SPRAWOZDANIE Z ĆWICZENIA 3:
Przeniesienie współrzędnych geodezyjnych na powierzchni elipsoidy obrotowej

Maja Kret
nr 1, gr. 2
325693

Wydział Geodezji i Kartografii
Politechnika Warszawska

Warszawa, 20 grudnia 2023

Spis treści

1 Cel ćwiczenia

Celem ćwiczenia jest przeniesienie współrzędnych geodezyjnych na powierzchni elipsoidy obrotowej oraz wizualizacja i analiza ich na mapie.

2 Wstęp teoretyczny

2.1 Algorytm Kivioja

Algorytm Kivioja umożliwia przeniesienie współrzędnych geodezyjnych, czyli wyznaczenie współrzędnych punktu końcowego oraz azymutu na końcu odcinka linii geodezyjnej. Dane do wykonania obliczenia to współrzędne punktu początkowego, azymut do punktu końcowego oraz odległość geodezyjna pomiędzy punktami. Algorytm wykorzystuje metodę całkowania numerycznego - dzieli linię geodezyjną na n krótkich odcinków, a następnie iteracyjne oblicza przyrosty współrzędnych kolejnych punktów oraz korekty azymutu aż do punktu końcowego. Współrzędnymi punktu końcowego jest suma przyrostów współrzędnych.

2.2 Algorytm Vincentego

Algorytm Vincentego jest bardziej złożony od algorytmu Kivioja. Rozwiązuje on problem przeciwny, ponieważ za pomocą współrzędnych dwóch punktów oblicza azymut oraz odległość pomiędzy nimi. W tym ćwiczeniu użyto go do korekty współrzędnych obliczonych algorytmem Kivioja.

3 Dane do ćwiczenia

φ_1	$53°45'00.00000''$
λ_1	$15°15'00.00000''$

Tabela 1: Współrzędne punktu początkowego P_1

nr	długość $s[km]$	azymut $A[°]$
1 - 2	40	$0°00'00.000''$
2 - 3	100	$90°00'00.000''$
3 - 4	40	$180°00'00.000''$
4 - 1	100	$270°00'00.000''$

Tabela 2: Parametry linii geodezyjnych

4 Przebieg ćwiczenia

1. **Wyznaczenie współrzędnych punktów metodą Kivioja:** Punkty 2, 3, 4 i 1* zostały obliczone poprzez zastosowanie funkcji `kivioj` (kod 2), która za argumenty przyjmuje współrzędne punktu początkowego, azymut do kolejnego punktu, długość linii pomiędzy punktami oraz liczbę iteracji ustawioną na 1000.
2. **Analiza punktów na mapie:** Przedstawiono obliczone punkty na mapie i narysowany figurę, którą wyznaczają (kod 2). Następnie obliczono odległość pomiędzy punktami 1 i 1*.
3. **Zastosowanie algorytmu Vincentego:** Z użyciem uzyskanych współrzędnych punktu 4 oraz punktu początkowego, obliczono faktyczną odległość pomiędzy punktem 4 a 1 oraz azymut 4 - 1 i odwrotny (funkcja `vincenty`, kod 3). Następnie zastosowano te parametry do zamknięcia trapezu - wyliczenia punktu 1* algorytmem Kivioja, który będzie się pokrywał z punktem 1.
4. **Wizualizacja na mapie:** Zaznaczono obliczone punkty na mapie oraz narysowano zamkniętą figurę. Do stworzenia map oraz tabel użyto biblioteki `plotly.graph_objects`
5. **Obliczenie pola i obwodu figury:** Do obliczenia pola i obwodu figury użyto funkcji `geometry_area_perimeter` z biblioteki `pyproj` (kod 4).

