Python – Testat 2

1 Einleitung

Bei gewissen Sportarten (z.B. Wandern, Mountainbiken, ...) werden für die Planung der Routen oft topografische Karten eingesetzt. Diese enthalten Informationen über das Gelände und stellen diese in der Regel durch Höhenlinien dar, wie in Abb. 1 abgebildet.

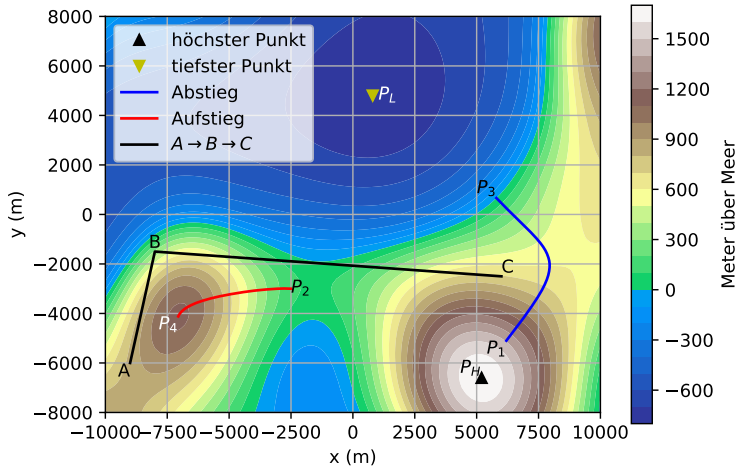

Abbildung 1: Topografische Karte mit Höheninformation.

Es soll eine eigene Klasse `TopoMap` implementiert werden, welche die Höhendaten aus einer Textdatei einliest. Zudem sollen die im Weiteren beschriebenen Methoden implementiert werden, damit die wichtigsten Informationen für Sportler extrahiert werden können.

2 Aufgabenstellung und Bewertungskriterien

☞ *Implementieren Sie die Klasse **TopoMap**, wie unten im Detail beschrieben und reichen Sie diese als Python-Datei dem Dozenten bis spätestens am 22. Mai 2019 um 10 Uhr per Email (nramagna@hsr.ch) ein.*

Bewertungskriterien: Für jedes Detail, das richtig implementiert wird, gibt es Punkte. Rechts neben jeder Detailbeschreibung wird die Zahl der möglichen Punkte in Klammern angegeben. In diesem Testat 2 gibt es total 38 Punkte. Die unten angegebenen Klassen-, Methoden-, Parameter- und Key-Namen müssen exakt übernommen werden, sonst werden für die betroffenen Teile keine Punkte vergeben. Es wird empfohlen, den eigenen Code ausgiebig zu testen. Ein erster Test kann mit dem Beispielcode in den grauen Listing-Boxen oder mit der mitgelieferten Jupyter-Datei durchgeführt werden.

Prüfungszulassung: Um zur Modulschlussprüfung zugelassen zu werden, müssen für beide Testate 1 & 2 insgesamt mindestens 38 Punkte (50% von 76 Punkten) erreicht werden.

3 Klasse `TopoMap`

- Nur die unten aufgelisteten Methoden dürfen *public* sein. (1 P)
- Nebst den unten beschriebenen Properties (`self.X`, `self.Y`, `self.Z`) gibt es keine *public* Instanzvariablen. (1 P)
- Die unten aufgelisteten Methoden besitzen aussagekräftige Docstrings. (4 P)
- Python-Codestil entspricht den PEP8-Empfehlungen. 1 P Abzug pro Stilfehler[1]. (6 P)

3.1 Methode `__init__(self, fname)`

Die Textdatei, welche die gerasterten Höhendaten beinhaltet, wird mit dem `fname`-Argument angegeben. Wie in Abb. 2 dargestellt, besteht die Datei aus mehreren komma-getrennten Zahlenwerten, die in drei Bereiche (rot, blau, grün) eingeteilt werden. Die Zahlen im grünen Bereich stellen die z-Werte (Höhe über Meer) für alle Punkte im Raster dar. In der ersten Zeile (roter Bereich) liegen die entsprechenden x-Werte der Punkte (z.B. $-10000\,\mathrm{m}, \ldots, 10000\,\mathrm{m}$) und in der ersten Spalte (blauer Bereich) liegen die y-Werte der Punkte (z.B. $-8000\,\mathrm{m}, \ldots, 8000\,\mathrm{m}$).

