Universiteit Utrecht

Bachelor scriptie

Efficiënt Algoritmisch Differentiëren

met behulp van gerichte acyclische grafen

10 juni 2017

Auteur:
Esther Middelaar
Studentnummer 4275527
Universiteit Utrecht
e.d.middelaar@students.uu.nl

Begeleider:
Dr. Tristan van Leeuwen
Universiteit Utrecht
Mathematical Sciences
T.vanLeeuwen@uu.nl

Abstract

Dankzij de groei binnen de technologie wordt algoritmische differentiatie een steeds groter onderwerp in de informatica. Het levert een nauwkeurige waarde van een willekeurige afgeleide op, terwijl de numerieke berekening ervan slechts een schatting oplevert. Binnen de algoritmische differentiatie zijn er twee bekende methoden: de *forward* en de *reverse method*. In dit verslag zullen de twee methoden worden besproken en zal er worden onderzocht of er een snellere methode bestaat. Dit wordt gedaan aan de hand van een voorbeeld van het inverse probleem van de één-dimensionale golfvergelijking. Om dit minimaliseringsprobleem op te kunnen lossen, is namelijk de afgeleide nodig. De waarden van deze afgeleide worden bepaald met behulp van een numerieke berekening en algoritmische differentiatie. Onder dat laatste valt de *forward method*, de *reverse method* en een tussenliggende methode, die gebruik maakt van parallelle berekeningen. Vervolgens worden de resultaten van deze methoden met elkaar vergeleken en volgt er een algoritme, die op een willekeurige (niet-)lineaire functie is toe te passen. Met behulp van gerichte acyclische grafen bepaalt dit algoritme welke volgorde van berekeningen resulteert in het snelste programma.

Voorwoord

Voor u ligt mijn bachelorscriptie. In de afgelopen drie jaar van mijn bacheloropleiding wiskunde aan de Universiteit Utrecht heb ik de nodige kennis en vaardigheden opgedaan om dit verslag te kunnen schrijven. In het laatste blok van mijn studie heb ik onderzoek gedaan naar de voortgang van algoritmisch differentiëren. Met deze kennis heb ik zelf een Python programma geschreven die de afgeleide van het inverse probleem van een één-dimensionale golfvergelijking kan bepalen.

Doelgroep Het doel van dit verslag is om de grondslagen van de theorie uit te leggen, zodanig dat er verder onderzoek en ontwikkeling wordt gestimuleerd. In dit verslag draait het voornamelijk om literatuuronderzoek, waarbij er vanuit gegaan wordt dat de lezer niveau 2 van de wiskunde studie als basiskennis bezit.

Dankwoord Dit verslag is in circa twee maanden geschreven. In deze twee maanden heb ik veel geleerd, op wetenschappelijk gebied, maar ook op persoonlijk vlak. Er zijn een aantal mensen zonder wie dit mij nooit was gelukt; ik wil graag even stilstaan bij de mensen die mij de afgelopen periode enorm hebben gesteund en geholpen.

Ik wil graag mijn studiebegeleider, Tristan van Leeuwen, bedanken voor de fijne begeleiding. Je hebt me goed geholpen bij het vinden van het juiste onderwerp. Ook stond je altijd voor me klaar wanneer ik een vraag had; je nam altijd de tijd om alles goed uit te leggen. Dankzij jouw goede tips heb ik mijn scriptie succesvol kunnen afronden.

Verder wil ik mijn vriend en familie ook bedanken voor hun geduld en luisterend oor. Dit verslag ging niet zonder enige frustratie, maar jullie bleven altijd kalm en schoten, wanneer dit kon, te hulp. Ook mijn vrienden stonden altijd voor mij klaar wanneer ik wel wat afleiding kon gebruiken.

Inhoudsopgave

Proloog

Algoritmische differentiatie is erg in opkomst, maar de principes ervan gaan terug naar het begin van de calculus. De implementatie hiervan komt daarentegen pas uit de tijd van de computers. Aangezien differentiatie een bekend en breed begrip is, bevat de literatuur over algoritmische differentiatie veel vaardigheden en herhaling van voorgaande theorie.

Het begon allemaal met het proefschrift van R.E. Moore uit 1962, die zowel interval rekenkunde als algoritmische differentiatie behandelt om de Taylor coëfficiënten van een stelsel van gewone differentiaalvergelijkingen te krijgen. In 1964 merkte R.E. Wengert op dat afgeleiden met behulp van de Taylor coëfficiënten verkregen kunnen worden en kwam met de *forward method*. De *reverse method* kwam vervolgens in het werk van S. Linnainmaa in 1976 en in het proefschrift van B. Speelpenning in 1980 voor. [1], [2, p. 169-170] Het werd uitgebreider gepubliceerd in het werk van M. Iri in 1984. Vervolgens heeft L.B. Rall zich nog lang met algoritmische differentiatie bezig gehouden, waarbinnen hij verschillende ontdekkingen heeft gedaan, waaronder de matrixformulering. Ondertussen hebben M. Iri en A. Griewank de analyse van algoritmen voor algoritmische differentiatie uitgevoerd op basis van grafentheorie. [2, p. 169-170]

In dit verslag zal de basistheorie van algoritmisch differentiëren met behulp van parallel programmeren worden uitgelegd. Hiervoor is eerst voorkennis nodig van gerichte acyclische grafen en topologische ordening. Vervolgens zullen de twee bekende manieren van algoritmisch differentiëren worden benoemd: de *forward* en *reverse method*. Hierna zal er gekeken worden naar het inverse probleem van de één-dimensionale golfvergelijking, waarbij het nodig is de gradiënt te bepalen. Er zal een nieuwe methode ontstaan, waarbij wordt onderzocht of deze sneller de afgeleide kan bepalen dan de andere twee methoden. Uiteindelijk ontstaat er een algoritme, die zorgt voor het snelste programma. Tot slot wordt alles besproken in de conclusie.

Opmerking. Bij de informatie, die uit andere werken is gehaald, zal altijd een bronvermelding staan. De afbeeldingen in dit verslag zijn met behulp van Python verkregen. De code hiervoor staat in de appendix.

HOOFDSTUK 1

Inleiding

Tegenwoordig zijn er computer programma's, die nauwkeurige numerieke waarden van een functie berekenen, gewenst. In dit verslag zal het specifiek om afgeleiden gaan. Deze worden, elementair gezien, onder andere gebruikt voor het oplossen van algebraïsche en differentiaalvergelijkingen. Dit kunnen afgeleiden van de eerste orde, maar ook afgeleiden met een hogere orde zijn. Ze spelen een centale rol in onder andere de gevoeligheidsanalyse en bij neurale netwerken. Verder is bij bijvoorbeeld de optimalisatie van een functie in meerdere dimensies de gradiënt, de Jacobiaan of de Hessiaan vereist. Bij het inverse probleem, dat in sectie 4.1.1 wordt besproken, is de gradiënt tevens erg belangrijk.

Nauwkeurige waarden voor de afgeleiden kunnen verkregen worden met behulp van algoritmisch differentiëren. Merk op dat dit betekent dat er geen numerieke differentiaties zullen plaatsvinden en er dus geen gebruik gemaakt wordt van het *divided difference* algoritme

$$D_{+h}f(x) \equiv \frac{f(x+h)-f(x)}{h} \qquad \text{of} \qquad D_{\pm h}f(x) \equiv \frac{f(x+h)-f(x-h)}{2h}. \quad [3, p.\ 1-12]$$

Dat deze methode geen nauwkeurige waarden geeft, is makkelijk te zien. Als h klein is, dan vermindert de fout het aantal significante cijfers in D_{+h}, maar als h niet klein is, worden truncatiefouten (termen als $h^2 f'''(x)$) significant. Zelfs als h optimaal is gekozen, zullen slechts $\frac{1}{2}$, resp. $\frac{2}{3}$ van de significante cijfers van $D_{+h}f(x)$, resp. $D_{\pm h}f(x)$ nauwkeurig zijn. Bij hogere-orde afgeleiden worden deze afrondfouten problematisch. Algoritmische differentiatie bevat daarentegen geen truncatiefouten en levert gewoonlijk afgeleiden met een grote nauwkeurigheid. [3, p. 1-12] Aangezien een grote nauwkeurigheid vaak veel rekenkracht vergt, is een snelle berekening vereist. Tegenwoordig kan er niet meer op toenemende kloksnelheden gerekend worden, waardoor parallelle programmering een dominante rol gaat spelen.

Een algoritmisch efficiënte manier om een nauwkeurige waarde van een differentiaalvergelijking te bepalen, staat in dit verslag centraal. Stapsgewijs de gradiënt berekenen kan bij grote dimensies of lastige algoritmes erg lang duren. Het is praktisch als er een methode bestaat waarbij zulke berekingen relatief snel gaan. Hiervoor bestaat al een methode, namelijk de zogeheten *reverse method*. Echter, in dit verslag zal met behulp van gerichte acyclische grafen onderzocht worden of er een methode bestaat, die met behulp van parallelle berekeningen sneller de afgeleide kan bepalen. Uiteindelijk ontstaat er een algoritme, die de snelste berekening weergeeft. Dit algoritme zal eenvoudig toe te passen zijn op verchillende (niet-)lineaire vergelijkingen.

HOOFDSTUK 2

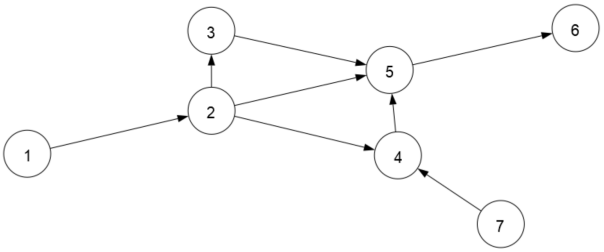
Parallel programmeren

De interesse in parallelle berekeningen kwam eind 1950 opzetten. In de jaren 60 en 70 kwamen de supercomputers in opkomst, deze hadden meerdere processoren, die gezamenlijk aan gedeelde data werkten. Halverwege de jaren 80 kwam er een nieuwe soort van parallele computer gelanceerd toen het Caltech Concurrent Computation project een supercomputer bouwde voor wetenschappelijke toepassingen. Dit systeem toonde aan dat er extreme prestaties bereikt kunnen worden met grote hoeveelheden microprocessoren. Deze massale parallelle processoren stonden bekend om hun sterke rekenkracht. In 1997 overschreed de ASCI Red supercomputer de barriére van een triljoen *floating points operations* per seconde (FLOPS). Sindsdien zijn parallelle processoren in omvang en kracht gegroeid. Echter, sinds eind jaren 80 kwamen er clusters[1], concurrenten van de parallelle processoren, in opmars. Uiteindelijk werden vele parallelle processoren vervangen door deze clusters. Tegenwoordig zijn clusters de werkplek van de informatica. [4]

Een multi-core processor maakt tegenwoordig gebruik van parallelle berekeningen. De meeste desktop- en laptop systemen bezitten nu een dual-core microprocessor en een quad-core microprocessor is eenvoudig te verkrijgen. Chipfabrikanten zijn begonnen met het verhogen van de totale werkingsprestatie door het toevoegen van extra CPU-kernen. De reden hiervoor is dat het verhogen van de prestaties door middel van parallelle berekeningen veel energiezuiniger kan zijn dan het verhogen van de microprocessorklokfrequenties. Aangezien de wereld steeds mobieler en energiebewuster wordt, is dit erg belangrijk. [4] Een goed voorbeeld om te laten zien dat parallel programmeren voor efficiënte berekeningen zorgt wordt hieronder neergelegd. Hiervoor wordt eerst informatie over gerichte acyclische grafen en topologische ordening gegeven.

2.1 Gerichte acyclische grafen

Een graaf G bestaat uit een eindige verzameling knopen V en een verzameling kanten E of pijlen A, waarbij de kanten en pijlen tussen twee knopen lopen. Hierbij is een pijl gericht; er kan alleen volgens de richting van de pijl over de graaf worden gelopen. Een kant is ongericht en kan in beide richtingen worden doorlopen. De notatie van een gerichte graaf is nu $G = (V, A)$ en voor een ongerichte graaf $G = (V, E)$. De notatie van een pijl van knoop v naar w wordt genoteerd als (v, w) en een kant tussen v en w als $\{v, w\}$. Een pad in een gerichte graaf kan worden beschreven door een reeks pijlen, die de eigenschap hebben dat het eindpunt van elke

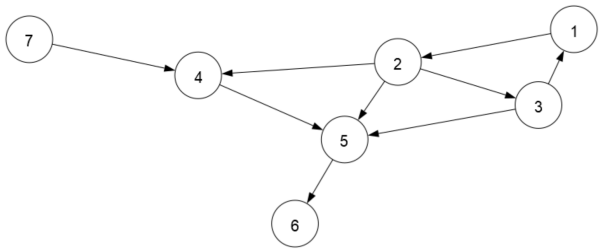


Figuur 2.1: Voorbeeld van een gerichte acyclische graaf.

[1]Clusters zijn parallelle computers, bestaande uit grote aantallen computers, die met elkaar zijn verbonden via een netwerk.

pijl hetzelfde is als het startpunt van de volgende pijl in de reeks. [5], [6], [7] Indien het beginpunt van de eerste pijl in de reeks gelijk is aan het eindpunt van de laatste pijl, is de graaf cyclisch. Met behulp van deze informatie volgt de volgende definitie.[6], [7]

Definitie 2.1.1. Een gerichte acyclische graaf is een graaf $G = (V, A)$ met eindig veel gerichte pijlen, waarbij het onmogelijk is om in een punt v te starten en via een reeks gerichte pijlen terug in v te komen. [6], [7]



Figuur 2.2: Voorbeeld van een gerichte cyclische graaf.

Dit betekent dat een gerichte acyclische graaf een topologische ordening heeft; een reeks van knopen zodanig dat de gerichte pijlen op volgorde liggen, waarbij de eerste pijlen in de reeks als eerste worden belopen. Topologische ordening wordt nog verder uitgelegd in sectie 2.2.

Knopen hebben in een gerichte graaf een ingraad en een uitgraad. De definitie hiervan volgt hieronder.

Definitie 2.1.2. De ingraad van een knoop $k \in V$ is het aantal pijlen $(m, k) \in A$. De uitgraad wordt gedefinieerd door alle pijlen $(k, n) \in A$. [7]

Met behulp van deze informatie volgt de stelling voor een gerichte acyclische graaf.

Theorema 2.1.1. *Een gerichte acyclische graaf bevat een punt met ingraad nul en een punt met uitgraad nul. [7], [8]*

Bewijs. Laat G een gerichte acyclische graaf zijn en neem aan dat de ingraad van elke knoop minimaal één is. Kies een willekeurige knoop v en volg de pijlen terugwaarts. Aangezien de ingraad minimaal één is, wordt een knoop w twee keer bezocht. Laat nu C de reeks knopen in het pad van w naar w zijn. Er geldt nu dat C een cykel is. Dit leidt tot een contradictie, dus een gerichte acyclische graaf bevat een punt met ingraad nul. Het bewijs voor de uitgraad volgt logischerwijs. [7], [8] ■

Een punt k heet bereikbaar vanuit punt l indien er een pad van l naar k bestaat in die graaf. Een gerichte graaf heet samenhangend indien ieder punt vanuit ieder ander punt bereikt kan worden. [5] Oftewel als er voor iedere partitie $V = V_1 \cup V_2$ een pijl loopt van V_1 naar V_2 of omgekeerd.

Opmerking. Een samenhangende ongerichte acyclische graaf wordt ook wel een boom genoemd. [5]

2.1.1 Gegevensverwerking

Een gerichte acyclische graaf kan gebruikt worden om een netwerk van verwerkingselementen te representeren. Hierin gaan de gegevens via de inkomende pijl naar het verwerkingselement, waarna ze het element met de uitgaande pijl verlaten. [9]

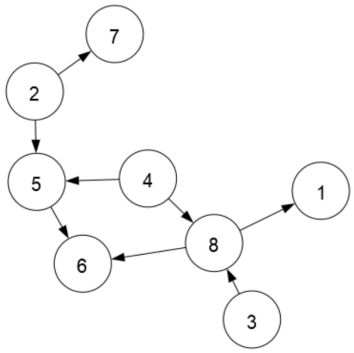
Met behulp van gerichte acyclische grafen kan er dus in kaart worden gebracht waar de onafhankelijkheden binnen een berekening zitten. Hiermee wordt er een beeld gecreëerd van welke berekeningen er gelijktijdig uitgevoerd kunnen worden, wat parallel programmeren mogelijk maakt.