5 Wyniki ćwiczenia

5.1 Algorytm Kivioja

Współrzędne punktów obliczone algorytmem Kivioja

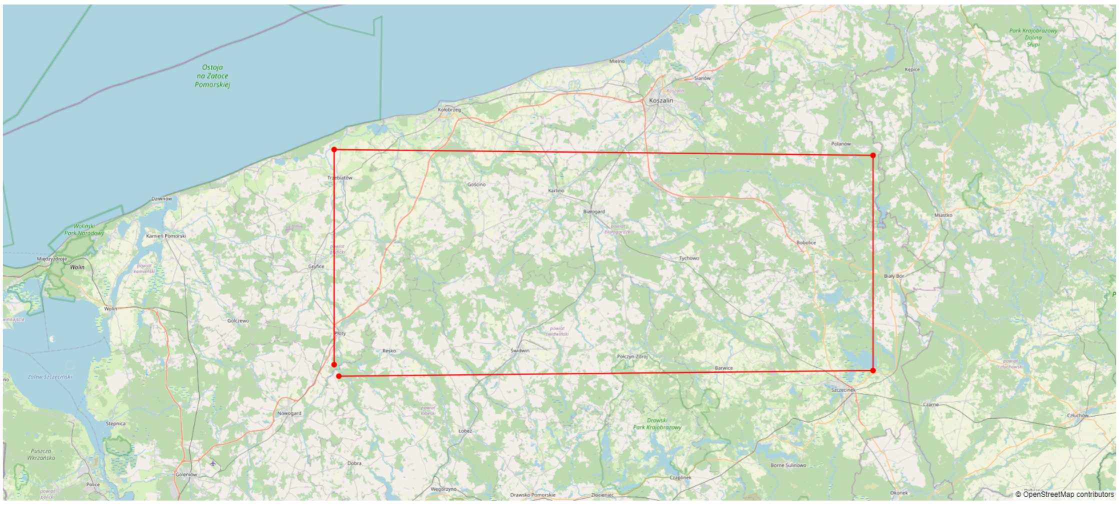
nr	phi	lambda	azymut
1	53° 45' 00.00000''	15° 15' 00.00000''	0° 00' 00.00000''
2	54° 06' 33.75763''	15° 15' 00.00000''	0° 00' 00.00000''
3	54° 05' 58.80144''	16° 46' 43.41406''	91° 14' 18.34038''
4	53° 44' 25.04170''	16° 46' 43.41406''	180° 00' 00.00000''
1*	53° 43' 50.55252''	15° 15' 48.31182''	268° 46' 41.48882''

Rysunek 1: Współrzędne punktów obliczone algorytmem Kivioja

W tabeli 1 przedstawiono obliczone współrzędne punktów - szerokość i długość geodezyjną oraz azymut na końcu linii geodezyjnej.

Na mapie wyraźnie widać, że punkty nie zamykają się w czworokąt, a punkt 1* nie pokrywa się z początkowym. Punkt 1* znajduje się $1'9.44748''$ na południe i $46.31182''$ na wschód od punktu 1. Co za pomocą funkcji `line_length` można wyliczyć na odległość aż $2322.516m$, czyli ponad $2.3km$. Rysowane boki trapezu nie są do siebie prostopadłe, z obliczonych azymutów wynika, że linia 2 - 3 była narysowana pod kątem mniejszym niż 90°, a linia przeciwna 4 - 1* pod kątem większym niż 90° w stosunku do poprzednich linii. Natomiast w przypadku linii o długości 40 km, czyli 1* - 2 oraz 3 - 4, azymut przy końcu linii pozostał taki sam i nie doszło tu do żadnej korekty.

Scattermapbox - Algorytm Kivioja



Rysunek 2: Otrzymana figura

5.2 Algorytm Vincentego

Algorytm Vincentego

A - B	Odl. AB	Az. AB	Az. odw. AB
4 - 1	100862.55117m	-88° 46' 10.88591''	89° 59' 51.09899''

