Automatische Differentiatie

Een Snel en Nauwkeurig Alternatief

M.A. van Griethuijsen

Universiteit Utrecht
Departement Wiskunde
Begeleider: dr. T. van Leeuwen
Juni, 2016

1 Introductie

Bij tal van wiskundige toepassingen wordt de afgeleide/gradiënt/Jacobiaan[1] van algoritmen of van functies gebruikt. Een algoritme of functie kan van alles doen, het kan iets simpels zijn als:

$$f(x) = x^2.$$

Maar het kan zijn dat het algoritme waarvan de afgeleiden worden bepaald ingewikkelder in elkaar zit, bijvoorbeeld bij de Euler-forward methode voor het oplossen van eerste orde differentiaal vergelijkingen:

```
functie(f,x0,y0,h,step){
    yi[0]=y0;
    xi[0]=x0;
        for i=1:step
            xi[i]=xi[i-1]+h;
            yi[i]=yi[i-1]+h*f(xi[i-1],yi[i-1]);
        end
    end
}
```

Bij dit voorbeeld is het al een stuk moeilijker om de afgeleiden te bepalen. Toch kan dit handig zijn om de invloed van $x0$, $y0$ en h op het eindresultaat te bepalen. Als dit bekend is kunnen de optimale beginwaarden bepaald worden.

Bovenstaand voorbeeld is een voorbeeld van een gevoeligheidsanalyse, ook bij het doen van simulaties en het doen van voorspellingen(bijvoorbeeld van het weer) wordt vaak gebruik gemaakt van informatie uit afgeleiden. Over het algemeen worden computers gebruikt om deze afgeleiden te bepalen en het is gewenst om dit zo nauwkeurig en zo efficiënt mogelijk te doen. Het zou het beste zijn als de afgeleiden even nauwkeurig te bepalen zijn als de uitkomst van de functie of het algoritme. In het vervolg wordt een algoritme en een functie omschreven met de term functie en in het kort als f waarbij f meerdere inputs(n) heeft en meerdere outputs(m) kan genereren.

Het bepalen van afgeleiden kan op een aantal manieren gedaan worden, zoals numeriek, symbolisch of door middel van Automatische Differentiatie (AD). Deze methoden hebben allemaal zo hun voor- en nadelen. Als de afgeleiden numeriek worden bepaald dan wordt dit meestal gedaan door een benadering van de differentiequotiënt:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \text{ of } f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x-h)}{2h}.$$

Het nadeel van deze methode is dat de afgeleide benaderd wordt, door h dicht bij 0 te kiezen, dit zorgt voor onnauwkeurigheid in het resultaat. Wanneer h te groot wordt gekozen geeft dit een onnauwkeurige inschatting, terwijl wanneer h te klein gekozen wordt afrondingsfouten ontstaan bij het aftrekken van de twee evaluaties. Dit kan opgelost worden door te werken met imaginaire getallen (complex-step [5]):

$$f'(x) = real\left(\lim_{h\to 0} \frac{f(x+ih)}{h}\right),$$

hierdoor kan met erg kleine getallen gewerkt worden zonder dat dit een probleem veroorzaakt. Met deze methode is het alleen mogelijk om per afhankelijke variabelen de afgeleiden te bepalen. Als een functie meerdere variabelen heeft en er wordt gezocht naar de gradiënt dan moet dit per afhankelijke variabele gedaan worden, dit kan erg lang duren. Daarentegen is deze methode wel makkelijk te implementeren.

[1] In het vervolg wordt gesproken over de afgeleide, alles kan echter gegeneraliseerd worden tot gradiënten en Jacobi-Matrixen.

Bij Symbolische Differentiatie wordt gebruik gemaakt van differentieer regels om de expressie van de afgeleiden op te bouwen uit de originele functie. Deze expressie kan vervolgens symbolisch weergegeven en geëvalueerd worden. Echter is er geen differentiatie regel, die vertelt hoe omgegaan moet worden met conditionele expressies als een if-statement, loops of datastructuren, die worden geïntroduceerd. Over het algemeen ontstaat zo een erg grote expressie waar veel termen dubbel in verwerkt zitten. Deze dubbele termen komen bijvoorbeeld door de product regel:

$$\frac{d(u(x)v(x))}{dx} = u(x)\frac{dv(x)}{dx} + \frac{du(x)}{dx}v(x),$$

merk op dat $u(x)$ en $du(x)/dx$ los van elkaar voorkomen in de afgeleiden van het product. Over het algemeen hebben $u(x)$ en $du(x)/dx$ termen gemeenschappelijk, door de product regel komen deze termen dubbel voor, als dit vaak gebeurt zorgt dit voor een exponentiële toename van de evaluatie tijd[9].

Automatische Differentiatie lijkt erg op Symbolische Differentiatie, behalve dat met de numerieke waarde van de afgeleiden gewerkt wordt in plaats van de symbolische expressie. In het volgende hoofdstuk zal in gegaan worden op de twee manieren om AD toe te passen, vervolgens wordt in hoofdstuk 3 gekeken naar twee manieren om AD te implementeren en wordt een voorbeeld gegeven in de programmeertaal Julia. In hoofdstuk 4 worden twee toepassingen van AD getoond, een voor het zoeken naar nulpunten en een voor het zoeken naar een optimale oplossing. Er wordt afgesloten met een korte discussie.

2 Automatische Differentiatie

Er zijn verschillende manieren om Automatische Differentiatie (AD) uit te voeren. Deze manieren verschillen in de wijze waarop de kettingregel doorlopen wordt, van binnen naar buiten (Forward mode) of van buiten naar binnen (Backward mode).

