KU LEUVEN kulak

Wetenschap & Technologie Kulak

Bachelor in de fysica, informatica, wiskunde
Ingenieurswetenschappen

Wetenschappelijk rekenen
{python, sympy, numpy, matplotlib}

Handleiding bij de vakken

`X0B53A` – Probleemoplossen en ontwerpen, deel 1

`X0A22B` – Informaticawerktuigen

Stijn Rebry

stijn.rebry@kuleuven.be

Inhoudsopgave

Introductie

`Python` is in eerste plaats een programmeertaal. In een typische programmeercontext ligt de focus op het implementeren van algoritmes, op een zo algemeen mogelijke manier, met gestructureerde code die efficiënt is, makkelijk leesbaar, bruikbaar en uitbreidbaar.

In een wetenschappelijke context, bij het maken van theoretische of numerieke berekeningen, ligt de initiële focus vaak elders. Specifieke berekeningen gebeuren vaak ad hoc, verkennend: de gebruiker wil van elke rekenstap eerst het resultaat zien vooraleer te beslissen wat de volgende berekening zal zijn. In plaats van algemene, compileerbare code te schrijven, wordt *interactief* gewerkt, zeker in een eerste fase.

Om een concreet voorbeeld te gebruiken. Als een student het verloop van een welbepaalde reële functie wil bestuderen, wil die snel antwoord: Wat zijn de nulpunten? Wat is het teken van de functie tussen elke twee opeenvolgende nulpunten? Zijn er kritieke punten en wat is de aard? Dit alles kan misschien wel in een programma gegoten worden, maar zo een functie-analyse in volle algemeenheid programmeren zal veel te ver leiden. In plaats van compileerbare code te schrijven, gebruikt de student enkel de commando's die in zijn geval van toepassing zijn.

Om op deze manier te werken moet `Python` worden uitgebreid met extra pakketten die focussen op symbolische en numerieke berekeningen, het maken van grafieken en dergelijke meer. Ook het werken in de grafische front-end zal anders verlopen. In plaats van compileerbare scripts, wordt code lijn voor lijn ingevoerd in een interactieve shell. Hieronder staan talloze voorbeelden, waarvan de zinvolheid telkens kort geïntroduceerd is in de tekst. Het is de bedoeling dat de lezer de code zelf uitvoert en daarbij voor zichzelf nagaat de syntax en de betekenis van de code te hebben begrepen. Maak gebruik van de help-functie en de online documentatie van `Python` en de gebruikte pakketten waar de uitleg hieronder ontoereikend is.

De tekst bestaat uit vier grote delen. Het eerste deel focust op de basissyntax, symbolisch rekenen en functie-analyse. Daarna komen de plotfuncties uitgebreid aan bod, met zowel aandacht voor grafieken van wiskundige functies als voor het voorstellen van experimentele gegevens. Het derde deel handelt over matrices en stelsels. In het laatste deel worden de structuren uit het vak Programmeren samengevoegd met de concepten van symbolisch en numeriek rekenen, teneinde zelf functies te schrijven die basisalgoritmen uit de wiskunde implementeren.

Hoofdstuk 1

Wetenschappelijk rekenen

1.1 Basisbewerkingen, variabelen, functies

Aritmetiek. `Python` kent uiteraard dezelfde aritmetische basisbewerkingen als andere rekensystemen: som (`+`), verschil (`-`), product (`*`), deling (`/`) en machtsverheffing (`**`). De eenvoudigste *statements* in `Python` zijn *expressies*, berekeningen bestaande uit deze operatoren, getallen en haakjes. Al wat in een statement achter een hekje (`#`) komt, wordt door `Python` genegeerd en dient als commentaar voor de gebruiker.

```
1+1
2
2-3
-1
5*8
40
13/21  # float: eindige mantisse
0.6190476190476191
21**34 # integer: oneindige precisie
9025183088777951914332401034032563746234570 81
```

Data-types voor getallen. Berekeningen met gehele getallen (type `int`) gebeuren in Python met oneindige precisie, uiteraard gelimiteerd door het computergeheugen. Decimale resultaten (type `float`) kennen slechts een eindig aantal beduidende cijfers en worden afgerond: wiskundig gezien zijn dit allemaal getallen, maar voor een computer zijn gehele en decimale getallen van een heel ander *type*. Dit type bepaalt welke functies en methoden er precies allemaal op een specifiek object van toepassing zijn en wat het effect er van is.

```
type(13/21)
float
type(21**34)
int
```

Python-functies. Een commando van de vorm `functie(argument)` is een functie-aanroep. Meer informatie over de werking van een specifieke functie is te bekomen met `?functie` of `help(functie)`. Voorbeelden zijn de functies `abs()` die de absolute waarde van een getal berekent en `round()` die een getal afrondt. Help-commando's en foute commando's zowel als de bijhorende uitvoer staan in deze tekst om technische redenen in commentaar: verwijder het commentaarsymbool `#` om deze te testen.

```
abs(-89)
89
round(144/233)
1
type(abs)
builtin_function_or_method
# ?abs
## abs(number) -> number
## Return the absolute value of the argument.
## Type:      builtin_function_or_method
```

Modulorekenen en logische operatoren. Andere veelgebruikte operatoren zijn de gehele deling (`//`), de rest bij gehele deling (ook gekend als modulo) (`%`), logische operatoren (`and`, `or`, `not`) en vergelijkinsoperatoren (`<`, `<=`, `>`, `>=`, `==`, `!=`). Het resultaat van een logische bewerking is waar (`True`) of vals (`False`). Een boolean (type `bool`) wordt in berekeningen automatisch naar 1 of 0 omgezet.

```
144 // 89
1
144 % 89
55
type(True)
bool
True and False
False
True or False
True
not False
True
233 < 377
True
610 != 987
True
1597 + True
1598
```

Complexe getallen. `Python` volgt de conventie uit de elektrotechniek en gebruikt het symbool `j` voor de imaginaire eenheid: merk op dat een coëfficiënt de letter `j` zonder operator moet voorafgaan. Het commando `complex(1,2)` resulteert in identiek hetzelfde als `1+2j`. Complexe getallen hebben enkele specifieke methoden. De syntax voor methoden is `object.methode(argument)`. Het is niet essentieel voor de gebruiker om te weten (het is ook niet altijd duidelijk) waarom een techniek als functie dan wel als methode is geïmplementeerd.

```
1+2j
(1+2j)
complex(1,2)
(1+2j)
(1+2j)+(3+4j)
(4+6j)
(5+6j)*(7+8j)
(-13+82j)
# (j)**2
## NameError: name 'j' is not defined
(1j)**2
(-1+0j)
(3+4j).real
3.0
(3+4j).imag
4.0
(3+4j).conjugate()
(3-4j)
abs(3+4j)
5.0
```

Toekenningen. Variabelen in Python hebben een naam én een waarde. De toekenning (`=`) mag niet verward worden met de wiskundige gelijkheid. Is er geen waarde aan toegekend, dan bestaat de variabele niet en kan de naam niet gebruikt worden. Verwijderen van een specifieke variabele kan met `del`, om alles uit het geheugen te wissen is er `%reset`, de optie `-f` onderdrukt de vraag om te bevestigen.

```
a = 1
a
1
a = a + 1
a
2
# b
## NameError: name 'b' is not defined
b = 2.0
b
2.0
del a
# a
## NameError: name 'a' is not defined
# %reset -f
# b
## NameError: name 'b' is not defined
```

Variabelen in Python (of in een programmeeromgeving in het algemeen) verschillen van wat in wiskunde variabelen worden genoemd. Om in Python met onbekenden te kunnen werken, moet een specifieke module worden ingeladen, wat aan bod komt in de volgende sectie.

1.2 Modules voor wiskundig rekenen

Python kent standaard niet de meest voor de hand liggende wiskundige functies. Er zijn echter ontelbare modules die de standaardmogelijkheden van Python quasi onbeperkt uitbreiden. Tabel 1.1 geeft een overzicht van de belangrijkste `Python`-uitbreidingen die in deze tekst aan bod komen. Voor het maken wiskundige berekeningen worden meteen twee modules geïntroduceerd: `sympy` en `numpy`. Het eerste pakket focust op exacte, symbolische resultaten, terwijl het tweede eerder voor numerieke berekeningen bedoeld is.

Importeren van modules. Importeren van alle commando's uit zo een module gebeurt met `import`. Het commando `help(module)` geeft een overzicht van alle commando's in de module. De commando's kunnen dan worden aangeroepen met `module.commando(argument)`. De veelgebruikte functie `sqrt()` berekent de vierkantswortel en heeft in beide pakketten een andere implementatie. In de code hieronder lijkt het misschien alsof `sympy.sqrt(2)` niet wordt uitgerekend, maar er is gewoon geen eenvoudiger exacte vorm om dat getal te schrijven. De methode `.n()` laat toe de numerieke waarde te berekenen.

Tabel 1.1: Belangrijkste `Python`-uitbreidingen in deze tekst

Pakket	Functionaliteit
IPython	Interactieve shell
SymPy	Symbolisch rekenen
NumPy	Numeriek rekenen
Matplotlib	Grafieken

```
# sqrt(2)
## NameError: name 'sqrt' is not defined
import numpy
# help(numpy)
## ...
numpy.sqrt(2)
1.4142135623730951
import sympy
# help(sympy)
## ...
sympy.sqrt(2)
sqrt(2)
sympy.sqrt(2).n()
1.41421356237310
```

`sympy` versus `numpy`. Kennelijk is de boogsinus in `numpy` als `arcsin()` en in `sympy` als `asin()` geïmplementeerd. Voor het gemak kan de gebruiker beide functies dezelfde naam geven. De wiskundige constante π is eveneens in beide pakketten gedefinieerd: in `numpy` is dat louter een getal, in `sympy` een object met naast de getalwaarde nog tal van eigenschappen. Het is in het algemeen niet mogelijk objecten en functies uit verschillende modules zomaar door elkaar te gebruiken.

```
numpy.asin = numpy.arcsin
numpy.asin(1)
1.5707963267948966
sympy.asin(1)
pi/2
numpy.pi
3.141592653589793
type(numpy.pi)
float
sympy.pi
pi
type(sympy.pi)
sympy.core.numbers.Pi
sympy.pi.n()
3.14159265358979
sympy.sin(sympy.pi)
0
sympy.exp(sympy.pi*sympy.I)
-1
sympy.sin(sympy.pi/5)
sqrt(-sqrt(5)/8 + 5/8)
# numpy.sin(sympy.pi)
## AttributeError: 'Pi' object has no attribute 'sin'
```

Complexe getallen in `sympy`. De imaginaire eenheid `j` blijkt enkel geschikt voor numeriek gebruik, in `sympy` bestaat alternatief de constante `I` die geschikt is voor symbolische berekeningen. Het commando `arg()` berekent het argument van een complex getal.

```
sympy.exp(1j * sympy.pi)
exp(1.0*I*pi)
sympy.exp(1j * sympy.pi).n()
-1.0 + 0.e-21*I
sympy.exp(sympy.I * sympy.pi)
-1
sympy.arg(1+sympy.I)
pi/4
```