Rysunek 3: Właściwa odległość oraz azymut 4 - 1

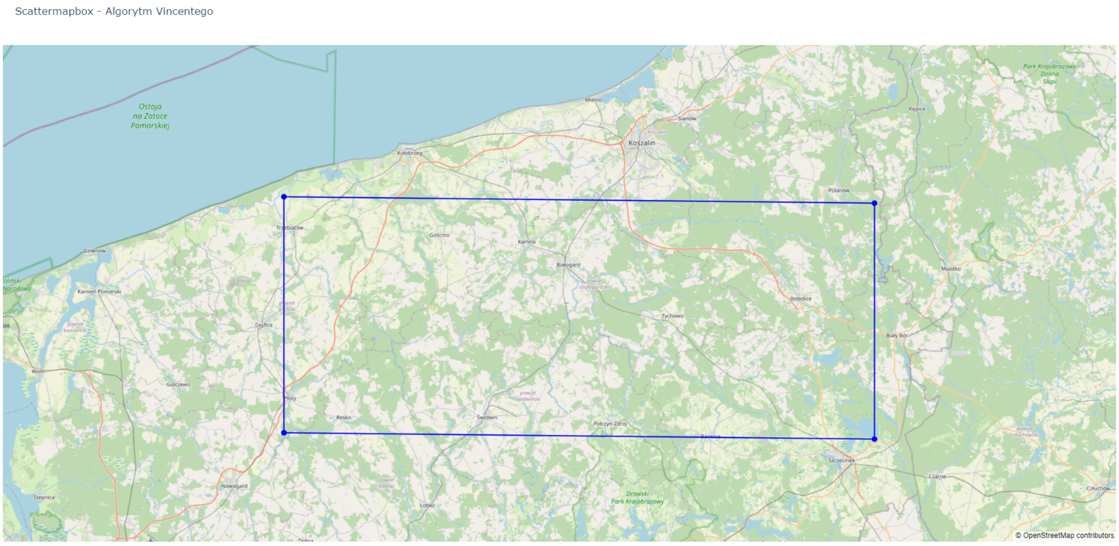
Przy pomocy algorytmu Vincentego wyliczono odległość oraz azymut pomiędzy punktami 4 i 1, które znajdują się w tabeli 3. Dane te można następnie użyć do narysowania zamkniętego trapezu o linii 4 - 1 odpowiednio skorygowanej, aby była równoległa do linii 2 - 3.

Współrzędne właściwe punktów obliczone algorytmem Vincentego

nr	phi	lambda	azymut
1	53° 45' 00.00000''	15° 15' 00.00000''	0° 00' 00.00000''
2	54° 06' 33.75763''	15° 15' 00.00000''	0° 00' 00.00000''
3	54° 05' 58.80144''	16° 46' 43.41406''	91° 14' 18.34038''
4	53° 44' 25.04170''	16° 46' 43.41406''	180° 00' 00.00000''
1*	53° 45' 00.00000''	15° 15' 00.00000''	-90° 00' 08.90101''

Rysunek 4: Właściwe współrzędne zamkniętej figury

Na podstawie współrzędnych zawartych w tabeli 4 można stwierdzić, że współrzędne punktu pierwszego oraz ostatniego są identyczne, a więc tworzą one zamknięty trapez. Jego pole oraz obwód obliczono przy użyciu funkcji `geometry_area_perimeter` z biblioteki `pyproj`. Pole wynosi **4 016 880 873.853** m^2, czyli w przybliżeniu $4016.881 km^2$, a obwód **280 862.551**m co odpowiada około $280.863 km$.



Rysunek 5: Otrzymany trapez

6 Wnioski

Powodem, dlaczego otrzymana figura po zastosowaniu algorytmu Kivioja się nie zamyka jest kształt Ziemi. Rysując prostą linię geodezyjną na powierzchni elipsoidy, trzeba brać pod uwagę zakrzywienie Ziemi i dokonywać ciągłej korekcji azymutu. Jest to zaimplementowane w algorytmie Kivioja, który uwzględniając zakrzywienie, wylicza współrzędne z skorygowanym azymutem, który różni się od przyjętego na początku. Odchyłki kształtu figury od oczekiwanego prostokąta wynikają z odległości punktów od biegunów, gdzie zakrzywienia są największe. Gdy przyjmiemy za współrzędne punktu początkowego wartości $(0°, 0°)$ różnice współrzędnych pomiędzy punktem pierwszym a ostatnim będą niewielkie. Również azymuty przy końcach linii geodezyjnych będą bliższe azymutom danym. Pole trapezu będzie bliskie polu prostokąta, ponieważ będzie wynosić 4000.135 km^2. Gdy zmienimy współrzędne punktu początkowego na bliskie równikowi, linie 2 - 3 oraz 4 - 1* będą rysowane z większym zakrzywieniem, a punkty 1 i 1* będą jeszcze dalej od siebie. Wynika z tego, że na dokładność obliczeń bezpośredni wpływ ma szerokość geograficzna punktów. Natomiast niezależnie od długości geograficznej, linie 1 - 2 oraz 3 - 4 będą zawsze równoległe, a azymut przy ich końcu nie będzie korygowany dla długości 40km.