In een computer wordt een $y = f(x)$ in stapjes berekend, eerst wordt de binnenste elementaire operatie berekend vervolgens wordt de uitkomst hiervan ingevuld in de elementaire operatie daarbuiten. Deze elementaire operaties zullen in het vervolg weergegeven worden als w_i. De functie f kan dan geschreven worden als een serie van samengestelde elementaire operaties:

$$f = w_N \circ w_{N-1} \circ w_{N-2} \circ \ldots \circ w_1.$$

Bij AD wordt vervolgens de kettingregel herhaald op deze elementaire operaties toegepast, evenals andere differentiatie regels zoals die voor vermenigvuldigen, optellen en delen. Door herhaalde toepassing van de kettingregel ontstaat een expressie als hieronder:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} = \ldots. \tag{1}$$

2.1 Forward Automatische Differentiatie

Bij Forward AD wordt expressie (1) van binnen naar buiten doorlopen, in de expressie hieronder wordt van rechts naar links gewerkt:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \frac{\partial y}{\partial w_2}\left(\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x}\right) = \frac{\partial y}{\partial w_3}\left(\frac{\partial w_3}{\partial w_2}\left(\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x}\right)\right) = \ldots. \tag{2}$$

Als f bepaald wordt door (n) onafhankelijke variabelen x_k dan kunnen de partiële afgeleiden naar x_k worden opgeslagen in een gradiënt vector($\dot{w}_i$):

$$\dot{w}_i = \left(\frac{\partial w_i}{\partial x_1}, \frac{\partial w_i}{\partial x_2}, \ldots, \frac{\partial w_i}{\partial x_n}\right).$$

Als f meerdere resultaten geeft, ofwel $f : \mathbb{R}^n \to \mathbb{R}^m$, dan kan met deze gradiënt vectoren de Jacobi Matrix worden opgebouwd waarbij deze vectoren de rijen vormen. Als de Jacobi matrix te groot is om in het geheugen te bewaren kan er voor gekozen worden om per onafhankelijke variabele(x_i) de invloed op y te bepalen door voor de gewenste x_i de afgeleiden op 1 te zetten en voor alle andere op 0.

Als voorbeeld kan de volgende functie $f : \mathbb{R}^2 \to \mathbb{R}$ worden bekeken:

$$f(x_1, x_2) = \cos(2x_1) + x_1 x_2^2. \tag{3}$$

Door elke sub-expressie te labelen met w_i is deze functie als volgt uit te drukken.

$$\begin{aligned} \cos(2x_1) + x_1 x_2^2 &= \cos(2w_1) + w_1 w_2^2 \\ &= \cos(w_3) + w_1 w_4 \\ &= w_5 + w_6 \\ &= w_7. \end{aligned}$$

Voor elke onafhankelijke variabele(x_i) kunnen de beginwaarden $\dot{w}_i$ worden gegeven door de eenheidsvector e_i, in het voorbeeld betekent dit dat $\dot{w}_1 = e_1$ en $\dot{w}_2 = e_2$:

$$\dot{w}_1 = \left(\frac{\partial x_1}{\partial x_1}, \frac{\partial x_1}{\partial x_2}\right) = (1, 0), \dot{w}_2 = \left(\frac{\partial x_2}{\partial x_1}, \frac{\partial x_2}{\partial x_2}\right) = (0, 1).$$

Nu kan van elke sub-expressie de afgeleide bepaald worden door gebruik te maken van eerder bepaalde sub-expressies en hun afgeleide. Zo kan stapsgewijs de afgeleide bepaald worden van de originele functie (3).

Waarden	Afgeleiden
$w_1 = x_1$	$\dot{w}_1 = (1, 0)$
$w_2 = x_2$	$\dot{w}_2 = (0, 1)$
$w_3 = 2w_1$	$\dot{w}_3 = 2\dot{w}_1$
$w_4 = w_2^2$	$\dot{w}_4 = 2w_2\dot{w}_2$
$w_5 = \cos(w_3)$	$\dot{w}_5 = -\sin(w_3)\dot{w}_3$
$w_6 = w_1 w_4$	$\dot{w}_6 = w_1\dot{w}_4 + w_4\dot{w}_1$
$w_7 = w_5 + w_6$	$\dot{w}_7 = \dot{w}_5 + \dot{w}_6$

Omdat f een functie van $\mathbb{R}^2$ naar $\mathbb{R}$ is, is het resultaat de gradiënt van f. In een computationele graaf(Figuur 1) is te zien dat de volgorde voor het berekenen van de afgeleide gelijk is aan de normale evaluatie van de functie. Hierdoor is het niet nodig om de relaties tussen sub-expressies op te slaan. Bij Backward AD is de volgorde om de afgeleide te bepalen omgekeerd.

2.2 Backward Automatische Differentiatie

In tegenstelling tot Forward AD wordt bij Backward AD bij de kettingregel gewerkt van buiten naar binnen. Bij Backward AD wordt bij expressie (1) van links naar rechts gewerkt.

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \dots. \tag{4}$$

Bij Backward AD zijn de waarden waarnaar gekeken wordt de adjoints($\bar{w}_i$). Dit zijn de waarden, afgeleid naar de sub-expressie w_i. Wanneer f bijvoorbeeld m afhankelijke variabelen heeft kan dit weer als vector geschreven worden:

$$\bar{w}_i = \left(\frac{\partial y_1}{\partial w_i}, \frac{\partial y_2}{\partial w_i}, \dots, \frac{\partial y_m}{\partial w_i}\right)^T.$$

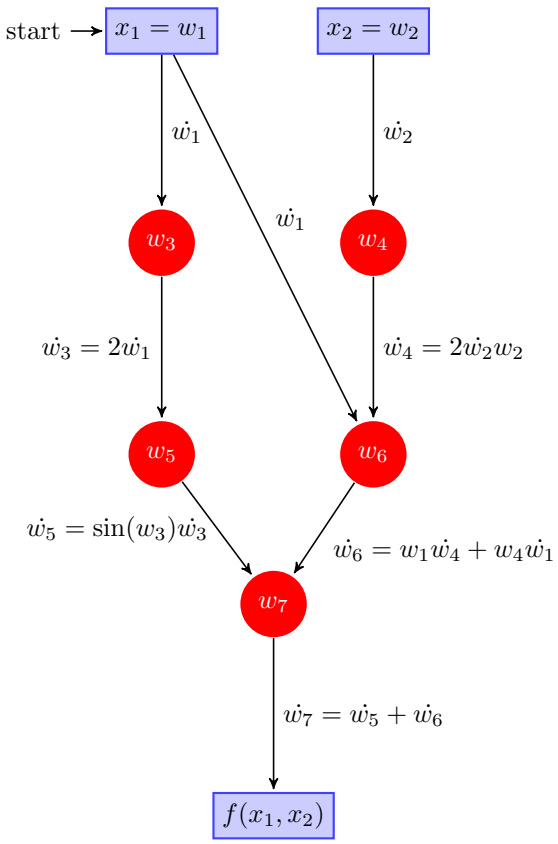

Figure 1: Graaf van Forward AD

De waarden van $\bar{w}_i$ kunnen nu bepaald worden met de volgende formule:

$$\bar{w}_i = \sum_{j \in \pi(i)} \bar{w}_j \frac{\partial w_j}{\partial w_i}, \text{ met } i = N - m, N - (m+1), \ldots, 1, \tag{5}$$

waarbij $\pi(i)$ de verzameling is van indices van de elementaire operaties, die direct gebruik maken van de elementaire operatie w_i. Verder is N de hoogste index van de sub-expressies w_i. In het voorbeeld dat gebruikt is bij Forward AD is dit 7. De beginwaarden kunnen net als bij Forward AD gegeven worden door eenheidsvector e_i ofwel:

$$\bar{w}_{(N+1-i)} = e_i, \text{ met } i = 1, \ldots, m.$$

Wanneer de vectoren van de adjoints als kolommen gebruikt worden in een matrix ontstaat weer de Jacobi Matrix. Net als bij Forward AD kan er voor gekozen worden om voor maar één afhankelijke variabele de afgeleiden te bepalen. Dit gebeurt weer door voor één $\bar{w}_i$ de waarde op 1 te zetten en voor de rest op 0.