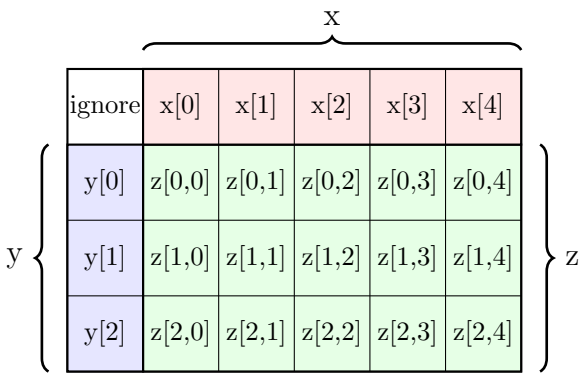

Abbildung 2: Datenstruktur der Kartendatei `map_data.txt`.

- Die Methode liest die angegebene Textdatei ein und speichert die $x/y/z$-Daten intern in nicht-öffentliche Variablen ab. (2 P)
  Hinweis: Benutzen Sie die `numpy.loadtxt()`-Funktion für das Einlesen der Datei.
- Die Kopien (keine Referenzen) der $x/y/z$-Daten können als zweidimensionale NumPy-Arrays über die folgenden Properties erhalten werden: `self.X`, `self.Y`, `self.Z`, d.h. diese Properties haben nur Get-Methoden und keine Set-Methoden. (2 P)
  Hinweis: Erstellen Sie die zweidimensionalen x/y-Arrays mit der `np.meshgrid()`-Funktion.

```
>>> tmap = TopoMap('map_data.txt')
>>> tmap.X.shape, tmap.Y.shape, tmap.Z.shape
((81, 101), (81, 101), (81, 101))
```

```
>>> tmap.X = [0, 1, 2]
AttributeError: can't set attribute
```

[1] der Codestil wird mit der Flake8-Software geprüft, `http://flake8.pycqa.org/en/latest/`

3.2 Methode `highest_point(self)`

- Die Methode liefert ein Tupel von 3D-Koordinaten (x, y, z) des höchsten Punktes auf der Karte. (1 P)

```
>>> tmap.highest_point()
(5200.0, −6600.0, 1680.0)
```

3.3 Methode `lowest_point(self)`

- Die Methode liefert ein Tupel von 3D-Koordinaten (x, y, z) des tiefsten Punktes auf der Karte. (1 P)

```
>>> tmap.lowest_point()
(800.0, 4800.0, −771.2)
```

3.4 Methode `elevation(self, position)`

- Die Methode liefert die interpolierte Höhe z an der angegebenen Position $[x, y]$. (2 P)

```
>>> tmap.elevation([1.1e3, −2.2e3])
array(229.89353933)
```

- Das Argument kann auch ein zweidimensionales NumPy-Array (oder Liste) sein, welches mehrere Positionen $[[x_0, y_0], \ldots, [x_n, y_n], \ldots, [x_N, y_N]]$ beinhaltet. (2 P)

```
>>> tmap.elevation([[1.1e3, −2.2e3], [3.3e3, −4.4e3]])
array([ 229.89353933, 1114.56379161])
>>> tmap.elevation(np.array([[1.1e3, −2.2e3], [3.3e3, −4.4e3]]))
array([ 229.89353933, 1114.56379161])
```

Hinweis: Benutzen Sie die `scipy.interpolate.RectBivariateSpline()`-Klasse[2] und deren `ev()`-Methode[3] für die 2D-Interpolation, ähnlich wie in der Übung 12.

[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RectBivariateSpline.html`

[3] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RectBivariateSpline.ev.html`

3.5 Methode `elevation_profile(self, path)`

Das `path`-Argument definiert einen Pfad auf der Karte und besteht aus einem zweidimensionalen NumPy-Array (oder Liste) von Punkten $[[x_0, y_0], \ldots, [x_n, y_n], \ldots, [x_N, y_N]]$. Das Höhenprofil gibt die Höhen z_n in Abhängigkeit von der Distanz d_n entlang dem angegebenen Pfad an.

