Python i MEK1100

En oversettelse fra Matlab til Python
av deler av kapittel 3 og 5 i boka

Feltteori og vektoranalyse

av

Bjørn Gjevik og Morten Wang Fagerland

2017

oversatt av

Karsten Trulsen

med bistand fra

Susanne Støle–Hentschel og Hans Petter Langtangen

Seksjon for Mekanikk
Matematisk institutt
Universitetet i Oslo
16/1–2018

Kapittel 3

Bruk av Python

3.1 Innledning

I det følgende har vi oversatt kapittel 3 i boka Gjevik & Fagerland (2017) fra Matlab til Python. Mens bokas kapittel 3 er skrevet uten å anta at man kan noe om Matlab fra før, vet vi at dagens typiske student i MEK1100 har forkunnskaper i Python som går langt forbi den introduksjonen som gis her. Vi har likevel prøvd å oversette kapittel 3 i boka forholdsvis "ordrett".

For grunnleggende introduksjon til Python anbefales læreboka i IN1900 (Langtangen, 2016) samt Python-versjonen (Ryan, 2015) av MATLAB-appendikset i læreboka i MAT1110 (Lindstrøm & Hveberg, 2015). Det er også en elektronisk versjon av MATLAB-appendikset i læreboka i MAT1110 (Lindstrøm & Hveberg, 2016).

Vi skal basere oss på Matplotlib for å generere grafikk, dette er dokumentert på nettsiden `http://matplotlib.org/`.

3.2 Litt grunnleggende Python

3.2.1 Oppstart, grensesnitt, kommentarer

På Linux kan du starte Python ved å skrive `python3` eller `ipython3` i et terminalvindu.[1] Det er mange andre måter å starte Python, som vi ikke diskuterer her.

I terminalvinduet kan du skrive inn Python-kommandoer som vil bli utført med en gang du trykker på Enter. I `python3` betyr symbolet `>>>` at Python er klar til å ta imot en kommando. I `ipython3` betyr symbolet `In [#]:` (hvor `#` er et tall) det samme. Resultatet av denne vil bli skrevet ut rett under (se eksempler i neste avsnitt).

Når du skal skrive Python-kode er det lurt (som i alle andre programmeringsspråk) å venne seg til å skrive kommentarer slik at det er lettere både for deg selv og for andre å lese koden ved en senere anledning. Linjekommentarer i Python starter med et *nummertegn* `(#)` og alt fra dette tegnet og ut linja vil oppfattes som kommentar. Kommentarer over flere linjer er definert ved `'''` (3 tegn) før og etter kommentaren.

[1] Eksemplene i det følgende ble testet med Python 3.6.3.

Importere `numpy` for numerikk

Matematiske konstanter og funksjoner som kan ta argumenter som er både skalarer, lister og arrayer gjøres tilgjengelig fra `numpy`:

```
>>> import numpy as np
>>> print (np.pi, np.sin(1))
3.14159265359  0.84147098
>>> print (np.sin([1, 2]), np.sin(np.array([1, 2])))
[ 0.84147098  0.90929743] [ 0.84147098  0.90929743]
```

Her merker vi oss at "`[1, 2]`" er ei liste mens "`np.array([1, 2])`" er et objekt av type numpy.ndarray, sistnevnte er å foretrekke for numerisk arbeid. `array()` diskuteres i seksjon 3.2.2.

Importere `matplotlib.pyplot` for grafikk

Vi skal bruke `matplotlib.pyplot` for å generere grafikk. Følgende tre linjer tegner opp ei rett linje mellom punktene (0,1) og (1,2):

```
import matplotlib.pyplot as plt
plt.plot([0, 1], [1, 2])
plt.show()
```

Dersom Python ikke er i interaktiv modus så vil kommandoen `show()` være nødvendig for at grafikkvinduet skal komme opp på skjermen.

Interaktiv modus

Når vi jobber på datamaskin med grafisk framstilling av felt kan det være behagelig at grafikken som vises på skjermen blir oppdatert etterhver som vi gir kommandoene. Dette kan oppnås ved å gi kommandoen `interactive(True)`. I så fall skal ikke kommandoen `show()` være nødvendig:

```
import matplotlib.pyplot as plt
plt.interactive(True)
plt.plot([0, 1], [1, 2])
```

Interaktiv modus kan også slås på og av med kommandoene `plt.ion()` og `plt.ioff()`, og man kan sjekke modus med kommandoen `plt.isinteractive()`.

Importere `pylab` for både numerikk og grafikk

For arbeid i dette kurset kan det være greit å inkludere `pylab` med "stjerne-import", da blir `numpy` og `matplotlib.pyplot` importert til samme navnerom slik at man ikke trenger å skrive forstavelsene "`np.`" og "`plt.`". Dette oppnås ved:

```
from pylab import *
```

Dette har den ulempen at man mister kontroll over om funksjonene man bruker kommer fra `numpy` eller `pyplot`.