Opmerking. In de computertechnologie worden gerichte acyclische grafen ook wel wait-for-graphs genoemd. Een gerichte acyclische graaf kan worden gebruikt om een deadlock[2] te detecteren, deze graaf laat dan zien dat een hulpbron moet wachten op een ander proces om door te gaan. [10]

[2]Een situatie waarin twee computerprocessen op elkaar wachten, waardoor het systeem geen antwoord meer geeft.

2.2 Topologische ordening

Een topologische ordening van een gerichte graaf is een ordening van de lijnen in een reeks, zodanig dat voor elke lijn het startpunt eerder in de reeks voorkomt dan het eindpunt. In het figuur hiernaast is de topologische ordening bijvoorbeeld 2,3,4,5,7,8,6,1. Merk op dat de graaf in dit figuur acyclisch is. Er geldt de onderstaande stelling.



Figuur 2.3: Voorbeeld van een gerichte graaf.

Theorema 2.2.1. *Een gerichte graaf heeft een topologische ordening dan en slechts dan als de graaf acyclisch is. [6], [7]*

Bewijs. Neem een gerichte graaf G met de topologische ordening $v_1, \ldots, v_n$. Stel dat G een gerichte cykel C bevat en laat v_i de knoop in C zijn met de kleinste index. Laat tevens v_j de knoop voor v_i in C zijn, oftewel (v_j, v_i) is een lijn waarbij $i < j$. Echter, vanwege de topologische ordening volgt dat er moet gelden dat $j < i$. Dit leidt tot een contradictie. Dus een gerichte graaf met een topologische ordening bevat geen cykel en is dus een gerichte acyclische graaf. [6], [7]
Laat nu G een gerichte acyclische graaf zijn met $n + 1$ elementen. Merk op dat bij een gerichte acyclische graaf met één element de topologische ordening gelijk aan de graaf zelf is. Neem nu aan dat een gerichte acyclische graaf met een kardinaliteit $\leq n$ een topologische ordening heeft en laat v de knoop in G met ingraad nul zijn. Construeer de deelgraaf G' met alle knopen van G behalve v. Er geldt dat de verzameling van G' een gerichte acyclische graaf is, aangezien het verwijderen van v in de graaf G niet voor een cykel kan zorgen. Construeer nu een topologische ordening van G door eerst v in de reeks te stoppen en vervolgende de topologische ordening van G' te gebruiken. Dit is toegestaan aangezien de ingraad van v nul is. Oftewel als G een gerichte acyclische graaf is, dan heeft G een topologische ordening. [6], [7] ■

Met behulp van de wiskundige notatie volgt de definitie van topologische ordening.

Definitie 2.2.1. Een topologische ordening van een gerichte acyclische graaf $G = (V, A)$ is een nummering van de knopen met volgnummers $1, \ldots, n$ zodanig dat $(i, j) \in A$ met $i < j$. [7]

Het algoritmische probleem, om een topologische ordening van een gegeven gerichte acyclische graaf $G = (V, A)$ te vinden, kan in lineaire tijd opgelost worden. Het algoritme van Kahn bouwt zo'n topologische ordening direct op.

Definitie 2.2.2. Het algoritme van Kahn handhaaft een lijst van knopen met ingraad nul, die nog niet zijn opgenomen in de gedeeltelijk geconstrueerde topologische ordening T. Deze lijst bestaat in het begin uit de knopen zonder ingevoerde pijlen. Vervolgens voegt het algoritme herhaaldelijk een knoop uit deze lijst toe aan T en controleert of zijn buren aan de lijst moeten worden toegevoegd. Het algoritme eindigt wanneer alle knopen op deze manier zijn verwerkt. De verzameling T is dan de topologische ordening van een gerichte acyclische graaf. Een alternatief voor het opstellen van een topologische ordening is het *depth-first search* algoritme. [7], [8], [11], [12]

Het is ook mogelijk om in lineaire tijd te controleren of een gegeven gerichte graaf acyclisch is. Dit kan door ofwel een topologische ordening te vinden en te controleren of deze klopt, ofwel door te bewijzen dat het algoritme daadwerkelijk alle knopen succesvol indeelt zonder een fout te maken. [11]

Theorema 2.2.2. *Laat G een gerichte acyclische graaf zijn, dan produceert het algoritme van Kahn een topologische ordening van G. [11]*

Bewijs. Stel dat aan het eind alle knopen in de topologische ordening staan, dan hebben al deze knopen een ingraad nul gehad en zijn de knopen dus goed gesorteerd. Stel dat er een verzameling knopen overblijft, waarbij alle knopen een ingraad ≥ 1 hebben. Kies nu een willekeurige knoop v uit deze groep en loop via de pijlen van knoop naar knoop. Ga zo door tot er een knoop twee keer wordt bezocht. Er is nu een cykel gevonden, dus er bestaat geen topologische ordening. [11] ■

Indien in een topologische ordening alle opeenvolgende paren knopen met elkaar verbonden zijn door middel van pijlen, dan vormen deze pijlen een gericht Hamilton-pad in de gerichte acyclische graaf. Als er een Hamilton-pad bestaat, dan is de topologische ordening uniek. Omgekeerd, als een topologische ordening geen Hamilton-pad vormt, dan heeft de gerichte acyclische graaf twee of meer geldige topologische ordeningen. Er is dan altijd mogelijk om een tweede geldige ordening te vormen door twee opeenvolgende knopen, die niet door een pijl zijn verbonden, met elkaar te verwisselen. Merk op dat het hierom mogelijk is om in lineaire tijd te testen of er een unieke sortering bestaat en of er een Hamilton-pad bestaat, ondanks de geldende NP-hardheid van het Hamilton-padprobleem voor (meer algemene) gerichte grafen. [13]

Opmerking. Dit betekent ookwel dat een gerichte acyclische graaf een unieke topologische ordening heeft als de graaf een gericht pad, die alle knopen van de graaf bevat, heeft. In dit geval is de ordening hetzelfde als de orde waarin de knopen in het pad voorkomen. [13]

2.3 Parallelle berekeningen

Met behulp van de informatie uit de voorgaande secties, kan nu het belang van parallelle berekeningen worden besproken. Dit wordt gedaan aan de hand van een voorbeeld.

Stel er is een programma dat een *floating point output* y wilt berekenen aan de hand van een *floating point input* x. Terwijl het programma dit berekent, worden er een aantal tussenwaarden berekend. Sommige tussenwaarden worden opgeslagen of overschreven. Stel nu dat voor $y = f(x_1, x_2)$ wordt weergegeven door

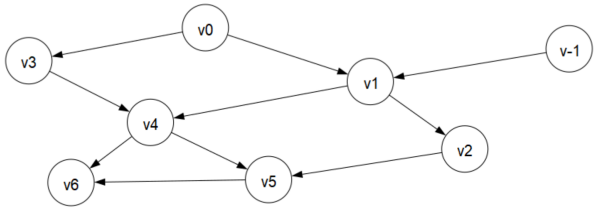
$$y = \left[\sin\left(\frac{x_1}{x_2}\right) + \frac{x_1}{x_2} - e^{x_2}\right] \cdot \left[\frac{x_1}{x_2} - e^{x_2}\right]. \tag{2.1}$$

Indien y met $x_1 = \frac{3}{2}$ en $x_2 = \frac{1}{2}$ berekend wordt, worden de operaties, die hieronder zijn geïllustreerd, door het programma uitgevoerd. [3, p. 1-12]

$v_{-1} = x_1$	$= 1.5000$	
$v_0 = x_2$	$= 0.5000$	
$v_1 = {}^{v_{-1}}/_{V_0}$	$= {}^{1.5000}/_{0.5000}$	$= 3.0000$
$v_2 = \sin(v_1)$	$= \sin(3.0000)$	$= 0.1411$
$v_3 = \exp(v_0)$	$= \exp(0.5000)$	$= 1.6487$
$v_4 = v_1 - v_3$	$= 3.0000 - 1.6487$	$= 1.3513$
$v_5 = v_2 + v_4$	$= 0.1411 + 1.3513$	$= 1.4924$
$v_6 = v_5 * v_4$	$= 1.4924 * 1.3513$	$= 2.0167$
$y = v_6$	$= 2.0167$	

Tabel 2.1: De volgorde waarin de operaties door het programma worden uitgevoerd. [3, p. 1-12]

In dit geval gaat het om een kleine berekening. Echter, indien er meer variabelen bijkomen, wordt het een grote lijst. De berekeningen kunnen dan erg lang gaan duren; parallel programmeren biedt hierin een oplossing. Als er een gerichte acyclische graaf van deze berekeningen wordt uitgeschreven, dan leidt dit tot het figuur hiernaast. Deze graaf bevat geen unieke topologische ordening, wat betekent dat er een aantal onafhankelijke berekeningen bestaan. Uit dit figuur valt op te halen dat in dit geval v_1 en v_3 tegelijk berekend kunnen worden, waarna v_2 en v_4 gezamenlijk te bepalen zijn. Als laatste kunnen v_5 en v_6 berekend worden. Met behulp van deze parallelle berekeningen zal de rekentijd aanzienlijk kleiner worden.



Figuur 2.4: Gerichte acyclische graaf van het tabel hierboven.

HOOFDSTUK 3

Algoritmisch differentiëren

Algoritmische differentiatie, ook wel automatische differentiatie genoemd, is een set van technieken om de afgeleide van een, door een computerprogramma gespecificeerde, functie numeriek te bepalen. Afgeleiden van willekeurige orde kunnen automatisch worden berekend, met een nauwkeurigheid die afhangt van de werkprecisie. Hierbij gebruikt het programma vaak net iets meer rekenkundige operaties dan het oorspronkelijke programma. [14] Er zijn twee bekende manieren voor algoritmisch differentiëren, namelijk de *forward* en de *reverse method*. Deze twee methoden zullen hieronder worden besproken.

Stel vergelijking 2.1 moet algoritmisch gedifferentieerd worden; hierbij moet de uitvoervariabele y naar zowel x_1 als x_2 afgeleid worden. Laat het hier alleen om de afgeleide naar x_1 gaan.

3.1 Forward method

Bij de *forward method* wordt eerst de onafhankelijke variabele, naar welke differentiatie wordt uitgevoerd, vastgelegd, waarna recursief de afgeleiden van de alle subexpressies worden berekend. Laat $f : \mathbb{R}^n \to \mathbb{R}^m$ met $f = f_k \circ \ldots \circ f_2 \circ f_1$ een functie zijn met zijn Jacobiaan $J_x f$ voor een willekeurige x. Hierbij is k het aantal primitieve operaties. Definieer de partiële functie

$$f^i = f_i \circ \ldots \circ f_2 \circ f_1$$

Er volgt dat de volledige Jacobiaan van f zich ontbindt in

$$J_x f = J_{f^{k-1}(x)} f_k \cdot \ldots \cdot J_{f^{i-1}(x)} f_i \cdot \ldots \cdot J_{f^1(x)} f_2 \cdot J_x f_1,$$

waarbij de punten, waarop elke tussenliggence Jacobiaan wordt genomen, worden bepaald door de berekening van f tot daaraan toe. De notatie $J_{f^i-1(x)} f_i$ staat dan voor de Jacobiaan van de functie f_i, die wordt bepaald met het punt $f^{i-1}(x)$. Merk op dat de Jacobiaan een $n \times m$ matrix is, wat in sommige gevallen erg groot kan zijn. Echter, elk stuk Jf_i is relatief klein, omdat elke primitieve f_i alleen op een klein deel van de gegevens opereert. De *forward method* is dus een impliciete weergave van de matrices $J_x f$ in de vorm van een programma dat het product $J_x f \cdot v$ berekend voor een gegeven vector v. [14] Op papier kan het gezien worden als herhaaldelijk de afgeleide van de binnenfunctie in de kettingregel te substitueren, wat eruit komt te zien als:

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v_{-1}} \frac{\partial v_{-1}}{\partial x_1} = \frac{\partial y}{\partial v_{-1}} \left(\frac{\partial v_{-1}}{\partial v_0} \frac{\partial v_0}{\partial x_1} \right) = \frac{\partial y}{\partial v_{-1}} \left(\frac{\partial v_{-1}}{\partial v_0} \left(\frac{\partial v_0}{\partial v_1} \frac{\partial v_1}{\partial x_1} \right) \right) = \ldots \qquad [15]$$

Alle variabelen worden dus naar x_1 gedifferentieerd, oftewel $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$ wordt berekend voor alle i, waarbij $\dot{v}_{-1} = 1$ en $\dot{v}_0 = 0$. Als dit wordt uitgevoerd, geeft dit de volgende rij operaties. [3, p. 1-12]

$v_{-1} = x_1$	$= 1.5000$	
$\dot{v}_{-1} = \dot{x}_1$	$= 1.0000$	
$v_0 = x_2$	$= 0.5000$	
$\dot{v}_0 = \dot{x}_2$	$= 0.0000$	
$v_1 = {}^{v_{-1}}/_{v_0}$	$= {}^{1.5000}/_{0.5000}$	$= 3.0000$
$\dot{v}_1 = {}^{(\dot{v}_{-1} - v_1 * \dot{v}_0)}/_{v_0}$	$= {}^{1.0000}/_{0.5000}$	$= 2.0000$
$v_2 = \sin(v_1)$	$= \sin(3.0000)$	$= 0.1411$
$\dot{v}_2 = \cos(v_1) * \dot{v}_1$	$= -0.9900 * 2.0000$	$= -1.9800$
$v_3 = \exp(v_0)$	$= \exp(0.5000)$	$= 1.6487$
$\dot{v}_3 = v_3 * \dot{v}_0$	$= 1.6487 * 0.0000$	$= 0.0000$
$v_4 = v_1 - v_3$	$= 3.0000 - 1.6487$	$= 1.3513$
$\dot{v}_4 = \dot{v}_1 - \dot{v}_3$	$= 2.0000 - 0.0000$	$= 2.0000$
$v_5 = v_2 + v_4$	$= 0.1411 + 1.3513$	$= 1.4924$
$\dot{v}_5 = \dot{v}_2 + \dot{v}_4$	$= -1.9800 + 2.0000$	$= 0.0200$
$v_6 = v_5 * v_4$	$= 1.4924 * 1.3513$	$= 2.0167$
$\dot{v}_6 = \dot{v}_5 * v_4 + v_5 * \dot{v}_4$	$= 0.0200 * 1.3513 + 1.4924 * 2.0000$	$= 3.0118$
$y = v_6$	$= 2.0167$	
$\dot{y} = \dot{v}_6$	$= 3.0118$	

Tabel 3.1: De volgorde waarin de operaties door het programma worden uitgevoerd. [3, p. 1-12]

Merk op dat een grotere berekening met meerdere uitvoervariabelen nog steeds $\dot{y}_i = \frac{\partial y_i}{\partial x_1}$ geeft. Dit is de basis *forward method* van algoritmische differentiatie. Het wordt "*forward*" genoemd, omdat de afgeleide waarden $\dot{v}_i$ gelijktijdig met de waarden v_i worden uitgevoerd. [3, p. 1-12] Om de gradiënt van dit voorbeeld te bepalen, moet er ook naar x_2 gedifferentieerd worden, met $\dot{v}_{-1} = 0$ en $\dot{v}_0 = 1$.