Als voorbeeld wordt weer gebruik gemaakt van functie (3). Alleen wordt nu vanaf w_7 begonnen. Het is nodig om de functie normaal te evalueren en daar alle tussenberekeningen van op te slaan. Omdat de functie maar één uitkomst heeft is de startwaarde makkelijk bepaald:

$$\bar{w}_7 = \frac{\partial y}{\partial w_7} = \frac{\partial y}{\partial y} = 1.$$

Daarna kunnen de Adjoints achtereenvolgens bepaald worden

Waarden	Adjoints
$w_7 = w_5 + w_6$	$\bar{w_7} = 1$
$w_6 = w_1 w_4$	$\bar{w_6} = \bar{w_7}\frac{\partial w_7}{\partial w_6} = \bar{w_7}$
$w_5 = \cos(w_3)$	$\bar{w_5} = \bar{w_7}\frac{\partial w_7}{\partial w_5} = \bar{w_7}$
$w_4 = w_2^2$	$\bar{w_4} = \bar{w_6}\frac{\partial w_6}{\partial w_4} = \bar{w_6} w_1$
$w_3 = 2w_1$	$\bar{w_3} = \bar{w_5}\frac{\partial w_5}{\partial w_3} = -\sin(w_3)\bar{w_5}$
$w_2 = x_2$	$\bar{w_2} = \bar{w_4}\frac{\partial w_4}{\partial w_2} = 2\bar{w_4} w_2$
$w_1 = x_1$	$\bar{w_1} = \bar{w_3}\frac{\partial w_3}{\partial w_1} + \bar{w_6}\frac{\partial w_6}{\partial w_1} = 2\bar{w_3} + \bar{w_6} w_4$

Aangezien de waarden van w_i bekend moeten zijn is het eerst nodig om f normaal te berekenen en deze waarden bij te houden. In een computationele graaf is te zien dat de berekening van de afgeleiden omgekeerd is aan die van de normale evaluatie. De normale evaluatie loopt van boven naar onder de berekening van de afgeleiden van onder naar boven. Het nadeel van deze methode is,

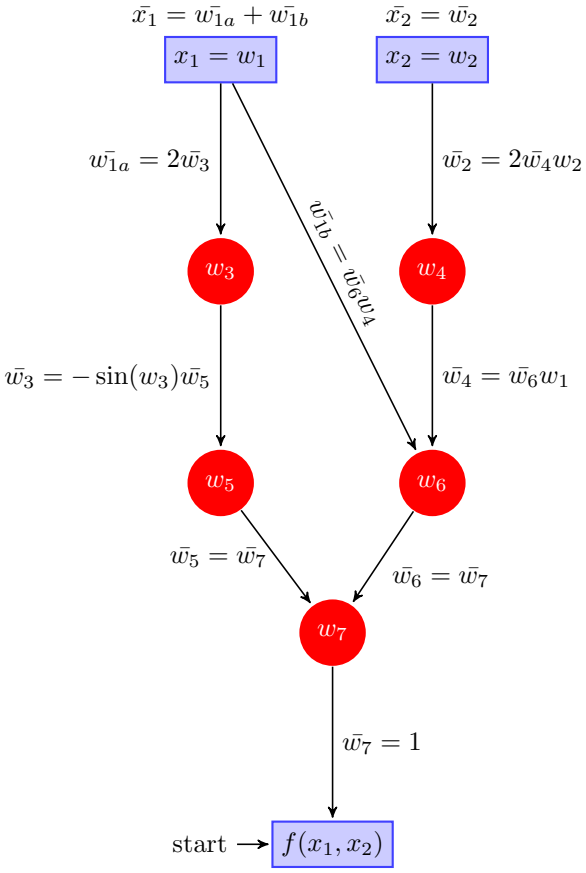

Figure 2: Graaf van Backward AD

dat de waarden van de tussenberekeningen w_i opgeslagen moeten worden. Tevens moet opgeslagen worden van welke waarden elke w_i origineel afhangt. Om de relaties tussen de sub-expressies op te slaan wordt over het algemeen gebruik gemaakt van een Wengert lijst [7, p. 174-177]. Een Wengert lijst kan gezien worden als het pad door een algoritme voor een specifieke input. Hierbij worden alle loops uitgerold, en conditionele statements vervangen door een specifiek pad. Bij het maken van een Wengert lijst worden alle tussenberekeningen opgeslagen in een aparte Wengert variabel, deze worden worden nooit overschreven. Hierdoor correspondeert een functie variabele over het algemeen met meerdere Wengert variabelen. Dit betekent dat de beschrijving van een Wengert

lijst veel groter is dan de beschrijving van het algoritme. Doordat elke tussenberekening wordt opgeslagen, wordt meer geheugen gebruikt. Dit zou eventueel verholpen kunnen worden door de tussenwaarden opnieuw te berekenen. Dit kost echter wel extra tijd maar is voor grote problemen vaak wel nodig.

Het implementeren van Forward AD is makkelijker dan Backward AD, maar als gebruik gemaakt wordt van gradiënten in plaats van scalaire waarden als afgeleiden dan maakt het niet veel uit welke methode gebruikt wordt. Dit kan echter niet altijd, het opslaan van deze gradiënten kan namelijk erg veel geheugen in beslag nemen bij grote problemen. In dat geval is het mogelijk om met een scalaire waarden te rekenen. Dan wordt bij de Forward mode berekend welke invloed één input heeft op alle eindresultaten en bij de Backward AD welke invloed alle eindresultaten hebben op één eindresultaat. Door dit vervolgens voor alle input of output variabelen te doen kan alsnog de volledige Jacobi Matrix worden opgebouwd.