1.3 Symbolisch rekenen

Werken met één module. Uit de voorbeelden in de voorgaande sectie blijkt duidelijk waarom de modulenaam steeds voor de functienaam moet worden vermeld. Is er echter geen verwarring mogelijk, of zal slechts één variant worden gebruikt, dan kunnen specifieke functies één voor één worden gedefinieerd zonder module-prefix. Met `from module import *` kunnen zelfs alle commando's ineens verkort worden ingevoerd. Het is absoluut af te raden om dit te doen als er meerdere pakketten door elkaar gebruikt worden. Let steeds op met samenvallende functienamen bij gebruik van meerdere pakketten!

```
125 sin = sympy.sin
126 pi = sympy.pi
127 sin(pi)
128 0
129 from sympy import *
130 exp(I*pi)
131 -1
```

Breuken. Alle bewerkingen in het vervolg van dit hoofdstuk gebruiken dus functies uit de module `sympy`. Om consequent exacte berekeningen te kunnen maken, zijn nog wat extra technieken vereist. In uitkomsten van berekeningen in `sympy` blijkt dat breuken exact bewaard blijven, maar in aritmetische berekeningen is dat niet zo. Om expliciet met breuken te werken in plaats van met decimale getallen is er de functie `Rational()`. Teller en noemer van een breuk selecteren, zal vaak nuttig blijken.

```
132 sin(pi/6)
133 1/2
134 1/2
135 0.5
136 q = Rational(1,2); q
137 1/2
138 numer(q)
139 1
140 denom(q)
141 2
```

Wiskundige veranderlijken. Om interessante dingen te kunnen doen, zijn wiskundige variabelen nodig. Deze moeten worden geïnitialiseerd met het commando `symbols()` dat een `Python`-object creëert waarvan de *waarde* een wiskundig symbool is. In het voorbeeld hieronder worden de `Python`-objecten `abscis` en `ordinaat` gemaakt met als wiskundige symbolen `x` en `y`. In de praktijk zullen de naam van het object en die van de wiskundige veranderlijke meestal hetzelfde zijn, maar het is niet onbelangrijk het verschil tussen beide te begrijpen.

```
142 abscis = symbols('x')
143 ordinaat = symbols('y')
144 type(abscis)
145 sympy.core.symbol.Symbol
146 abscis + ordinaat
147 x + y
148 # abscis == x
149 # NameError: name 'x' is not defined
```

Polynomen. Met symbolen en aritmetische bewerkingen kunnen bijvoorbeeld polynomen worden geconstrueerd, die met `expand()` en `factor()` kunnen worden herschreven. Noteer dat de test voor gelijkheid (`==`) nagaat of `Python`-objecten identiek zijn, niet of de corresponderende objecten wiskundig gelijk zijn: daarvoor is er de methode `.equals()`.

```
x,y,z = symbols('x y z')
polynoom = x**2 + 2*x +1; polynoom
x**2 + 2*x + 1
factorisatie = factor(polynoom); factorisatie
(x + 1)**2
ontbinding = expand(factorisatie); ontbinding
x**2 + 2*x + 1
factorisatie == ontbinding
False
factorisatie.equals(ontbinding)
True
```

Rationale functies. Gangbare bewerkingen bij rationale functies zijn het schrappen van gemeenschappelijke factoren in teller en noemer (`cancel()`) en het schrijven in partieelbreuken (`apart()`). Net zoals bij breuken kunnen teller en noemer worden geselecteerd, handig om later nulpunten, perforaties en asymptoten te berekenen.

```
breuk = (x**3+2*x**2-x-2) / (x**2-x-2); breuk
(x**3 + 2*x**2 - x - 2)/(x**2 - x - 2)
cancel(breuk)
(x**2 + x - 2)/(x - 2)
apart(breuk)
x + 3 + 4/(x - 2)
numer(breuk)
x**3 + 2*x**2 - x - 2
denom(breuk)
x**2 - x - 2
```

Exponentiële en logaritmische functies. Het getal van Euler is in `sympy` gedefinieerd als `E` maar meestal wordt in rekensoftware de exponentiële functie `exp()` gebruikt. Het logaritme `log()` staat zoals in de meeste computerpakketten voor het natuurlijke logaritme. Om met tiendelige (of andere) logaritmen te werken, moet de basis als tweede argument aan het commando `log()` worden meegegeven. De logaritmische functie is enkel gedefinieerd voor positieve (reële) getallen, de exponentiële functie ook voor complexe getallen, waardoor een misschien verwachte vereenvoudiging niet optreedt. Voor positieve getallen gebeurt dat wel.

```
N(E)
2.71828182845905
E**x == exp(x)
True
ln(x)
log(x)
log(1000,10)
3
exp(ln(x))
x
ln(exp(x))
log(exp(x))
x = symbols('x',positive=True)
ln(exp(x))
x
```

1.4 Functie-analyse

Expressies versus functies. De objecten `polynoom` en `breuk` uit de vorige sectie zijn geen functies. In `Python` blijkt dat onder meer omdat ze niet als een functie kunnen geëvalueerd worden. In deze tekst worden dit soort uitdrukkingen expressies genoemd. Evaluatie van expressies kan indirect door de gewenste waarde in de veranderlijke te substitueren met de methode `.subs()`. Vaak is het in de context van wiskundig rekenen aangewezen om met functie te kunnen werken, gebruik daarvoor de syntax `def f(x): return ...`. Later blijkt dat dit ook in programmeercontext een enorm belangrijke constructie is.

```
x = symbols('x')
expressie = sin(x)/x
# expressie(pi)
## TypeError: 'Mul' object is not callable
expressie.subs(x,pi)
0
def f(x): return sin(x)/x
f(pi)
0
```

Limieten. Het commando voor limietberekeningen is `limit()` en heeft vier argumenten: een expressie, het argument, het limietpunt en de richting, links (`'-'`) of rechts (`'+'`). Het commando `oo` (2 keer de letter o) staat voor (plus) oneindig $+\infty$. Let op: als het vierde argument niet gespecifieerd is, berekent `Python` zonder meer de rechter limiet.

```
limit(f(x),x,0)
1
limit(f(x),x,oo)
0
limit(1/x,x,0)
oo
limit(1/x,x,0,'+')
oo
limit(1/x,x,0,'-')
-oo
```

Oneindig en meer. Rekenen met oneindig volgt de bekende rekenregels, maar waar een functie niet gedefinieerd is (en de functiewaarde zou kunnen verward worden met de limiet), is `Python` niet steeds consequent. Het resultaat varieert van een foutmelding, over complex oneindig (`zoo`) tot een onbekende scalaire uitkomst (`nan`). De meeste bewerkingen met deze laatste waarden resulteren opnieuw in het onbepaalde `nan`.

```
oo-2
oo
-3*-oo
oo
oo+oo
oo
oo-oo
nan
# 1/0
## ZeroDivisionError: division by zero
tan(pi/2)
zoo
f(0)
nan
nan+1
nan
```

Afgeleiden. Met het commando `diff(expr,x,n=1)` worden afgeleiden berekend, zonder derde argument `n` standaard de eerste orde afgeleide. Vaak zijn de belangrijkste bewerkingen op (wiskundige) functies enkel gedefinieerd voor expressies en niet voor `Python`-functies. De output is doorgaans ook een expressie, evaluatie kan via `.subs()`. Omzetten van deze expressie naar een functie is wat omslachtig, omdat de `def`-constructie het argument substitueert vooraleer de code wordt uitgevoerd, terwijl de methode `.subs()` eerst het object bepaalt en dan pas de waarde substitueert: let op de verschillende rol van `x` en `x0`.

```
n = symbols('n')
diff(x**n,x)
n*x**n/x
diff(sin(x),x)
cos(x)
diff(sin(x),x,2)
-sin(x)
diff(sin(x),x,3)
-cos(x)
diff(sin(x),x,4)
sin(x)
diff(sin(x),y)
0
diff(sin(x),x).subs(x,0)
1
# def Df(x): return diff(f(x),x)
# Df(pi)
## ValueError: Can't calculate 1-th derivative wrt pi.
def Df(x0): return diff(f(x),x).subs(x,x0)
Df(x)
cos(x)/x - sin(x)/x**2
Df(1)
-sin(1) + cos(1)
```

Integralen. Het commando `integrate(expr,var)` berekent een primitieve (de onbepaalde integraal zonder integratieconstante) of met argumenten van een andere vorm `integrate(expr,(var,a,b))` een bepaalde integraal. Ook als een functie integreerbaar is, is er vaak geen elementaire functie voor bekend. Soms heeft zo een primitieve een naam, zoals `Si` in het voorbeeld hieronder, zoniet zijn in- en uitvoer gelijk. Dat de primitieve geen elementaire functie is, wil niet noodzakelijk zeggen dat de functie niet integreerbaar zou zijn: vaak kan de berekening wel numeriek. De resulterende expressie evalueren gebeurt zoals anders.

```
integrate(sin(x),x)
-cos(x)
integrate(sin(x),(x,0,pi))
2
integrate(sin(x)/ln(x),x)
Integral(sin(x)/log(x), x)
integrate(sin(x)/x,x)
Si(x)
integrate(sin(x)/x,(x,0,oo))
pi/2
integrate(sin(x)/x,(x,0,pi))
Si(pi)
N(integrate(sin(x)/x,(x,0,pi)))
1.85193705198247
integrate(sin(x),x).subs(x,0)
-1
```

Reeksontwikkelingen. Het commando voor reeksontwikkelingen `series()` berekent de eerste termen van de Laurent-reeks. Voor toepassing in dit vak volstaat het te weten dat dit een uitbreiding is van de Taylorreeks voor complexe functies waarbij ook negatieve machten kunnen voorkomen. De term `O(x**6)` geeft aan vanaf welke orde de termen verwaarloosd worden. Deze term kan weggelaten worden met de methode `.removeO()`, pas dan wordt evaluatie van de reeksbenadering zinvol. Standaard wordt de functie ontwikkeld rond de oorsprong, maar indien gewenst kan in het derde argument een ander punt worden meegegeven.

```
series(sin(x),x)
```

```
x - x**3/6 + x**5/120 + O(x**6)
```

```
series(sin(x),n=10)
```

```
x - x**3/6 + x**5/120 - x**7/5040 + x**9/362880 + O(x**10)
```

```
series(1/x,x)
```

```
1/x
```

```
series(1/x,x,1)
```

```
2 + (x - 1)**2 - (x - 1)**3 + (x - 1)**4 - (x - 1)**5 - x + O((x -
1)**6, (x, 1))
```

```
series(sin(x)/x**2,x)
```

```
1/x - x/6 + x**3/120 - x**5/5040 + O(x**6)
```

```
series(sin(x)/x**2,x).subs(x,1)
```

```
O(1)
```

```
series(sin(x)/x**2,x).removeO()
```

```
-x**5/5040 + x**3/120 - x/6 + 1/x
```

```
series(sin(x)/x**2,x).removeO().subs(x,1)
```

```
4241/5040
```

```
N(4241/5040)
```

```
0.841468253968254
```

```
N(sin(1))
```

```
0.841470984807897
```