3.2 Reverse method

Bij de *reverse method*, ookwel *adjoint method* genaamd, wordt eerst de afhankelijke variabele, die wordt gedifferentieerd, vastgelegd. Vervolgens worden de afgeleiden van de alle subexpressies recursief berekend. In plaats van dat het product $Jf_x \cdot v$ wordt berekend, zoals bij de *forward method*, wordt hier $u^T \cdot J_x f$ berekend voor een gegeven vector u, waardoor de berekeningen terugwaarts worden uitgevoerd. [14] Op papier kan het gezien worden als herhaaldelijk de afgeleide van de buitenfunctie in de kettingregel te substitueren, oftewel:

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v_{-1}}\frac{\partial v_{-1}}{\partial x_1} = \left(\frac{\partial y}{\partial v_0}\frac{\partial v_0}{\partial v_{-1}}\right)\frac{\partial v_{-1}}{\partial x_1} = \left(\left(\frac{\partial y}{\partial v_1}\frac{\partial v_1}{\partial v_0}\right)\frac{\partial v_0}{\partial v_{-1}}\right)\frac{\partial v_{-1}}{\partial x_1} = \dots \qquad [15]$$

In plaats van een invoervariabele te kiezen en de gevoeligheid van elke tussenliggende variabele ten opzichte van die invoer te berekenen, wordt er nu een een uitvoervariabele gekozen en de gevoeligheid van die uitvoer met betrekking tot elk van de tussenvariabelen berekend. De gevoeligheid van de uitvoer moet achterwaarts worden berekend; beginnende bij de uitvoervariabelen. Deze methode wordt de "*reverse method*" genoemd, vanwege het feit dat de naam *backward differentiation* al wordt gebruikt voor een andere methode.[3, p. 1-12]

In dit voorbeeld is er maar één uitvariabele, namelijk y. Naast de variabele v_i, wordt nu ook de variabele $\bar{v}_i = \frac{\partial y}{\partial v_i}$, de *adjoint* variabele genaamd, berekend. [3, p. 1-12]

Opmerking. Hiermee wordt bedoeld dat $\bar{v}_i = \frac{\partial y}{\partial \delta_i}$, waarbij δ_i een nieuwe onafhankelijke variabele, toegevoegd aan de rechterkant van de vergelijking, is. Deze nieuwe variabele definieert v_i. Door een kleine numerieke waarde δ_i aan v_i toe te voegen, verandert de berekende numerieke waarde van y met $\bar{v}_i\delta_i$ naar de eerste orde in δ_i. [3, p. 1-12]

Schrijf y nu als functie van v_6, die weer een functie is van v_5 en v_4. Na verder uitwerken volgt dat

$$f(x_1, x_2) = y\Big(v_6\Big(v_5\Big(v_2\big(v_1(v_{-1}, v_0)\big), v_4\big(v_1(v_{-1}, v_0), v_3(v_0)\big)\Big), v_4\big(v_1(v_{-1}, v_0), v_3(v_0)\big)\Big)\Big). \tag{3.1}$$

Merk op dat er geldt dat $\frac{\partial y}{\partial v_6} = \bar{y} = \frac{\partial v_6}{\partial v_6} = \bar{v}_6 = 1$. Gebruikmakend van de notatie $\frac{\partial v_i}{\partial v_j} = \breve{v}_{i,j}$ en met behulp van uitdrukking (3.1) volgt nu het volgende stelsel vergelijkingen.

$$\begin{cases}
\bar{v}_6 &= \breve{v}_{6,6} \\
\bar{v}_5 &= \bar{v}_6\breve{v}_{6,5} \\
\bar{v}_4 &= \bar{v}_6\left(\breve{v}_{6,4} + \breve{v}_{6,5}\breve{v}_{5,4}\right) \\
\bar{v}_3 &= \bar{v}_6\left(\breve{v}_{6,4}\breve{v}_{4,3} + \breve{v}_{6,5}\breve{v}_{5,4}\breve{v}_{4,3}\right) \\
\bar{v}_2 &= \bar{v}_6\breve{v}_{6,5}\breve{v}_{5,2} \\
\bar{v}_1 &= \bar{v}_6\left(\breve{v}_{6,4}\breve{v}_{4,1} + \breve{v}_{6,5}\breve{v}_{5,4}\breve{v}_{4,1} + \breve{v}_{6,5}\breve{v}_{5,2}\breve{v}_{2,1}\right) \\
\bar{v}_0 &= \bar{v}_6\left(\breve{v}_{6,4}\breve{v}_{4,1}\breve{v}_{1,0} + \breve{v}_{6,4}\breve{v}_{4,3}\breve{v}_{3,0} + \breve{v}_{6,5}\breve{v}_{5,4}\breve{v}_{4,1}\breve{v}_{1,0}\right. \\
&\quad \left. + \breve{v}_{6,5}\breve{v}_{5,4}\breve{v}_{4,3}\breve{v}_{3,0} + \breve{v}_{6,5}\breve{v}_{5,2}\breve{v}_{2,1}\breve{v}_{1,0}\right) \\
\bar{v}_{-1} &= \bar{v}_6\left(\breve{v}_{6,4}\breve{v}_{4,1}\breve{v}_{1,-1} + \breve{v}_{6,5}\breve{v}_{5,4}\breve{v}_{4,1}\breve{v}_{1,-1} + \breve{v}_{6,5}\breve{v}_{5,2}\breve{v}_{2,1}\breve{v}_{1,-1}\right)
\end{cases}$$

Het valt gelijk op dat sommige partiële afgeleiden in relatie tot elkaar staan. De waarden van deze partiële afgeleiden kunnen in elkaar worden gesubstitueerd, waarbij $\frac{\partial y}{\partial v_6}$ gegeven is en de waarden van de variabelen v_i uit tabel 2.1 gehaald kunnen worden. Wanneer dit verder wordt uitgewerkt, leidt dit tot het volgende stelsel.

$$\begin{cases}
\bar{v}_6 &= 1 \\
\bar{v}_5 &= v_4 \\
\bar{v}_4 &= v_5 + v_4 \\
\bar{v}_3 &= -\bar{v}_4 \\
\bar{v}_2 &= \bar{v}_5 \\
\bar{v}_1 &= \bar{v}_5\cos(v_1) + \bar{v}_4 \\
\bar{v}_0 &= -{}^{v_1}\!/_{v_0}\cdot\bar{v}_1 + v_0 \\
\bar{v}_{-1} &= {}^{\bar{v}_1}\!/_{v_0}
\end{cases}$$

Een computerprogramma kan dit stapsgewijs oplossen. Wanneer dit wordt omgeschreven, dan zal het programma de waarden voor x_1 en x_2 verkrijgen door stapsgewijs de expressies uit tabel 3.2 te berekenen.

$$\begin{array}{l}
v_{-1} = x_1 = 1.5000 \\
\quad v_0 = x_2 = 0.5000 \\
\quad\quad v_1 = {}^{v_{-1}}\!/_{V_0} = {}^{1.5000}\!/_{0.5000} = 3.0000 \\
\quad\quad\quad v_2 = \sin(v_1) = \sin(3.0000) = 0.1411 \\
\quad\quad\quad\quad v_3 = \exp(v_0) = \exp(0.5000) = 1.6487 \\
\quad\quad\quad\quad\quad v_4 = v_1 - v_3 = 3.0000 - 1.6487 = 1.3513 \\
\quad\quad\quad\quad\quad\quad v_5 = v_2 + v_4 = 0.1411 + 1.3513 = 1.4924 \\
\quad\quad\quad\quad\quad\quad\quad v_6 = v_5 * v_4 = 1.4924 * 1.3513 = 2.0167 \\
\quad\quad\quad\quad\quad\quad\quad\quad y = v_6 = 2.0167 \\
\quad\quad\quad\quad\quad\quad\quad\quad \bar{v}_6 = \bar{y} = 1.0000 \\
\quad\quad\quad\quad\quad\quad\quad \bar{v}_5 = \bar{v}_6 * v_4 = 1.0000 * 1.3513 = 1.3513 \\
\quad\quad\quad\quad\quad\quad\quad \bar{v}_4 = \bar{v}_6 * v_5 = 1.0000 * 1.4924 = 1.4924 \\
\quad\quad\quad\quad\quad\quad \bar{v}_4 = \bar{v}_4 + \bar{v}_5 = 1.4924 + 1.3513 = 2.8437 \\
\quad\quad\quad\quad\quad\quad \bar{v}_2 = \bar{v}_5 = 1.3513 \\
\quad\quad\quad\quad\quad \bar{v}_3 = -\bar{v}_4 = -2.8437 \\
\quad\quad\quad\quad\quad \bar{v}_1 = \bar{v}_4 = 2.8437 \\
\quad\quad\quad\quad \bar{v}_0 = \bar{v}_3 * v_3 = -2.8437 * 1.6487 = -4.6884 \\
\quad\quad\quad \bar{v}_1 = \bar{v}_1 + \bar{v}_2 * \cos(v_1) = 2.8437 + 1.3513 * (-0.9900) = 1.5059 \\
\quad\quad \bar{v}_0 = \bar{v}_0 - \bar{v}_1 * {}^{v_1}\!/_{v_0} = -4.6884 - 1.5059 * \frac{3.0000}{1.5000} = -13.7239 \\
\quad\quad \bar{v}_{-1} = -{}^{\bar{v}_1}\!/_{v_0} = {}^{1.5059}\!/_{0.5000} = 3.0118 \qquad \bar{x}_2 = \bar{v}_0 = -13.7239 \\
\bar{x}_1 = \bar{v}_{-1} = 3.0118
\end{array}$$

Tabel 3.2: De volgorde waarin de operaties door het programma worden uitgevoerd. [3, p. 1-12]

Ook voor een uitvoer y, die afhankelijk is van een miljoen invoervariabelen x_i, valt de *reverse method* te gebruiken om de gradiënt $\nabla_x y$ te verkrijgen. Voor functies met vectoren valt het product van de getransponeerde Jacobiaan met een vector $\bar{y} = (\bar{y}_i)$ te krijgen. Hierbij moet wel gelden dat de grenzen van y_i naar $\bar{y}_i$ worden geïnitialiseerd. [3, p. 1-12]

Opmerking. Deze methode geeft een nauwkeurige waarde voor eerste-orde afgeleiden met één tot vier functie evaluaties als kosten. Aangezien de huidige computers getallen afronden, zal er nog steeds een kleine afrondfout in het eindantwoord zitten. Met behulp van de variabelen $\bar{v}_i$ kan het effect van de afrondfouten van de berekende y worden geschat, aangezien de afrondfout δ_i begrensd is door $\epsilon|v_i|$, waarbij ϵ een onafhankelijke constante is. Er volgt nu dat de berekende waarde van y maximaal $\epsilon \sum_i |\bar{v}_i v_i|$ van de werkelijke waarde van y verschilt. [3, p. 1-12]

Opmerking. De *forward* en *reverse method* zijn slechts twee manieren om de kettingregel om te schrijven. Het probleem om een volledige Jacobiaan van $f : \mathbb{R}^n \to \mathbb{R}^m$ met een minimal aantal rekenkundige operaties te berekenen, staat bekend als het *optimal Jacobian accumulation* probleem, welke NP-compleet is. [15] Er geldt dat de *forward method* efficiënter is dan de *reverse method* als bij de functie $f : \mathbb{R}^n \to \mathbb{R}^m$ geldt dat $m \gg n$. Er zijn dan maar n stappen nodig in de *forward method*, in tegenstelling tot de m stappen die nodig zijn in de *reverse method*.

HOOFDSTUK 4

Toepassing van algoritmische differentiatie

De *forward* en *reverse method* zijn niet de enige twee manieren, die algoritmisch differentiëren mogelijk maken. Sommige manieren gebruiken eigenschappen van beide methoden. In deze sectie zal algoritmische differentiatie worden toegepast op het inverse probleem van de één-dimensionale golfvergelijking. Het zal vervolgens worden vergeleken met het *divided difference* algoritme.

Opmerking. Voor de sectie hieronder is informatie verschaft uit appendix B.

4.1 Eén-dimensionale golfvergelijking

Golven zijn een veelvoorkomend verschijnsel. Zo maken muziekinstrumenten geluid door het produceren van golven en kan een magnetron met behulp van microgolven verwarmen. Ook in de natuur komen golven voor: bijvoorbeeld in de oceaan en bij aardbevingen. Dit fenomeen werd in de achttiende eeuw onderzocht door Jean Le Rond d'Alambert. Hij bedacht in 1746 de één-dimensionale golfvergelijking (4.1), die hij in 1747 publiceerde. Hierdoor raakten Euler en Bernoulli tevens geïnteresseerd in golfvergelijkingen, waarop zij deze uitdrukking uitbreidde. [16, p. 608-611]

De één-dimensionale golfvergelijking wordt geschreven als

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \tag{4.1}$$

waarbij $x \in \mathbb{R}$, $t \in [0, T]$ en $c \in \mathbb{R}$ de golfsnelheid is. De beginvoorwaarden zijn hier $u(0, x) = u_0(x)$ en $\frac{\partial u}{\partial t}(0, x) = 0$ voor alle $x \in [0, l]$. De oplossing hiervan kan, door de differentiaalvergelijking te discretiseren, numeriek berekend worden. Hierbij wordt de uitdrukking voor $\frac{\partial^2 u}{\partial x^2}$ bepaald, waarna dit in vergelijking (4.1) wordt gesubstitueerd. Noem de gediscretiseerde oplossing van (4.1), op positie i en op tijdstap j, $u_{i,j}$. Uit een Taylor expansie van $u(x_i + \Delta x, t_j) = u_{i+1,j}$ en $u(x_i - \Delta x, t_j) = u_{i-1,j}$ volgt het stelsel

$$\begin{cases} u_{i+1,j} &= u_{i,j} + \Delta x \frac{\partial u_{i,j}}{\partial x} + \frac{1}{2}\Delta x^2 \frac{\partial^2 u_{i,j}}{\partial x^2} + \mathcal{O}(\Delta x^3), \\ u_{i-1,j} &= u_{i,j} - \Delta x \frac{\partial u_{i,j}}{\partial x} + \frac{1}{2}\Delta x^2 \frac{\partial^2 u_{i,j}}{\partial x^2} - \mathcal{O}(\Delta x^3). \end{cases}$$

Door de eerste vergelijking van de tweede vergelijking af te trekken en verdere termen te negeren, volgt dat

$$\frac{\partial^2 u_{i,j}}{\partial x^2} \approx \frac{1}{\Delta x^2} \left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right).$$

Door dit in (4.1) te substitueren en het verder uit te werken, volgt dat de één-dimensionale golfvergelijking ookwel wordt benaderd door

$$\ddot{\mathbf{u}}(t) = c^2 L\mathbf{u}(t), \tag{4.2}$$

waarbij $\mathbf{u}(t) = [u(-1+\Delta x, t), u(-1+2\Delta x, t), \ldots, u(1-2\Delta x, t), u(1-\Delta x, t)]^T$ en

$$L = \frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & & \\ 1 & -2 & 1 & \\ & & \ddots & \\ & & 1 & -2 \end{pmatrix} \tag{4.3}$$

de $n_x n_t \times n_x n_t$ differentiatie matrix is, met $n_x n_t = {}^2/_{\Delta x} - 1$ het aantal rasterpunten.

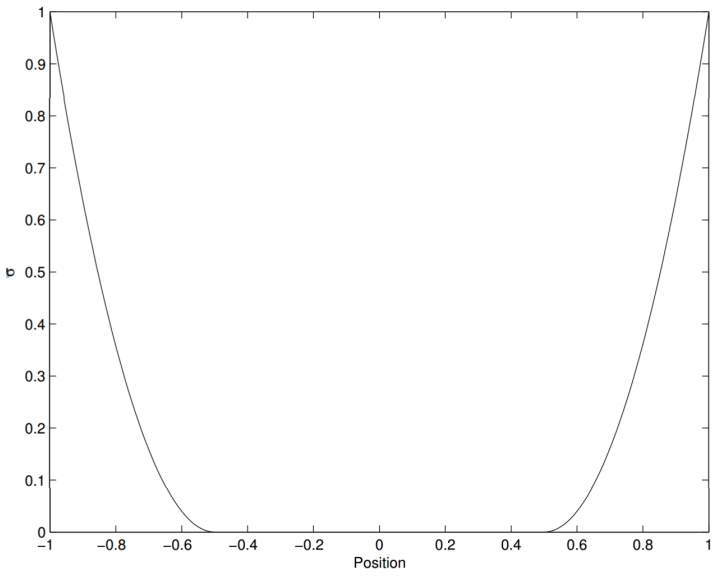
Opmerking. Hierbij wordt bedoeld dat n_x het aantal ruimtestappen is en n_t het aantal tijdstappen.

Deze vergelijking kan nu zodanig worden aangepast dat met een eindig interval een oneindig domein kan worden nagestreeft. Dit wordt bereikt door een tijdelijke eerste-orde afgeleide term toe te voegen, die de oplossing dicht bij de grens afdempt. De vergelijking wordt

$$\ddot{\mathbf{u}}(t) + 2\Sigma\dot{\mathbf{u}}(t) = c^2 L\mathbf{u}(t),$$

waarbij Σ een $n_x n_t \times n_x n_t$ positieve diagonaalmatrix is met elementen σ_i.

Opmerking. Bij σ_i duidt i de positie $-1+\Delta x$ tot $1-\Delta x$ aan. In het midden van de matrix ligt σ_i dichtbij nul en bij de randen van de matrix wordt σ_i steeds groter, zoals te zien is in afbeelding 4.1.



