: Optional[tuple[float, float]] = None*, *hot_streams_: list[pyheatintegration.streams.streams.Stream] = []*, *cold_streams_: list[pyheatintegration.streams.streams.Stream] = []*)

cold_plot_segments: list[pyheatintegration.plots.plot_segment.PlotSegment]

cold_plot_segments_separated: list[pyheatintegration.plots.plot_segment.PlotSegment]

cold_plot_segments_split: list[pyheatintegration.plots.plot_segment.PlotSegment]

cold_streams: list[*pyheatintegration.streams.streams.Stream*]

cold_temmperature_range: Optional[pyheatintegration.ranges.temperature_range.TemperatureRange]

heat_range: pyheatintegration.ranges.heat_range.HeatRange

heat_ranges: list[pyheatintegration.ranges.heat_range.HeatRange]

hot_plot_segments: list[pyheatintegration.plots.plot_segment.PlotSegment]

hot_plot_segments_separated: list[pyheatintegration.plots.plot_segment.PlotSegment]

hot_plot_segments_split: list[pyheatintegration.plots.plot_segment.PlotSegment]

hot_streams: list[*pyheatintegration.streams.streams.Stream*]

hot_temperature_range:
Optional[pyheatintegration.ranges.temperature_range.TemperatureRange]

init_plot_segments_separated_streams(*streams: list[*pyheatintegration.streams.streams.Stream*]*, *temperature_range: pyheatintegration.ranges.temperature_range.TemperatureRange*) → list[pyheatintegration.plots.plot_segment.PlotSegment]

classmethod round(*x: float*) → float

split(*minimum_approach_temp_diff: float*) → None
最小接近温度差を満たすように熱交換器を分割する。

class pyheatintegration.segment.**Segments**(*initlist=None*)

cold_lines() → list[tuple[tuple[float, float], tuple[float, float]]]

cold_lines_separated() → list[tuple[tuple[float, float], tuple[float, float]]]

cold_lines_split() → list[tuple[tuple[float, float], tuple[float, float]]]

get_lines(*plot_segments: list[pyheatintegration.plots.plot_segment.PlotSegment]*) → list[tuple[tuple[float, float], tuple[float, float]]]

hot_lines() → list[tuple[tuple[float, float], tuple[float, float]]]

hot_lines_separated() → list[tuple[tuple[float, float], tuple[float, float]]]

hot_lines_split() → list[tuple[tuple[float, float], tuple[float, float]]]

split(*minimum_approach_temp_diff: float*) → None

3.15 temperature_range

第 4 章

Indices and tables

- genindex
- modindex
- search

Python モジュール索引

p

索引