Onbepaalde functies. In theoretische berekeningen is het soms handig om niet nader bepaalde functies te kunnen gebruiken. Zo een object wordt gemaakt met het commando `Function()`. Achteraan het commando kunnen de argumenten van de functie worden toegevoegd.

```
f = Function('f')
g = Function('g')
diff(f(x)*g(x),x)
```

```
f(x)*Derivative(g(x), x) + g(x)*Derivative(f(x), x)
```

1.5 Vergelijkingen en ongelijkheden

Vergelijkingen. Eerder bleek al het verschil tussen de toekenning (`=`) en de logische test voor gelijkheid (`==`). Vergelijkingen zoals bijvoorbeeld $x^2 = 1$, $x^2 = -1$ en zelfs het triviale $x = 5$ zijn nog helemaal wat anders: ze zijn (voor een onbekende x) waar noch vals en impliceren evenmin een toekenning, ongeacht het feit of er oplossingen bestaan en of deze bekend zijn. Vergelijkingen in `sympy` worden bepaald door het commando `Eq(expr1,expr2)` en opgelost met het commando `solve(eqn,var)`. Is het rechter lid van een vergelijking 0, dan volstaat enkel de uitdrukking uit het linkerlid. Veranderlijken worden beschouwd als complex, tenzij anders ingesteld. Als er meerdere onbekenden voorkomen, moet worden gespecifieerd wat de hoofdonbekende is. Het is belangrijk om te beseffen dat (1) niet alle vergelijkingen oplossingen hebben, (2) het niet steeds mogelijk is om de oplossing/alle oplossingen van een vergelijking (symbolisch of numeriek) te vinden, zelfs als die bestaan en dus (3) een vergelijking best wel (meer) oplossingen kan hebben, ook al vindt `sympy` er geen of slechts enkele. In het geval het commando `solve()` oplossingen niet kan vinden, is het mogelijk om deze numeriek te proberen benaderen. Ook dat probleem is veel te technisch om hier in detail te behandelen, maar een snelle (en feilbare) manier is `nsolve(expr,var,start)`, waarbij steeds een startpunt moet worden meegegeven. Let op dat numerieke oplossingen slechts bij benadering juist zijn en afhangen van het startpunt.

```
x==5
False
solve(x==5)
False
Eq(x,5)
Eq(x, 5)
solve(Eq(x,5))
[5]
solve(x**2==1)
False
solve(x**2-1)
[-1, 1]
solve(x**2+1)
[-I, I]
solve(x**3-1)
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
x = symbols('x',real=True)
solve(x**2+1)
[]
solve(x**3-1)
[1]
a,b,c,x = symbols('a b c x')
solve(a*x**2+b*x+c)
[{a: -(b*x + c)/x**2}]
solve(a*x**2+b*x+c,x)
[(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
solve(sin(x),x)
[0, pi]
# solve(x+cos(x),x)
## NotImplementedError: No algorithms are implemented to solve equation
nsolve(cos(x)+x,x,0.5)
-0.739085133215161
nsolve(sin(x),x,-3)
-3.14159265358979
nsolve(sin(x),x,6)
6.28318530717959
```

Stelsels. Een stelsel is niets meer dan een verzameling van vergelijkingen. Deze worden in `sympy` opgesomd tussen vierkante haken (in lijstvorm, zo blijkt later), verder gelden dezelfde principes als bij vergelijkingen. Lineaire stelsels oplossen gebeurt doorgaans via het matrixformalisme, dat komt aan bod in hoofdstuk 3.

```
a,b,c,d,e,f,x,y = symbols('a b c d e f x y')
solve([x/y-y/x,x-y**2],x,y)
[(1, -1), (1, 1)]
solve([x*(x - a), x**2 + y],x,y)
[(0, 0), (a, -a**2)]
solve([a*x+b*y-c,d*x+e*y-f],x,y)
{x: (-b*f + c*e)/(a*e - b*d), y: (a*f - c*d)/(a*e - b*d)}
```

Ongelijkheden. Dezelfde technieken als hierboven gelden voor het oplossen van ongelijkheden, met als grootste verschil dat het resultaat vaak een (unie van) interval(len) is.

```
solve(x**2>1)
((-oo < x) & (x < -1)) | ((1 < x) & (x < oo))
```

Hoofdstuk 2

Grafieken

De meeste wetenschappelijke publicaties bevatten grafische voorstellingen van gegevens of resultaten. Opdat deze beelden de juiste informatie zouden overbrengen, is het belangrijk die zo goed mogelijk te verzorgen. Daarom wordt in dit hoofdstuk uitgebreid aandacht besteed aan het maken, verzorgen en exporteren van grafieken zodat deze als duidelijke illustraties in wetenschappelijke teksten kunnen worden gebruikt

2.1 Smart-plots in `sympy`

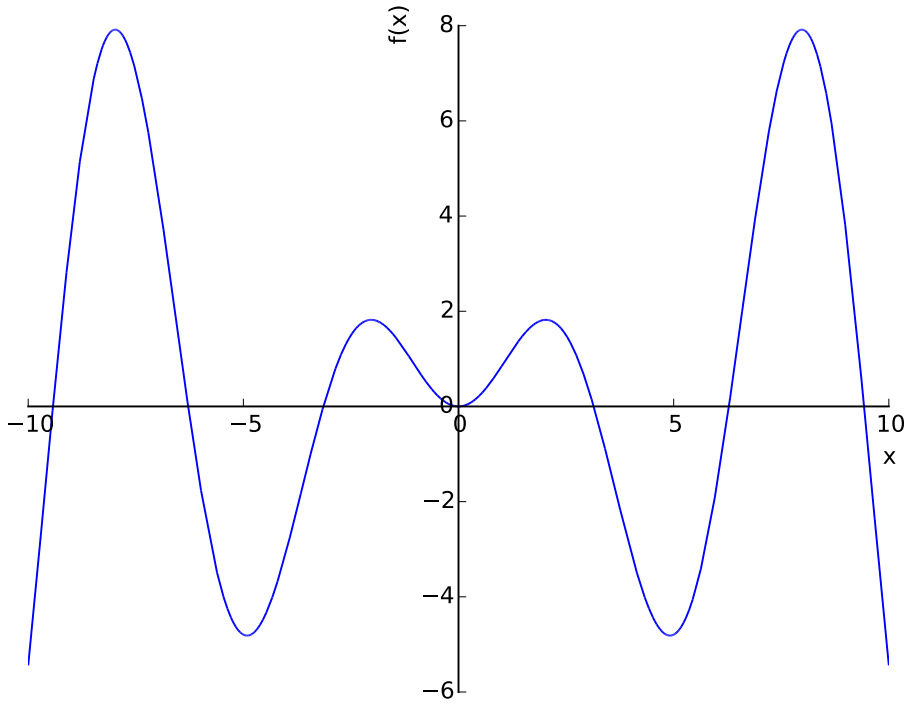
Het pakket `sympy` maakt zogenaamde smartplots van functies en parameterkrommen in twee en drie dimensies: op basis van het voorschrift, worden deze krommen of oppervlakken automatisch getekend. Hoe glad een computergrafiek er ook uit ziet, het is allerbelangrijkst altijd te onthouden dat zo een grafiek enkel bestaat uit een aantal punten die één voor één werden uitgerekend om daarna door het computerprogramma zijn getekend en verbonden. De module `sympy.plotting` doet dit dus automatisch, maar om meer gespecialiseerde dingen te doen, zeker bijvoorbeeld bij het werken met meetwaarden, dient de gebruiker zelf punten te kiezen en te berekenen.

De module `sympy.plotting`. Commando's voor het tekenen van `sympy`-functies zijn gebundeld in een submodule `plotting` die afzonderlijk dient te worden ingeladen. Hieronder worden enkele functies gedefinieerd en al een eerste grafiek gemaakt, het resultaat staat in Figuur 2.1.

```
330 from sympy import *
331 from sympy.plotting import *
332 x,y,t,s = symbols('x y t s')
333 def f(x): return x*sin(x)
334 plot(f(x))
```

`Plot`-objecten. Indien niet gespecifieerd, kiest het commando `plot()` als grenzen -10 en 10. Met als tweede argument `(x,a,b)` kan het bereik voor de veranderlijke `x` van linkergrens `a` tot rechtergrens `b` worden ingesteld. Een heel aantal andere opties laten toe de lay-out van de afbeelding te wijzigen. Het is tevens zeer eenvoudig om meerdere grafieken te tekenen, weliswaar allemaal in dezelfde kleur. De output van onderstaande commando's is niet in de tekst opgenomen.

```
335 plot(f(x),(x,0, 8*pi))
336 plot(f(x),line_color='red',
337     title='Grafiek van de functie',
```



Figuur 2.1: De grafiek van een functie

```
    xlim = (-5,15),
    ylim = (-2,2))
def Df(x0): return diff(f(x),x).subs(x,x0)
def D2f(x0): return diff(Df(x),x).subs(x,x0)
plot(f(x),Df(x),D2f(x),(x,0,2*pi))
```

In wezen is een grafiek een object, dat kan worden gemanipuleerd, afgebeeld of weggeschreven naar een bestand. Elke afgebeelde functie krijgt een index (startend bij 0) en heeft methoden `.line_color` en `.label`. Het resultaat tonen kan met de methode `.show()`, wegschrijven met `.save()`.

De extensie van de bestandsnaam bepaalt het formaat van de afbeelding. Kies altijd voor een vectorformaat (`.pdf`) want rasterafbeeldingen zijn nooit even gedetailleerd: ofwel onduidelijk ofwel zeer groot.

```
plot_f = plot(f(x),Df(x),D2f(x),(x,0,2*pi), show=False,
  legend=True, xlabel=latex('$x$'), ylabel=latex('$y$'))
plot_f[0].line_color='red'
plot_f[1].line_color='blue'
plot_f[2].line_color='green'
plot_f[0].label = latex('$f$')
plot_f[1].label = latex('$Df$')
plot_f[2].label = latex('$D^2 f$')
plot_f.show()
plot_f.save('figuur.pdf')
```

Het resultaat is te zien in figuur 2.2.