Figuur 4.1: De waarden van σ ten opzichte van de positie. [Appendix B]

Als vervolgens de Taylor expansie van $u(x_i, t_j + \Delta t) = u_{n+1}$ en $u(x_i, t_j - \Delta t) = u_{n-1}$ wordt bepaald, dan volgt:

$$\begin{cases} u_{n+1} &= u_n + \Delta t\frac{\partial u_n}{\partial t} + \frac{1}{2}\Delta t^2\frac{\partial^2 u_n}{\partial t^2} + \mathcal{O}(\Delta t^3), \\ u_{n-1} &= u_n - \Delta t\frac{\partial u_n}{\partial t} + \frac{1}{2}\Delta t^2\frac{\partial^2 u_n}{\partial t^2} - \mathcal{O}(\Delta t^3). \end{cases}$$

Door optelling en aftrekking van de twee vergelijkingen en door het weglaten van de overige termen, volgt een stelsel gewone differentiaalvergelijkingen

$$\begin{cases} \dot{\mathbf{u}}(n\Delta t) & \approx & \frac{1}{2\Delta t}\left(\mathbf{u}_{n+1} - \mathbf{u}_{n-1}\right), \\ \ddot{\mathbf{u}}(n\Delta t) & \approx & \frac{1}{\Delta t^2}\left(\mathbf{u}_{n+1} - 2\mathbf{u}_n + \mathbf{u}_{n-1}\right), \end{cases} \tag{4.4}$$

waarbij $\dot{\mathbf{u}}(n\Delta t) = \frac{\partial u_n}{\partial t}$ en $\ddot{\mathbf{u}}(n\Delta t) = \frac{\partial^2 u_n}{\partial t^2}$. Laat nu $\mathbf{u}_n \equiv \mathbf{u}(n\Delta t)$, dan leidt (4.4) tot

$$\mathbf{u}_{k+1} = 2(I + \Delta t\Sigma)^{-1}\mathbf{u}_k - (I + \Delta t\Sigma)^{-1}(I + \Delta t\Sigma)\mathbf{u}_{k-1} + \Delta t^2 c^2 (I + \Delta t\Sigma)^{-1} L\mathbf{u}_k. \tag{4.5}$$

Hieruit volgt ook dat de golfvergelijking kan worden weergegeven als

$$A(c)\mathbf{u}(t) = c^2 L\mathbf{u}(t), \tag{4.6}$$

waarbij de $n_x n_t \times n_x n_t$ matrix $A(c)$ uit vergelijking (4.5) gehaald kan worden, oftewel

$$A(c) = \begin{pmatrix} I & & & & & \\ -I & I & & & & \\ P & Q & R & & & \\ & & & \ddots & & \\ & & & & P & Q & R \end{pmatrix},$$

met $P = \frac{I - \Delta\Sigma}{\Delta t^2}$, $Q = \frac{-(2I + \Delta t^2 c^2 L)}{\Delta t^2}$ en $R = \frac{I + \Delta\Sigma}{\Delta t^2}$.

4.1.1 Het inverse probleem

Noem $\mathbf{u}_n(c)$ de oplossing van de golfvergelijking voor een waarde c op tijdstip $n \leq N \in \mathbb{N}$, die voldoet aan uitdrukking (4.5). De metingen worden gemodelleerd als een lineaire bemonstering van de oplossing

$$d_n = \mathbf{p}^T \mathbf{u}_n,$$

waarbij $\mathbf{p}$ een $n_t \times n_x n_t$ vector van gewichten is, die bepalen waar de oplossing wordt gemeten. Laat $\phi(c)$ de fout tussen de meting en de oplossing voor een willekeurige waarde van c zijn. Het inverse probleem kan vervolgens worden gesteld als een niet-lineaire kleinste-kwadraten probleem

$$\min_c \phi(c),$$

met

$$\phi(c) = \frac{1}{2}\sum_{n=0}^{N}\left(\mathbf{p}^T\mathbf{u}_n(c) - d_n\right)^2. \tag{4.7}$$

4.1.1.1 De afgeleide van het inverse probleem

De waarde van c, die ϕ minimaliseert, is de oplossing van het inverse probleem. Merk op dat in dit geval de oplossing makkelijk te vinden is door de functie ϕ nauwkeurig te meten. Echter, voor een inverse probleem in meerdere dimensies wordt vaak een iteratieve methode gebruikt. Er geldt dat voor de waarde c, die ϕ minimaliseert, $\phi'(c) = 0$. Het vinden van zo'n stationair punt gaat met behulp van de Newton-Raphson methode,

$$c_{k+1} = c_k - \frac{\phi'(c_k)}{\phi''(c_k)},$$

beginnende met een initiële schatting c_0. Om deze methode te kunnen gebruiken, moeten de eerste en tweede afgeleide van $\phi(c)$ berekend worden. Wanneer het inverse probleem wordt gedifferentieerd, dan geeft dit

$$\phi'(c) = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{p}^T \mathbf{u}_n}{\partial c} \right) r_n = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p} r_n, \tag{4.8}$$

$$\phi''(c) = \sum_{n=0}^{N} \frac{\partial \left(\left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p} r_n \right)}{\partial c} = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p}\mathbf{p}^T \left(\frac{\partial \mathbf{u}_n}{\partial c} \right) + \left(\frac{\partial^2 \mathbf{u}_n}{\partial c^2} \right)^T \mathbf{p} r_n, \tag{4.9}$$

met $r_n = (\mathbf{p}^T \mathbf{u}_n(c) - d_n)$. De eerste afgeleide van $\phi(c)$ wordt in dit hoofdstuk met behulp van het *divided difference* algoritme en algoritmische differentiatie berekend, waarna de twee met elkaar worden vergeleken. Om het wat eenvoudiger te houden, wordt de matrix Σ in dit voorbeeld gezien als een nulmatrix. In beide gevallen moeten de waarden van $\mathbf{u}_n$ en d_n worden bepaald. Hiervoor zijn functies geschreven, die deze waarden kunnen berekenen. De Python code is hieronder weergegeven.

```
import numpy as np
from scipy.sparse import spdiags
import matplotlib.pyplot as plt

def solveA(c,nt,dt,f):
    u = np.copy(f)
    for i in range(1, nt-1):
        u[i+1] = 2 * u[i] - u[i - 1] + dt ** 2 * c ** 2 * L @ u[i]
                 + 2 * dt ** 2 * c * L @ f[i+1]
    return u

#definieer een functie die d uitrekent
def data(c0,nt,dt,p,f):
    # Bepaal de vector u
    u = solveA(c0,nt,dt,f)

    #Bepaal de vector d
    d=np.zeros(nt)
    for i in range(nt):
        d[i] = p@u[i]
    return d
```

Hierbij geldt dat d_n de oplossing van het inverse probleem voor $c = c_0$ is. De functie `solveA(c,nt,dt,f)` is zodanig geschreven, dat het in meerdere situaties bruikbaar is.

4.1.1.1.1 Divided difference

Een manier om de afgeleide numeriek te bepalen, is met behulp van het *divided difference* algoritme van Newton, welke hieronder gegeven is.

Definitie 4.1.1. Gegeven de punten $x_0, x_1, \ldots, x_n$ met willekeurige indices $0 \leq i < j \leq n$, het *divided difference* algoritme wordt gedefinieerd als

$$f[x_i] = f(x_i),$$
$$f[x_i, x_{i+1}, \ldots, x_j] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_j] - f[x_i, x_{i+1}, \ldots, x_{j-1}]}{x_j - x_i}. \qquad [17, p.\ 308 - 2012]$$

Laat nu $f(x)$ een differentieerbare functie zijn en z_0, z_1 in het domein van f liggen. Indien z_1 naar z_0 nadert, dan geldt

$$f[z_0, z_1] = \frac{f(z_1) - f(z_0)}{z_1 - z_0} \longrightarrow f'(z_0). \qquad [17, p.\ 308 - 2012] \tag{4.10}$$

Op deze manier kan de waarde van $\phi'(c)$ worden benadert, waarbij $z_1 - z_0$ zeer klein wordt gekozen. Dit resulteert in

$$\phi'(c) \approx \frac{\phi(c+h) - \phi(c-h)}{2h}.$$

Aangezien deze methode te maken heeft met truncatiefouten, mag h niet te klein gekozen worden. Er is een omslagpunt waarbij de totale fout weer groter wordt als h te klein gekozen wordt. [17, p. 9] In dit verslag is er gekozen voor $h = 1/1.000.000$, wat in dit geval goed gaat. Eerst moeten dus de waarden van d_n en $\mathbf{u}_n$ worden bepaald, waarna r_n wordt bepaald en uiteindelijk, door $\phi(c+h)$ en $\phi(c-h)$ te bepalen, volgt de waarde van $\phi'(c)$. De Python code ziet er als volgt uit:

```
def dphi(c0,c1,nx,nt,dt,pk):
    # Bepaal benodigde gegevens
    o = np.ones(nx)
    elementen = np.array([o, -2 * o, o])
    diags = np.array([-1, 0, 1])
    global L
    L = spdiags(elementen, diags, nx, nx).toarray()

    # Bepaal de vector f, is nodig voor u
    f = np.zeros((nt, nx))
    x = np.linspace(0, 1, nx)
    f[0] = np.exp(x ** 2)
    f[1] = np.exp(x ** 2)

    # bepaal p, d en h,c2,c3
    p = np.zeros(nx)
    p[pk] = 1
    d = data(c0, nt, dt, p, f)
    h=1/1000000
    c2=c1+h
    c3=c1-h

    # Bepaal de vector u en de afgeleide vector du
    u2 = solveA(c2, nt, dt, f)
    u3 = solveA(c3, nt, dt, f)

    # bepaal r
    r2 = np.zeros(nt)
    r3 = np.zeros(nt)
    for i in range(nt):
        r2[i] = np.transpose(p) @ u2[i] - d[i]
        r3[i] = np.transpose(p) @ u3[i] - d[i]

    # Bepaal phi en de afgeleide dphi
    phi2=0
    phi3 = 0
    for i in range(0, nt):
        phi2 += 1 / 2 * r2[i] ** 2
        phi3 += 1 / 2 * r3[i] ** 2
    dphi = (phi2 - phi3) / (2 * h)
    return dphi
```

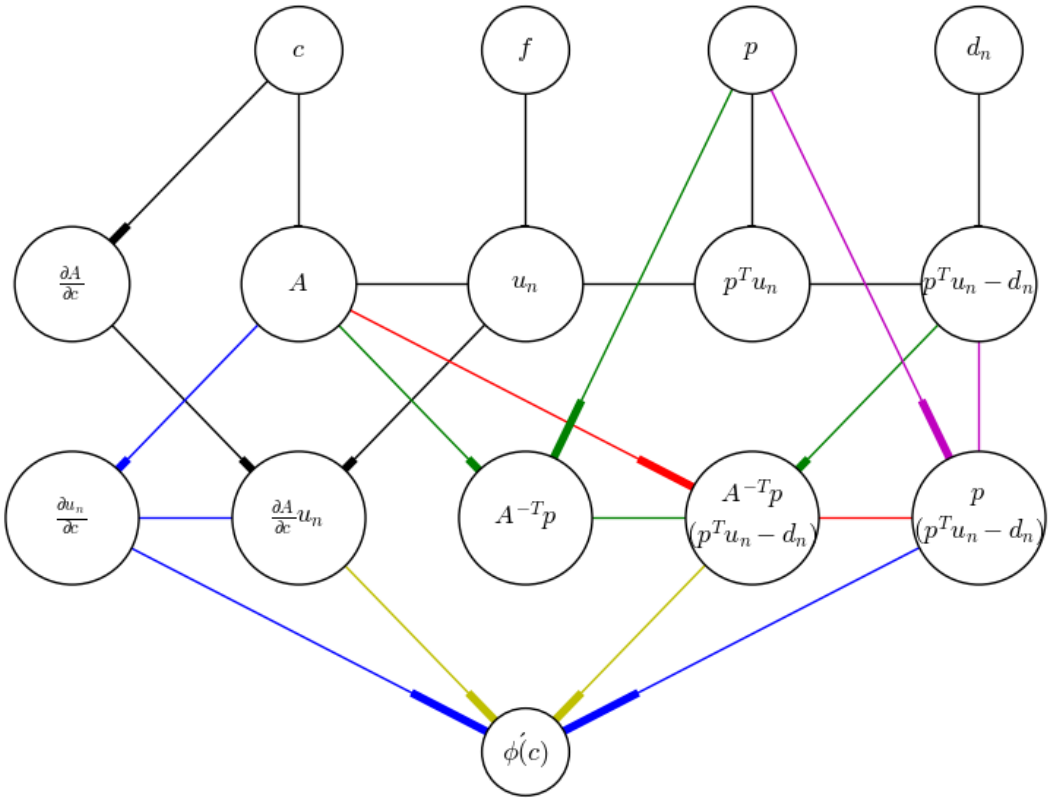
De functie `dphi(c0,c1,nx,nt,dt,pk)` roept `data(c0,nt,dt,p,f)` en `solveA(c,nt,dt,f)` aan, waarbij de benodigde gegevens worden meegestuurd. Hierbij is $c = c_1$ de waarde waarvoor de afgeleide wordt berekend. Er geldt dat $0 \leq$ `pk` $<$ `nt` de plek is waar de oplossing wordt gemeten. Om ervoor te zorgen dat de waarden $\mathbf{u}_n$ goed worden berekend, is de vector `f` aangemaakt, waarbij de eerste twee elementen in `f` door een exponentiële functie worden gedefinieerd.

Opmerking. Indien de lezer geïnteresseert is in numerieke berekeningen, dan is het boek "A First Course in Numerical Methods", geschreven door Uri M. Ascher and Chen Greif, een aanrader. Hier staat veel theorie, waaronder over het *divided difference* algoritme, en worden veel voorbeelden gebruikt.

4.1.1.1.2 Algoritmische differentiatie

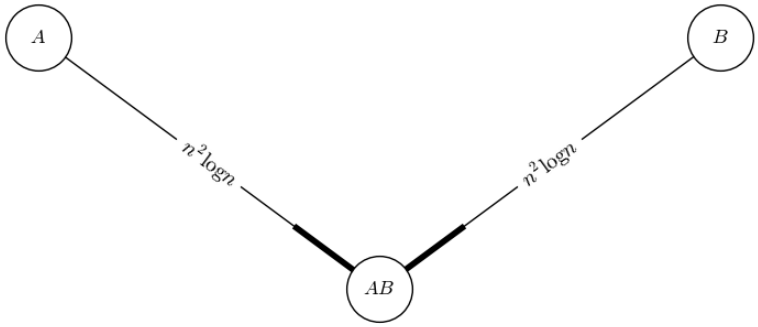
Met behulp van algoritmische differentiatie wordt er een nauwkeurige waarde van $\phi'(c)$ verkregen, omdat hier geen truncatiefouten bij komen kijken. Eerst zal met behulp van de *forward method* de afgeleide worden berekend. Dit kan echter sneller, namelijk met de *reverse method*. Vervolgens wordt onderzocht of het met behulp van parallelle berekeningen nog sneller kan. Als laatste worden de methoden met elkaar vergeleken.

Bij algoritmische differentiatie is het handig om eerst een gerichte acyclische graaf op te stellen van de berekeningen. De graaf van het inverse probleem ziet eruit zoals in figuur 4.2.



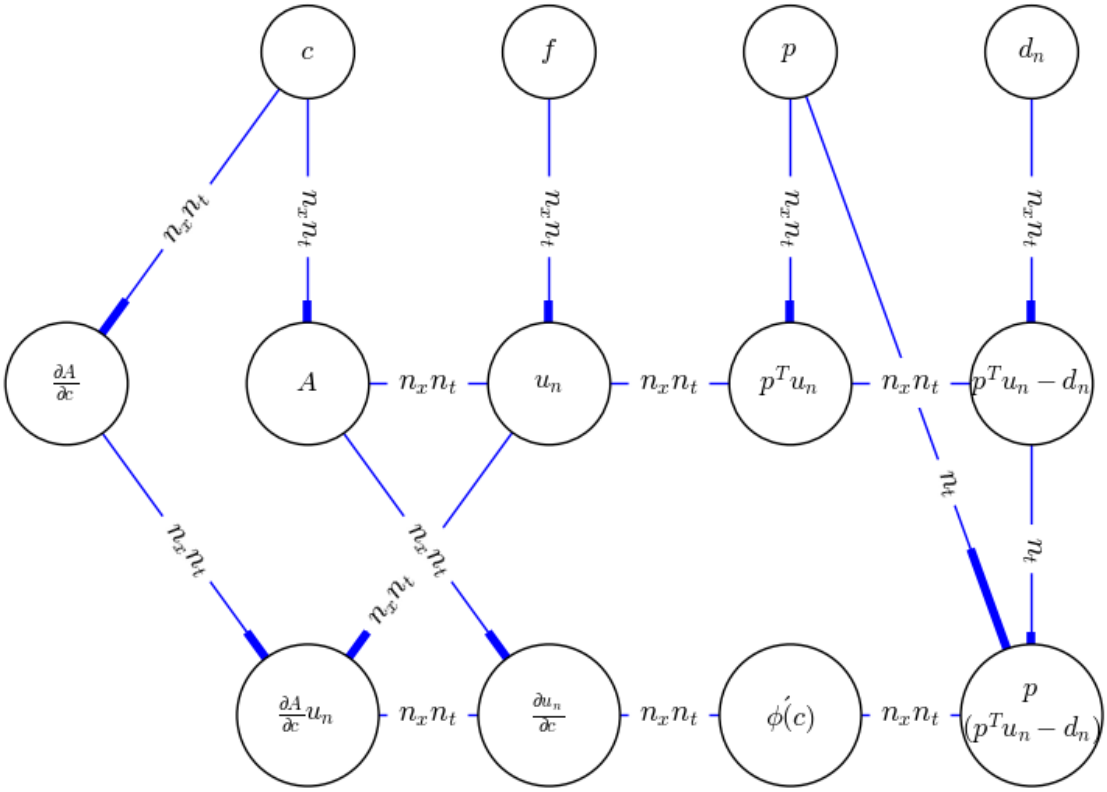
Figuur 4.2: Een gerichte acyclische graaf van de berekening van $\phi'(c)$.