Forward AD en Backward AD zijn twee uiterste manieren om AD uit te voeren. Het is mogelijk om een combinatie van deze twee methoden te gebruiken om de afgeleiden te bepalen. Dit is echter een stuk lastiger te implementeren dan gewoon Forward AD en Backward AD, maar in sommige gevallen is dit wel sneller.

3 Implementatie

De manier om AD te implementeren is door in een duaal programma tegelijk met de berekening van de waarden de afgeleide waarden te berekenen. Dit kan op twee manieren gedaan worden: Source Code Transformation (SCT) en Operator Overloading (OO).

3.1 Source code transformation

Bij Source Code Transformation (SCT) wordt door een programma de broncode van de functie vervangen door een nieuwe broncode, waarbij behalve het resultaat tevens de afgeleiden berekend worden bij de gegeven input. De voorbeeld functie (3) ziet er, opgesplitst in sub-expressies, in code als volgt uit.

```
f(x_1, x_2){
    w_3 = 2x_1;
    w_4 = x_2^2;
    w_5 = cos(w_3);
    w_6 = w_1 w_4;
    w_7 = w_5 + w_6;
    Return w_7;
}
```

Forward AD met SCT zou hier het volgende duale programma van maken:

```
f(x_1, x_2, dx_1, dx_2){
    w_3 = 2 * x_1;
    dw_3 = 2 * dx_1;
    w_4 = x_2^2;
    dw_4 = 2 * x_2 * dx_2;
    w_5 = cos(w_3);
    dw_5 = -sin(w_3) * dw_3;
    w_6 = x_1 * w_4;
    dw_6 = x_1 * dw_4 + w_4 * dx_1;
    w_7 = w_5 + w_6;
    dw_7 = dw_5 + dw_6;
    Return [w_7, dw_7];
```

```
}
```

Het voordeel van SCT is, dat het bij alle programmeertalen kan worden toegepast, en verder kan de verkregen code achteraf geoptimaliseerd worden. Doordat de source code van de afgeleide beschikbaar is kan hier nogmaals SCT op worden toegepast om zo hogere orde afgeleiden te krijgen. Het maken van het programma dat de SCT uitvoert is bijna even lastig als het maken van een eigen compiler[3, 120].

3.2 Operator Overloading

Door gebruik te maken van object georiënteerd programmeren en Operator Overloading (OO) kunnen zonder veranderingen aan het originele programma toch de afgeleiden worden berekend. Hiervoor is het nodig om een functie object te maken van twee getallen, de waarden en de afgeleide waarde. Dit ziet er als volgt uit $U = \{u, u'\}$, waarbij u de waarde is en u' de afgeleide. Vervolgens moeten de operatoren, die gebruikt worden door het programma, dat geanalyseerd wordt, worden gedefinieerd voor dit functie object, zodat op de juiste manier de afgeleide wordt berekend. In de basis zijn dit optellen, aftrekken, vermenigvuldigen en delen. Maar over het algemeen is het nodig om meer standaard functies te definiëren. Als $U = \{u, u'\}$ en $V = \{v, v'\}$ dan moeten de volgende regels worden gedefinieerd voor optellen, aftrekken, vermenigvuldigen en delen:

$$\begin{aligned} U + V &= \{u + v, u' + v'\}, \\ U - V &= \{u - v, u' - v'\}, \\ U * V &= \{u * v, vu' + uv'\}, \\ \frac{U}{V} &= \left\{\frac{u}{v}, \frac{vu' - uv'}{v^2}\right\}. \end{aligned}$$

Evenzo kan dit worden uitgebreid met bijvoorbeeld de volgende standaard functies:

$$\begin{aligned} \sin(U) &= \{\sin(u), u'cos(u)\}, \\ log(U) &= \left\{log(u), \frac{u'}{u}\right\}, \\ U^n &= \{u^n, nu^{n-1}u'\}, \text{als } n \neq 0. \end{aligned}$$

Ook moet een constructor functie gemaakt worden om het functie object te initialiseren en moeten methodes toegevoegd worden om de waarden van het object uit te lezen. Als alle benodigde functies zijn geïmplementeerd, dan kan de afgeleide worden berekend. Als $f(x)$ een scalaire functie is, is het berekenen van een afgeleide mogelijk door met $x \to \{x, 1\}$ te rekenen[2]. Bovenstaande kan uitgebreid worden voor functies met n variabelen door in het afgeleide deel van het functie object de gradiënt te plaatsen in de vorm van een vector met de lengte n. Alle bovenstaande regels voor het bepalen van de afgeleiden blijven gelden. Om de gradiënt van een functie $f : \mathbb{R}^n \to \mathbb{R}^m$ te berekenen hoeft alleen het afgeleide deel van het functie object vervangen te worden door een eenheidsvector. De beginwaarden zijn dan voor alle variabelen als volgt:

$$x_i \to \{x_i, e_i\},$$

waarbij e_i de i-de eenheidsvector is. In het voorbeeld van functie (3) ziet Forward AD met OO er als volgt uit:

$$x_1 = \{x_1, [1, 0]\}, x_2 = \{x_2, [0, 1]\},$$

$$\begin{aligned} f(x_1, x_2) &= \cos(2 * \{x_1, [1, 0]\}) + \{x_1, [1, 0]\} * \{x_2, [0, 1]\}^2 \\ &= \cos(\{2x_1, [2, 0]\}) + \{x_1, [1, 0]\} * \{x_2^2, [0, 2x_2]\} \\ &= \{\cos(2x_1), [-4\sin(2x_1), 0]\} + \{x_1x_2^2, [x_2^2, 2x_1x_2]\} \\ &= \{\cos(2x_1) + x_1x_2, [-4\sin(2x_1) + x_2^2, 2x_1x_2]\}. \end{aligned}$$

De waarde komt overeen met de originele functie (3), en de gradiënt wordt gegeven door $[-4\sin(2x_1)+x_2^2, 2x_1x_2]$. Het is erg makkelijk om OO te implementeren door een nieuw object te maken en de basisfuncties, die de te analyseren code gebruikt, te overschrijven. Jammer genoeg kan dit echter niet in elke programmeertaal, de taal moet het toestaan om functies te overschrijven voor nieuwe objecten, dit is bijvoorbeeld mogelijk bij C++, Matlab en Fortran. Niet elke compiler kan efficiënt met overschreven functies omgaan, zo kan het zijn dat SCT sneller werkt doordat hier optimalisaties toegepast kunnen worden. Zo is het voor $f(x) = \sin(x) + \cos(x)$ niet nodig om de afgeleide van $\sin(x)$ nog eens te berekenen omdat $\cos(x)$ al berekend is uit de normale evaluatie. Soms is het nodig dat de functie waarvan de afgeleide bepaald moet worden iets wordt bewerkt zodat het object als input geaccepteerd wordt. Door een combinatie te gebruiken van deze twee methoden kan zonder een SCT tool toch de broncode van de afgeleide functie verkregen worden. Dit kan gedaan worden door een object te maken dat niet de afgeleide berekent, maar de code om de afgeleide te berekenen in een bestand zet, zodat dit later geëvalueerd kan worden voor verschillende input waarden. Een voorbeeld van deze aanpak voor MATLAB wordt beschreven in [6].