Impliciete en parameterkrommen. De voorbeelden hierboven zijn Cartesische krommen, bepaald als de grafiek van een functie f,

$$\big\{\big(x, f(x)\big) \mid x \in [a,b]\big\}.$$

Een andere manier om krommen te bepalen is door een parametervoorstelling,

$$\big\{\big(f(t), g(t)\big) \mid t \in [a,b]\big\}.$$

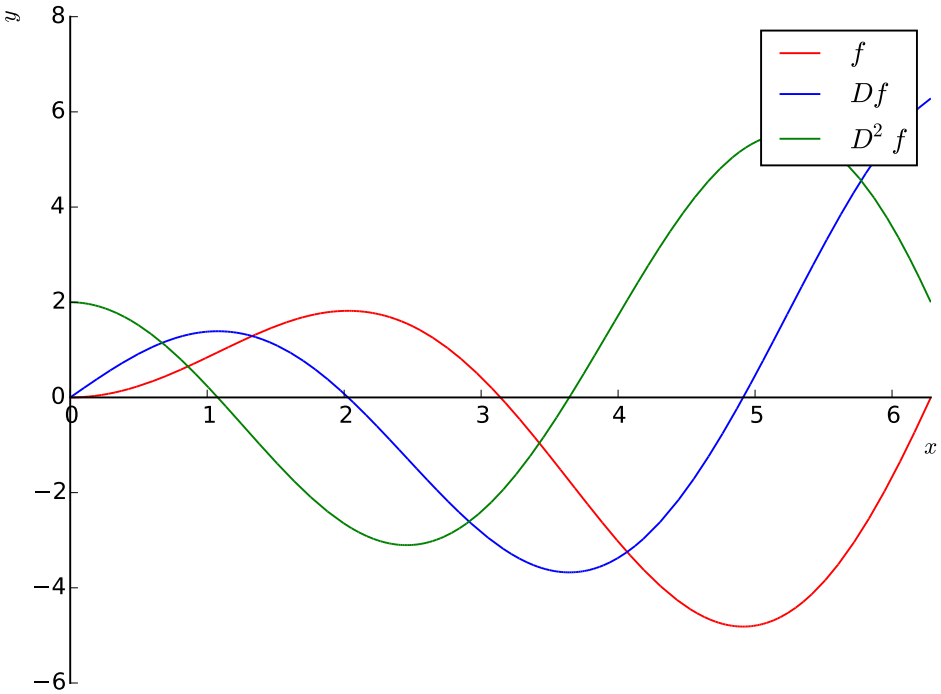
Impliciet bepaalde krommen ten slotte worden gegeven door een vergelijking $f(x,y) = 0$ die niet noodzakelijk (eenvoudig) oplosbaar is,

$$\big\{\big(x,y\big) \in [a,b] \times [c,d] \mid f(x,y) = 0\big\}.$$

Er is geen algemene methode om deze puntenverzameling te bepalen.

De eenheidscirkel, de verzameling van punten op afstand 1 van de oorsprong, kan op elk van de drie manieren omschreven worden:

- de definitie van de eenheidscirkel leidt dadelijk tot de impliciete vergelijking
$$x^2 + y^2 = 1;$$
- de eenheidcirkel is de unie van de cartesische krommen bepaald door de functies met voorschrift
$$\sqrt{1-x^2} \text{ en } -\sqrt{1-x^2} \text{ met telkens } x \in [-1,1];$$
- de grondformule van de goniometrie levert dat beide voorgaande equivalent zijn met de parametervoorstelling
$$\big(\cos(t), \sin(t)\big) \text{ met } t \in [0, 2\pi].$$



Figuur 2.2: De grafieken van een functie en haar afgeleiden

Onderstaande commando's illustreren de syntax voor deze drie soorten grafieken, de output is dus drie keer identiek aan wat te zien is in Figuur 2.3.

```
plot(sqrt(1-x**2),-sqrt(1-x**2),(x,-1,1))
plot_parametric(cos(t),sin(t),(t,-pi,pi))
plot_implicit(x**2+y**2-1,(x,-1,1),(y,-1,1))
```

3D-grafieken in `sympy`. Er zijn ook commando's voor grafieken in drie dimensies: de grafiek van een functie van twee veranderlijken $\big(x, y, f(x, y)\big)\}$, een ruimtekromme $\big(f(t), g(t), h(t)\big)$ en een parameteroppervlak $\big(f(s,t), g(s,t), h(s,t)\big)$. Hieronder een voorbeeld van elk, het resultaat in Figuren 2.4, 2.5 en 2.6.

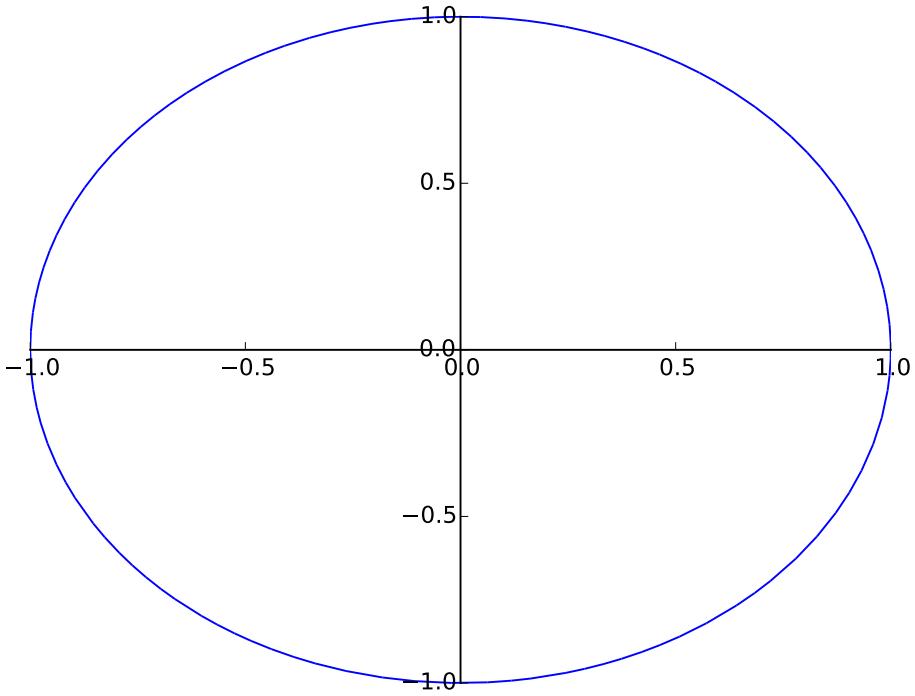
```
plot3d(x*sin(y),(x,-10,10),(y,-2*pi,2*pi))
plot3d_parametric_line(t,cos(t),sin(t),(t,0,4*pi))
plot3d_parametric_surface(s,cos(t),sin(t),(t,0,2*pi),(s,0,4*pi))
```

2.2 Grafieken met `matplotlib`

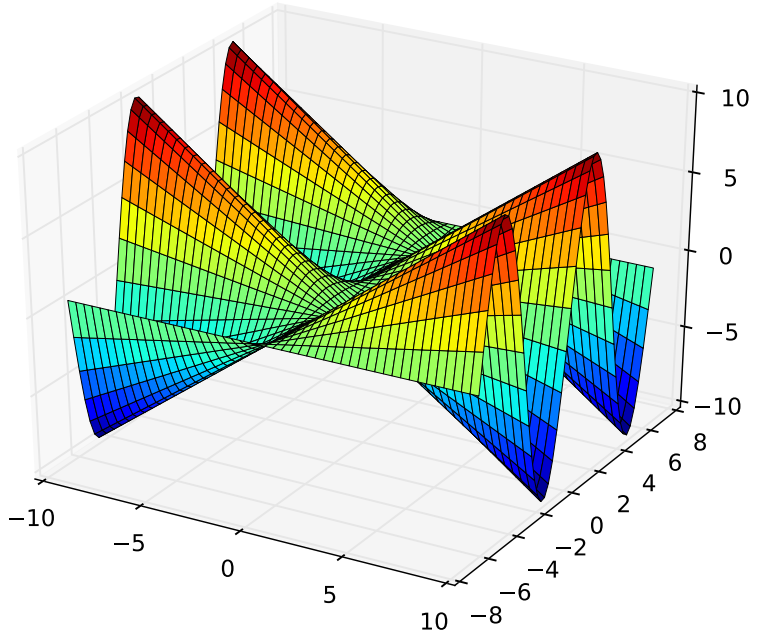
De grafieken in de vorige sectie zijn smart-plots: de module `sympy` bepaalt zelf hoeveel en welke punten tussen de opgegeven eindpunten worden berekend. In veel gevallen wil de gebruiker echter meer controle over het maken van grafieken. De module `matplotlib` breidt de plot-mogelijkheden van `Python` bijna ongelimiteerd uit. Omdat veel berekeningen nu expliciet zullen moeten gebeuren, wordt werken met de module `numpy` ook onontbeerlijk. Bij het combineren van meerdere modules is het altijd aangewezen om niet expliciet alle commando's te importeren maar enkel de modules, teneinde naamconflicten te vermijden. Is het niet de bedoeling om wiskundige eigenschappen te gebruiken, werk dan meteen met `numpy`-commando's, waarvan de rekentijd doorgaans veel korter is dan bij de corresponderende `sympy`-commando's.

Lijsten van waarden berekenen. Waar de syntax van een smart-plot commando enkel een functievoorschrift gebruikt, moeten in het vervolg doorgaans expliciet x- en y-waarden van alle te tekenen punten worden berekend. De eenvoudigste manier om waarden voor de abscis te kiezen, is punten op gelijke afstand van elkaar. De commando's `linspace()` en `arrange()` doen dat voor gegeven begin- en eindpunt respectievelijk met een vast aantal en met een vaste tussenafstand. De waarden voor de ordinaat moeten één voor één bij deze x-waarden worden berekend, wat `numpy`-commando's automatisch puntsgewijs doen. Dat is niet zo bij `sympy` dat functies beschouwt als afbeeldingen $f : \mathbb{R} \to \mathbb{R}$. Gebruik `lambdify()` als work-around.

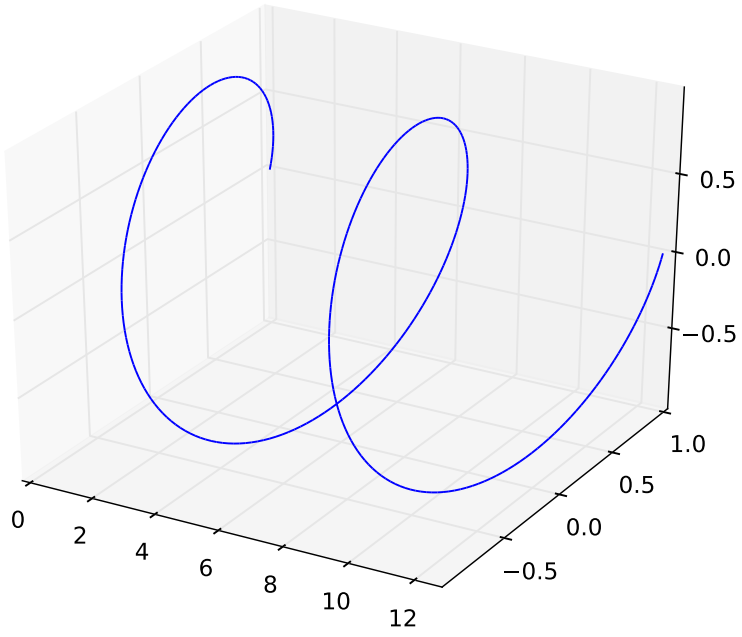
```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
np.arange(0,1,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
xx = np.linspace(0,1,10); xx
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
np.sin(xx)
array([0.        , 0.11088263, 0.22039774, 0.3271947 , 0.42995636,
       0.52741539, 0.6183698 , 0.70169788, 0.77637192, 0.84147098])
# sp.sin(xx)
# AttributeError: 'numpy.ndarray' object has no attribute 'is_Number'
x = sp.symbols('x')
sp.lambdify(x, sp.sin(x), "numpy")(xx)
array([0.        , 0.11088263, 0.22039774, 0.3271947 , 0.42995636,
       0.52741539, 0.6183698 , 0.70169788, 0.77637192, 0.84147098])
```