In deze graaf is het mogelijk om via verschillende paden naar $\phi'(c)$ te lopen. Dit resulteert in deelgrafen, die in bij de verschillende methoden te zien zijn. In deze deelgrafen is de complexiteit van elke berekening te zien. Deze complexiteit staat bij de pijl aangegeven, zoals bijvoorbeeld in figuur 4.3 te zien is. In figuur 4.3 zijn A en B een $n \times n$ matrix. Uit de graaf volgt dat de matrixvermenigvuldiging van A met B complexiteit $\mathcal{O}(n^2 \log n)$ heeft.



Figuur 4.3: Een voorbeeld van een gerichte acyclische graaf.

Forward method
Eén van de manieren om naar $\phi'(c)$ te lopen, is via de zwarte, paarse en blauwe paden. Dit resulteert in de deelgraaf 4.4 op de volgende pagina. Deze graaf kan topologisch gesorteerd worden. Merk op dat deze topologische ordening niet uniek is en er dus geschoven kan worden met sommige elementen. Een simpele topologische ordening is van links naar rechts, van boven naar beneden en geeft de ordening $[c,\ \mathbf{f},\ \mathbf{p},\ \mathbf{d},\ \frac{\partial A(c)}{\partial c},\ A(c),\ \mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n - d_n,\ A^{-1}(c)\frac{\partial A(c)}{\partial c},\ \frac{\partial \mathbf{u}_n}{\partial c},\ \mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ \phi'(c)]$. Door het weglaten van de triviale elementen en het schuiven van de onafhankelijke elementen wordt de ordening $[\mathbf{u}_n,\ A^{-1}(c)\frac{\partial A(c)}{\partial c},\ \frac{\partial \mathbf{u}_n}{\partial c},\ \mathbf{p}^T\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n - d_n,\ \mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ \phi'(c)]$ gevonden. Merk op dat alle termen binnen deze topologische ordening afhankelijk van $\mathbf{u}_n$ zijn, waardoor er geen gebruik gemaakt kan worden van parallelle berekeningen.



Figuur 4.4: Een gerichte acyclische graaf van de berekening van $\phi'(c)$ met behulp van de *forward method*.

Deze deelgraaf komt overeen met de *forward method*. Laat de rechte haken [aangeven waar een losstaande berekening begint en] waar deze eindigt. Het pad dat wordt belopen komt overeen met de volgende stappen:

$$\phi'(c) = \sum_{n=0}^{N} \left[\left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \right] \mathbf{p}\,[r_n]\,.$$

Eerst wordt $\mathbf{u}_n$ berekend, vervolgens $\frac{\partial \mathbf{u}_n}{\partial c}^T$, waarna r_n en als laatste $\phi'(c)$. Door de vergelijking (4.5) te differentiëren, volgt dat $\frac{\partial \mathbf{u}_n}{\partial c}$ voldoet aan:

$$\frac{\partial \mathbf{u}_{n+1}}{\partial c} = 2\frac{\partial \mathbf{u}_n}{\partial c} - \frac{\partial \mathbf{u}_{n-1}}{\partial c} + \Delta t^2 c^2 L \frac{\partial \mathbf{u}_n}{\partial c} + 2\Delta t^2 c L \mathbf{u}_n. \tag{4.11}$$

Met behulp van deze uitdrukking kan de afgeleide van het inverse probleem worden berekend. Gelukkig is `solveA(c,nx,nt,dt,f)` zodanig geschreven met deze functie ook $\frac{\partial \mathbf{u}_n}{\partial c}$ kan worden bepaald. Hiervoor hoeven alleen de waarden $\mathbf{u}_n$ worden gekopieerd, behalve de eerste twee, aangezien $\frac{\partial \mathbf{u}_0}{\partial c} = \frac{\partial \mathbf{u}_1}{\partial c} = 0$. De Python code wordt hieronder gegeven.

```
def dphi(c0,c1,nx,nt,dt,pk):
    # Bepaal benodigde gegevens
    o = np.ones(nx)
    elementen = np.array([o, -2 * o, o])
    diags = np.array([-1, 0, 1])
    global L
    L = spdiags(elementen, diags, nx, nx).toarray()

    # Bepaal de vector f, is nodig voor u
    f = np.zeros((nt, nx))
    x = np.linspace(0, 1, nx)
    f[0] = np.exp(x ** 2)
    f[1] = np.exp(x ** 2)

    # bepaal p en d
    p = np.zeros(nx)
    p[pk] = 1
    d = data(c0, nt, dt, p, f)

    # Bepaal de vector u en de afgeleide vector du
    u = solveA(c1, nt, dt, f)
    q = np.zeros((nt, nx))
    for i in range(2, nt):
        q[i] = np.copy(u[i - 1])
    du = solveA(c1, nt, dt, q)

    # bepaal r
    r = np.zeros(nt)
    for i in range(nt):
        r[i] = np.transpose(p) @ u[i] - d[i]

    # Bepaal de afgeleide dphi
    dphi = 0
    for i in range(0, nt):
        dphi += np.transpose(du[i]) @ p * r[i]
    return dphi
```

Opmerking. Uit de graaf volgt dat deze berekening een complexiteit van $\mathcal{O}(n_x n_t)$ heeft. Dit komt doordat $A(c)$ een ijle matrix is, waardoor de berekeningen sneller gaan. Echter, als ϕ van meerdere parameters zou afhangen, dan zou de complexiteit $\mathcal{O}(k \cdot n_x n_t)$ worden, waarbij k het aantal parameters voorstelt. Dit komt doordat op bij deze methode voor elk parameter $\frac{\partial \mathbf{u}_n}{\partial c}$ uitdrukking (4.11) moeten worden bepaald. Het is dus handiger om de berekeningen te herorganiseren, wat leidt tot de *reverse method*.

Reverse method

Met behulp van de Lagrange-multiplicator is de afgeleide van $\phi(c)$ te vinden. De Lagrange functie is gegeven door $\Lambda(\mathbf{u}, c, \mathbf{v}) = \frac{1}{2}(\mathbf{p}^T\mathbf{u} - \mathbf{d})^2 - \mathbf{v}^T(A(c)\mathbf{u} - \mathbf{f})$ waarbij $(A(c)\mathbf{u} - \mathbf{f})$ een golfvergelijking is. Er geldt hier dat $\mathbf{u} = [u_0, u_1, \ldots, u_N]^T$ en $\mathbf{f} = [u(0,x), 0, \ldots, 0]^T$. Bij het minimum van $\phi = \frac{1}{2}(\mathbf{p}^T\mathbf{u} - \mathbf{d})^2$ geldt dat $\nabla_{\mathbf{u},c,\mathbf{v}}\Lambda(\mathbf{u}, c, \mathbf{v}) = [0, \phi'(c), 0]^T$. Wanneer deze gradiënt verder wordt uitgewerkt, geeft dit:

$$\frac{\partial \nabla}{\partial \mathbf{u}} = \mathbf{p}(\mathbf{p}^T\mathbf{u} - \mathbf{d}) - A(c)^T\mathbf{v} = 0, \tag{4.12}$$

$$\frac{\partial \nabla}{\partial c} = -\left(\frac{\partial A(c)\mathbf{u}}{\partial c}\right)^T \mathbf{v} = \phi'(c), \tag{4.13}$$

$$\frac{\partial \nabla}{\partial \mathbf{v}} = -(A(c)\mathbf{u} - \mathbf{f}) = 0. \tag{4.14}$$

Met behulp van deze drie vergelijkingen kan de afgeleide op twee verschillende manieren worden omgeschreven. Uit vergelijking (4.12) volgt dat:

$$\begin{pmatrix} I & -I & P^T & & & & \\ & I & Q^T & & & & \\ & & P^T & & & & \\ & & & \ddots & & & \\ & & & & R^T & Q^T & P^T \\ & & & & & R^T & Q^T \\ & & & & & & R^T \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{N-2} \\ v_{N-1} \\ v_N \end{pmatrix} = \begin{pmatrix} \mathbf{p}r_0 \\ \mathbf{p}r_1 \\ \mathbf{p}r_2 \\ \vdots \\ \mathbf{p}_{N-2} \\ \mathbf{p}_{N-1} \\ \mathbf{p}_N \end{pmatrix}.$$

Als dit wordt uigewerkt, dan leidt dit tot de volgende (terugwaartse) recursieve betrekking:

$$\mathbf{v}_{n-1} = 2\mathbf{v}_n - \mathbf{v}_{n+1} + \Delta t^2 c^2 L^T \mathbf{v}_n + \Delta t^2 \mathbf{p} r_{n-1},$$

met $\mathbf{v}_N = \Delta t^2 \mathbf{p} r_N$ en $\mathbf{v}_{N-1} = 2\mathbf{v}_N + \Delta t^2 c^2 L^T \mathbf{v}_N + \Delta t^2 \mathbf{p} r_{N-1}$. Met deze uitdrukking wordt de berekende $\mathbf{v}_n$ opnieuw gebruikt om alle partiële afgeleiden te berekenen. Met behulp van vergelijking (4.6) en (4.13) volgt dat de afgeleide ook kan worden herleid tot:

$$\begin{aligned} \phi'(c) &= -\left(\frac{\partial A(c)\mathbf{u}}{\partial c}\right)^T \mathbf{v} \\ &= \sum_{n=1}^{N-1} \left(\frac{\partial (c^2 L \mathbf{u}_n)}{\partial c}\right)^T \mathbf{v}_{n+1} \\ &= \sum_{n=1}^{N-1} 2c \mathbf{u}_n^T L^T \mathbf{v}_{n+1} \end{aligned} \tag{4.15}$$

Dit wordt ook wel de *adjoint state approach* genoemd.

Opmerking. Deze methode komt overeen met de *reverse method*. Dit volgt uit het feit dat $A^T\mathbf{v} = \mathbf{p}(\mathbf{p}^T\mathbf{u} - d)$, wordt berekend. Hierdoor worden de waarden van $\mathbf{v}_n$ terugwaarts berekend.

De afgeleide van $\phi(c)$ kan ook met behulp van vergelijking (4.14) worden omgeschreven. Merk op dat geldt

$$\begin{aligned} A(c)\mathbf{u} &= \mathbf{f}, \\ \frac{\partial A(c)}{\partial c}\mathbf{u}_n + A(c)\frac{\partial \mathbf{u}_n}{\partial c} &= 0, \\ -A(c)^{-1}\frac{\partial A(c)}{\partial c}\mathbf{u}_n &= \frac{\partial \mathbf{u}_n}{\partial c}. \end{aligned}$$

Door deze term voor de afgeleide van $\mathbf{u}_n$ in vergelijking (4.8) te substitueren, volgt dat de afgeleide van $\phi(c)$ kan worden geschreven als

$$\begin{aligned} \phi'(c) &= \sum_{n=0}^{N} \left(-A(c)^{-1}\frac{\partial A(c)}{\partial c}\mathbf{u}_n\right)^T \mathbf{p} r_n, \\ &= -\sum_{n=0}^{N} \mathbf{u}_n^T \left(\frac{\partial A(c)}{\partial c}\right)^T A(c)^{-T} \mathbf{p} r_n. \end{aligned} \tag{4.16}$$

Het principe is hier hetzelfde als bij (4.15), alleen wordt er een andere notatie gebruikt.

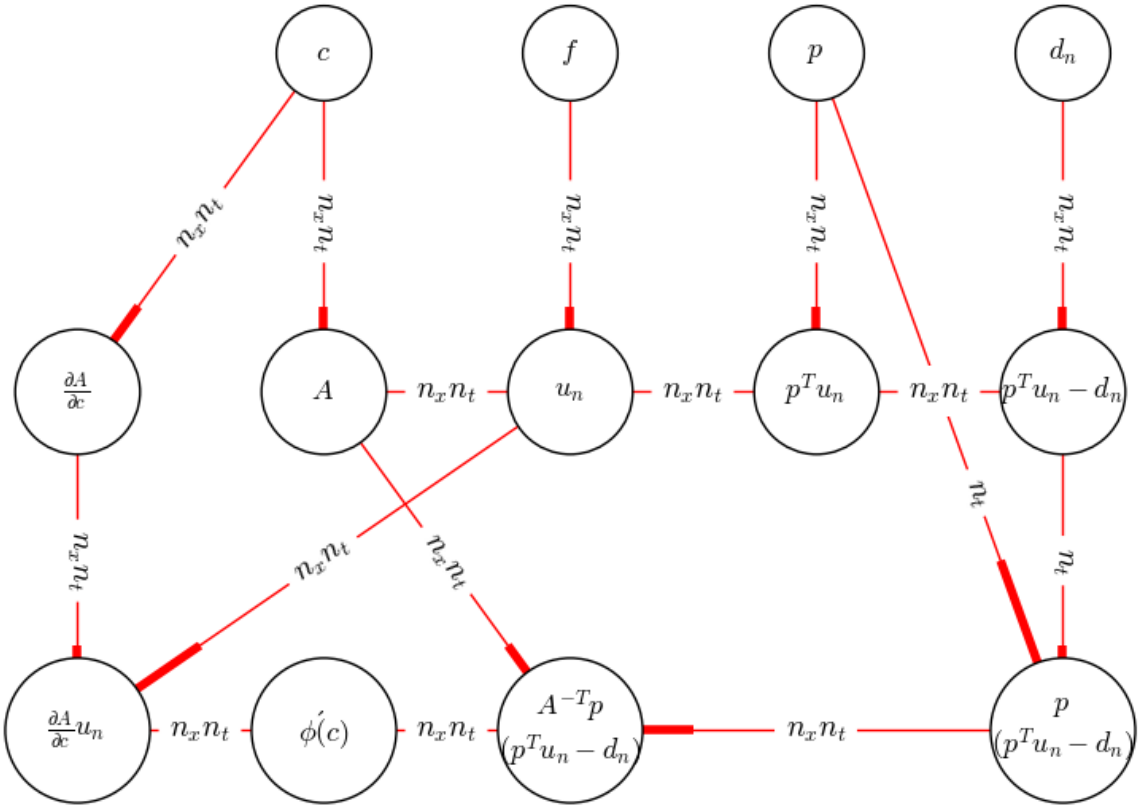
De berekening met behulp van deze methode kan op dezelfde manier worden weergegeven als bij de *forward method*. Als vergelijking (4.5) wordt gebruikt, dan volgen de volgende stappen:

$$\phi'(c) = \sum_{n=1}^{N-1} 2c \left[\mathbf{u}_n\right]^T L^T \left[\mathbf{v}_{n+1}\right].$$

Eerst wordt $\mathbf{u}_n$ berekend, vervolgens r_n en aan de hand daarvan $\mathbf{v}_{n+1}$. Als laatste wordt hiermee $\phi'(c)$ bepaald. Als het in termen van vergelijking (4.16) wordt opgeschreven, dan volgt:

$$\phi'(c) = -\sum_{n=0}^{N} \left[\left[\mathbf{u}_n^T\right] \left(\frac{\partial A(c)}{\partial c}\right)^T \right] \left[A(c)^{-T}\mathbf{p}\left[r_n\right]\right].$$

Eerst wordt $\mathbf{u}_n$ berekend, vervolgens r_n, $A(c)^{-T}\mathbf{p}r_n$ en $\mathbf{u}_n^T \left(\frac{\partial A(c)}{\partial c}\right)^T$. Als laatste wordt hiermee $\phi'(c)$ bepaald. Deze methode komt overeen met het pad dat belopen wordt met behulp van de zwarte, paarse, rode en gele pijlen. Dit resulteert in de deelgraaf hieronder.



Figuur 4.5: Een gerichte acyclische graaf van de berekening van $\phi'(c)$ met behulp van de *reverse method*.

De berekening (4.15) die hier uitgevoerd wordt, komt overeen met de topologische ordening $[\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n - d_n,\ \mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ A^{-T}(c)\mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ \mathbf{u}_n^T \frac{\partial A(c)}{\partial c}^T,\ \phi'(c)]$. Merk op dat alle termen binnen deze topologische ordening afhankelijk van $\mathbf{u}_n$ zijn, waardoor er geen gebruik gemaakt kan worden van parallelle berekeningen.