3.3 Julia voorbeeld

Julia[2] is een programmeertaal bedoeld voor scientific computing, net als Matlab en Python. Het heeft als voordeel dat het over het algemeen veel sneller rekent. Hier volgt een beschrijving hoe Forward Operator Overloading geïmplementeerd kan worden in Julia.

Om te beginnen is een Data type nodig om de waarden en afgeleide waarden in op te slaan. In Julia kan dit gedaan worden door een composite type te maken waar twee variabelen in zitten in het voorbeeld wordt dit type diffhelp genoemd.

```
type diffhelp
  value
  diff
end
```

Een nieuw object van het type diffhelp kan nu gemaakt door de naam te gebruiken als functie:

```
x1= diffhelp(1,0)
x2= diffhelp(1,[1,0,0])
```

Doordat de variabelen in het type, value en diff, geen type aanduiding hebben is, kunnen zowel numerieke waarden als vectoren meegegeven worden aan dit object. De waarden van een object kunnen opgevraagd worden met de standaard [object].[variabele-naam] notatie.

```
x1.value
x2.diff
```

Het uitvoeren van deze twee regels geeft de waarden van x1 en de afgeleide van x2. Vervolgens kan een variabele waarde van een object worden aangepast, dit kan door ze gelijk te stellen aan een nieuwe waarde:

```
#waarde x1 wordt 2
x1.value= 2
#afgeleide x2 wordt 5
x2.diff=5
```

Julia stelt de gebruiker in staat om operatoren te overloaden, een operator in Julia is een functie en kan op verschillende manieren gebruikt worden, de plus kan bijvoorbeeld als volgt gebruikt worden:

[2] http://julialang.org/

```
1+1
+(1,1)
+(1,1,1,-1)
```

Van alle drie deze expressies is de uitkomst 2, het is dus mogelijk om meer dan 2 waarden aan een operator mee te geven. Voor de eerste twee waarden wordt eerst een uitkomst berekend en vervolgens wordt de + toegepast op de rest, achtereenvolgens gebeurt:

```
+(1,1,1,-1)
+(2,1,-1)
+(3,-1)
```

Het overloaden van een operator is in Julia erg simpel, dit kan door een nieuwe functie te definiëren met als naam het teken van de operator. Het maken van een nieuwe functie kan op twee manieren, een standaard en een compacte:

```
#standaard
function f(x,y)
  x + y
end

#compact
f(x,y)= x+y
```

Als de standaard notatie gebruikt wordt is het resultaat van de functie wat op de laatste regel van de beschrijving wordt uitgerekend. Dit betekent dat het niet nodig is om return te schrijven, dit is wel nodig als een functie eerder gestopt wordt bijvoorbeeld bij een loop of if statement.

Omdat de nieuwe functie omschrijving alleen moet werken voor objecten van het diffhelp type moeten de input variabele van de functie gespecificeerd worden, dit kan met de :: notatie.

```
#standaard
function +(x::diffhelp,y::diffhelp)
  diffhelp(x.value+y.value, x.diff+y.diff)
end
```

Het is nu mogelijk om twee diffhelp objecten bij elkaar op te tellen, echter kan het gebeuren dat een constante bij een diffhelp object worden opgeteld. Er moet dus een beschrijving komen hoe constanten bij een diffhelp object worden opgeteld. Dit kan door nog twee regels toe te voegen.

```
#standaard
function +(x::Number,y::diffhelp)
  diffhelp(x+y.value, y.diff)
end

function +(x::diffhelp,y::Number)
  diffhelp(x.value+y, x.diff)
end
```

Als voor alle operatoren en functies, die het algoritme gebruikt de de correcte beschrijving wordt gegeven, is het nog handig om aan te geven hoe een numerieke waarden geconvergeerd moet worden naar een diffhelp waarden. Dit kan door voor de convert functie een versie te schrijven die een numerieke warden omzet in een diffhelp waarden.

```
function convert{T<:Number}(::Type{diffhelp}, x::T)
  diffhelp(x,0)
end
```

Deze functie maakt van een object van het type Number, of een sub-type hiervan, een diffhelp object met afgeleide waarden 0. Nu dit gedaan is kan de afgeleide berekend worden door met het diffhelp object te rekenen. Dit alles kan uitgebreid worden door functionaliteit voor vectoren en matrixen toe te voegen. Een diffhelp object kan bijvoorbeeld in een vector gestopt worden. Om hiermee te kunnen rekenen is het nodig om operatoren te definiëren voor matrix vector berekeningen. Voor elementsgewijze operatoren is dit niet nodig, omdat hiervoor de zelfde functies als voor het scalair voorbeeld worden gebruikt. Voor vermenigvuldigen met een matrix en een functie als de norm is het nog wel nodig om een nieuwe beschrijving te geven.

```
function norm(x::Vector{diffhelp},p=2)
  result=0
  for xi in x
    result+= norm(xi)^p
  end
  return result^(1/p)
end

function *{T<:diffhelp}(M::Matrix,x::Vector{T})
  mult=zerosDiffHelp(size(M,1),size(x,1))
  for i =1:size(M,2)
    for j =1:size(M,1)
     mult[j]=mult[j]+x[i]*M[j,i]
    end
  end
  mult
end
```

Zoals te zien is wordt voor de norm standaard de Euclidische norm gekozen. Het is echter mogelijk om een willekeurige norm toe te passen. Voor de vermenigvuldiging moet per element de waarden berekend worden. In Appendix A is een overzicht te vinden van alle code die nodig is voor een basis OO module in Julia

4 Toepassingen

Bij problemen is het nodig om het nulpunt of minimum van een functie te bepalen. Dit kan in veel gevallen gedaan worden met behulp van de afgeleide of gradiënt. Hieronder worden twee manieren besproken die makkelijk gebruikt kunnen worden als de afgeleide nauwkeurig bekend is.