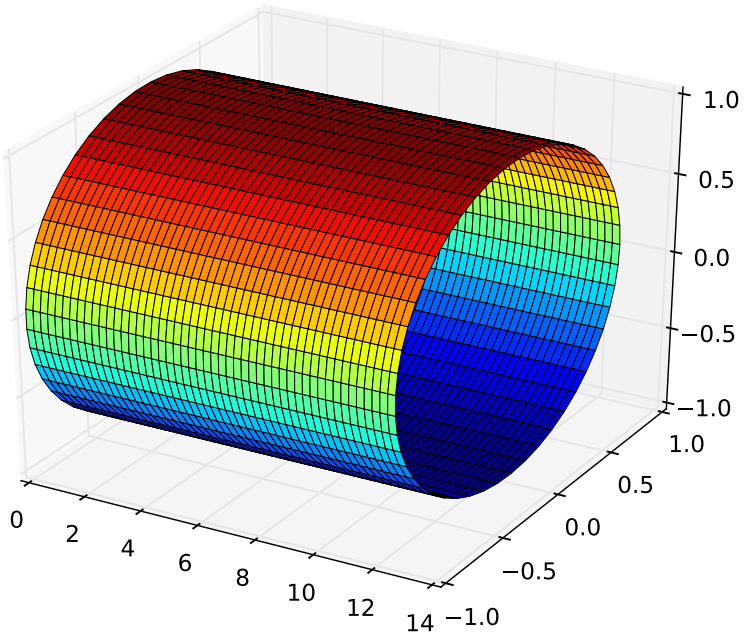
Figuur 2.3: De eenheidscirkel



Figuur 2.4: De grafiek van de functie $(x, y) \mapsto x \cdot \sin(y)$



Figuur 2.5: De ruimtekromme $t \mapsto \big(t, \cos(t), \sin(t)\big)$



Figuur 2.6: Het parameteroppervlak $(s, t) \mapsto \big(s, s \cdot \cos(t), s \cdot \sin(t)\big)$

Grafieken in `matplotlib`. Een eerste verschil met `sympy`-grafieken is dat plotpunten expliciet berekend moeten worden, dat is hierboven al gebeurd. Verder opent het commando `figure()` een plot-object waaraan opeenvolgende `plot()`-commando's informatie toevoegen, tot het geheel wordt weggeschreven (`savefig()`) of gesloten (`close()`). Onderstaande grafiek is niet opgenomen in de tekst.

```
xx = np.linspace(0,2*np.pi,100)
plt.figure()
plt.plot(xx,np.sin(xx))
plt.show()
plt.savefig('plot.pdf')
plt.close()
```

Lijn- en puntstijl. Korte codes laten toe de lijn- en puntstijl van een grafiek in te stellen: de code `'b-'` tekent een blauwe lijn, `'g:*'` een groene stippellijn met sterretjes als meetpunten en `'r--o'` een rode streepjeslijn met cirkelvormige meetpunten. Een overzicht van de meest gebruikte opties staat in Tabel 2.1. Lijndikte wordt ingesteld met een afzonderlijke optie `linewidth`, en de optie `label` stelt tekst in de legende in. Het bereik van de grafiek kan worden ingesteld met `xlim()` en `ylim()`. Er zijn nog meer mogelijkheden, waarvan een overzicht is te vinden met `?plt.plot`. Het resultaat van onderstaande code is te zien in Figuur 2.7.

```
xx = np.linspace(0,2*np.pi,20)
plt.figure()
plt.plot(xx,np.sin(xx),'b-', label='Sinus')
plt.plot(xx,np.cos(xx),'g:*',lw=2, label='Cosinus')
plt.plot(xx,np.tan(xx),'r--o', label='Tangens')
plt.ylim([-2,2])
plt.xlim([np.pi/2,3*np.pi/2])
plt.legend()
```

Poolkrommen. Een bijzondere categorie van parameterkrommen die nog niet aan bod kwam, zijn de poolkrommen, bepaald door de poolvergelijking $r = f(\theta)$ en gevormd door de verzameling punten

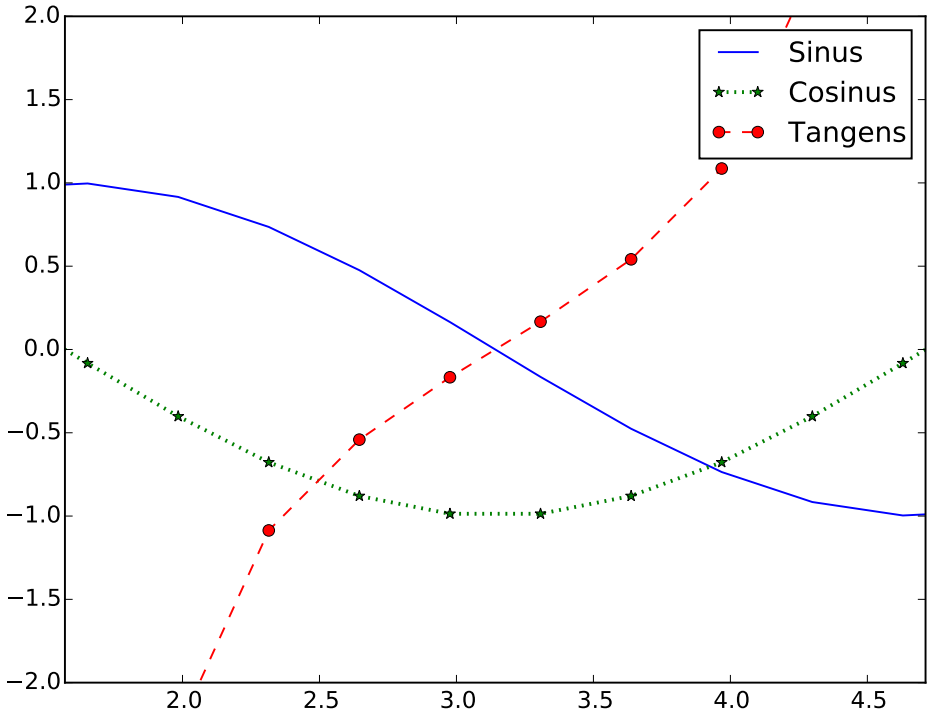
$$\big\{\big(f(\theta)\cos(\theta), f(\theta)\sin(\theta)\big) \mid \theta \in [\alpha, \beta]\big\}.$$

Hieronder enkele voorbeelden met het resultaat in Figuur 2.8.

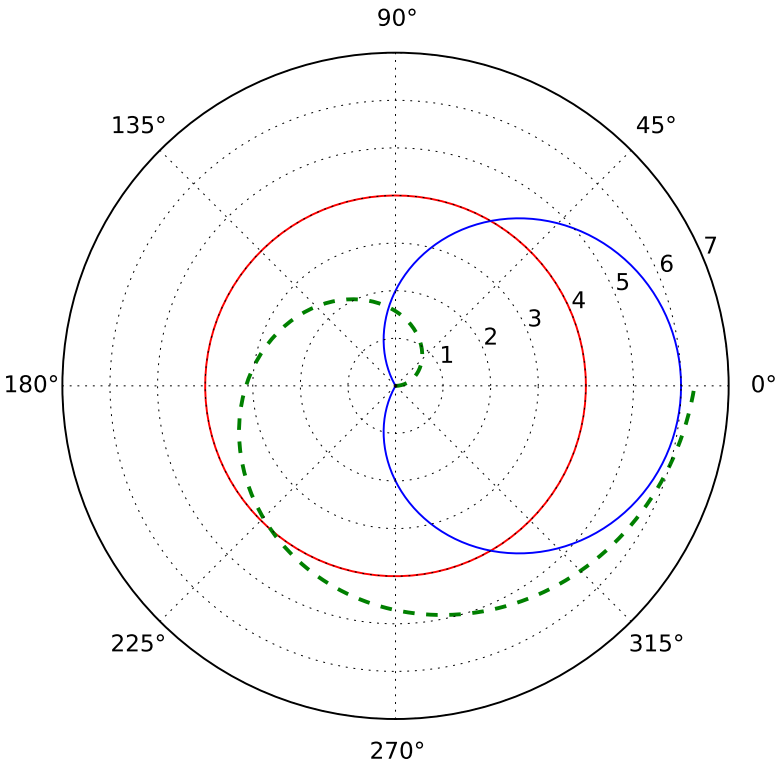
```
xx = np.linspace(0,2*np.pi,200)
plt.figure()
plt.polar(xx,np.repeat(4,200),'r-')
plt.polar(xx,xx,'g--',lw=2)
plt.polar(xx,2+4*np.cos(xx),'b')
```

Tabel 2.1: Verschillende lijn- en puntstijlen van grafieken

Puntstijl		Kleur		Lijnstijl	
`'.'`	punt	`'b'`	blauw	`'-'`	doorlopende lijn
`'o'`	cirkel	`'g'`	groen	`'--'`	streepjeslijn
`'s'`	vierkant	`'r'`	rood	`'-.'`	punt-streeplijn
`'p'`	vijfhoek	`'c'`	cyaan	`':'`	stippellijn
`'*'`	ster	`'m'`	magenta		
`'h'`	achthoek	`'y'`	geel		
`'+'`	plusteken	`'k'`	zwart		
`'d'`	ruit	`'w'`	wit		



Figuur 2.7: Drie grafieken met `matplotlib`



Figuur 2.8: Drie poolkrommen

Subplots. Is het nodig verschillende grafieken te vergelijken, dan staan die best steeds in hetzelfde assenstelsel. Is dat niet mogelijk, bijvoorbeeld omdat de schalen verschillen, dan kunnen de grafieken naast of onder elkaar gezet worden met het commando `subplot(i,j,k)`. Dit deelt het grafiekgebied op in `i` rijen en `j` kolommen en selecteert de `k`-e grafiek (van links naar rechts en van boven naar beneden geteld). Als voorbeeld de grafiek van de exponentiële functie met alle combinaties van lineaire en logaritmische schalen, het loont de moeite de output in Figuur 2.9 te bestuderen.

```
xx = np.linspace(0,2,100)
plt.figure()
plt.subplot(2,2,1)
plt.plot(xx,np.exp(xx))
plt.subplot(2,2,2,xscale='log')
plt.plot(xx,np.exp(xx))
plt.subplot(2,2,3,yscale='log')
plt.plot(xx,np.exp(xx))
plt.subplot(2,2,4,xscale='log',yscale='log')
plt.plot(xx,np.exp(xx))
```

