Essay „Perfect Peak“

in „Python – If error: continue“

Modul des Masterstudienganges „Physische Geographie“

Philipps-Universität Marburg, WS 2017/18

von Laura Giese & Johannes Schnell

Datum: 09.04.2018

Einführung

Der „perfekte Gipfel“ (engl. perfect peak) für den ambitionierten Bergsteiger kann nach C. RAUCH (2012) anhand des *Eigenständigkeitswertes*, einem Maß für die „Einzigartigkeit“ eines Berges in Bezug auf die umliegende Landschaft, berechnet werden. Die Berechnung basiert dabei auf den orographischen Kennzahlen *Dominanz* und *Prominenz* (vgl. Abb. 1). Im Zuge dieser Arbeit wird eine Funktion zur Berechnung des Eigenstands eines Berges mithilfe eines Digitalen Höhenmodells (DGM) in der Computersprache *Python* entwickelt. Abbildung 1 stellt das konzeptionelle Vorgehen dabei schematisch dar.

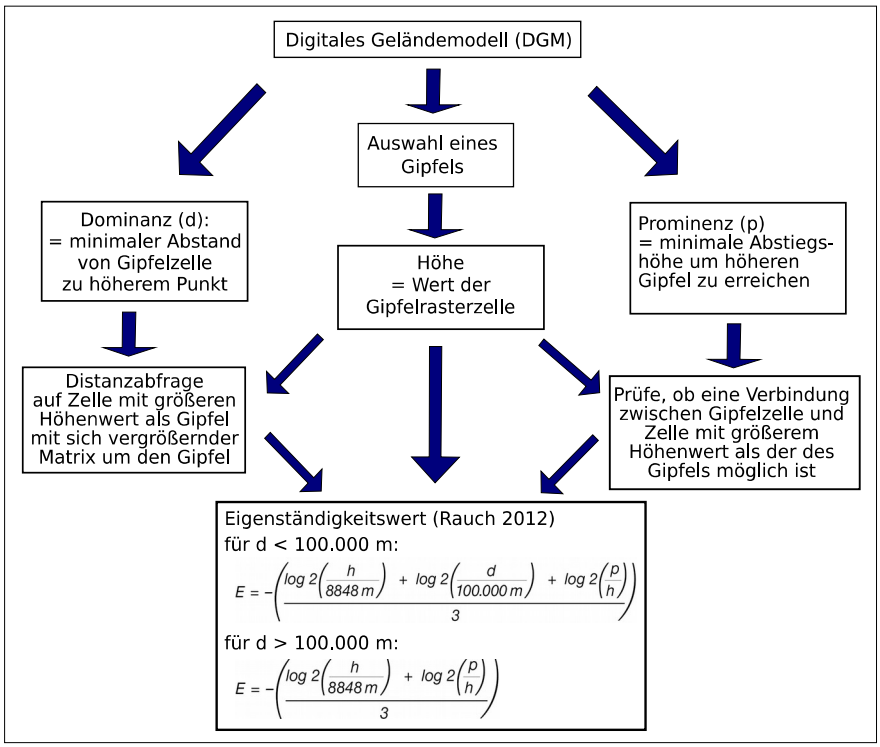

Abb. 1: Workflow

Erläuterung des Codes

Erste Schritte - „Höhenwert“

Zunächst wird in einem DGM ein Gipfel identifiziert. Die Höhe des Gipfels ist die Basisinformation des DGM und wird diesem entnommen. Für den folgenden Ansatz wird das DGM in einen *Array* umgewandelt (vgl. Codeblock 1). Die Zeilen- und Spaltennummer des betreffenden Gipfels im *Array* wird mithilfe der Koordinaten- oder der Höheninfromation ermittelt. In dieser Arbeit wurde die zweite Vorgehensweise gewählt.

```
#%%
import math
import gdal, ogr, osr, os
import numpy as np
import time

#%%
# nach https://pcjericks.github.io/py-gdalogr-
cookbook/raster_layers.html#replace-no-data-value-of-raster-with-new-value
def raster2array(rasterfn):
    raster = gdal.Open(rasterfn)
    band = raster.GetRasterBand(1)
     return band.ReadAsArray()
```

Codeblock 1: Importieren der benötigten Bibliotheken und Umwandlung des Rasterdatensatzes zu einem *Array* (natives Python Format)

„Dominanz“

Zur Bestimmung der *Dominanz*, dem Abstand zum nächstgelegenen Punkt, der höher als der Gipfel liegt, wird eine *nxn*-Matrix (*n:* ungerade Zahl) mit dem Gipfel als Zentrum auf das DGM gelegt. Es wird überprüft, ob innerhalb dieser Matrix mit Höhenwerten ein Wert existiert, der größer ist, als jener, der den Gipfel repräsentiert. Sobald ein Pixel mit größerem Wert entdeckt wird, muss die Matrixgröße um den Faktor $\sqrt{2}$ vergrößert werden. Der Grund dafür ist, dass nicht kreisförmig, sondern quadratisch gesucht wird.