Die Distanz d_n bis zum n-ten Punkt $[x_n, y_n]$ berechnet sich aus der kumulierten Summe der Teildistanzen bis zu diesem Punkt:

$$d_0 = 0$$
$$d_n = \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

- Die Methode liefert zwei NumPy-Arrays als Rückgabewert: (2 P)
  - Das erste Array beinhaltet die Distanzen d_n $(n = 0, \ldots, N)$ und
  - das zweite Array beinhaltet die interpolierten Höhen z_n $(n = 0, \ldots, N)$.

```
>>> path = np.array([[0, −2e3], [1e3, −2e3], [1e3, −3e3], [0, −3e3]])
>>> dn, zn = tmap.elevation_profile(path)
>>> dn
array([   0., 1000., 2000., 3000.])
>>> zn
array([ 70., 170., 347., 186.])
```

3.6 Methode `distance_travelled(self, path)`

Zwei Kennzahlen sind beim Radsport besonders wichtig: die totale Länge des zurückgelegten Weges und die überwundenen Höhenmeter.

- Die totale Weglänge d berechnet sich aus der Summe aller 3D-Teildistanzen entlang dem Pfad, der aus den Punkten $[[x_0, y_0], \ldots, [x_n, y_n], \ldots, [x_N, y_N]]$ besteht:

  $$d = \sum_{i=1}^{N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2 + (z_i - z_{i-1})^2}\ ,$$

  wobei z_i die interpolierte Höhe am Punkt $[x_i, y_i]$ ist.

- Für die Berechnung der Höhenmeter werden nur die positiven Höhendifferenzen $(z_i - z_{i-1}) \geq 0$ aufsummiert. Die Abwärtsbewegungen werden ignoriert, da diese üblicherweise weniger bedeutend sind.

- Die Methode liefert zwei Zahlen zurück: (3 P)
  - die Länge des zurückgelegten Weges und
  - die Summe der positiven Höhendifferenzen.

```
>>> path = np.array([[0, −2e3], [1e3, −2e3], [1e3, −3e3], [0, −3e3]])
>>> d, hm = tmap.distance_travelled(path)
>>> d
3033.408842774067
>>> hm
277.0
```

3.7 Methode `fall_line(self, start, descent=True, gamma=50, max_iter=10000, precision=0.1)`

Beim Mountainbiken oder Skifahren ist man an der Falllinie eines Hanges interessiert. Die Falllinie ist jene Linie auf einer Oberfläche, die der Richtung des grössten Gefälles (Gradient) folgt. Sie schneidet die Höhenlinien stets rechtwinklig. Der Gradient der Geländeoberfläche $Z(x, y)$ ist definiert als:

$$\text{grad}(Z) = \frac{\partial Z}{\partial x}\,\hat{\mathbf{x}} + \frac{\partial Z}{\partial y}\,\hat{\mathbf{y}} = \begin{bmatrix} \frac{\partial Z}{\partial x} \\ \frac{\partial Z}{\partial y} \end{bmatrix},$$

wobei $\hat{\mathbf{x}}$ und $\hat{\mathbf{y}}$ die Basisvektoren des Koordinatensystems sind. Weiter sind $\partial Z/\partial x$ und $\partial Z/\partial y$ die Steigungen (Ableitungen) in x- bzw. y-Richtung.

Hinweis: In NumPy kann der numerische Gradient mit der `numpy.gradient()`-Funktion[4] berechnet werden:

```
>>> dy = np.abs(y[1] - y[0])   # Abstand in y-Richtung
>>> dx = np.abs(x[1] - x[0])   # Abstand in x-Richtung
>>> dZdy, dZdx = np.gradient(Z, dy, dx)   # y: axis=0, x: axis=1
```

Die Punkte entlang der Falllinie können mittels dem sogenannten Gradientenverfahren gefunden werden:

1. Beim `start`-Punkt $[x_0, y_0]$ starten.
2. Die interpolierten Steigungen $\partial Z/\partial x$ und $\partial Z/\partial y$ am aktuellen Punkt $[x_i, y_i]$ bestimmen. **Hinweis:** Benutzen Sie wieder die `scipy.interpolate.RectBivariateSpline()`-Klasse und deren `ev()`-Methode für die 2D-Interpolation.
3. Nächster Punkt $[x_{i+1}, y_{i+1}]$ in entgegengesetzte Richtung des Gradienten verschieben:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} - \gamma \begin{bmatrix} \frac{\partial Z}{\partial x} \\ \frac{\partial Z}{\partial y} \end{bmatrix},$$

   wobei γ der Schrittweiten-Faktor ist, der mit dem `gamma`-Argument definiert werden kann.
4. Das Verfahren anhalten, falls eine der folgenden Bedingungen erfüllt ist:
   - Die max. Anzahl der Iterationen (`max_iter`-Argument) wurde erreicht.
   - Die Schrittweite ist genügend klein: $\gamma \cdot (|\partial Z/\partial x| + |\partial Z/\partial y|) <$ `precision`-Argument.
   - Der Kartenrand wurde erreicht.
   - Das Meer wurde erreicht ($Z \leq 0$).

   Sonst nach oben zum Punkt 2 springen und weitermachen.

- Die Methode liefert den Pfad als zweidimensionales NumPy-Array von Punkten $[[x_0, y_0], \ldots, [x_n, y_n], \ldots, [x_N, y_N]]$, wobei $n = 0, \ldots, N$ ist. (6 P)
- Mit dem Argument `descent=False` wird die Richtung invertiert, d.h. bergauf anstatt bergab steigen. (1 P)

```
>>> P1 = [6200, -5100]
>>> path = tmap.fall_line(P1)
>>> path[-1]  # letzter Punkt ist vor dem Wasser
array([5799.35851985,  661.59193054])
```

[4] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.gradient.html`

3.8 Methode `aligned_map(self, p, q)`

Manchmal kann es nützlich sein, eine Karte neu auszurichten. Dafür werden zunächst zwei Referenzpunkte $\mathbf{P} = [p_x, p_y]$ und $\mathbf{Q} = [q_x, q_y]$ definiert:

- Der Punkt **P** soll der neue Ursprung des Koordinatensystems sein. Dazu müssen alle Kartenpunkte entsprechend verschoben werden.
- Der Punkt **Q** soll anschliessend genau im Norden des neuen Ursprungs zu liegen kommen, wie in Abb. 3 dargestellt. Dafür muss die bereits verschobene Karte noch um den Winkel φ gedreht werden.

Hinweis: Ein Punkt $[x_n, y_n]$ kann wie folgt mit der Rotationsmatrix **A** um den Ursprung gedreht werden:

$$\begin{bmatrix} \tilde{x}_n \\ \tilde{y}_n \end{bmatrix} = \mathbf{A} \begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \qquad (*)$$

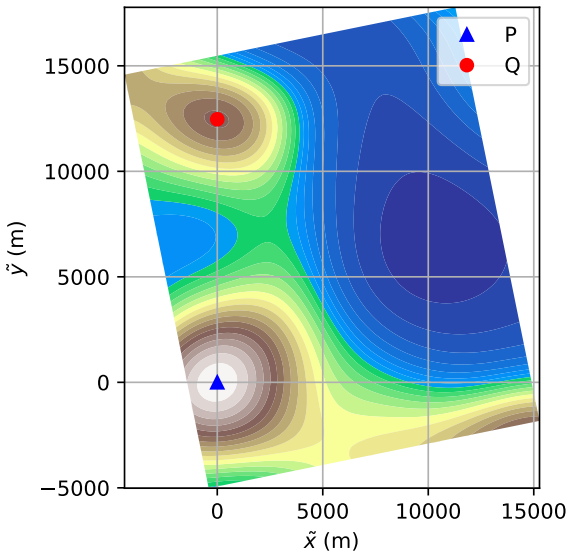

Abbildung 3: Die an den Punkten **P** und **Q** ausgerichtete Karte.

- Die Methode liefert vier zweidimensionale NumPy-Arrays als Rückgabewerte: (4 P)
  - `X_new`: Array der neuen x-Werte (rotierte Kartenpunkte)
  - `Y_new`: Array der neuen y-Werte (rotierte Kartenpunkte)
  - `Z_new`: die kopierten z-Werte der Kartenpunkte
  - `A`: die Rotationsmatrix (gemäss (*) eine 2×2-Matrix)

```
>>> p = [5165, −6601]
>>> q = [−7052, −4115]
>>> X_new, Y_new, Z_new, A = tmap.aligned_map(p, q)
>>> X_new.shape, Y_new.shape, Z_new.shape, A.shape
((81, 101), (81, 101), (81, 101), (2, 2))
```