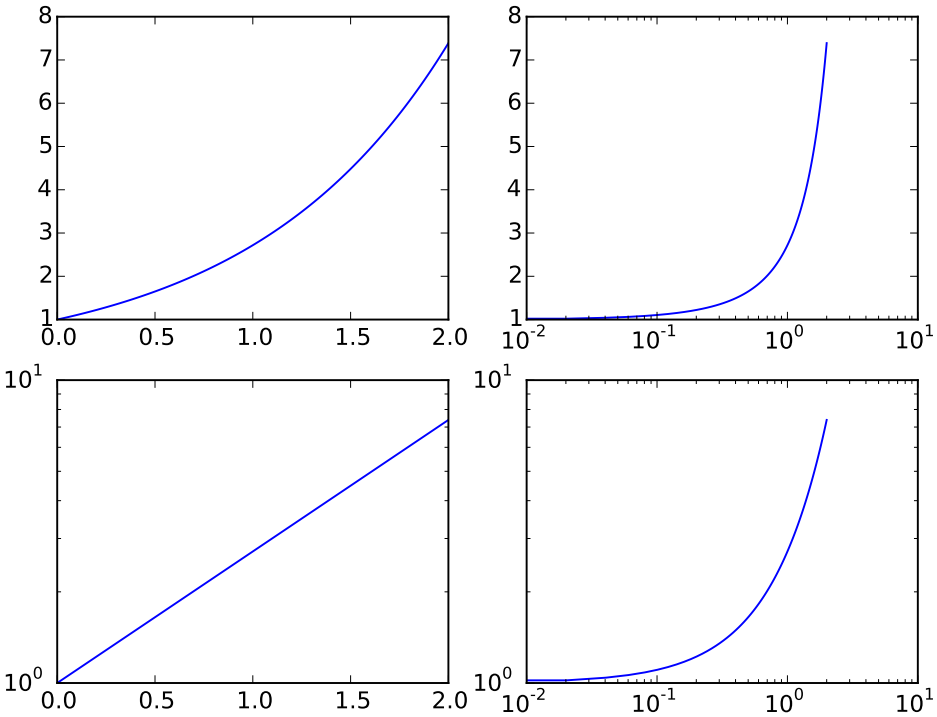
Andere grafiektypes. Interessante grafiektypes waarvan geen voorbeelden in deze tekst zijn opgenomen maar die bij gelegenheid zeker van pas komen, zijn de volgende.

- `fill(x,y)`: ingekleurd gebied begrensd door de grafiek waarbij het eerste en het laatste punt ook worden verbonden;
- `quiver(x,y,u,v)`: vectorveld, bestaat uit een pijl (u_i, v_i) in elk punt (x_i, y_i);
- `scatter(x,y)`: spreidingsdiagram, enkel de punten (x_i, y_i) worden getekend, niet onderling verbonden.

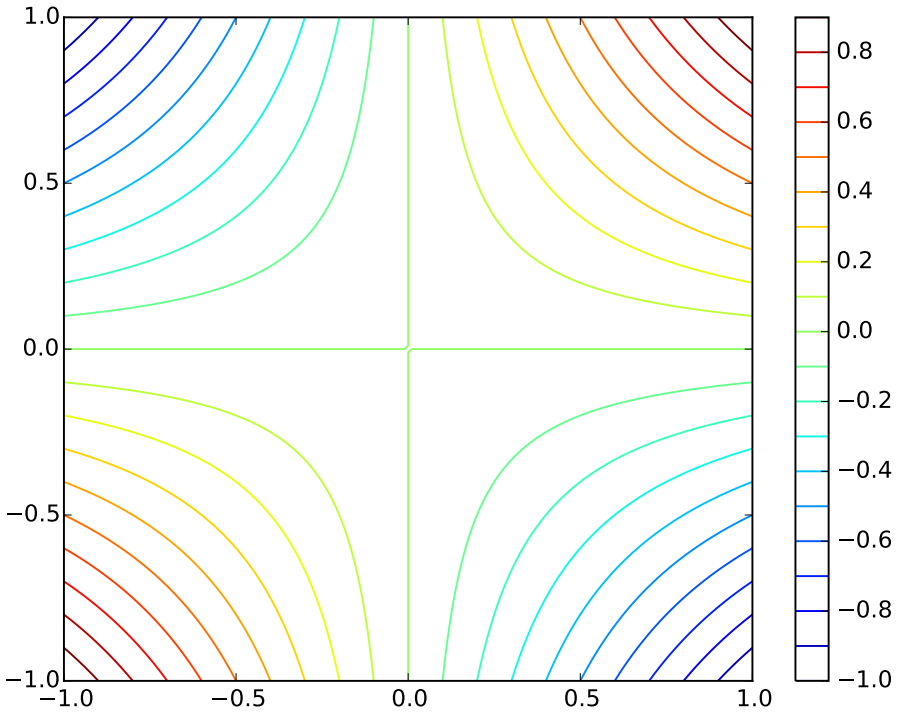
Contourlijnen. Manueel waarden berekenen voor een driedimensionale plot heeft wat meer voeten in de aarde. Er moeten immers functiewaarden $f(x, y)$ worden berekend bij elk koppel (x, y). Twee lijsten van bijvoorbeeld telkens honderd (of algemeen, n) x- en y-waarden leiden met het commando `meshgrid()` tot tienduizend (n^2) koppels. Contourlijnen (`contour()`) zijn impliciete krommen $f(x, y) = c$ voor verschillende waarden van c, deze tweedimensionale grafiek toont een hoogtekaart van het driedimensionale oppervlak bepaald door de cartesische vergelijking $z = f(x, y)$. Een voorbeeld is afgebeeld in Figuur 2.10.

```
xx = np.linspace(-1,1,100)
yy = np.linspace(-1,1,100)
X, Y = np.meshgrid(xx, yy)
Z = X*Y
plt.figure()
CP = plt.contour(X,Y,Z,levels=np.arange(-1,1,0.1))
plt.colorbar(CP)
```

Cartesische oppervlakken in `matplotlib`. Om driedimensionale afbeeldingen te maken is naast de module `matplotlib` ook de submodule `Axes3D` vereist. Aan het grafiekgebied moet eerst en vooral een assenstelsel met drie dimensies worden toegevoegd, pas dan komen grafieken aan bod. De code is heel wat technischer dan bij `sympy` maar de mogelijkheden zijn ook veel uitgebreider. Dezelfde grafiek $\big(x, y, f(x, y)\big)$ kan als ingekleurd oppervlak (`plot_surface`), doorzichtig rooster (`plot_wireframe`) of set van hoogtelijnen (`contour()` of `contourf()`) worden weergegeven. Dit laatste commando laat ook toe om de contourlijnen op een coördinaatvlak te projecteren (opties `zdir` en `offset`), zoals te zien in Figuur 2.11.



Figuur 2.9: De exponentiële functie op lineaire en logaritmische schalen

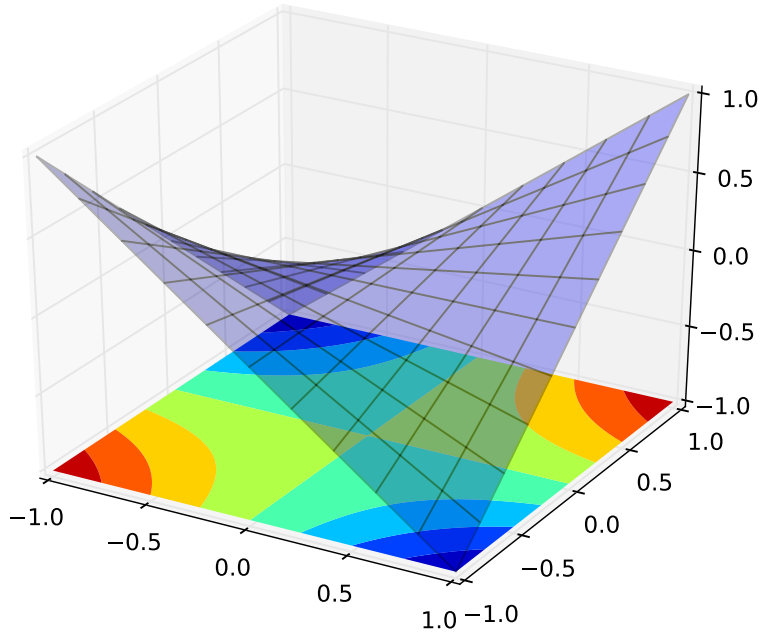


Figuur 2.10: Contourlijnen van de functie $f : (x, y) \mapsto x \cdot y$

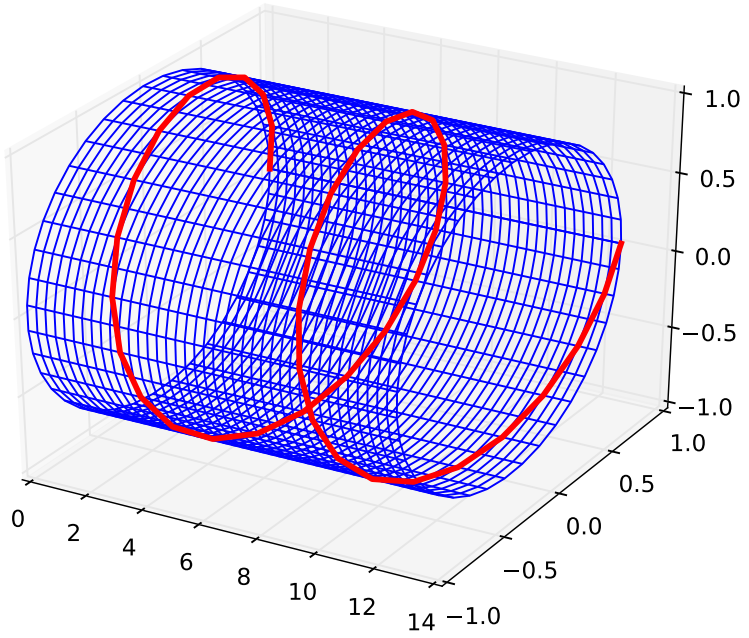
```
412 from mpl_toolkits.mplot3d import Axes3D
413 fig = plt.figure()
414 ax = fig.add_subplot(1,1,1, projection='3d')
415 ax.plot_surface(X,Y,Z, alpha=0.3)
416 ax.contourf(X, Y, Z, zdir='z', offset=-1)
```

Andere 3D-grafieken in `matplotlib`. Hieronder een voorbeeld van een parameterkromme en -oppervlak in drie dimensies met de resulterende grafiek in Figuur 2.12. Door een coördinaat toe te voegen zijn ook de commando's `quiver()` en `scatter()` net als `plot()` uitbreidbaar naar drie dimensies.

```
417 tt = np.linspace(0,2*np.pi,40)
418 ss = np.linspace(0,4*np.pi,40)
419 T, S = np.meshgrid(tt,ss)
420 fig = plt.figure()
421 ax = fig.gca(projection='3d')
422 ax.plot_wireframe(S,np.cos(T),np.sin(T))
423 ax.plot(ss,np.cos(ss),np.sin(ss),color='red',linewidth=3)
```



Figuur 2.11: De grafiek van de functie $f : (x, y) \mapsto x \cdot y$ met contourlijnen.



Figuur 2.12: Parameteroppervlak met parameterkromme in drie dimensies.

Hoofdstuk 3

Vectoren en matrices

3.1 Lijsten

Lijsten vormen een datatype dat uitgebreid aan bod komt in de context van programmeren, maar ze komen ook bij technisch wetenschappelijk rekenen zo vaak voor dat de eigenschappen hieronder snel worden overlopen.