Om deze methode te kunnen gebruiken, moeten de waarden van $\mathbf{v}_n$ bepaald worden. Dit gebeurt aan de hand van de functie `solveAt(c,nx,nt,dt,f)`.

```
def solveAt(c,nx,nt,dt,f):
    v = np.zeros((nt, nx))
    v[nt - 1] = dt ** 2 * f[nt - 1]
    v[nt - 2] = 2 * v[nt - 1] + dt ** 2 * c ** 2 * np.transpose(L) @ v[nt - 1]
              + dt ** 2 * f[nt - 2]
    for i in range(nt-2,-1,-1):
        v[i - 1] = 2 * v[i] - v[i + 1] + dt ** 2 * c ** 2 * np.transpose(L) @ v[i]
                 + dt ** 2 * f[i-1]
    return v
```

Met behulp van deze functie en de functies die $\mathbf{u}_n$ en d_n bepalen, kan de $\phi'(c)$ worden bepaald aan de hand van de methode. De Python code die zo volgt is:

```
def dphi(c0,c1,nx,nt,dt,pk):
    # Bepaal benodigde gegevens
    o = np.ones(nx)
    elementen = np.array([o, -2 * o, o])
    diags = np.array([-1, 0, 1])
    global L
    L = spdiags(elementen, diags, nx, nx).toarray()

    # Bepaal de vector f, is nodig voor u
    f = np.zeros((nt, nx))
    x = np.linspace(0, 1, nx)
    f[0] = np.exp(x ** 2)
    f[1] = np.exp(x ** 2)

    # bepaal p en d
    p = np.zeros(nx)
    p[pk] = 1
    d = data(c0, nt, dt, p, f)

    # Bepaal de vector u
    u = solveA(c1,nt,dt,f)

    # bepaal r
    r = np.zeros(nt)
    for i in range(nt):
        r[i] = np.transpose(p) @ u[i] - d[i]

    #bepaal q en v
    q = np.zeros((nt,nx))
    for i in range(1, nt-1):
        q[i] = p * r[i]
    v=solveAt(c1,nx,nt,dt,q)

    # Bepaal de afgeleide dphi
    dphi = 0
    for i in range(0,nt-1):
        dphi += 2 * c1 * np.transpose(u[i]) @ np.transpose(L) @ v[i + 1]
    return dphi
```

Uit de graaf volgt dat deze methode een complexiteit van $\mathcal{O}(n_x n_t)$ heeft. Merk op dat het bij deze methode niet uitmaakt of ϕ van één of meer parameters afhangt, de complexiteit blijft altijd hetzelfde. Deze methode is dus in sommige gevallen sneller dan de vorige methode. Echter, misschien kan dit met behulp van parallelle berekeningen nog sneller. Dit leidt tot een tussenliggende methode.

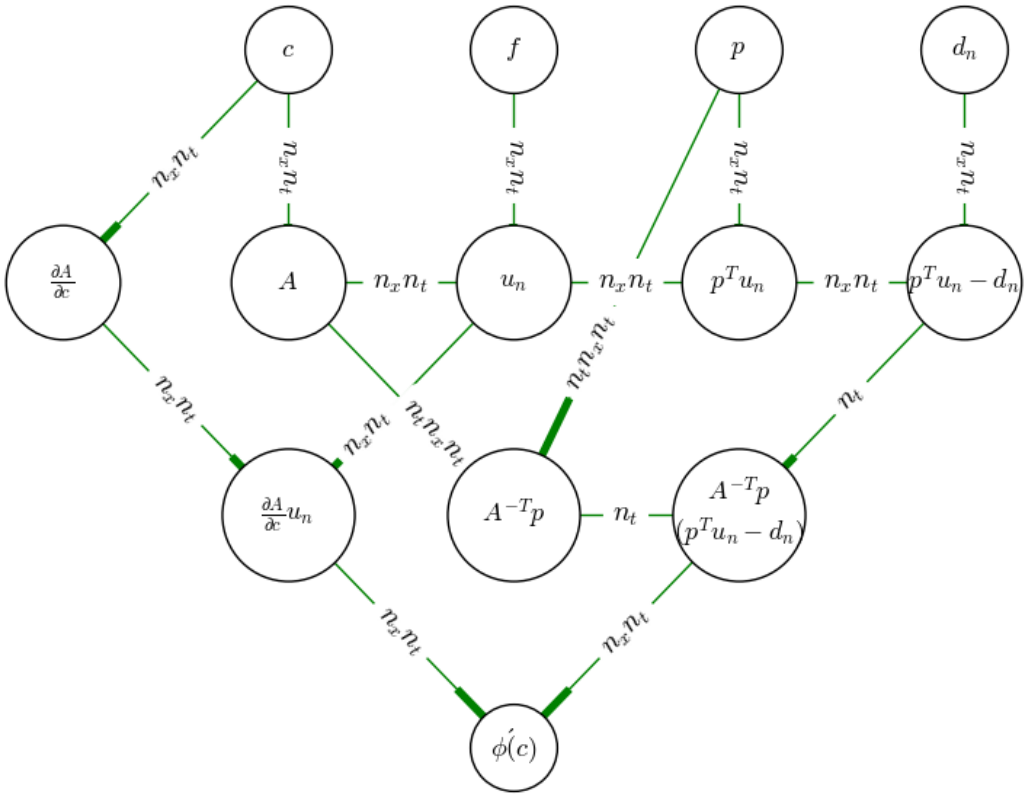
Tussenliggende methode

De tussenliggende methode maakt gebruik van parallelle berekeningen. Hierbij wordt de berekening van $\mathbf{v}_{n+1}$ opgesplitst in twee verschillende berekeningen. Het pad wat zo belopen wordt, is het pad via de zwarte, gele en groene pijlen. De deelgraaf is in figuur 4.6 te zien is. Deze graaf kan topologisch gesorteerd worden. Merk op dat deze topologische ordening niet uniek is en er dus geschoven kan worden met sommige elementen. Een simpele topologische ordening is van links naar rechts, van boven naar beneden en geeft de ordening $[c,\ \mathbf{f},\ \mathbf{p},\ \mathbf{d},\ \frac{\partial A(c)}{\partial c},\ A(c),\ \mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n - d_n,\ A^{-T}(c)\mathbf{p},\ A^{-T}(c)\mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ \mathbf{u}_n^T\frac{\partial A(c)}{\partial c}^T,\ \phi'(c)]$. Door het weglaten van de triviale elementen en het schuiven van de onafhankelijke elementen wordt de ordening $[\mathbf{u}_n,\ A^{-T}(c)\mathbf{p},\ \mathbf{p}^T\mathbf{u}_n,\ \mathbf{p}^T\mathbf{u}_n - d_n,\ \mathbf{u}_n^T\frac{\partial A(c)}{\partial c}^T,\ A^{-T}(c)\mathbf{p}(\mathbf{p}^T\mathbf{u}_n - d_n),\ \phi'(c)]$ gevonden. Merk op dat $A(c)^{-T}\mathbf{p}$ niet van $\mathbf{u}_n$ afhankelijk is. Hierom is het mogelijk om gebruik te maken van parallelle berekeningen. Het pad dat op deze manier wordt belopen komt overeen met de volgende

stappen:

$$\phi'(c) = -\sum_{n=0}^{N} \left[[\mathbf{u}_n^T] \left(\frac{\partial A(c)}{\partial c} \right)^T \right] [A(c)^{-T}\mathbf{p}] [r_n]. \tag{4.17}$$

Eerst worden $\mathbf{u}_n$ en $A(c)^{-T}\mathbf{p}$ gelijktijdig berekend, vervolgens r_n en $\mathbf{u}_n^T \left(\frac{\partial A(c)}{\partial c} \right)^T$ en als laatste wordt hiermee $\phi'(c)$ bepaald.



Figuur 4.6: Een gerichte acyclische graaf van de berekening van $\phi'(c)$ met behulp van een tussenliggende methode.

Gebruikmakend van vergelijking (4.15) volgt dat deze methode $\mathbf{v}_n$ in twee delen opsplitst: $\mathbf{w}_n$ en r_n. Deze methode voert dan de berekening

$$\phi'(c) = \sum_{n=1}^{N-1} 2c \left[\mathbf{u}_n\right]^T L^T \left[\mathbf{w}_{n+1}\right] r_{n+1}$$

uit, waarbij

$$\mathbf{w}_{n-1} = 2\mathbf{w}_n - \mathbf{w}_{n+1} + \Delta t^2 c^2 L^T \mathbf{w}_n + \Delta t^2 \mathbf{p}.$$

Merk op dat $\mathbf{w}_n$ onafhankelijk van $\mathbf{u}_n$ is en ze dus gelijktijdig berekend kunnen worden.
Deze methode maakt, net als de *reverse method*, gebruik van de functies `solveA(c,nt,dt,f)`, `data(c,nt,dt,p,f)` en `solveAt(c,nx,nt,dt,f)`. De Python code voor de afgeleide van ϕ is hieronder weergegeven.

```
def dphi(c0,c1,nx,nt,dt,pk):
    # Bepaal benodigde gegevens
    o = np.ones(nx)
    elementen = np.array([o, -2 * o, o])
    diags = np.array([-1, 0, 1])
    global L
```

```
    L = spdiags(elementen, diags, nx, nx).toarray()

    # Bepaal de vector f, is nodig voor u
    f = np.zeros((nt, nx))
    x = np.linspace(0, 1, nx)
    f[0] = np.exp(x ** 2)
    f[1] = np.exp(x ** 2)

    #bepaal p en d
    p = np.zeros(nx)
    p[pk] = 1
    d=data(c0,nt,dt,p,f)

    # Bepaal de vector u
    u = solveA(c1,nt,dt,f)

    # bepaal r
    r = np.zeros(nt)
    for i in range(nt):
        r[i] = np.transpose(p) @ u[i] - d[i]

    w=np.zeros((nt,nx,nt))
    for j in range (0,nt):
        q = np.zeros((nt, nx))
        q[j]=p
        #bepaal w
        w[:,:,j]=solveAt(c1,nx,nt,dt,q)

    #bepaal de afgeleide dphi
    dphi=0
    for j in range(0,nt):
        for i in range(0,nt-1):
            dphi += 2 * c1 * np.transpose(u[i])
                @ np.transpose(L) @ w[j,:,i+1] * r[i+1]
    return dphi
```

Opmerking. Uit de graaf volgt dat deze methode een complexiteit van $\mathcal{O}(n_t n_x n_t)$ heeft. Eigenlijk is dit $\mathcal{O}(\frac{(n_t+1)n_x n_t}{2})$ vanwege het feit dat $A(c)$ een ijle matrix is en $\mathbf{p}$ ook uit veel triviale elementen bestaat.

De complexiteit is groter dan bij de *adjoint state approach*. Echter, in dit geval zijn $\mathbf{u}_n$ en $A^{-T}p$ gelijktijdig te berekenen, wat ervoor zorgt dat de berekening iets sneller gaat. Als de $A(c)$ een diagonaal matrix zou zijn, dan zou de complexiteit $\mathcal{O}(n_x n_t)$ worden, omdat de waarden $A(c)^{-T}p$ dan parallel berekend kunnen worden. Indien dit geldt, zou deze methode dus even snel als de *reverse method* zijn.

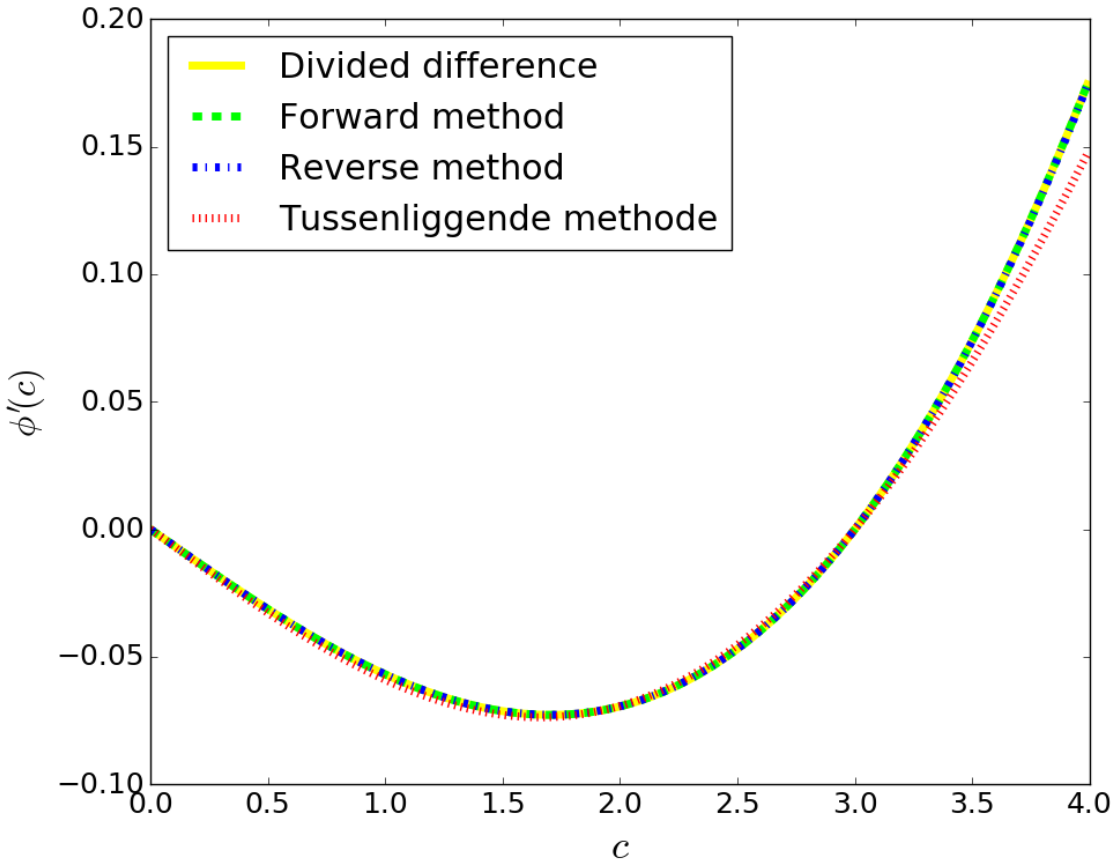
4.1.1.1.3 Discussie

In deze paragraaf worden de methoden met elkaar vergeleken, om te kijken welke methode bij dit voorbeeld de voorkeur krijgt. Om de methoden met elkaar te kunnen vergelijken, is het handig om de functie te plotten. Laat $\texttt{h} = {}^1\!/_{1.000.000}$, $\texttt{nx} = 100$, $\texttt{nt} = 150$, $\texttt{dt} = \frac{1}{1000}$, $\texttt{pk} = 0$ en $\texttt{c0} = 2$. Met de code

```
d = np.arange(0.0, 4.0, 0.01)

vecfunc_forward = np.vectorize(dphi_forward)
forward = vecfunc_forward(3, d, 100, 150, 1 / 1000, 0)
vecfunc_reverse = np.vectorize(dphi_reverse)
reverse = vecfunc_reverse(3, d, 100, 150, 1 / 1000, 0)
vecfunc_numeriek = np.vectorize(dphi_numeriek)
numeriek = vecfunc_numeriek(3, d, 100, 150, 1 / 1000, 0)
vecfunc_forerse = np.vectorize(dphi_forerse)
forerse = vecfunc_forerse(3, d, 100, 150, 1 / 1000, 0)
```

```
fig = plt.figure()
ax  = fig.add_subplot(111)


ax.plot(d, numeriek, c = 'yellow', label = "Divided difference",
        linewidth = 5, linestyle = '-')
ax.plot(d, forward, c = 'lime', label = "Forward method",
        linewidth = 5, linestyle = '--')
ax.plot(d, reverse, c = 'b', label = "Reverse method",
        linewidth = 5, linestyle = '-.')
ax.plot(d, forerse, c = 'r', label = "Tussenliggende methode",
        linewidth = 5, linestyle = ':')

plt.rc('font', size=18)
plt.xlabel('$c$', fontsize=25)
plt.ylabel('$\phi\'(c)$', fontsize = 22)

leg = plt.legend(loc = 2)

plt.show()
```

kunnen de waarden voor $\phi'(c)$ met $c \in [0, 4]$ worden geplot, wat figuur 4.7 oplevert.



Figuur 4.7: De waarden van $\phi'(c)$, waarbij $\texttt{h} = {}^{1}/_{1.000.000}$, $\texttt{nx} = 100$, $\texttt{nt} = 150$, $\texttt{dt} = \frac{1}{1000}$, $\texttt{pk} = 0$ en $\texttt{c0} = 2$.