Starte opp `ipython` i interaktiv modus

Dersom `ipython` startes opp med kommandoen "`ipython --pylab`" vil man både importere `pylab` og sette Python i interaktiv modus, dette er ekvivalent med

```
from pylab import *
interactive(True)
```

Da vil Python oppføre seg ganske likt Matlab, men kanskje ikke slik en erfaren Python-programmerer vil foretrekke.

Utskrift av grafikk til fil

Den enkleste måten å plotte ut grafikk til fil med Matplotlib er å bruke det grafiske grensesnittet i plottevinduet.

Utskrift av grafikk til fil kan også gjøres med `plt.savefig("filnavn.filtype")`. Fila vil bli skrevet i det formatet som angis av `filtype`. Her er et eksempel som skriver et tomt koordinatsystem til fila "`utskrift.pdf`":

```
plt.plot([0, 1], [1, 2])
plt.savefig("utskrift.pdf")
```

3.2.2 Matriser

En matrise kan lages med kommandoen `array` som hører til modulen `numpy`. Objektet som returneres er av type numpy.ndarray, som er å foretrekke framfor vanlige lister for numerisk arbeid. Elementene separeres med *komma* (`,`). Hver rad skal rammes inn med *klammeparenteser* ([ ]). Hele matrisen skal også rammes inn med *klammeparenteser* ([ ]). Resultatet kan vises med kommandoen `print`.

```
>>> A = np.array([[1, 2 ,3],[4, 5, 6], [7, 8, 9]])
>>> print (A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Det er flere måter å transponere matrisen, for eksempel ved å bruke kommandoen `transpose`.

```
>>> print (A.transpose())
[[1 4 7]
 [2 5 8]
 [3 6 9]]
>>> print (np.transpose(A))
[[1 4 7]
 [2 5 8]
 [3 6 9]]
>>> print (A.T)
[[1 4 7]
```

```
 [2 5 8]
 [3 6 9]]
```

For å finne den inverse til en matrise har vi funksjonen `linalg.inv`.

```
>>> B = np.array([[1,3,0],[2,0,4],[1,3,1]])
>>> print (B)
[[1 3 0]
 [2 0 4]
 [1 3 1]]
>>> C = np.linalg.inv(B)
>>> print (C)
[[ 2.          0.5        -2.        ]
 [-0.33333333 -0.16666667  0.66666667]
 [-1.          0.          1.        ]]
```

Vi sjekker deretter at produktet av matrisene B og C gir enhetsmatrisen

```
>>> print (np.dot(C,B))
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Funksjonen `diag` returnerer matrisens diagonal.

```
>>> print (np.diag(A))
[1 5 9]
```

Funksjonen `sum` returnerer summen av alle matriseelementene.

```
>>> print (np.sum(A))
45
```

Med et ekstra argument vil funksjonen `sum` summere opp kolonnene eller radene avhengig av verdien til `axis`-parameteren (0=rad, 1=kolonne).

```
>>> print (np.sum(A, axis=0))
[12 15 18]
>>> print (np.sum(A, axis=1))
[ 6 15 24]
```

Elementet i rad i og kolonne j av A er $A[i, j]$ (husk at i Matlab starter indekseringen fra 1, mens i Python starter indekseringen fra 0, alle indeksene er derfor redusert med 1 i forhold til Matlab-kapitlet i boka).

```
>>> print (A[1,0] + A[2,1])
12


>>> A[0,2] = 0
>>> print (A)
[[1 2 0]
 [4 5 6]
 [7 8 9]]
```

Kommandoen `shape` returnerer antall rader og kolonner til en matrise.

```
>>> print (np.shape(A))
(3, 3)
```

Kommandoen `size` returnerer antall elementer i en matrise

```
>>> print (np.size(A))
9
```

Det finnes flere Python-kommandoer som gjør det enkelt å generere matriser. Vi har sett at funksjonen `diag(A)` returnerer diagonalen til `A`, men den kan også brukes til å bygge opp en matrise:

```
>>> A = np.diag([1,2,3])
>>> print (A)
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Andre nyttige kommandoer som kan nevnes er `ones`, `zeros` og `eye` som lager matriser med henholdsvis enere, nuller, og identitetsmatrisen. Du kan lese mer om dem ved å bruke `help` (f.eks. `help(np.zeros)`). Disse fungere ikke på samme måte som i Matlab, for eksempel returnerer `ones(3)` en tre-dimensjonal vektor mens `ones([3,3])` returnerer en 3×3 matrise.

Matematiske operasjoner på arrayer

Det matematiske funksjonene i `numpy` kan operere komponentvis på arrayer