7 Kod źródłowy

Kod źródłowy 1: Zamiany jednostek kątowych

```
degree_sign = u"\N{DEGREE SIGN}"

# Radiany na stopnie, minuty i sekundy
def rad2dms(rad):
  dd = np.rad2deg(rad)
  dd = dd
  deg = int(np.trunc(dd))
  mnt = int(np.trunc((dd-deg) * 60))
  sec = ((dd-deg) * 60 - mnt) * 60
  mnt = abs(mnt)
  sec = abs(sec)
  if sec > 59.99999:
      sec = 0
      mnt += 1
  sec = f"{sec:.5f}"
  dms = (f"{deg}{degree_sign} {mnt}' {sec}''")
  return dms

# Stopnie dziesiętne na stopnie, minuty i sekundy
def deg2dms(dd):
  deg = int(np.trunc(dd))
  mnt = int(np.trunc((dd-deg) * 60))
  sec = ((dd-deg) * 60 - mnt) * 60
  mnt = abs(mnt)
  sec = abs(sec)
  sec = f"{sec:.5f}"
  dms = (f"{deg}{degree_sign} {mnt}' {sec}''")
  return dms
```

Kod źródłowy 2: Algorytm Kivioja

```
# Parametry elipsoidy
a = 6378137
e2 = 0.00669438002290
g = Geod(ellps='WGS84')

# Współrzędne punktu początkowego
phi_1_deg = 53 + 45/60
lam_1_deg = 15 + 15/60
s = [40000, 100000, 40000, 100000]
Az_1_deg = [0, 90, 180, 270]

phi_1 = np.deg2rad(phi_1_deg)
lam_1 = np.deg2rad(lam_1_deg)
Az_1 = np.deg2rad(Az_1_deg)