4.1 Kleinste-kwadratenmethode

Bij de kleinste-kwadratenmethode wordt gezocht naar een $\mathbf{x}$ zodanig dat:

$$A\mathbf{x} = \mathbf{b}.$$

Als $A \in \mathbb{R}^{m\times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ met $m \geq n$, dan is er mogelijk geen $\mathbf{x}$ die dit probleem exact oplost. Over het algemeen wordt gezocht naar de best passende $\mathbf{x}$, ofwel de minimale $\mathbf{x}$. Dit betekent dat gezocht wordt naar een $\mathbf{x}$ zodanig dat de som van de kwadraten van de verschillen minimaal is:

$$\min_{\mathbf{x}} \|A\mathbf{x} - \mathbf{b}\|_2^2,$$

hierbij is $\|.\|_2$ de euclidische norm en door te kwadrateren verdwijnt de wortel en wordt de som van de kwadraten berekend. De fout kan nu geschreven worden als functie van $\mathbf{x}$:

$$\mathbf{r}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}.$$

Het probleem kan nu geschreven worden als:

$$\min_{\mathbf{x}} f(\mathbf{x}), \text{ waar } f(\mathbf{x}) = \|\mathbf{r}(\mathbf{x})\|^2.$$

Het minimum wordt aangenomen als voor elke x_i de partiële afgeleiden gelijk zijn aan 0:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = 0, \quad i = 1, \ldots, n.$$

De oplossing voor dit probleem kan algebraïsch bepaald worden [1, H6.1], dit geeft de oplossing:

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}.$$

Echter is dit soms niet te berekenen, bijvoorbeeld als de matrix A te groot is om expliciet op te slaan of als A wordt gegeven door een functie. In zulke gevallen kan met een iteratief algoritme het minimum gevonden worden. Hiervoor wordt de afgeleide gebruikt en stapsgewijs naar het minimum geconvergeerd. Een manier om dit te doen is met de gradiënt descent:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \bigtriangledown f(\mathbf{x}_i),$$

waarbij α, de stapgrootte, klein genoeg moet zijn[1, H7.4]. De α moet zo gekozen worden dat per stap de fout kleiner wordt, het is ook mogelijk om α per iteratie anders te kiezen voor een nog betere convergentie. In code ziet dit er als volgt uit:

```
function steepdescent(x0,f,alpha,niter)
  xk=x0
  for k=1:niter
    gk=f(xk).diff
    xk=xk-alpha*gk
  end
  return xk
end
```

Hierbij is niter het gekozen aantal iteraties. Het voordeel van deze methode is dat deze breed ingezet kan worden om het minimum van een willekeurige functie te bepalen. In het geval van het kleinste-kwadratenmethode is er geen matrix matrix vermenigvuldiging meer nodig maar alleen een matrix vector vermenigvuldiging deze kost maar $O(n^2)$ tegen $O(n^3)$ echter moet het algoritme wel vaak worden uitgevoerd aangezien het slechts lineair convergeert[2, H7.4].

De kleinste-kwadratenmethode kan bijvoorbeeld gebruikt worden om de best passende lijn door een aantal punten te vinden. Hierbij wordt gezocht naar een polynoom van graad n, waarbij de kwadratische afstand tussen alle punten minimaal is. Voor elk punt (x_i, y_i) kan een lineaire vergelijking van de volgende vorm worden opgesteld:

$$y_i = p_n x_i^n + p_{n-1} x_i^{n-1} + \ldots + p_0.$$

Van alle punten zijn de paren (x_i, y_i) bekend, daardoor kan dit stelsel beschreven worden als matrix vector product:

$$\mathbf{y} = A\mathbf{p},$$

waarbij A, $\mathbf{p}$ en $\mathbf{y}$ de volgende vorm hebben:

$$A = \begin{pmatrix} x_0^0 & x_0 & \cdots & x_0^n \\ x_1^0 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m & \cdots & x_m^n \end{pmatrix}, \mathbf{y} = \begin{pmatrix} y_0 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}, \mathbf{p} = \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix}.$$

Over het algemeen is het aantal punten groter dan de graad van het polynoom, dit betekent dat het polynoom niet alle punten kan raken. Wel kan een polynoom gevonden worden, waarbij de som van de verschillen minimaal is. Dit kan gedaan worden met de gradiënt descent methode. Hieronder staat een voorbeeld van het bepalen wat de best passende lijn is bij de volgende punten:

x	y	x	y
1.1	1.532448493	2.1	8.682626599
1.2	2.082122583	2.2	3.877877689
1.3	2.072737476	2.3	4.131255574
1.4	1.978480808	2.4	3.921487814
1.5	5.417335135	2.5	4.681913627
1.6	2.395115853	2.6	5.158560222
1.7	3.341710394	2.7	4.799509460
1.8	2.911108284	2.8	11.41569220
1.9	3.678764707	2.9	5.276665333
2.0	3.227427748	3.0	5.339524615

Als deze punten geplot worden, is te zien dat er op een paar punten na een redelijke lineaire samenhang is, zie in Figuur 3 de rechter plot. De best passende lijn door deze punten kan met de hierboven beschreven methode worden gevonden. Als deze methode wordt toegepast, wordt de volgende lijn gevonden:$f(x) = -0.9045 + 2.5369x$. De code om deze berekening uit te voeren is

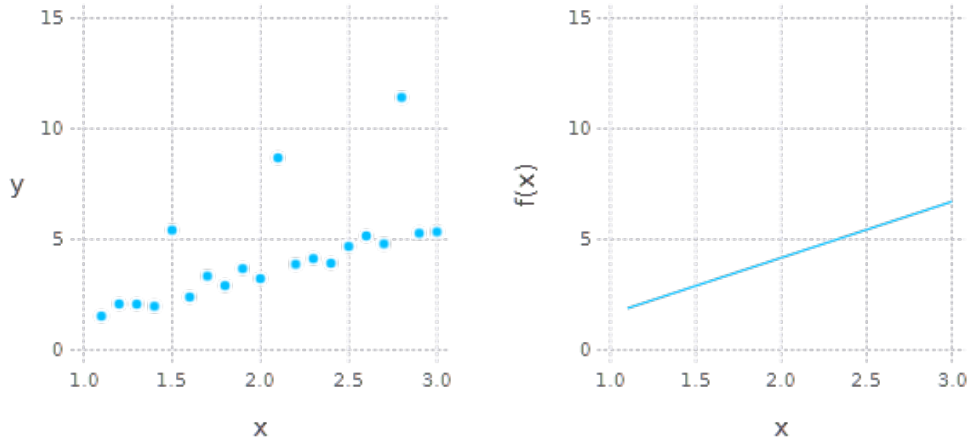

Figure 3: Punten plot en gevonden lijn

terug te vinden in B. Het voordeel van deze aanpak is dat met een willekeurige algoritme gewerkt kan worden, dit hoeft niet een lineaire operatie te zijn zoals een matrix vector product. De functie f kan een willekeurige functie zijn die iets doet met meerdere input waarden. Met deze methode kan dan nog steeds om een minimale oplossing gevonden worden.