```
>>> A = np.array([[1, 2]])
>>> print (np.sin(A))
[[ 0.84147098  0.90929743]]
```

3.2.3 Kolon-operatoren, arange og linspace

For å generere en vektor med heltallige verdier fra en aritmetisk tallfølge kan vi bruke `arange`. For eksempel, en vektor (tallrekke) fra -5 til 5 med intervall 1

```
>>> a = np.arange(-5,6)
>>> print (a)
[-5 -4 -3 -2 -1  0  1  2  3  4  5]
```

Det genereres tall større eller lik første argument, og mindre enn andre argument. En vektor fra -15 til 15 med intervall 3

```
>>> a = np.arange(-15, 16, 3)
>>> print (a)
[-15 -12  -9  -6  -3   0   3   6   9  12  15]
```

For ikke-heltallige verdier anbefales det å bruke `linspace` for å unngå avrundingsfeil, `linspace` genererer det antall tall som angitt ved tredje argument, jevnt fordelt fra og med første og til og med andre argument

```
>>> a = np.linspace(-15, 15, 11)
>>> print (a)
[-15. -12.  -9.  -6.  -3.   0.   3.   6.   9.  12.  15.]
```

Det finnes en annen kommando `range` som fungerer tilsynelatende likt som `arange` for heltallige argumenter, men mens `arange` genererer objekter av type numpy.ndarray så genererer `range` ei liste som ikke anbefales for numerisk arbeid.

I Python kan kolon-operatoren brukes til å indeksere matriser og vektorer, og inkluderer alt men ikke det siste tallet

```
>>> A = np.array([[1, 2, 3, 4],[ 5, 6, 7, 8],[ 9, 10, 11, 12],[ 13, 14, 15, 16]])
>>> print (A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
>>> print (A[1:3,:])
[[ 5  6  7  8]
 [ 9 10 11 12]]
```

Første til tredje rad, andre til fjerde kolonne

```
>>> print (A[0:3,1:4])
[[ 2  3  4]
 [ 6  7  8]
 [10 11 12]]
```

Andre og tredje rad, alle kolonnene

```
>>> print (A[1:3,:])
[[ 5  6  7  8]
 [ 9 10 11 12]]
```

Steglengde 2 langs kolonnene

```
>>> print (A[1:3,1:4:2])
[[ 6  8]
 [10 12]]
```

Steglengde 3 langs kolonnene

```
>>> print (A[1:3,0:4:3])
[[ 5  8]
 [ 9 12]]
```

Det burde nå være åpenbart at kolon-operatoren fungerer på dramatisk forskjellig måte i Matlab og Python!

3.2.4 py-filer og funksjoner

Isteden for å taste inn kommandoer i terminal-vinduet kan vi samle kommandoer i ei fil ved hjelp av en teksteditor og så kjøre fila i Python. Her antar vi at fila slutter på ".py" og kalles derfor ei *py-fil*.

Her er ei fil med kommandoer som definerer en matrise A og skriver ut noen egenskaper til matrisen:

```
import numpy as np
A = np.array([[1,3,0],[2,1,1]])
print (A)
print (A.size)
print (A.shape)
print (np.diag(A))
print (np.sum(A))
print (A.transpose())
```

Denne fila kan du kalle hva som helst, la oss si at du kaller den `MatriseEksempel.py`, og den kan da kjøres ved å skrive "`python MatriseEksempel.py`" i et terminalvindu.

Under følger et eksempel på hvordan vi kan definere en funksjon som ligger i ei fil ved navn `tridiag.py`:

```
import numpy as np

def tridiag(n):
    '''Konstruerer en nxn matrise med -2 paa
    diagonalen og 1 paa de to subdiagonalene'''

    A = -2*np.eye(n) + np.diag(np.ones(n-1),1) + np.diag(np.ones(n-1),-1)
    return A
```

Linja som starter med `def` forteller Python at dette er en funksjon som heter "tridiag" og tar en innparameter (n). De to neste linjene er en kort beskrivelse av funksjonen. Denne hjelpeteksten er alltid lurt å ta med ettersom det er denne teksten som `help` bruker for å beskrive funksjonen. Deretter beregnes matrisen A som returneres.

Funksjonen kan nå importeres:

```
from tridiag import *
```

Brukerveiledning fås med:

```
>>> help(tridiag)
Help on function tridiag in module tridiag:

tridiag(n)
    Konstruerer en nxn matrise med -2 paa
    diagonalen og 1 paa de to subdiagonalene
```

(Trykk på "q" for å komme ut av `help`.)
Funksjonen kalles opp slik:

```
>>> print (tridiag(5))
[[-2.  1.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.]
 [ 0.  1. -2.  1.  0.]
 [ 0.  0.  1. -2.  1.]
 [ 0.  0.  0.  1. -2.]]
```

3.2.5 Enkel plotting i 2D

Kommandoen `plot` kan brukes til enkel plotting i to dimensjoner. La oss først plotte funksjonen $y = x^3$ der x er definert over et område fra -2π til 2π:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2*np.pi, 2*np.pi, 41)
y = x**3
plt.plot(x,y)
plt.show()
```

Her har vi valgt å diskretisere x-intervallet med 41 punkt som gir steg på $\pi/10$. Velges dette steget for stort blir grafen hakkete. Prøv f.eks. med et steg på $\pi/2$.