```

```
# Promienie główne krzywizny
def M_and_N(phi):
  sin_phi = np.sin(phi)
  M = a * (1 - e2) / (1 - e2 * sin_phi**2)**(3/2)
  N = a / np.sqrt(1 - e2 * sin_phi**2)
  return M, N

# Algorytm Kivioja
def kivioj(phi_1, lam_1, Az_1, s, n=1000):
  ds = s / n

  phi = phi_1
  lam = lam_1
  Az = Az_1

  for i in range(n):
    M, N = M_and_N(phi)
    dphi = ds * np.cos(Az) / M
    dAz = ds * np.sin(Az) * np.tan(phi) / N

    phi_mid = phi + dphi / 2
    Az_mid = Az + dAz / 2

    M_mid, N_mid = M_and_N(phi_mid)
    dphi_mid = ds * np.cos(Az_mid) / M_mid
    dlam_mid = ds * np.sin(Az_mid) / (N_mid * np.cos(phi_mid))
    dAz_mid = ds * np.sin(Az_mid) * np.tan(phi_mid) / N_mid

    phi += dphi_mid
    lam += dlam_mid
    Az += dAz_mid

  return phi, lam, Az

phis_kiv = [phi_1_deg]
lambdas_kiv = [lam_1_deg]
azimuths_kiv = [Az_1_deg[0]]

phis_kiv_dms = [deg2dms(phi_1_deg)]
lambdas_kiv_dms = [deg2dms(lam_1_deg)]
azimuths_kiv_dms = [deg2dms(Az_1_deg[0])]

# Obliczenie współrzędnych  punktów
for i in range(4):
  phi, lam, Az = kivioj(phi_1, lam_1, Az_1[i], s[i])

  phis_kiv.append(np.rad2deg(phi))
  lambdas_kiv.append(np.rad2deg(lam))
  azimuths_kiv.append(np.rad2deg(Az))
```

```
  phis_kiv_dms.append(rad2dms(phi))
  lambdas_kiv_dms.append(rad2dms(lam))
  azimuths_kiv_dms.append(rad2dms(Az))

  phi_1 = phi
  lam_1 = lam

nr = [’1’, ’2’, ’3’, ’4’, ’1*’]

# Tabela współrzędnych
fig = go.Figure(go.Table(
  header = dict(values = [’nr’, ’phi’, ’lambda’, ’azymut’]),
  cells = dict(values = [nr, phis_kiv_dms, lambdas_kiv_dms, azimuths_kiv_dms])
))
fig.update_layout(
  title = ’Współrzędne punktów obliczone algorytmem Kivioja’,
  width = 1000, height = 400
)
fig.show()

# Mapa współrzędnych
fig = go.Figure(
  go.Scattermapbox(
    lat = phis_kiv,
    lon = lambdas_kiv,
    mode = ’markers+lines’,
    marker = dict(size = 10, color = ’red’),
    line = dict(width = 2, color = ’red’),
    text = nr,
    hoverinfo=’text’))
fig.update_layout(
  title = ’Scattermapbox - Algorytm Kivioja’,
  mapbox_style = "open-street-map",
  width = 2000,
  height = 1000,
  mapbox = dict(
    center = go.layout.mapbox.Center(
      lat = phi_1_deg,
      lon = lam_1_deg
    ),
    zoom = 7
  ),
)
fig.show()
```