Elementen selecteren. Lijsten worden gebruikt om verschillende waarden te bundelen. Een lijst van lengte `n` is een opeenvolging van `n` elementen. Elk element correspondeert met een index: het eerste element heeft index `0` en het laatste index `n-1`. Bij negatieve indices, selecteert `Python` elementen van achter naar voor, het laatste element heeft dan als index `-1`, het eerste `-n`. Met `a:b` kunnen elementen met index `a` tot (maar *niet* tot en met) `b` worden geselecteerd. Zonder begin- (of eind-)punt wordt de lijst van het begin tot aan index `b` geselecteerd (of vanaf index `b` tot het einde). Met het plusteken kunnen twee lijsten worden samengevoegd. Dit verklaart de markante keuze die `Python` maakt om de laatste index niet in een selectie op te nemen: de code `a[:i] + a[i:]` vormt de originele lijst `a`.

```
a = [2,3,5,7,11,13]
type(a)
list
a[0]
2
a[1]
3
# a[6]
# IndexError: list index out of range
a[-1]
13
a[-6]
2
a[0:2]
[2, 3]
a[-3:-1]
[7, 11]
a[2:]
[5, 7, 11, 13]
a[:2]
[2, 3]
a + [17]
[2, 3, 5, 7, 11, 13, 17]
a == a[:3] + a[3:]
True
```

Bijzondere lijsten. Het commando `range(n)` genereert de indices `0`, `1`, ..., `n-1`. Met twee (of drie) argumenten kunnen ook het startpunt (en de stapgrootte) worden gekozen. De output van dit commando samen met een `for-` (en eventueel `if-`) constructie kan worden gebruikt om gestructureerde lijsten te maken: hieronder eerst de eerste tien kwadraten en daarna enkel de even kwadraten. Een lege lijst zal in veel toepassingen nuttig blijken.

```
449 range(10)[3]
450 3
451 range(3,10)[3]
452 6
453 range(3,10,2)[3]
454 9
455 [x**2 for x in range(10)]
456 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
457 [x**2 for x in range(10) if x%2==0]
458 [0, 4, 16, 36, 64]
459 type([])
460 list
```

Lijsten manipuleren. Met het commando `del()` kunnen elementen uit een lijst worden geschrapt. De indices schuiven in de resulterende lijst uiteraard op. Een element kan makkelijk overschreven worden. Door de lijst te splitsen kan een element worden tussengevoegd.

```
461 b = [17,18,19,20,29]
462 del(b[1]); b
463 [17, 19, 20, 29]
464 del b[1]; b
465 [17, 20, 29]
466 b[1] = 19; b
467 [17, 19, 29]
468 b = b + [31]; b
469 [17, 19, 29, 31]
470 b = b[:2] + [23] + b[2:]; b
471 [17, 19, 23, 29, 31]
```

Commando's met lijsten. Veelvoorkomende operaties met lijsten zijn de lengte, de som van de elementen (indien dit getallen zijn), het minimum en het maximum bepalen. De operator `in` resulteert in een boolean `True` of `False` naargelang het element in de lijst voorkomt of niet.

```
472 len(a)
473 6
474 sum(a)
475 41
476 max(a)
477 13
478 min(a)
479 2
480 3 in a
481 True
482 3 in b
483 False
```

Geneste lijsten. De elementen in een lijst hoeven geen getallen te zijn, het type van elk element kan anders zijn. In het bijzonder kunnen lijsten zelf element zijn van een andere lijst.

```
lijst = [1,a,['a','b','c'],'hallo']
lijst[1]
[2, 3, 5, 7, 11, 13]
lijst[2][2]
'c'
```

Gekoppelde lijsten. Een belangrijk verschil tussen lijsten en niet samengestelde data-types is dat bij toekenning van een lijst aan een nieuwe naam, enkel de referentie naar de lijst wordt gemaakt, de afzonderlijke waarden blijven identiek. Om twee afzonderlijke lijsten te maken volstaat de work-around met `list()` hieronder. Het resultaat van een functie op een lijst is doorgaans een kopie (zie `sorted(b)`). Een methode werkt meestal in op het object zelf (zie `b.sort()`).

```
B = b; B[1] = 2; b
[17, 2, 23, 29, 31]
A = list(a); A[1] = 2; a
[2, 3, 5, 7, 11, 13]
sorted(b)
[2, 17, 23, 29, 31]
b
[17, 2, 23, 29, 31]
b.sort()
b
[2, 17, 23, 29, 31]
```

3.2 Symbolisch matrixrekenen

Het datatype `Matrix`. De lijsten uit de vorige sectie gedragen zich duidelijk niet als vectoren, denk aan de optelling van twee lijsten. Het commando `Matrix()` zet lijsten (of lijsten van lijsten) om in `sympy`-objecten die zich gedragen als wiskundige vectoren (of matrices): de som gebeurt elementsgewijs, maalteken en machtsverheffing voeren matrixvermenigvuldigingen uit, bewerkingen die in de wiskunde niet zijn gedefinieerd geven foutmeldingen.

```
import sympy as sp
v1 = sp.Matrix([1,2,3]); v1
Matrix([
[1],
[2],
[3]])
v2 = sp.Matrix([4,5,6])
v3 =  sp.Matrix([7,8,9,0])
A = sp.Matrix([[1,2,3],[4,5,6],[7,8,9]]); A
Matrix([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
type(v1)
sympy.matrices.dense.MutableDenseMatrix
v1 + v2
```

```
Matrix([
[5],
[7],
[9]])
# v1 + v3
# sympy.matrices.matrices.ShapeError: Matrix size mismatch.
2 * v1
Matrix([
[2],
[4],
[6]])
A*v1
Matrix([
[14],
[32],
[50]])
A**2
Matrix([
[ 30,  36,  42],
[ 66,  81,  96],
[102, 126, 150]])
# 1 + v1
# TypeError: cannot add matrix and <class 'int'>
# sp.sin(A)
# AttributeError: ImmutableMatrix has no attribute could_extract_minus_sign.
```

Elementen uit matrices selecteren. De methode `.shape` geeft de dimensies van een matrix. Elementen in matrices worden geïdentificeerd met een rij- en kolomindex. Een dubbelpunt in plaats van een index, selecteert de ganse rij of kolom. Verder gelden dezelfde afspraken als bij lijsten. Een vector is in wezen een $1 \times n$- of een $n \times 1$-matrix maar één index volstaat.

```
A.shape
(3, 3)
A[2,2]
9
A[1,:]
Matrix([[4, 5, 6]])
A[:,2]
Matrix([
[3],
[6],
[9]])
A[1:,:2]
Matrix([
[4, 5],
[7, 8]])
v1.shape
(3, 1)
v1[0,0]
1
v1[0]
1
```

Symbolische matrixberekeningen. De meest courante matrixeigenschappen zijn geïmplementeerd als methoden van het matrixobject: getransponeerde, spoor, determinant, rang, inverse en eigenstructuur. De methode `.eigenvects()` berekent een lijst van drietallen met telkens de eigenwaarde, multipliciteit en bijhorende eigenvector(en).

```
562 B = sp.Matrix([[1,2],[3,5]]); B
```

```
563 Matrix([
564 [1, 2],
565 [3, 5]])
```

```
566 B.transpose()
```

```
567 Matrix([
568 [1, 3],
569 [2, 5]])
```

```
570 B.trace()
```

```
571 6
```

```
572 B.det()
```

```
573 -1
```

```
574 B.rank()
```

```
575 2
```

```
576 B.inv()
```

```
577 Matrix([
578 [-5,  2],
579 [ 3, -1]])
```

```
580 B*B.inv()
```

```
581 Matrix([
582 [1, 0],
583 [0, 1]])
```

```
584 Beigen = B.eigenvects()
585 Beigen[0]
```

```
586 (3 + sqrt(10), 1, [Matrix([
587   [-2/(-sqrt(10) - 2)],
588   [                 1]])])
```

```
589 [Beigen[k][0] for k in range(2)]
```

```
590 [3 + sqrt(10), -sqrt(10) + 3]
```

3.3 Numeriek matrixrekenen

De symbolische berekeningen uit de vorige sectie laten in het beste geval toe om exacte oplossingen te berekenen of met onbekenden te werken. Doorgaans is symbolisch rekenen erg traag en soms is het niet echt nodig of mogelijk om een exacte oplossing te vinden. In veel situaties zijn numerieke methoden sneller en volstaat een benadering. De module `numpy` voorziet een eigen datatype `ndarray` voor numerieke matrixberekeningen en kent analoge methoden als hierboven, zij het met een heel eigen syntax. Net zoals in de eerdere hoofdstukken bepaalt het doel van een berekening welke module is aangewezen.

Het datatype `ndarray`. Het commando `array()` zet lijsten (of lijsten van lijsten) om in `numpy`-objecten die zich gedragen als wiskundige vectoren (of matrices). Desgewenst kan het datatype van de elementen worden gespecificeerd (`bool`, `int`, `real`, `complex`). Let wel op dat deze objecten de wiskundige rekenregels niet strikt volgen: veel bewerkingen die wiskundig zinloos zijn, gebeuren toch, namelijk elementsgewijs.

```
import numpy as np
v1 = np.array([1,2,3]); v1
array([1, 2, 3])
v2 = np.array([4,5,6])
A = np.array([[1,2,3],[4,5,6],[7,8,9]]); A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
z = np.array([1,2,3,4],dtype=complex); z
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
type(v1)
numpy.ndarray
v1 + v2
array([5, 7, 9])
# v1 + z
# ValueError: operands could not be broadcast together
2 * v1
array([2, 4, 6])
1 + v1
array([2, 3, 4])
np.sin(v1)
array([0.84147098, 0.90929743, 0.14112001])
```