Beispiel: Der Abstand des Punktes 50|50 vom Ursprung (0|0) ist mit einem Wert von 70,71 größer als der des Punktes 0|70 (Abstand: 70).

Anschließend werden alle Distanzen zwischen Gipfelpixel und Pixeln mit größeren Höhenwerten gebildet und die kleinste wird mit der Zellengröße multipliziert. Die Funktion *bestimmung()* (vgl. Codelblock 2) verhindert, dass die Matrix über die Größe des Arrays hinausgeht.

```
#%%
def distanz(z,s,y,x):
    dis = math.sqrt((z-y)**2 + (s-x)**2)
    return(dis)
#%%
def bestimmung(z,s,dgm_ar, step):
    if z-step < 0:
        y1 = 0
    else:
        y1 = z-step
    if z+step >= len(dgm_ar):
        y2 = len(dgm_ar)-1
    else:
        y2 = z+step
    if s-step < 0:
        x1 = 0
    else:
        x1 = s-step
    if s+step >= len(dgm_ar[0]):
        x2 = len(dgm_ar[0])-1
    else:
        x2 = s+step
    return(y1,y2,x1,x2)
#%%
def dominanzB(z,s, dgm_ar, step, res):
    end_list = []
    while len(end_list) == 0:
        b = bestimmung(z,s,dgm_ar,step)
        for y in range(b[0], b[1]+1):
          for x in range(b[2], b[3]+1):
              if dgm_ar[z][s] < dgm_ar[y][x]:
                  dissi = distanz(z,s,y,x)
                  end_list.append(dissi)
        step = step+1
    end_list = []
    step = int((step-1) * math.sqrt(2)) + 1
    b = bestimmung(z,s,dgm_ar,step)
    for y in range(b[0], b[1]+1):
        for x in range(b[2], b[3]+1):
            if dgm_ar[z][s] < dgm_ar[y][x]:
                  dissi = distanz(z,s,y,x)
                  end_list.append(dissi)
    if len(end_list)> 0:
        #print(end_list)
        print("dominanz: ", min(end_list)*res)
     return(min(end_list)*res)
```

Codeblock 2: Funktionen zur Berechnung der *Dominanz dominanzB()* und die zugehörige Hilfsfunktion *bestimmung()*

„Prominenz“

Die Berechnung der *Prominenz*, also der minimalen Abstiegshöhe, die überwunden werden muss, um einen höheren Gipfel zu erreichen erfolgt nach dem Ansatz "Badewanne leeren". Der Ansatz wird im Folgenden an einem Gedankenbeispiel erläutert:

Gedankenbeispiel: Das Gebirge wird bis zur Gipfelspitze mit Wasser „aufgefüllt“. Dann wird schrittweise Wasser abgelassen und überprüft, ob man "trockenen" Fußes einen höheren Gipfel erklimmen kann.

Dies wird wie folgt im Skript umgesetzt: Die *canigo_fill()*-Funktion (vgl. Codeblock 3a,b) überprüft in einer 3x3-Matrix um jeden bekannten Punkt (Variable: *kennichschon*) ob diese Pixel kleinere Werte aufweisen als der Gipfel und größer sind als der aktuelle Wasserstand (Variable: *fill*). Trifft beides zu werden die Koordinaten in die Variable *neu* geschrieben. Anschließend wird diese mit *kennichschon* verglichen, um die noch nicht bekannten Koordinaten zu ermitteln und diese als neue *start*-Punkte zu setzen. Die Schleife endet, wenn ein Pixel einen größeren Höhenwert hat als der Gipfelpixel.