Kod źródłowy 3: Algortym Vincentego

```
# Funkcja udostępniona przez prowadzącego - algorytm Vincentego
def vincenty(BA,LA,BB,LB):
  b = a * np.sqrt(1-e2)
  f = 1-b/a
  dL = LB - LA
  UA = np.arctan((1-f)*np.tan(BA))
  UB = np.arctan((1-f)*np.tan(BB))
  L = dL
  while True:
    sin_sig = np.sqrt((np.cos(UB)*np.sin(L))**2 +\
      (np.cos(UA)*np.sin(UB) - np.sin(UA)*np.cos(UB)*np.cos(L))**2)
    cos_sig = np.sin(UA)*np.sin(UB) + np.cos(UA) * np.cos(UB) * np.cos(L)
    sig = np.arctan2(sin_sig,cos_sig)
    sin_al = (np.cos(UA)*np.cos(UB)*np.sin(L))/sin_sig
    cos2_al = 1 - sin_al**2
    cos2_sigm = cos_sig - (2 * np.sin(UA) * np.sin(UB))/cos2_al
    C = (f/16) * cos2_al * (4 + f*(4 - 3 * cos2_al))
    Lst = L
    L = dL + (1-C)*f*sin_al*(sig+C*sin_sig*(cos2_sigm+\
      C*cos_sig*(-1 + 2*cos2_sigm**2)))
    if abs(L-Lst)<(0.000001/206265):
      break

  u2 = (a**2 - b**2)/(b**2) * cos2_al
  A = 1 + (u2/16384) * (4096 + u2*(-768 + u2 * (320 - 175 * u2)))
  B = u2/1024 * (256 + u2 * (-128 + u2 * (74 - 47 * u2)))
  d_sig = B*sin_sig * (cos2_sigm + 1/4*B*(cos_sig*(-1+2*cos2_sigm**2)\
    - 1/6 *B*cos2_sigm * (-3 + 4*sin_sig**2)*(-3+4*cos2_sigm**2)))
  sAB = b*A*(sig-d_sig)
  A_AB = np.arctan2((np.cos(UB) * np.sin(L)),(np.cos(UA)*np.sin(UB) - np.sin(UA)*np.cos(UB)*np.cos(L)))
  A_BA = np.arctan2((np.cos(UA) * np.sin(L)),(-np.sin(UA)*np.cos(UB) + np.cos(UA)*np.sin(UB)*np.cos(L))) + np.pi
  return sAB, A_AB, A_BA

length41, az41, az_odw41 = vincenty(
  np.deg2rad(phis_kiv[3]), np.deg2rad(lambdas_kiv[3]),
  np.deg2rad(phis_kiv[0]), np.deg2rad(lambdas_kiv[0]))
length = [f"{length41:.5f}"]
az_deg = rad2dms(az41)
az_odw_deg = rad2dms(az_odw41)

# Tabela odległości i azymutów 4 - 1
fig = go.Figure(go.Table(
  header = dict(values = ['A - B', 'Odl. AB [m]', 'Az. AB', 'Az. odw. AB']),
  cells = dict(values = ['4 - 1', length, az_deg, az_odw_deg])
))
fig.update_layout(
```

```
  title = 'Algorytm Vincentego',
  width = 1000, height = 300
)
fig.show()

phis_vin = phis_kiv[0:4]
lambdas_vin = lambdas_kiv[0:4]
azimuths_vin = azimuths_kiv[0:4]

phis_vin_dms = phis_kiv_dms[0:4]
lambdas_vin_dms = lambdas_kiv_dms[0:4]
azimuths_vin_dms = azimuths_kiv_dms[0:4]

phi_vin5, lam_vin5, Az_vin5 = kivioj(
  np.deg2rad(phis_kiv[3]), np.deg2rad(lambdas_kiv[3]), az41, length41)
phis_vin.append(np.rad2deg(phi_vin5))
lambdas_vin.append(np.rad2deg(lam_vin5))
azimuths_vin.append(np.rad2deg(Az_vin5))

phis_vin_dms.append(rad2dms(phi_vin5))
lambdas_vin_dms.append(rad2dms(lam_vin5))
azimuths_vin_dms.append(rad2dms(Az_vin5))

# Tabela skorygowanych współrzędnych
fig = go.Figure(go.Table(
  header = dict(values = ['nr', 'phi', 'lambda', 'azymut']),
  cells = dict(values = [nr, phis_vin_dms, lambdas_vin_dms, azimuths_vin_dms])
))
fig.update_layout(
  title = 'Współrzędne właściwe punktów obliczone algorytmem Vincentego',
  width = 1000, height = 400
)
fig.show()

# Mapa skorygowanych współrzędnych
fig = go.Figure(
  go.Scattermapbox(
    lat = phis_vin,
    lon = lambdas_vin,
    mode = 'markers+lines',
    marker = dict(size = 10, color = 'blue'),
    line = dict(width = 2, color = 'blue'),
    text = nr,
    hoverinfo='text'))
fig.update_layout(
  title = 'Scattermapbox - Algorytm Vincentego',
  mapbox_style = "open-street-map",
  width = 2000,
  height = 1000,
```

```
    mapbox = dict(
      center = go.layout.mapbox.Center(
        lat = phi_1_deg,
        lon = lam_1_deg
      ),
      zoom = 7
    ),
)
fig.show()
```

Kod źródłowy 4: Pole i obwód figury

```
# Obliczenie pola i obwodu
area, perimeter = g.geometry_area_perimeter(
  Polygon(
    LineString([
      Point(lambdas_vin[i], phis_vin[i]) for i in range(len(lambdas_vin))])
    ))

area = abs(area)
area_km = area / 1000000
perimeter_km = perimeter / 1000
area = f"{area:.3f}"
area_km = f"{area_km:.3f}"
perimeter = f"{perimeter:.3f}"
perimeter_km = f"{perimeter_km:.3f}"

fig = go.Figure(go.Table(
  header = dict(values = [r'','Pole', r'Obwód']),
  cells = dict(values = [['[km]', '[m]'], [area, area_km], [perimeter,
 perimeter_km]])))
fig.update_layout(
  title = 'Pole i obwód wielokąta',
  width = 1000, height = 300)
fig.show()
```

Spis tabel

Spis rysunków

Spis kodów źródłowych