Vi kan også enkelt plotte en kurve som er gitt ved en parametrisering. En sirkel med radius r kan uttrykkes ved $x(t) = r\cos t$, $y(t) = r\sin t$ der t går fra 0 til 2π. La oss plotte en sirkel med radius 3:

```
r = 3
t = np.linspace(0, 2*np.pi, 101)
x = r*np.cos(t)
y = r*np.sin(t)
plt.plot(x,y)
plt.axis("equal")
```

For at sirkelen skal se ut som en sirkel og ikke som en ellipse, så er det et viktig poeng at en enhet langs x-aksen er like lang som en enhet langs y-aksen. Dette oppnås med kommandoen `plt.axis("equal")` — prøv å utføre kommandosekvensen ovenfor uten å kalle `plt.axis("equal")` og se at figuren da ser ut som en ellipse!

Dersom du nå opplevde at kurven til $y = x^3$ og sirkelen $x^2 + y^2 = 9$ endte opp i samme plott, så burde plottevinduet ha blitt lukket mellom disse to plottingene. Det kan enten gjøres fra kommandolinja med `plt.close()` eller fra plottevinduet.

3.2.6 Elementvise operasjoner

Noen operasjoner (f.eks. multiplikasjon, divisjon og potens) kan man ønske å bruke på matriser på to forskjellige matematiske måter. Når variablene disse operasjonene

virker på er skalarer, vil den vanlige betydningen (multiplikasjon, divisjon og potens) gjelde slik at f.eks. `3*2` vil gi det forventede svaret 6. Hvis derimot variablene er enten vektorer eller matriser, kan vi ønske at operasjonene enten er vanlige matriseoperasjoner eller elementvise operasjoner. **Matlab og Python oppfører seg ikke likt!** I Matlab betyr "`*`", "`/`", "`**`" og "`^`" matriseoperasjoner (de to siste er synonymer for potens), mens man ved å tilføye et "`.`" foran operatorene får elementvise operasjoner. I Python betyr "`*`", "`/`" og "`**`" elementvise operasjoner ("`**`" betyr potens, mens derimot "`^`" betyr noe helt annet). I eksempelet over der vi ønsker å regne ut $y = x^3$, og vi allerede har definert en vektor med x-verdier, vil vi at hvert element i x skal opphøyes i tredje; vi ønsker ikke å bruke matrisemultiplikasjonen av x med seg selv to ganger. Noen eksempler:

```
>>> a = np.arange(1,5)
>>> print (a)
[1 2 3 4]
>>> print (3*a)
[ 3  6  9 12]
>>> print (a*a)
[ 1  4  9 16]
```

Indreprodukt beregnes med `dot`

```
>>> print (np.dot(a,a))
30
```

Ytre eller dyadisk produkt beregnes med `outer`

```
>>> print (np.outer(a,a))
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]
 [ 4  8 12 16]]
```

Ta en titt i boka for å finne ut hvordan disse operasjonene gjøres i Matlab!

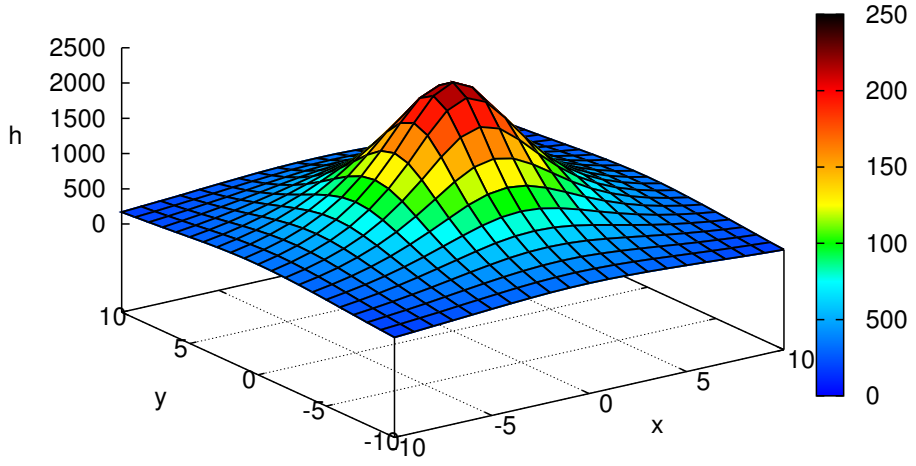
3.3 Plotting av flater

Vi skal nå se på hvordan vi kan lage plottet av flaten i figur 1.7 (i boka). Dette er den samme flaten som vi har studert på side 10 i kapittel 1 og er gitt matematisk ved formelen:

$$h = \frac{h_0}{1 + \frac{x^2+y^2}{R^2}}. \tag{3.1}$$

Formelen modellerer en isolert fjelltopp der h er høyden over havflaten, h_0 er høyden på toppen av fjellet og R er et mål for radius i fjellet. Konstantene h_0 og R er her satt til 2277 m og 4000 m.

Det anbefales at du først prøver ut de påfølgende kommandoene direkte i Python for senere å samle dem i ei py-fil.



Figur 3.1: En modell av fjelltoppen Beerenberg på Jan Mayen.

Det første vi skal gjøre er å definere de to konstantene h_0 og R. Vi ser av figuren at x- og y-aksen er oppgitt i km og h-aksen i meter. Vi definerer derfor h_0 i meter og R i km:

```
h0 = 2277.
R = 4.
```

x- og y-aksen skal spenne over et område på 20×20 km med sentrum i origo. For å lage et passende grid med x- og y-verdier kan vi bruke kommandoen `meshgrid` som tar to vektorer som parametere. Vektorene vi bruker som innparametere kan vi lage ved å bruke `linspace`:

```
t = np.linspace(-10,10,41)
x,y = np.meshgrid(t,t, indexing="ij")
```

Størrelsen på gridet må tilpasses hvert eksempel. Her har vi brukt 41 punkter per akse, noe som gir en passende maskestørrelse samtidig som flaten ser glatt ut.

Merk at `meshgrid` må angis med opsjon `indexing="ij"` for at etterfølgende operasjoner skal fungere korrekt: Prøv å lage figur 3.3 uten bruk av `indexing="ij"`.[2]

Vi er nå klare til å regne ut h som angitt i (3.1):

```
h = h0/(1 + (x**2+y**2)/(R**2))
```

[2] 2D grid kan organiseres på to måter: Kartesisk indeksering (standard `meshgrid` i Python/`numpy` og i Matlab), eller matrise-indeksering (`meshgrid` med `indexing="ij"` i Python/`numpy`, `ndgrid` i Matlab). Kommandoen `gradient` (se kapittel 3.5) ønsker kartesisk indeksering i Matlab og matrise-indeksering i Python.

For å tegne opp 3D grafikk skal vi bruke Matplotlib mplot3d toolkit.[3] For å komme i gang kan vi først gjøre følgende:

```
from mpl_toolkits.mplot3d import axes3d
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
```

Deretter kan vi be om å få tegnet opp omtrent som Matlab `surf`

```
surf = ax.plot_surface(x, y, h, rstride=1, cstride=1, cmap=plt.cm.jet,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

eller omtrent som Matlab `mesh` med skyggevirkning

```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(x, y, h, rstride=1, cstride=1, color="w", linewidth=0.1)
plt.show()
```

eller vi kan lage kurver som ikke har skyggevirkning

```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_wireframe(x, y, h, rstride=10, cstride=10)
plt.show()
```

For å endre fargene som skriv `cmap=plt.cm.MAP` hvor `MAP` er et gyldig navn på colormap, for eksempel `jet`, `hot`, `bone`, `cool`, etc., dette står forklart i seksjon 3.4 om konturlinjer.

For å se hvilke farger som representerer hvilke verdier kan man skrive `colorbar()`, men det er selvfølgelig kun meningsfullt ved bruk av tilsvarende colormap.

Til slutt kan vi pynte litt på figuren. For å angi navn på aksene bruker vi

```
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("h")
```

Samler vi alle kommandoene kan det se slik ut:

```
from mpl_toolkits.mplot3d import axes3d
h0 = 2277. # Hoyden av toppen av fjellet (m)
R = 4.     # Maal for radius av fjellet (km)
t = np.linspace(-10,10,21)
x,y = np.meshgrid(t,t, indexing="ij") # Grid for x- og y-verdiene (km)
h = h0/(1 + (x**2+y**2)/(R**2))    # Beregn hoyden h (m)
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
```

[3] `http://matplotlib.org/mpl_toolkits/mplot3d/`