Bei einem ersten Ansatz wurde gleichmäßig "das Wasser aus der Badewanne" gelassen (vgl. Codeblock 3a). Problematisch dabei ist die Rechenzeit. Ein schnellerer Ansatz ist eine alternierende Methode (vgl. Codeblock 3b), bei der zunächst in beispielsweise 128 m Schritten (hier sind Werte zu wählen, die Teil der 2er Potenzreihe sind) das „Wasser” abgelassen wird. Aus dem ersten Wert, welcher *TRUE* zurückgibt und somit größer ist als die Höhe des ursprünglich gewählten Gipfels und aus der Hälfte der Schrittgröße (in diesem Beispiel: ½ x 128 = 64) wird die Summe gebildet. Das Ergebnis entspricht dem neuen „Füllstand“. Daraufhin wird abermals geprüft, ob ein höherer Gipfel „trockenen Fußes“ erreichbar ist. Sollte wieder *TRUE* zurückgegeben werden, wiederholt sich der Vorgang mit Schrittgröße 32 m. Wird *FALSE* zurückgegeben, werden 32 m vom letzten *fill*-Wert subtrahiert und erneut in *canigo_fill()* eingesetzt. Das Resultat ist die Höhe des zu überschreitenden Grates. Dieser Wert muss abschließend von der Gipfelhöhe abgezogen werden, um die *Prominenz* im Sinne der Formel für den *Eigenständigkeitswert* zu ermitteln.

```
#%%
def bestimmung_cfill(start,ar, st):
    if start[st][0]-1< 0:
        y1 = 0
    else:
        y1 = start[st][0]-1
    if start[st][0]+2 >= len(ar):
        y2 = len(ar)-1
    else:
        y2 = start[st][0]+2
    if start[st][1]-1 < 0:
        x1 = 0
    else:
        x1 = start[st][1]-1
    if start[st][1]+2 >= len(ar[0]):
        x2 = len(ar[0])-1
    else:
        x2 = start[st][1]+2
    return(y1,y2,x1,x2)
```

```
#%%
def canigo_fill(start, ar, fill, startval):
    kennichschon=start[:]
    neu=start[:]
    #neustart=start
    while len(neu) > 0:
        neu=[]
        for st in range(len(start)):
            b = bestimmung_cfill(start,ar,st)
            for y in range(b[0], b[1]):
                for x in range(b[2], b[3]):
                    if ar[y][x] > ar[startval[0]][startval[1]]:
                        #print("groesseren Punkt gefunden")
                        return(True,(y,x))
                    if ar[y][x] > fill:
                        neu.append((y,x))
                        neu = list(set(neu))
        start=[]
        for i in range(len(neu)):
            if neu[i] not in kennichschon:
                start.append((neu[i][0], neu[i][1]))
                start = list(set(start))
        kennichschon = kennichschon + start
        #kennichschon = list(set(kennichschon))
        #print(len(neu), len(kennichschon), len(start))
    else:
        #print("kein weiterer Gipfel")
        return(False, kennichschon)
        #start = neustart
        #return(False, start)

#%%
def prominenz(start, ar, dwn_stp):
    fillv = ar[start[0][0]][start[0][1]]
    startval = (start[0][0], start[0][1])
    gipfel = False
    while gipfel == False:
        fillv = fillv - dwn_stp
        print(fillv)
        t1 = time.clock()
        gval = canigo_fill(start, ar, fillv, startval)
        start = gval[1]
        gipfel = gval[0]
        t2 = time.clock()
        print(t2-t1)
    return(fillv, gval[1])
```

Codeblock 3a: Funktion *prominenz()* zur Berechnung der *Prominenz*. Die Hilfsfunktion *canigo_fill()* tastet die Umgebung der Pixel nach und nach ab und prüft, ob sie als mögliche „Verbindungspixel“ gelten. *bestimmung_cfill()* übernimmt hier eine ähnliche Funktion wie *bestimmung().*

Mit alternierendem Ansatz:

```
#%%


def prominenz(start, ar, dwn_step):
    fillv = ar[start[0][0]][start[0][1]]
    startval = (start[0][0], start[0][1])
    gipfel = False
    fillserie = [int(dwn_step/2**y) for y in range(int(math.log(x,2)+1))]
    while gipfel == False:
        fillv = fillv - fillserie[0]
        print(fillv)
        t1 = time.clock()
        gval = canigo(start, ar, fillv, startval)
        start = gval[1]
        gipfel = gval[0]
        neuer_fill = gval[2]
        print(neuer_fill, "im while")
        t2 = time.clock()
        print(t2-t1)
    serie_c = fillserie[1:]
    neuer_fill = neuer_fill + fillserie[1]
    print(neuer_fill, "vor for")
    for i in range(len(serie_c)-2):
        if i == 0:
            gval_sec = canigo([(startval[0], startval[1])], ar,neuer_fill,
startval)
            print(gval_sec[2], gval_sec[0])
            start = gval_sec[1]
        else:
            gval_sec = canigo([(startval[0], startval[1])], ar,neuer_fill,
startval)
            print(gval_sec[2], gval_sec[0])

        if gval_sec[0] == True:
            neuer_fill = gval_sec[2] + serie_c[i+1]
            print("ping", serie_c[i+1], neuer_fill)
        if gval_sec[0] == False:
            neuer_fill = gval_sec[2] - serie_c[i+1]
            print("pong", serie_c[i+1], neuer_fill)
     return(neuer_fill, "")
```

Codeblock 3b: Funktion *prominenz()* zur Berechnung der *Prominenz*. Die Hilfsfunktion *canigo_fill()* tastet die Umgebung der Pixel nach und nach ab und prüft, ob sie als mögliche „Verbindungspixel“ gelten. *bestimmung_cfill()* übernimmt hier eine ähnliche Funktion wie *bestimmung().*