Uit dit figuur blijkt dat deze methoden dicht bij elkaar liggen. De tussenliggende methode lijkt het meest af te wijken van de rest. Om het verschil goed te bekijken, zijn de specifieke waarden van $\phi'(c)$ in tabel 4.1 te zien.

	$\phi'(c)$			
c	Divided difference	Forward method	Reverse method	Tussenliggende methode
0	0.0	0.0	0.0	0.0
1	-0.0569024136768	-0.0569023398048	-0.0569095095859	-0.0590349090903
2	-0.0694513538832	-0.0694514265537	-0.0694114877326	-0.0692519913116
3	-9.12611612342e-14	0.0	0.0	0.0
4	0.176737076883	0.176737025462	0.176200196704	0.148993795366

Tabel 4.1: De waarden van $\phi'(c)$ voor verschillende waarden van c. Hierbij zijn $\texttt{h} = {}^1/_{1.000.000}$, $\texttt{nx} = 100$, $\texttt{nt} = 150$, $\texttt{dt} = \frac{1}{1000}$, $\texttt{pk} = 0$ en $\texttt{c0} = 3$.

Het is opvallend dat het *divided difference* algoritme als enige voor $c = 3$ niet de waarde 0 krijgt. Dit komt doordat numerieke berekingen altijd een schatting zijn en er altijd een truncatiefout in voorkomt. Echter, uit dit tabel valt op te halen dat de *forward method* en het *divided difference* algoritme het dichtst bij elkaar liggen, gevolgd door de *reverse method* en de tussenliggende methode. Dat de drie methoden van algoritmische differentiatie van elkaar verschillen, komt waarschijnlijk door het feit dat ze de afgeleide op verschillende manieren berekenen. Doordat de computer de getallen afrond, wordt er verder gerekend met getallen met een kleine fout. Hierdoor kunnen de gevonden waarden van elkaar verschillen. Er geldt hier dat hoe beter de werkprecisie van de computer is, hoe nauwkeuriger de waarden van $\phi'(c)$ worden. Dit gaat niet op voor numerieke berekeningen, omdat daar altijd afrondfouten bij komen kijken. Dat de tussenliggende methode het meest afwijkt van de rest, komt waarschijnlijk doordat deze methode een extra rekenstap maakt, waardoor er meer ruimte is voor afrondfouten. Al met al lijkt het wel te kloppen dat algoritmische differentiatie een nauwkeurige waarde van $\phi'(c)$ geeft, terwijl het *divided difference* algoritme slechts een schatting is.
Zoals al eerder was opgemerkt, gaat de voorkeur eerder uit naar de *reverse method* dan naar de *forward method* vanwege de complexiteit. Ondanks dat deze hetzelfde was bij dit voorbeeld, zou de *forward method* een grotere complexiteit krijgen indien ϕ van meerdere parameters zou afhangen. De vraag was of een tussenliggende methode misschien sneller zou zijn. Helaas is de tussenliggende methode ondanks zijn parallelle berekeningen duurder dan *reverse method*. Dit is een goed argument om toch voor de *reverse method* te kiezen.

Snelheid is niet het enige waar rekening mee gehouden moet worden. Ook de kosten van het geheugen is erg belangrijk bij zulke berekeningen, aangezien een computer hierin beperkt is. Doordat de tussenliggende methode een extra rekenstap maakt, moeten meer variabelen worden opgeslagen, wat extra geheugen kost. In plaats van dat $\mathbf{v}_n$ direct wordt bepaald, wordt eerst $\mathbf{w}_n$ bepaald. De *reverse method* gebruikt dus relatief minder geheugen. Aangezien zo'n vector $\mathbf{w}_n$ voor grote n behoorlijk groot kan zijn, is dit iets om rekening mee te houden wanneer er een afweging wordt gemaakt welke methode te gebruiken.

HOOFDSTUK 5

Het algoritme voor de snelste berekening

Ondanks dat de tussenliggende methode niet sneller werkt dan de *reverse method*, betekent dit niet dat de *reverse method* altijd het snelst is. Zoals al eerder benoemd werd, is voor de functie $f : \mathbb{R}^n \to \mathbb{R}^m$ de *forward method* sneller indien $m \gg n$. Bovendien kan het voorkomen dat in sommige gevallen er een sneller programma ontstaat met behulp van parallelle berekeningen. Om de beste manier van algoritmische differentiatie te vinden, kan gebruikt worden van het volgende algoritme:

1. Stel een gerichte acyclische graaf op met alle losstaande berekeningen van de functie. Hierbij mag zo'n berekening uit enkel één operatie bestaan. Indien meerdere berekeningen tot dezelfde uitdrukking leiden, is het verstandig verschillende kleuren te gebruiken. Dit behoudt het overzicht.
2. Splits de graaf op in deelgrafen, waarbij elke deelgraaf een ander pad doorloopt (en dus andere berekeningen uitvoert) naar de afgeleide functie.
3. Bepaal de complexiteit van elke berekening en zet deze bij de deelgrafen.
4. Zoek onafhankelijke berekeningen in de deelgrafen. Dit kan door simpelweg de pijlen in de graaf te volgen.
5. Bepaal de totale rekentijd van de graaf, rekening houdend met dat de onafhankelijke berekeningen gelijktijdig uitgevoerd kunnen worden. Dit is gelijk aan de grootste complexiteit van de graaf min de winst, die wordt gewonnen door de parallelle berekening.
6. Kies de graaf met de kleinste rekentijd.
7. Maak een topologische ordening van de graaf, waarbij de parallelle berekeningen elkaar opvolgen. Dit is de uiteindelijke volgorde waarmee de afgeleide wordt bepaald.

Dit algoritme is op verschillende (niet-)lineaire systemen toe te passen. Merk op dat indien er parallelle berekeningen te vinden zijn, dit niet betekent dat hier gelijk gebruik van gemaakt gaat worden. De totale rekentijd van de berekening kan namelijk nog steeds groter zijn dan bij een andere graaf. Bovendien is het ook verstandig om, zoals al eerder vermeld is, rekening te houden met de kosten van het geheugen.

HOOFDSTUK 6

Conclusie

In dit verslag is met behulp van het *divided difference* algoritme en algoritmische differentiatie de afgeleide van het inverse probleem van de één-dimensionale golfvergelijking berekend. Vervolgens werden deze methoden met elkaar vergeleken en ontstond er een algoritme, die van een functie bepaalt welke berekeningen het beste opeenvolgend uitgevoerd kunnen worden, zodat de rekentijd zo kort mogelijk is.

Het *divided difference* algoritme kwam erg in de buurt van de exacte waarden van $\phi'(c)$, maar vanwege truncatiefouten blijft het slechts een schatting. Met behulp van algoritmische differenitatie is wel de exacte waarden van $\phi'(c)$ te berekenen. Alleen vanwege het feit dat de computer geen onbeperkt geheugen bevat en hierom de waarden afrond, bevatte de waarden van $\phi'(c)$ nog steeds een kleine fout.

De *forward method* werd als eerst gebruikt om de afgeleide te bepalen en kwam het meest in de buurt van de waarden van het *divided difference* algoritme. Deze berekening had een complexiteit van $\mathcal{O}(n_x n_t)$, maar zodra $\phi(c)$ van meer parameters zou afhangen, zou de rekentijd stijgen. Dit was een goede reden om naar de *reverse method* te gaan kijken. Deze maakt gebruik van terugwaartse berekeningen, waardoor deze altijd een complexiteit $\mathcal{O}(n_x n_t)$ zou hebben. Dit is dus beter dan bij de *forward method*. Vervolgens werd er onderzocht of het nog sneller kon met behulp van parallelle berekeningen. Dit leidde tot een tussenliggende methode, welke een complexiteit van $\mathcal{O}(\frac{(n_t+1)n_x n_t}{2})$ heeft. Helaas is deze methode dus niet sneller. Bovendien bleken de waarden van $\phi'(c)$ bij de tussenliggende methode het meest af te wijken van de andere methoden. De reden hiervan is dat de tussenliggende methode een extra rekenstap maakt, waardoor er meer ruimte is voor de afrondfouten van de computer.

Door middel van het voorbeeld ontstond er een algoritme, die van een functie bepaalt welke volgorde van berekeningen zorgen voor het snelste programma. Door dit algoritme te volgen, volgt het snelste programma. Echter, wanneer men de verschillende methoden tegen elkaar opweegt, is de snelheid niet het enige waarop gelet moet worden. Er moet ook rekening gehouden worden met het geheugengebruik, waarbij de *reverse method* beter naar voren kwam dan de tussenliggende methode. Door de extra rekenstap gebruikt de tussenliggende methode namelijk meer geheugenruimte.

Al met al lijkt het erop dat de *reverse method* de beste keus is om de afgeleide van het inverse probleem te berekenen. Zowel qua snelheid als qua geheugengebruik laat deze methode zich van zijn beste kant zien. Als het gaat om een willekeurige functie, dan kan het algoritme worden gebruikt om de beste methode te vinden. Indien het gaat om een functie $f : \mathbb{R}^n \to \mathbb{R}^m$, dan krijgt de *forward method* de voorkeur indien $m \gg n$. Echter, het kan zijn dat bij bepaalde functies de berekening van zijn afgeleide sneller gaat met behulp van parallelle berekeningen. Dit valt nog verder te onderzoeken. Bovendien geldt dat ondanks dat het gelukt is om $\phi'(c)$ te bepalen, er altijd ruimte is voor verbetering. Om het inverse probleem op te kunnen lossen, is het nodig om ook een programma te hebben, die de tweede afgeleide bepaalt. Een mooie aanpassing zou zijn om vervolgens deze programma's ook werkende te krijgen voor $\phi'(c_1, c_2, \ldots, c_k)$, met

$k \in \mathbb{N}$. Wanneer dit gelukt is, kan de Newton-Raphson methode worden gebruikt om het minimum van het inverse probleem te vinden.

Met de technologie die continu verbeterd wordt, is er een goede toekomst voor algoritmische differentiatie. Het staat al erg bekend om zijn efficiëntie, maar er valt nog veel in dit onderwerp te onderzoeken. Een verdere onderzoek naar de combinatie met parallelle berekeningen is ook zeker aan te raden.

Bibliografie

[1] B. Pearlmutter, "*Automatic Differentiation: History and Headroom.*" https://autodiff-workshop.github.io/slides/BarakPearlmutter.pdf. Accessed: 2017-05-15.

[2] C. Floudas and P. Pardalos, *Encyclopedia of Optimization.* Springer Science and Business Media, second ed., 2008.

[3] A. Griewank and A. Walther, *Evaluating Derivative.* Society for Industrial and Applied Mathematics, Philadelphia, second ed., 2008.

[4] "*Parallel Computing: Background .*" http://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf. Accessed: 2017-05-19.

[5] J. Hoogeveen, "*Grafen.*" http://www.cs.uu.nl/docs/vakken/b3dw/Grafen.pdf. Accessed: 2017-05-15.

[6] "*DAGs and Topological Ordering.*" https://ocw.tudelft.nl/wp-content/uploads/Algoritmiek_DAGs_and_Topological_Ordering.pdf. Accessed: 2017-05-15.

[7] "*DAGs and Topological Ordering.*" http://homes.cs.washington.edu/~jrl/421slides/lec5.pdf. Accessed: 2017-05-15.

[8] C. Agarwal, "*Kahns algorithm for Topological Sorting.*" http://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/. Accessed: 2017-05-15.

[9] "*Directed Acyclich Graph.*" https://en.wikipedia.org/wiki/Directed_acyclic_graph. Accessed: 2017-05-15.

[10] "*Wait-for-graph.*" https://en.wikipedia.org/wiki/Wait-for_graph. Accessed: 2017-05-15.

[11] "*Data Structures.*" http://www.cs.nthu.edu.tw/~wkhon/ds/ds11/lecture/lecture12.pdf. Accessed: 2017-05-15.

[12] L. v. Iersel, "*Optimalisering in Netwerken.*" http://homepages.cwi.nl/~iersel/2WO12/2WO12-lec5-handout.pdf. Accessed: 2017-05-15.

[13] "*Topological Sorting.*" https://en.wikipedia.org/wiki/Topological_sorting. Accessed: 2017-05-15.

[14] A. Radul, "*Introduction to Automatic Differentiation.*" http://alexey.radul.name/ideas/2013/introduction-to-automatic-differentiation. Accessed: 2017-05-28.

[15] Wikipedia, "*Introduction to Automatic Differentiation.*" https://en.wikipedia.org/wiki/Automatic_differentiation#cite_note-8. Accessed: 2017-05-15.

[16] V. Katz, *A History of Mathematics: An Introduction.* Addison-Wesley, third ed., 2009.

[17] U. Ascher and C. Greif, *A First Course in Numerical Methods.* Society for Industrial and Applied Mathematics, Philadelphia, first ed., 2011.

Appendices

BIJLAGE A

Python code

In dit hoofdstuk staan alle Python codes, die gebruikt zijn voor het maken van de grafen. De Python codes zijn waarschijnlijk niet het meest praktisch en ook qua geheugen en snelheid is er misschien het één en ander te verbeteren. Merk op dat er verschillende packages zijn die hetzelfde resultaat kunnen geven. Hierbij heeft elke package zijn voor- en nadelen.

Voor het maken van de grafen is er gekozen voor de package *igraph* en *networkx*. Let op dat bij de elke graaf die met de package *igraph* wordt gemaakt, de indeling er anders uit kan zien. De graaf is verder wel hetzelfde.

Parallel programmeren

Hieronder staan de codes die voor de figuren in hoofdstuk 2 zorgden. De grafen uit dit hoofdstuk zijn alle met de package *igraph* gemaakt.

Gerichte acyclische grafen

```
import igraph
import numpy as np

vertices = ["1", "2", "3", "4", "5", "6", "7"]
edges = [(0,1),(1,2),(1,3), (1,4), (2,4), (4,5), (3,4), (6,3)]
#By the directed cyclic graph there's also an edge (2,0)

g = igraph.Graph(vertex_attrs={"label": vertices}, edges=edges, directed=True)

#plot(g)
visual_style = {}

# Scale vertices based on degree
outdegree = g.outdegree()
visual_style["vertex_size"] = [60]
visual_style["vertex_label_size"] = 20

# Set bbox and margin
visual_style["bbox"] = (800, 350)
visual_style["margin"] = 50
```

```
# Set colors of edges and vertices
g.es["color"] = "black"
g.vs["color"] = "white"

# Don't curve the edges
visual_style["edge_curved"] = False

# Plot the graph
igraph.plot(g, **visual_style)
```

Topologische ordening

De code voor deze graaf is grotendeels gelijk aan de vorige code, alleen wordt hier

```
vertices = ["1", "2", "3", "4", "5", "6", "7"]
edges = [(0,1),(1,2),(1,3), (1,4), (2,4), (4,5), (3,4), (6,3)]

# Set bbox and margin
visual_style["bbox"] = (800, 350)
```

vervangen door

```
vertices = ["1", "2", "3", "4", "5", "6", "7", "8"]
edges = [(2,7),(3,7),(3,4), (1,4), (1,6), (4,5), (7,5), (7,0)]

# Set bbox and margin
visual_style["bbox"] = (400, 400)
```

Parallelle berekeningen

De code voor deze graaf is grotendeels gelijk aan de vorige code, alleen wordt hier

```
vertices = ["1", "2", "3", "4", "5", "6", "7"]
edges = [(0,1),(1,2),(1,3), (1,4), (2,4), (4,5), (3,4), (6,3)]

# Set bbox and margin
visual_style["bbox"] = (800, 350)
```

vervangen door

```
vertices = ["v-1", "v0", "v1", "v2", "v3", "v4", "v5", "v6", "y"]
edges = [(0,2),(1,2),(2,3), (2,5), (1,4), (4,5), (3,6), (5,6), (6,7), (5,7)]

# Set bbox and margin
visual_style["bbox"] = (800, 300)
```

Toepassing van algoritmische differentiatie

Hieronder staan de codes die voor de figuren in hoofdstuk 4 zorgden. De grafen uit dit hoofdstuk zijn alle met de package *networkx*, verkregen uit *SciPy*, gemaakt. Voor deze package is het nodig om ook de package *numpy+klm* te hebben.