4.2 Nul-puntbepaling

Een veel gebruikte methode voor het bepalen van nulpunten van een functie is de Newton methode. Hierbij wordt op een iteratieve manier gezocht naar het nulpunt door gebruik te maken van de afgeleide. Het iteratie proces gaat als volgt voor scalaire functies:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Voor functies van $f : \mathbb{R}^n \to \mathbb{R}^m$ kan dit gegeneraliseerd worden tot:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - J_f(\mathbf{x_i})^{-1} f(\mathbf{x}_i),$$

waarbij $J_f(\mathbf{x_i})$ de Jacobi Matrix van f in het punt x_i is.

Door gebruik te maken van AD is het gemakkelijk om tegelijk met de evaluatie van $f(x)$ de waarden $f'(x)$ te bepalen. Het berekenen van nulpunten kan op deze manier veel exacter berekend worden zonder dat daarvoor eerst zelf de afgeleide bepaald hoeft te worden. In code zou dit als volgt gedaan kunnen worden:

```
function newton(f::Function,x0)
  xi=x0
  for i=1:100
    pointi= f(xi)
    xi=xi-pointi.value/pointi.diff
  end
  return xi
end
```

Als voorbeeld zou bijvoorbeeld gezocht kunnen worden naar het nulpunt van het volgende algoritme:

```
function example(x1)
  b=x1
  for i=1:100
    b=b+0.01*sin(b)
  end
  return b+0.5
end
```

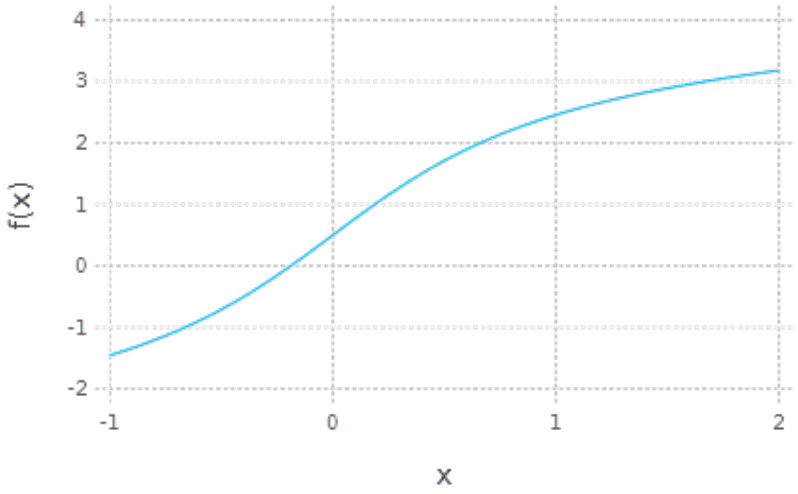

Figure 4: Plot van example

Een plot(Figure 4) laat zien dat het nulpunt rond -0.2 ligt. Door de Newton methode te gebruiken kan dit nulpunt gevonden worden. Als begonnen wordt bij $x_0 = 2$ dan wordt als resultaat $x = -0.18819743031887956$ gevonden, als dit wordt ingevuld in de formule dan is dit inderdaad het nulpunt.

5 Discussie

Hoewel de techniek achter AD vrij simpel is, biedt het diverse mogelijkheden voor het op een nauwkeurige manier berekenen van afgeleiden, zonder dat dit ten koste gaat van veel extra rekentijd. Hierdoor is deze methode uitermate geschikt om te gebruiken in tal van problemen. Maar voor grote problemen zijn er nog wel een aantal obstakels. Zo is het nog onmogelijk om AD te gebruiken op gedistribueerde systemen, waar gewerkt wordt met verschillende software componenten die tegelijk op verschillende platforms draaien[3, 138-139]. Desondanks is het wel mogelijk om AD parallel uit te voeren op een systeem met gedeeld geheugen door middel van bijvoorbeeld OpenMP[4]. Dit biedt mogelijkheden om voor parallelle problemen op een nauwkeurige manier de afgeleide te bepalen. Ook kunnen de principes van AD gebruikt worden om afgeleiden van willekeurige orde te berekenen door het herhaald toepassen van deze methode of door te werken met een matrix van partiële afgeleiden[8]. Automatische differentiatie kan voor veel problemen toegepast worden en biedt over het algemeen een beter alternatief dan Numerieke en Symbolische Differentiatie voor het nauwkeurig bepalen van afgeleiden. Hierdoor zijn problemen op te lossen hetgeen vroeger niet gebeurde omdat het berekenen van de afgeleiden te veel tijd kostte.