```
surf = ax.plot_surface(x, y, h, rstride=1, cstride=1, cmap=plt.cm.jet,
                       linewidth=0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("h")
plt.show()
```

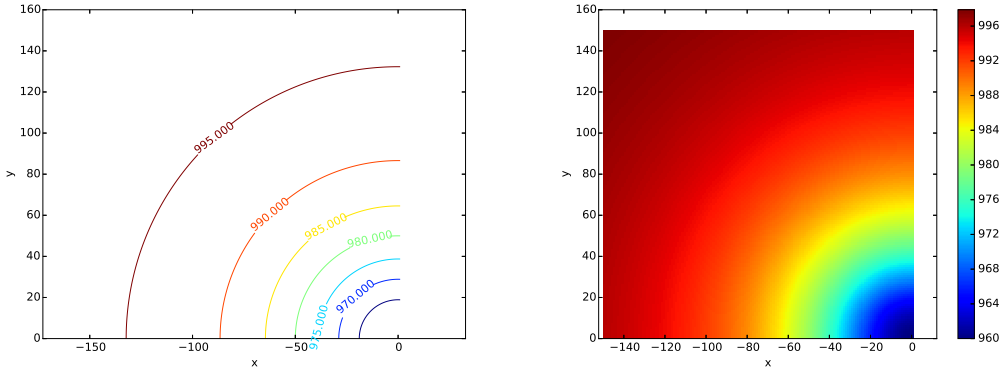
Her kan man bytte ut de to linjene med `plot_surface` og `colorbar` med alternativene vist ovenfor.

3.4 Plotting av konturlinjer

Vi skal ta for oss et nytt eksempel fra kapittel 1, side 10; lufttrykket ved havflaten i tilknytning til et stormsenter. Formelen vi skal bruke er gitt ved

$$p = p_0 - \frac{\Delta p}{1 + \frac{x^2+y^2}{R^2}} \tag{3.2}$$

der p_0 er lufttrykket langt borte fra sentrum, Δp er trykkfallet inn mot sentrum og R er et mål for utstrekningen av lavtrykket. Konturlinjene vi skal plotte er gjengitt i figur 3.2.



Figur 3.2: Til venstre konturlinjer for konstant trykk (isobarer) rundt et lavtrykkssentrum. Til høyre de samme med pcolor. Enhetene på aksene er kilometer.

Vi starter med konstantene som skal oppgis i km (R) og hPa (p_0 og Δp):

```
R = 50
p0 = 1000
dp = 40
```

Til forskjell fra boka skal vi her la x-aksen være definert over et område fra -150 til 0 km, mens vi lar y-aksen være definert over et område fra 0 til 150 km, deretter regnes p-verdiene ut:

```
tx = np.linspace(-150, 0, 151)
ty = np.linspace(0, 150, 151)
x,y = np.meshgrid(tx,ty, indexing="ij")
p = p0 - dp/(1+(x**2+y**2)/R**2)
```

Konturlinjene kan nå plottes med kommandoen `contour`:

```
plt.contour(x,y,p)
plt.colorbar()
```

For å plotte et valgfritt antall konturlinjer bruk `contour(x,y,p,n)` der n er antall linjer. Hvis vi kun er interessert i noen utvalgte trykkverdier (f.eks. 960, 970, 980 og 990) så kan vi gi verdiene som en vektor til `contour`:

```
v = [960, 970, 980, 990]
plt.contour(x,y,p,v)
```

eller enda enklere: `plt.contour(x,y,p,[960, 970, 980, 990])`.

Vil du bruke en annen fargeskala prøv `cmap=plt.cm.MAP` hvor `MAP` er et gyldig navn for en "colormap", for eksempel `autumn`, `bone`, `cool`, `copper`, `flag`, `gray`, `hot`, `hsv`, `jet`, `pink`, `prism`, `spring`, `summer` eller `winter`.

For å se hvilke verdier linjene har kan man skrive eksplisitt på hver linje, og linjene bør da ha en mørk farge. Man kan benytte kommandoen `clabel` som vist nedenfor til å skrive på verdiene i plottet (legg merke til variabelen `C` som brukes til å overføre informasjon fra `contour` til `clabel`).

Samler vi alle kommandoene for å generere figur 3.2 kan nå se slik ut:

```
R = 50    # Utstrekningen av lavtrykket (km)
p0 = 1000 # Lufttrykket langt borte fra sentrum (hPa)
dp = 40   # Trykkfallet inn mot sentrum (hPa)

tx = np.linspace(-150,1,151)
ty = np.linspace(0,150,151)
x,y = np.meshgrid(tx,ty, indexing="ij") # Grid for x- og y-verdiene (km)
p = p0 - dp/(1+(x**2+y**2)/R**2)     # Beregn trykket p (hPa)
C=plt.contour(x,y,p)
plt.clabel(C)
plt.axis("equal")
plt.xlabel("x") # Sett aksenavn
plt.ylabel("y")
plt.show()
```

For en alternativ type plott som inneholder samme informasjon, prøv å bytte ut linjene med `contour` og `clabel` med `pcolor` og `colorbar`:

```
plt.pcolor(x,y,p)
plt.colorbar()
```

3.5 Plotting av vektorfelt og beregning av gradient-vektor.

Et vektorfelt i kartesiske koordinater $\boldsymbol{v} = v_x(x,y)\boldsymbol{i} + v_y(x,y)\boldsymbol{j}$ kan plottes med

```
plt.quiver(x,y,vx,vy,[faktor],[farge])
```

der `faktor` og `farge` er valgfrie opsjoner som endrer henholdsvis vektorenes lengde med en skaleringsfaktor og vektorenes farge. La oss ta et eksempel:

$$\boldsymbol{v} = (x^2 + 2y - \frac{1}{2}xy)\boldsymbol{i} - 3y\boldsymbol{j}.$$

Vi lar x og y variere mellom -5 og 5. Når vi skal definere et grid for x og y, må vi passe på å bruke et større intervallsteg enn vi vil bruke for overflateplott eller konturlinjer. Det blir fort uryddig med for mange vektorer i en og samme figur.

```
t = np.linspace(-5,5,11)
x,y = np.meshgrid(t,t, indexing="ij")
vx = x**2 + 2*y - .5*x*y
vy = -3*y
plt.quiver(x,y,vx,vy,scale=25,units="x",color="b") # skalareringsfaktor, blaa far
plt.axis("equal")
```

Her har vi skalert lengden til alle vektorene med en faktor 25 relativt til x-aksen som referanse, og vi har bedt om at vektorene skal tegnes i blå farge.[4]

Hvis vi har definert et skalarfelt $F(x,y)$ over et område utspent av vektorene x og y i Python kan vi regne ut gradientvektoren til feltet med `gradient(F)`. Denne funksjonen returnerer de partielt deriverte ($\partial F/\partial x$ og $\partial F/\partial y$ i 2D) til feltet.[5] Med disse for hånden kan vi plotte vektorfeltet med `quiver`:

```
dFx,dFy = np.gradient(F)
plt.quiver(x,y,dFx,dFy)
```

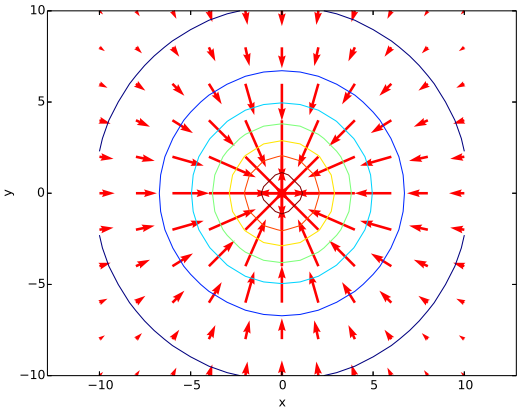
Vi skal videre presentere koden som er brukt for å lage figur 3.3 som er et plott av noen konturlinjer for h gitt ved (3.1) og gradientvektoren ∇h. Bortsett fra kommandoene `hold("on")` og `hold("off")`, som sørger for at konturlinjene og vektorfeltet blir plottet sammen og med samme akseegenskaper, skulle resten av koden nå være kjent.

```
h0 = 2277  # Hoyden av toppen av fjellet (m)
R = 4      # Maal for radius av fjellet (km)
tt = np.linspace(-10.,10.,11)
xx,yy = np.meshgrid(tt, tt, indexing="ij") # Definer et grovere grid til vektorfe
```

[4] NB! I Matlab og i Octave øker pilenes lengde med skaleringsfaktor. I Python/Matplotlib avtar pilenes lengde med skaleringsfaktor.

[5] Dersom `gradient(F)` kalles med kun ett argument antas det at avstanden mellom gridpunktene er lik én. Dersom avstanden er ulik én, men konstant lik H (en skalar) i alle retninger kan skrive `gradient(F,H)`. Dersom avstanden mellom gridpunkter er konstant lik `HX` i x-retning og konstant lik `HY` i y-retning kan man skrive `gradient(F,HX,HY)`.