Letzte Schritte

Abschließend dient die Funktion *eigenstand()* im Codeblock 4 nach C. RAUCH (2012) und der Wrapper *estand()*, welcher alle Funktionen zusammenfasst, zur Berechnung des *Eigenständigkeitswerts*.

```
def eigenstand(h, d, p):
  if d < 100000:
    E1 = math.log(h/8848, 2) + math.log(d/100000, 2) + math.log(p/h, 2)
    E = -(E1/3)
  else:
    E1 = math.log(h/8848, 2) + math.log((p/h), 2)
    E = -(E1/3)
  return(E)

#%%
def estand(y,x,ar,step,res, dwn_stp):
    t1=time.clock()
    h = float(ar[y][x])
    print("start dom")
    d = dominanzB(y,x,ar,step,res)
    t3 = time.clock()
    print("dauer ", t3-t1, "start prom")
    p1 = prominenz([(y,x)], ar, dwn_stp)
    p = h-p1[0]
    t4 = time.clock()
    print("dauer: ", t4-t3, "hoehe: ", h, "dominanz: ", d, "prominenz: ", p)
    t2=time.clock()
    print("eigenstand: ", eigenstand(h,d,p), "in:", (t2-t1)/3600," stunden")
     return(eigenstand(h,d,p), p1[1])
```

Codeblock 4: Berechnung des *Eigenständigkeitswerts* mit *eigenstand()* und Wrapper-Funktion *estand*()

Fazit

Die Berechnungszeit der *Prominenz* ist um so höher, je *prominenter* der Berg ist. Sucht man sich visuell einen Gipfel heraus tendiert man dazu, besonders *prominente,* beziehungsweise *dominante* Berge zu nehmen. Auf hochaufgelösten DGM kann eine Berechnung mehrere Stunden bis Tage dauern. Es empfiehlt sich daher, die Auflösung der DGM entsprechend zu skalieren. Allerdings wird beim Verändern der Auflösung unter Umständen auch der Höhenwert verändert. Man könnte die Funktion noch optimieren, indem man die Höhe und die *Dominanz* dem DGM mit der höchsten Auflösung entnimmt beziehungsweise berechnet, und die *Prominenz* mithilfe eines gering aufgelösten DGM ermittelt. Die hier berechneten Werte für den Eigenstand unterscheiden sich meist erst ab der zweiten bis dritten Nachkommastelle von Literaturwerten, was für Zwecke, wie beispielsweise dem Bergsteigen, mehr als ausreichend ist.

Eine Zusammenfassung von Ergebnissen, sowie der Vergleich mit Literaturwerten ist in der Datei „bsp.py“ zu finden. Dort sind auch die Code-Teile zum Einladen der .TIF-Dateien enthalten. Zum Einladen der Funktionen sind „eigenstand.py“ und „altern.py“ vorgesehen.

Es empfiehlt sich zuerst „eigenstand.py“ einzuladen, dann die Beispiele durchzugehen und abschließend „altern.py“ einzuladen, um die alternierende Variante ebenfalls zu sehen. Dabei wird die Funktion *prominenz()* überschrieben. Wichtig ist außerdem in den Beispielen die variable *dwn_step* (unterstrichen) entsprechend zu ändern:

```
estand(367,227, ar, 1, 200, 1) gleichmäßig
wird zu
estand(367,227, ar, 1, 200, 128) alternierend
```

Die entsprechenden Skripte und Rasterdaten, die in den Beispielen verwendet werden, sind unter

https://github.com/logmoc/msc-phygeo-class-of-2017-schnell7-1/tree/master/Python/Abgabe

zu finden. Dieser Abgabe sind außerdem die Beispiele des Harzes, Taunus und des Kufsteins angehängt.

Quellen und weiterführende Literatur

BUNDESAMT FÜR KARTOGRAPHIE UND GEODÄSIE 1950-2016: DGM200. GeoBasis-DE / BKG. (Bezug: 2018, Daten verändert). Abrufbar unter: http://www.geodatenzentrum.de/geodaten/gdz_rahmen.gdz_div?gdz_spr=deu&gdz_akt_zeile=5&gdz_anz_zeile=1&gdz_unt_zeile=3&gdz_user_id=0.

ERICKSON, J., DANIEL, C., PAYNE, M. (2013): Python GDAL/OGR Cookbook 1.0. Abrufbar unter: https://pcjericks.github.io/py-gdalogr-cookbook/raster_layers.html#replace-no-data-value-of-raster-with-new-value. Stand: 15.03.18.

RAUCH, C. (2012): Der perfekte Gipfel. In: DAV Panorama H. 2. Seite 112-117.