A Code voorbeeld AD met OO

```
module diffhelper

import Base.sqrt
import Base.exp
import Base.log
import Base.sin
import Base.cos
import Base.tan
import Base.zero
import Base.zeros
import Base.norm
import Base.isless
import Base.isnan
import Base.convert
import Base.promote_rule
import Base.print

export diffhelp
export getDiff

#the diffhelp object
type diffhelp
  value
  diff
end

# methods for diffhelp
+(x::diffhelp,y::diffhelp)= diffhelp(x.value+y.value,
        x.diff+y.diff)
+(x::Number,y::diffhelp)= diffhelp(x+y.value,y.diff)
+(x::diffhelp,y::Number)= diffhelp(x.value+y,x.diff)


-(x::diffhelp,y::diffhelp)= diffhelp(x.value-y.value,
        x.diff-y.diff)
-(x::Number,y::diffhelp)= diffhelp(x-y.value,y.diff)
-(x::diffhelp,y::Number)= diffhelp(x.value-y,x.diff)
-(x::diffhelp)=(-1)*x

*(x::diffhelp,y::diffhelp)= diffhelp(x.value*y.value,
        x.diff*y.value+y.diff*x.value )
*(x::Number,y::diffhelp)= diffhelp(x*y.value,x*y.diff)
*(x::diffhelp,y::Number)= diffhelp(x.value*y,x.diff*y)

function *{T<:diffhelp}(M::Matrix,x::Vector{T})
  mult=zerosDiffHelp(size(M,1),size(x,1))
  for i=1:size(M,2)
    for j=1:size(M,1)
     mult[j]=mult[j]+x[i]*M[j,i]
    end
```

```
  end
  mult
end


/(x::diffhelp,y::diffhelp)= diffhelp(x.value/y.value,
        (x.diff*y.value-y.diff*x.value)/y.value^2 )
/(x::Number,y::diffhelp)= diffhelp(x/y.value,
        (-y.diff*x)/y.value^2)
/(x::diffhelp,y::Number)= diffhelp(x.value/y,x.diff/y)

sqrt(x::diffhelp)=diffhelp(sqrt(x.value),
        x.diff/(2*sqrt(x.value)))
exp(x::diffhelp)=diffhelp(exp(x.value),
        x.diff*exp(x.value))
log(x::diffhelp)=diffhelp(log(x.value),x.diff/x.value)

^(x::diffhelp,y::diffhelp)=exp(log(x)*y)
^(x::Number,y::diffhelp)=diffhelp(x^y.value,
        x^y.value*log(x)*y.diff)
^(x::diffhelp,y::Number)=diffhelp(x.value^y,
        y*x.value^(y-1)*x.diff)

sin(x::diffhelp)=diffhelp(sin(x.value),
        cos(x.value)*x.diff)
cos(x::diffhelp)=diffhelp(cos(x.value),
        -sin(x.value)*x.diff)
tan(x::diffhelp)=diffhelp(tan(x.value),
        (1+tan(x.value)^2)*x.diff)

isless(a::diffhelp,b::diffhelp)= isless(a.value,b.value)
isless(a::Number,b::diffhelp)= isless(a,b.value)
isless(a::diffhelp, b::Number)= isless(a.value,b)

#checks if a diffhelp object is still a number
isnan(a::diffhelp)= isnan(a.value)
#returns a 0 diffhelp object
zero(x::diffhelp)= diffhelp(0,1)

print(x::diffhelp)= print("value: ",
        x.value," diff: ",x.diff)

#returns a zerro vector with no gradient
function zeros{T<:Int}(::Type{diffhelp}, x::T)
  result=diffhelp[]
  for i=1:x
    push!(result, diffhelp(0, zeros(T,x)))
  end
  result
end

#creates a zerro diffhelp vector with gradient of size mxn
```

```
function zerosDiffHelp(vecSize,gradientWidth)
  result=diffhelp[]
  for i=1:vecSize
    push!(result, diffhelp(0, zeros(Int,gradientWidth)))
  end
  result
end

#converts a number to a diffhelp object
convert{T<:Number}(::Type{diffhelp}, x::T)= diffhelp(x,0)

#converts normal vector to vector with diffhelp elements
function convert{T<:Vector}(::Type{diffhelp},x::T)
  result=diffhelp[]
  count=1
  for xi in x
    directionVec= zero(x)
    directionVec[count]=1
    push!(result,diffhelp(xi, directionVec))
    count = count+1
  end
  return result
end

#describes that a number shoud be converted to a diffhelp object,
#not the other way around
promote_rule{T<:Number}(::Type{diffhelp},::Type{T})= diffhelp

#calculates the norm of a diffhelp object
norm(a::diffhelp)= diffhelp(norm(a.value),
        a.value*a.diff/norm(a.value))

#calculates the p norm of a vector with diffhelp elements
###
function norm(x::Vector{diffhelp},p=2)
  result=0
  for xi in x
    result+= norm(xi)^p
  end
  return result^(1/p)
end
###
function norm(x::Array{diffhelp,2},p=2)
  if size(x,2)==1
    norm(vec(x),p)
  else
    error("size not suported")
  end
end

end
```

B Code voor kleinste kwadraaten dtafitting

```
importall diffhelper
using Gadfly

xas=ones(20)+[1:20]*0.1

A=[ones(20) xas]

y=[1.532448493, 2.082122583, 2.072737476,
   1.978480808, 5.417335135, 2.395115853,
   3.341710394, 2.911108284, 3.678764707,
   3.227427748, 8.682626599, 3.877877689,
   4.131255574, 3.921487814, 4.681913627,
   5.158560222, 4.799509460, 11.41569220,
   5.276665333, 5.339524615]

r(x) = A*x-y
f(x,p=2)= 1/p*norm(r(x))^p

x0= convert(diffhelp,[0;0])

lambda = eigmax(A'*A)
alpha = 1.5/lambda
niter=7500

function steepdescent(x0,f,alpha,niter,p=2)
  xk=x0
  for k=1:niter
    gk=f(xk,p).diff
    xk=xk-alpha*gk
    print(norm(gk,p),'\n')
  end
  return xk
end

xresult=steepdescent(x0,f,alpha,niter)

fresult(x)=xresult[1].value+xresult[2].value*x

p1=plot(x=xas,y=y,Scale.y_continuous(minvalue=0, maxvalue=15),
        Scale.x_continuous(minvalue=1, maxvalue=3))

p2=plot(fresult,1.1,3, Scale.y_continuous(minvalue=0, maxvalue=15),
        Scale.x_continuous(minvalue=1, maxvalue=3))

draw(PNG("linedot.png",16cm,8cm),hstack(p1,p2))
```

References

[1] Uri M. Ascher and Chen Greif. *A First Course in Numerical Methods.* Society for Industrial and Applied Mathematics, 2011.

[2] Michael Bartholomew-Biggs. *Nonlinear Optimization with Engineering Applications*. Springer, 2008.

[3] Andreas Griewank and Andrea Walther. *Evaluating Derivatives, Principles and Techniques of Alforitmic Differentiation*. Society for Industrial and Applied Mathematics, second edition, 2008.

[4] Dieter an Mey H. Martin Bücker, Bruno Lang and Christian H. Bischof. Bringing together automatic differentiation and openmp. *ICS '01 Proceedings of the 15th international conference on Supercomputing*, pages 246–251, June 2001.

[5] Peter Sturdza Joaquim R. R. A. Martins and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262, September 2003.

[6] Matthew Weinstein Michael A. Patterson and Anil V. Rao. An efficient overloaded method for computing derivatives of mathematical functions in matlab. *ACM Transactions on Mathematical Software*, 39(3), April 2013.

[7] Bruce Christianson Michael Bartholomew-Biggs, Steven Brown and Laurence Dixon. Nonlinear equations and optimisation. *Journal of Computational and Aplied Mathematics*, 4(124):171–190, 2001.

[8] Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. *ACM Transactions on Mathematical Software*, 18(2):159–173, June 1992.

[9] Alexey Radul. Introduction to automatic differentiation. `http://alexey.radul.name/ideas/2013/introduction-to-automatic-differentiation/`, August 2013. [Online accessed 1-June-2016].