Arrays maken. Veel toepassingen starten met vectoren of matrices met een duidelijke structuur. Om die te maken zijn er specifieke commando's of kan eerst een gepaste lijst (van lijsten) worden gemaakt (zie hoger) die daarna wordt omgezet.

```
np.arange(0.0, 1.0, 0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
np.linspace(0.0, 1.0, 10)
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
np.ones(10)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
np.eye(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
np.array([x**2 for x in range(10)])
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
np.array([[i+j for i in range(3)] for j in range(3)])
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Rekenen met matrices en vectoren. De operatoren `+`, `*`, `/` en `**` voeren in `numpy` allemaal puntsgewijze bewerkingen uit en dus resulteert enkel de som in de wiskundige matrixbewerking: in deze module is er het commando `dot()` om de matrixvermenigvuldiging uit te voeren. Ook in dit pakket (of in de submodule `linalg`) zijn de meest courante matrixeigenschappen geïmplementeerd. Matrixbewerkingen zijn vaak rekenintensief, waardoor afrondingsfouten zich opstapelen: de onoplettende gebruiker zou hieronder kunnen denken dat de matrix `A` inverteerbaar is.

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]]); A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
B = np.array([[(-j)**i for i in range(1,4)] for j in range(1,4)]); B
array([[ -1,   1,  -1],
       [ -2,   4,  -8],
       [ -3,   9, -27]])
A+B
array([[  0,   3,   2],
       [  2,   9,  -2],
       [  4,  17, -18]])
A*B
array([[  -1,    2,   -3],
       [  -8,   20,  -48],
       [ -21,   72, -243]])
np.dot(A,B)
array([[ -14,   36,  -98],
       [ -32,   78, -206],
       [ -50,  120, -314]])
B.shape
(3, 3)
np.transpose(B)
array([[ -1,  -2,  -3],
       [  1,   4,   9],
       [ -1,  -8, -27]])
np.trace(B)
-24
np.linalg.det(B)
12.0
np.linalg.matrix_rank(B)
3
Binv = np.linalg.inv(B); Binv
array([[-3.        ,  1.5       , -0.33333333],
       [-2.5       ,  2.        , -0.5       ],
       [-0.5       ,  0.5       , -0.16666667]])
np.dot(B,Binv)
array([[1.00000000e+00, 0.00000000e+00, 5.55111512e-17],
       [4.44089210e-16, 1.00000000e+00, 0.00000000e+00],
       [2.77555756e-16, 0.00000000e+00, 1.00000000e+00]])
np.linalg.eig(B)
(array([-24.55034695,  -0.4761666 ,   1.02651355]),
 array([[ 0.02931426,  0.84634969, -0.3221482 ],
        [ 0.27160207,  0.52615158, -0.91085105],
        [ 0.96196309,  0.08280534, -0.25801336]]))
np.poly(B)
array([  1.,  24., -14., -12.])
np.linalg.det(A)
-9.51619735392994e-16
np.linalg.inv(A)
array([[ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15],
       [-6.30503948e+15,  1.26100790e+16, -6.30503948e+15],
       [ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15]])
np.linalg.matrix_rank(A)
2
```

Hoofdstuk 4

Programmeren

4.1 Programmeerstructuren

Bij het werken met matrices duiken al snel programmeerstructuren op: lussen en voorwaardelijke opdrachten. Deze worden meer in detail behandeld in het handboek `Python`, hieronder wat voorbeelden ter opfrissing van de syntax.

`if`-statements. Een voorwaardelijke opdracht wordt enkel uitgevoerd als aan de voorwaarde is voldaan. Met `else` kunnen alternatieve instructies worden opgegeven voor het geval niet aan de voorwaarde is voldaan. Om achtereenvolgens verschillende voorwaarden na te gaan is er `elif`.

```
temperatuur = 20
if temperatuur > 100:
   toestand = "stoom"
elif temperatuur > 0:
   toestand = "vloeibaar"
else:
   toestand = "ijs"
print("Bij",temperatuur,"graden","is water",toestand)
Bij 20 graden is water vloeibaar
```

`for`-lussen. Met een `for`-lus kan een lijst met elementen $L_i, i = 1, \ldots, n$, worden opgebouwd. De constructie `[f(n) for n in range(n)]` is enkel mogelijk indien er een gesloten vorm $L_i = f(i)$ bekend is. Is er enkel een recursieve formule $L_n = f(L_{n-1})$ voorhanden, dan dringt de algemenere syntax `for k in ...` zich op. Welk van beide methodes het snelst is, hangt af van de omstandigheden, maar als de berekeningen identiek zijn, is de eerste vorm het efficiëntst. Een veelgemaakte fout is het vergeten initialiseren van objecten die in een lijst worden gebruikt: let op het voorbeeld hieronder op de initialisatie `lijst=[0]`. Door twee `for`-lussen in elkaar te nesten, kunnen lijsten van lijsten en dus matrices worden gegenereerd.

```
[n*(n+1)*sp.Rational(1,2) for n in range(10)]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
lijst = [0]
for n in range(1,10):
   lijst = lijst + [lijst[-1]+n]
lijst
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
import sympy as sp
sp.Matrix([[(-1)**(i+j) for i in range(3)] for j in range(3)])
Matrix([
[ 1, -1,  1],
[-1,  1, -1],
[ 1, -1,  1]])
```

while-lussen. Waar bij for-lussen het aantal stappen in principe op voorhand bekend is, is dat bij while-lussen typisch niet zo. De lus loopt zolang aan een criterium is voldaan. Elke for-lus kan zodoende door een while-lus worden voorgesteld, maar niet omgekeerd. Met een for-lus kan bijvoorbeeld van de eerste tien natuurlijke getallen worden getest of ze priem zijn: het is op voorhand duidelijk dat de priemgetaltest precies tien keer zal moeten worden uitgevoerd. Om de eerste tien priemgetallen te berekenen, moet met een while-lus worden gewerkt: het aantal getallen waarvan primaliteit zal moeten worden getest, is op voorhand immers niet gekend. Vaak worden voorwaardelijke opdrachten gebruikt in combinatie met lusstructuren. Let op de insprongen: if, elif en else op hetzelfde niveau, de instructies verder geïntendeerd.

```
from sympy.ntheory import isprime
priemgetallen = []
for n in range(10):
   if isprime(n):
      priemgetallen = priemgetallen + [n]
priemgetallen
[2, 3, 5, 7]
n = 0
priemgetallen = []
while len(priemgetallen)<10:
   n = n+1
   if isprime(n):
      priemgetallen = priemgetallen + [n]
priemgetallen
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Functies. Eerder werd al de def-constructie geïntroduceerd voor het maken van wiskundige functies. De mogelijkheden van def zijn echter veel uitgebreider. Er kunnen een willekeurig aantal commando's, lussen of voorwaardelijke opdrachten worden uitgevoerd voor het return-commando. Meer details over het schrijven van Python-functies worden gegeven in de context van het vak Programmeren.

```
def priem(N):
   aantal = 0
   n = 0
   while aantal<N:
      n = n+1
      if isprime(n):
         aantal = aantal + 1
   return n
priem(1000)
7919
```

4.2 Programmeertips

Interactief werken. In deze lessenreeks wordt tot nu toe steeds interactief gewerkt. Dat is de meest efficiënte manier om dingen uit te proberen, syntax te testen, snelle berekeningen te maken. Blijf de terminal gebruiken om te experimenteren: test stukken code eerst stap voor stap op de prompt uit vooraleer ze in een groter programma te integreren.

Een bruikbaar programma wordt echter nooit aangeboden als interactieve code: dit vergt te veel inzicht en aandacht van de gebruiker. Deze sectie belicht hoe de technieken uit de vorige hoofdstukken worden samengebracht tot een goed computerprogramma.

Variabelen. Hou volgende regels in acht om de leesbaarheid van je programma's te verhogen:

- Gebruik duidelijke variabelenamen. Namen als `rijIndex` en `kolomIndex` zeggen meer dan de symbolen `i` en `j`. Dit lijkt meer typwerk, maar de programmeeromgeving helpt doorgaans door deze namen automatisch aan te vullen.
- Werk niet met onbenoemde getallen (`9.81`), maar introduceer steeds een constante (`VALVERSNELLING=9.81`). Zo zijn waarden makkelijk op te sporen en aan te passen als de code verbeterd of veralgemeend moet worden.
- Volg de hoofdletterconventies: constante waarden in hoofdletters, woorden gescheiden door underscores (`DIT_IS_EEN_CONSTANTE`) en variabelen in kleine letters, woorden startend met een hoofdletter vanaf het tweede woord (`ditIsEenVariabele`).

Subroutines. Vermijd dubbele code: giet een structuur die op verschillende plaatsen voorkomt in een aparte routine. Dit is veel makkelijker te onderhouden en het verhoogt de leesbaarheid doordat enkele regels code telkens vervangen worden door een betekenisvolle functie-naam.

Zelfs als een stuk code maar één keer voorkomt, kan het lonen om deze op te splitsen. Zorg dat elke subroutine een duidelijk deel van de taken op zich nemen maar nog voldoende compact is om makkelijk te worden begrepen. De hoofdroutine is dan typisch een opeenvolging van opnieuw betekenisvolle functie-namen die de werking van het programma beschrijven.

Leesbaarheid. De naamgeving van constanten, veranderlijken en functies is bedoeld om het geheel zo leesbaar mogelijk te maken. Ook de insprongen die `Python` in functies, lussen en voorwaardelijke opdrachten oplegt, dienen dat doel. Trek dit door in de rest van de code. Gebruik voldoende spatiëring en lijneindes: ellenlange regels lijken misschien compacte code op te leveren, maar zijn zeer moeilijk te begrijpen of aan te passen.

Commentaar. Voorzie voldoende commentaar in de code. Er hoort hoe dan ook uitleg aan het begin van een functie, met een korte toelichting over de werking van het programma en een overzicht van de in- en outputvariabelen.

Voeg ook commentaar toe om de werking van specifieke programmaregels toe te lichten. Commentaar schrijven doet nadenken over de werking van de code, kan leiden tot een beter begrip, opsporing van fouten en misschien een efficiëntere oplossing. Code die niet uit te leggen valt, is geen goede code (en waarschijnlijk ook geen juiste).

Debuggen. Wanneer (een stuk van) de code geschreven is, is het tijd om te testen of de code (correct) werkt. Dit is vaak moeilijker of tijdrovender dan het uitdenken en schrijven van de globale structuur van de code.

Gebruik voorbeelden die zo eenvoudig mogelijk zijn om de code te testen, voorbeelden waarmee de correctheid van het antwoord makkelijk kan worden geverifieerd en de routine indien nodig zelfs manueel kan worden nagerekend. Werk eventueel tijdelijk met dummy-code die enkel output van het gewenste formaat genereert zodat andere routines kunnen worden getest.

Zijn er foutmeldingen of onverwachte resultaten, druk dan tijdens het uitvoeren van de routine op strategische momenten de waarde van goed gekozen veranderlijken af, om te zien of deze veranderen zoals gewenst.

Programmastructuur. Een functie hoort te werken als een *black box*: er worden eventueel parameters meegegeven en output teruggegeven, wat er verder in de routine gebeurt en welke namen er in worden gebruikt, spelen geen rol. Gebruik geen globale veranderlijken, maar enkel de parameters van de invoer en variabelen die in de routine zelf zijn aangemaakt. Overschrijf ook nooit de waarde van de invoerparameters, maar schrijf deze desnoods weg onder een andere naam.

De structuur van een `Python`-bestand is steeds als volgt: eerst modules inladen, daarna de definitie van de functie `main()` die op dit punt nog niet wordt uitgevoerd. Vervolgens alle subroutines voorafgegaan door specifieke commentaar bij de werking, in- en uitvoer. Tot slot de aanroep van de functie `main()`, waarmee het programma wordt uitgevoerd.

```
735 ##
736 # Beschrijving van de globale werking van het script
737
738 import module as ...
739
740 def main():
741     <code>
742
743 ## Doel van de functie
744 # @param x betekenis en datatype van het eerste argument
745 # @param y betekenis en datatype van het tweede argument
746 # @return betekenis en datatype van de output
747 def functie(x,y):
748     <code>
749
750 main()
```

KU Leuven, Campus Kulak Kortrijk
Groep Wetenschap & Technologie
Etienne Sabbelaan 53, 8500 Kortrijk
Tel. +32 56 24 60 20
Fax +32 56 24 69 99
stijn.rebry@kuleuven.be