```
hh = h0/(1+(xx**2+yy**2)/(R**2)) # Beregn hoyden med det nye griddet
dhx,dhy = np.gradient(hh,2.0)      # Beregn gradientvektoren (dh/dx,dh/dy)
# andre argument 2.0 er avstanden mellom gridpunktene i x- og y-retning
plt.quiver(xx,yy,dhx,dhy)          # Plott vektorfeltet
plt.hold("on")                     # Behold konturlinjene og akse-egenskapene
t = np.linspace(-10.,10.,21)
x,y = np.meshgrid(t,t, indexing="ij")  # Grid for x- og y-verdiene (km)
h = h0/(1+(x**2+y**2)/(R**2)) # Beregn hoyden h (m)
plt.contour(x,y,h)                 # Kontur og sett akseenhetene like
plt.xlabel("x")                    # Sett aksenavn
plt.ylabel("y")
plt.axis("equal")                  # Sett akseenhetene like
plt.hold("off")                    # Trenger ikke flere plott i denne figuren
plt.show()
```



Figur 3.3: Konturlinjer for h og gradientvektoren ∇h. Enhetene på aksene er kilometer.

Merk: I denne figuren er det et viktig poeng at pilene skal stå vinkelrett på sirklene. Dette oppnår man ved å kreve at enhetene langs begge aksene er like, derfor er det viktig å ta med kommandoen `axis("equal")`.

Merk: Prøv å kjøre skriptet ovenfor uten å ta med de to forekomstene av `indexing="ij"` og legg merke til at figuren da blir helt feil!

3.6 Plotting av strømlinjer

I kapittel 4 møter vi strømfunksjonen ψ. Kjenner vi uttrykket for denne er det lett å plotte strømlinjene. Siden strømfunksjonen er konstant langs en strømlinje vil strømlinjene være konturlinjer for skalarfeltet ψ. Vi kan derfor definere et grid i x- og y-retning med `meshgrid`, regne ut diskrete verdier av strømfunksjonen, og til slutt bruke `contour` for å tegne opp strømlinjene, akkurat på samme måte som vi plottet isobarene til trykkfeltet i seksjon 3.4.

Merk: Noen ganger ønsker vi å plotte strømliner og ekviskalarflater i samme plott. I så fall er det et viktig poeng at de to kurvefamiliene står vinkelrett på hverandre. Dette oppnås ved å kreve at enhetene langs begge aksene er like, derfor er det viktig å ta med kommandoen `axis("equal")`.

3.7 Oppgaver

Oppgavene som står i boka anbefales og kan gjøres uansett om man bruker Matlab eller Python.

Kapittel 5

En praktisk anvendelse av ∇-operatoren i meteorologi

Vi presenterer kun forslag til:

5.8 En fil med hele Python-koden

```
import numpy as np
import pylab as plt

# transpose nedenfor tilpasser datasetten som har vaert lagret for bruk i Matlab
p = np.transpose(np.loadtxt('trykkfelt.dat'))
u = np.transpose(np.loadtxt('vindfelt_u.dat'))
v = np.transpose(np.loadtxt('vindfelt_v.dat'))

print (p.shape)
print (u.shape)
print (v.shape)

Nx, Ny = p.shape

isobarer = np.arange(980, 1025, 5)
x = np.linspace(0,1600,Nx)
y = np.linspace(0,1600,Ny)
xx, yy = np.meshgrid(x,y, indexing='ij')
CS = plt.contour(xx,yy,p, isobarer)
plt.clabel(CS, inline=1, fontsize=10, fmt='%1.0f', colors='k')


plt.figure()
plt.quiver(xx,yy,u,v)

l = np.sqrt(u**2 + v**2)
l_max = l.max()
```

```
print ('max vindhastighet ', l_max)

dudx = np.gradient(u, axis=0)
dvdy = np.gradient(v, axis=1)
div = dudx + dvdy
plt.figure()
isobarer = np.array([-9,-4,-3,-2,-1,0,1,2])
CS = plt.contour(xx,yy,div,10)
plt.clabel(CS, inline=1, fontsize=10, fmt='%1.0f', colors='k')

dudy = np.gradient(u, axis=1)
dvdx = np.gradient(v, axis=0)
curlz = dvdx - dudy
plt.figure()
CS = plt.contour(xx,yy,curlz)
plt.clabel(CS, inline=1, fontsize=10, fmt='%1.0f', colors='k')
plt.show()
```

Bibliografi

Gjevik, B. & Fagerland, M. W. 2017 *Feltteori og vektoranalyse*. Farleia Forlag.

Langtangen, H. P. 2016 *A Primer on Scientific Programming with Python*, 5th edn. Springer.

Lindstrøm, T. L. & Hveberg, K. 2015 *Flervariabel analyse med lineær algebra*, 2nd edn. Gyldendal Akademisk Forlag.

Lindstrøm, T. L. & Hveberg, K. 2016 Kort innføring i MATLAB. `http://www.uio.no/studier/emner/matnat/math/MAT1110/v16/fvlabokkort.pdf`.

Ryan, Ø. 2015 Python-versjon av MATLAB-appendikset i "Flervariabel analyse med lineær algebra". `http://folk.uio.no/oyvindry/fvla/FVLAbokpythonappendix.pdf`.

Register