Algoritmische differentiatie

Voor de grote graaf is de code zoals hieronder.

```
import networkx as nx
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 8))
G = nx.DiGraph()
```

```
nodelist_small = ["c", "f", "p", "d", "phi'(c)"]
nodelist_normal = ["dA", "A", "u", "x", "r"]
nodelist_large = ["du", "q", "w", "o", "v"]

pos = {"dA": (0, 3), "c": (1, 4), "f": (2, 4), "p": (3, 4), "d": (4, 4),
       "q": (1, 2), "A": (1, 3), "u": (2, 3), "x": (3, 3), "r": (4, 3),
       "du": (0, 2), "w": (2, 2), "o": (4, 2), "v": (3, 2), "phi'(c)": (2, 1)}

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_small,
                       node_color='w',
                       node_size=1850)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_normal,
                       node_color='w',
                       node_size=3200)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_large,
                       node_color='w',
                       node_size=4300)


edgelist_black = [("dA", "q"), ("u", "q"), ("c", "A"), ("c", "dA"), ("f", "u"),
                  ("p", "x"), ("d", "r"), ("A", "u"), ("u", "x"), ("x", "r")]
edgelist_blue = [("A", "du"), ("q", "du"),
                 ("o", "phi'(c)"), ("du", "phi'(c)")]
edgelist_red = [("A", "v"), ("o", "v")]
edgelist_green = [("p", "w"), ("A", "w"), ("w", "v"), ("r", "v")]
edgelist_magenta = [("p", "o"), ("r", "o")]
edgelist_yellow = [("v", "phi'(c)"), ("q", "phi'(c)")]

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_black,
                       edge_color='k', arrows=True)

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_blue,
                       edge_color='b', arrows=True)

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_red,
                       edge_color='r', arrows=True)

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_green,
                       edge_color='g', arrows=True)

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_magenta,
                       edge_color='m', arrows=True)

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_yellow,
                       edge_color='y', arrows=True)

labels={}
labels["dA"] = r'$\frac{\partial A}{\partial c}$'
labels["A"] = r'$A$'
labels["c"] = r'$c$'
labels["f"] = r'$f$'
labels["p"] = r'$p$'
```

```
labels["d"] = r'$d_n$'
labels["q"] = r'$\frac{\partial A}{\partial c} u_n$'
labels["u"] = r'$u_n$'
labels["x"] = r'$p^T u_n$'
labels["r"] = r'$p^T u_n - d_n$'
labels["du"] = r'$\frac{\partial u_n}{\partial c}$'
labels["w"] = r'$A^{-T} p$'
labels["o"] = r'$p$' '\n' '$(p^T u_n - d_n)$'
labels["v"] = r'$A^{-T}p$' '\n' '$(p^T u_n - d_n)$'
labels["phi'(c)"] = r'$\phi \'(c)$'
nx.draw_networkx_labels(G,pos,labels,font_size=13)

plt.show()
```

Voor de kleine graaf is de code hieronder gegeven.

```
import networkx as nx
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 4))
G = nx.DiGraph()
pos = {1: ( 1,1), 2: (2,0), 3: (3,1)}

nx.draw_networkx_nodes(G, pos, nodelist=[1,2,3],
                       node_color='w',
                       node_size=1850)

nx.draw_networkx_edges(G, pos,
                       edgelist=[(1,2), (3,2)],
                       edge_color='k', arrows=True)

labels={}
labels[1] = '$A$'
labels[2] = '$AB$'
labels[3] = '$B$'
nx.draw_networkx_labels(G,pos,labels,font_size=13)

edge_labels = nx.get_edge_attributes(G,'r')
edge_labels[(1,2)] = '$n^2 \log n$'
edge_labels[(3,2)] = '$n^2 \log n$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_labels,
                        color='g', font_size=14, label_pos=0.5)

plt.show()
```

Forward method

```
import networkx as nx
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 8))
G = nx.DiGraph()

nodelist_small = ["c", "f", "p", "d"]
nodelist_normal = ["dA", "A", "u", "x", "r"]
nodelist_large = ["du", "o", "q",  "phi'(c)"]

pos = {"dA": (0, 3), "c": (1, 4), "f": (2, 4), "p": (3, 4), "d": (4, 4),
       "q": (1, 2), "A": (1, 3), "u": (2, 3), "x": (3, 3), "r": (4, 3),
       "du": (2, 2), "o": (4, 2), "phi'(c)": (3, 2)}
```

```
nx.draw_networkx_nodes(G, pos, nodelist=nodelist_small,
                       node_color='w',
                       node_size=1850)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_normal,
                       node_color='w',
                       node_size=3200)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_large,
                       node_color='w',
                       node_size=4300)

edgelist_blue = [("c", "A"), ("c", "dA"), ("f", "u"), ("p", "x"),
                 ("d", "r"), ("A", "u"), ("u", "x"), ("x", "r"),
                 ("dA", "q"), ("u", "q"), ("A", "du"), ("q", "du"),
                 ("o", "phi'(c)"), ("du", "phi'(c)"), ("p", "o"), ("r", "o")]

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_blue,
                       edge_color='b', arrows=True)

labels={}
labels["dA"] = r'$\frac{\partial A}{\partial c}$'
labels["A"] = r'$A$'
labels["c"] = r'$c$'
labels["f"] = r'$f$'
labels["p"] = r'$p$'
labels["d"] = r'$d_n$'
labels["q"] = r'$\frac{\partial A}{\partial c} u_n$'
labels["u"] = r'$u_n$'
labels["x"] = r'$p^T u_n$'
labels["r"] = r'$p^T u_n - d_n$'
labels["du"] = r'$\frac{\partial u_n}{\partial c}$'
labels["o"] = r'$p$' '\n' '$(p^T u_n - d_n)$'
labels["phi'(c)"] = r'$\phi \'(c)$'
nx.draw_networkx_labels(G,pos,labels,font_size=13)

edge_labels = nx.get_edge_attributes(G,'r')
edge_labels[("c", "A")] = '$n_x n_t$'
edge_labels[("c", "dA")] = '$n_x n_t$'
edge_labels[("f","u")] = '$n_x n_t$'
edge_labels[("p","x")] = '$n_x n_t$'
edge_labels[("d","r")] = '$n_x n_t$'
edge_labels[("A","u")] = '$n_x n_t$'
edge_labels[("u","x")] = '$n_x n_t$'
edge_labels[("x","r")] = '$n_x n_t$'
edge_labels[("dA","q")] = '$n_x n_t$'
edge_labels[("A","du")] = '$n_x n_t$'
edge_labels[("q","du")] = '$n_x n_t$'
edge_labels[("o","phi'(c)")] = '$n_x n_t$'
edge_labels[("du","phi'(c)")] = '$n_x n_t$'
edge_labels[("r","o")] = '$n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_labels, color='g',
                             font_size=14, label_pos=0.5)

edge_label = nx.get_edge_attributes(G,'r')
edge_label[("p","o")] = '$n_t$'
edge_label[("u","q")] = '$n_x n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_label, color='g',
                             font_size=14, label_pos=0.35)
```

```
plt.show()
```

Reverse method

```
import networkx as nx
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 8))
G = nx.DiGraph()

nodelist_small = ["c", "f", "p", "d"]
nodelist_normal = ["A", "dA", "u", "x", "r"]
nodelist_large = ["o", "phi'(c)", "v", "z"]

pos = {"dA": (0, 3), "c": (1, 4), "f": (2, 4), "p": (3, 4), "d": (4, 4),
       "A": (1, 3), "u": (2, 3), "x": (3, 3), "r": (4, 3), "z": (0, 2),
       "o": (4, 2), "v": (2, 2), "phi'(c)": (1, 2)}

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_small,
                       node_color='w',
                       node_size=1850)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_normal,
                       node_color='w',
                       node_size=3200)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_large,
                       node_color='w',
                       node_size=4300)


edgelist_red = [("c", "A"), ("c", "dA"), ("f", "u"), ("p", "x"),
                ("d", "r"), ("A", "u"), ("u", "x"), ("x", "r"),
                ("A", "v"), ("o", "v"), ("u", "z"), ("z", "phi'(c)"),
                ("dA", "z"), ("p", "o"), ("r", "o"), ("v", "phi'(c)")]

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_red,
                       edge_color='r', arrows=True)

labels={}
labels["dA"] = r'$\frac{\partial A}{\partial c}$'
labels["A"] = r'$A$'
labels["c"] = r'$c$'
labels["f"] = r'$f$'
labels["p"] = r'$p$'
labels["d"] = r'$d_n$'
labels["u"] = r'$u_n$'
labels["x"] = r'$p^T u_n$'
labels["r"] = r'$p^T u_n - d_n$'
labels["z"] = r'$\frac{\partial A}{\partial c} u_n$'
labels["o"] = r'$p$' '\n' '$(p^T u_n - d_n)$'
labels["v"] = r'$A^{-T}p$' '\n' '$(p^T u_n - d_n)$'
labels["phi'(c)"] = r'$\phi \'(c)$'
nx.draw_networkx_labels(G,pos,labels,font_size=13)

edge_labels = nx.get_edge_attributes(G,'r')
edge_labels[("c", "A")] = '$n_x n_t$'
edge_labels[("c", "dA")] = '$n_x n_t$'
edge_labels[("f","u")] = '$n_x n_t$'
edge_labels[("p","x")] = '$n_x n_t$'
```

```
edge_labels[("d","r")] = '$n_x n_t$'
edge_labels[("A","u")] = '$n_x n_t$'
edge_labels[("u","x")] = '$n_x n_t$'
edge_labels[("x","r")] = '$n_x n_t$'
edge_labels[("r","o")] = '$n_t$'
edge_labels[("A","v")] = '$n_x n_t$'
edge_labels[("o","v")] = '$n_x n_t$'
edge_labels[("u","z")] = '$n_x n_t$'
edge_labels[("z","phi'(c)")] = '$n_x n_t$'
edge_labels[("dA","z")] = '$n_x n_t$'
edge_labels[("v","phi'(c)")] = '$n_x n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_labels, color='g',
                             font_size=14, label_pos=0.5)

edge_label = nx.get_edge_attributes(G,'r')
edge_label[("p","o")] = '$n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_label, color='g',
                             font_size=14, label_pos=0.35)

plt.show()
```

Tussenliggende methode

```
import networkx as nx
from matplotlib import pyplot as plt

plt.figure(figsize=(12, 8))
G = nx.DiGraph()

nodelist_small = ["c", "f", "p", "d", "phi'(c)"]
nodelist_normal = ["A", "dA", "u", "x", "r"]
nodelist_large = ["z", "w", "v"]

pos = {"dA": (0, 3), "c": (1, 4), "f": (2, 4), "p": (3, 4), "d": (4, 4),
       "A": (1, 3), "u": (2, 3), "x": (3, 3), "r": (4, 3), "z": (1,2),
       "w": (2, 2), "v": (3, 2), "phi'(c)": (2, 1)}

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_small,
                       node_color='w',
                       node_size=1850)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_normal,
                       node_color='w',
                       node_size=3200)

nx.draw_networkx_nodes(G, pos, nodelist=nodelist_large,
                       node_color='w',
                       node_size=4300)


edgelist_green = [("c", "A"), ("c", "dA"), ("f", "u"), ("p", "x"),
                  ("d", "r"), ("A", "u"), ("u", "x"), ("x", "r"),
                  ("p", "w"), ("A", "w"), ("w", "v"), ("r", "v"),
                  ("u", "z"), ("z", "phi'(c)"), ("dA", "z"),
                  ("v", "phi'(c)")]

nx.draw_networkx_edges(G, pos,
                       edgelist=edgelist_green,
                       edge_color='g', arrows=True)

labels={}
```

```
labels["dA"] = r'$\frac{\partial A}{\partial c}$'
labels["A"] = r'$A$'
labels["c"] = r'$c$'
labels["f"] = r'$f$'
labels["p"] = r'$p$'
labels["d"] = r'$d_n$'
labels["u"] = r'$u_n$'
labels["x"] = r'$p^T u_n$'
labels["r"] = r'$p^T u_n - d_n$'
labels["z"] = r'$\frac{\partial A}{\partial c} u_n$'
labels["w"] = r'$A^{-T} p$'
labels["v"] = r'$A^{-T}p$' '\n' '$(p^T u_n - d_n)$'
labels["phi'(c)"] = r'$\phi \'(c)$'
nx.draw_networkx_labels(G,pos,labels,font_size=13)

edge_labels = nx.get_edge_attributes(G,'r')
edge_labels[("c", "A")] = '$n_x n_t$'
edge_labels[("c", "dA")] = '$n_x n_t$'
edge_labels[("dA", "z")] = '$n_x n_t$'
edge_labels[("f","u")] = '$n_x n_t$'
edge_labels[("p","x")] = '$n_x n_t$'
edge_labels[("d","r")] = '$n_x n_t$'
edge_labels[("A","u")] = '$n_x n_t$'
edge_labels[("u","x")] = '$n_x n_t$'
edge_labels[("x","r")] = '$n_x n_t$'
edge_labels[("w","v")] = '$n_t$'
edge_labels[("r","v")] = '$n_t$'
edge_labels[("z","phi'(c)")] = '$n_x n_t$'
edge_labels[("v","phi'(c)")] = '$n_x n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_labels, color='g',
                             font_size=14, label_pos=0.5)

edge_label = nx.get_edge_attributes(G,'r')
edge_label[("u","z")] = '$n_x n_t$'
edge_label[("p","w")] = '$n_t n_x n_t$'
edge_label[("A","w")] = '$n_t n_x n_t$'
nx.draw_networkx_edge_labels(G, pos, edge_labels = edge_label, color='g',
                             font_size=14, label_pos=0.35)

plt.show()
```

BIJLAGE B

Eén-dimensionale golfvergelijking

Op de volgende pagina staat een artikel van universitair docent Tristan van Leeuwen, waarin het inverse probleem van de één-dimensionale golfvergelijking wordt neergelegd. De informatie die hierin gegeven staat, is gebruikt in sectie 4.1.

An inverse problem for the 1D wave-equation

Tristan van Leeuwen
Utrecht University, the Netherlands

Abstract

Partial differential equations are used to model model many natural phenomena. If the underlying parameters (typically coefficients in de PDE) are known, (numerical) solutions of the PDE can be used to predict or study the behaviour of a particular system. In some cases, however, we want to do the opposite; given observations of a system and an underlying model (PDE), determine the coefficients such that the predictions fit the observations. This is a prime example of an *inverse problem*. Solving inverse problems governed by PDEs involves several disciplines, in particular numerical solution of PDEs and optimization. In this homework assignment you will develop a numerical method for solving an inverse problem with the wave equation. Basic knowledge of linear algebra and undergraduate numerical analysis should be sufficient to complete the questions. The method is best implemented using a high-level programming language that supports scientific computing such as Matlab, Octave or Python (with SciPy).

Wave phenomena are harnessed in many applications to reconstruct images of the interior of an object. Examples are seismology and medical ultrasound imaging. The basic setup for all these applications is the same; generate a wave using a *source*, let it interact with the object under investigation and record the response with a *receiver*. If we repeat the procedure for various source-reveiver configurations we can collect enough data to determine the inner structure of the object. To illustrate the main characteristics of the problem from a mathematical point of view, we start with a simple one-dimensional situation. Imagine an infinitely long thin rod along which waves can travel with a constant speed c. We can generate a wave by tapping it with a hamopenmer. The aim is to determine c by measuring the response at a single location.

1 THE FORWARD MODEL

We model wave propagation in our infinitely long rod with a one dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \tag{1}$$

for $x \in \mathbb{R}$ and $t \in [0, T]$ with initial conditions $u(0, x) = u_0(x)$, $\frac{\partial u}{\partial t}(0, x) = 0$. Here, c denotes the soundspeed.

We are ultimately interested in retrieving the soundspeed given the initial conditions and measurements at a single location $x = r$: $d(t) = u(t, r)$.

1.1 Spatial discretization

We discretize (1) on the finite interval $x \in [-1, 1]$ with grid-spacing h and obtain a system of ODEs

$$\ddot{\mathbf{u}}(t) = c^2 L \mathbf{u}(t), \tag{2}$$

where $\mathbf{u}(t) = [u(-1+h, t), u(-1+2h, t), \ldots, u(1-2h, t), u(1-h, t)]^T$ and

$$L = \frac{1}{h^2} \begin{pmatrix} -2 & 1 & & \\ 1 & -2 & 1 & \\ & & \ddots & \\ & & 1 & -2 \end{pmatrix},$$

is the $n \times n$ differentiation matrix where $n = 2/h - 1$ denotes the number of gridpoints.

In order to emulate an infinite domain using a finite interval we introduce a so-called *sponge layer* that damps the solution near the boundary. This is achieved by adding a first-order temporal derivate term to the system of ODEs

$$\ddot{\mathbf{u}}(t) + 2\Sigma\dot{\mathbf{u}}(t) = c^2 L\mathbf{u}(t), \tag{3}$$

where Σ is an $n \times n$ positive diagonal matrix with elements σ_i (i denotes position from $-1+h$ to $1-h$) which are zero (or very small) in the center of the domain and increase towards the boundaries. An example of such a profile is shown in figure 1.

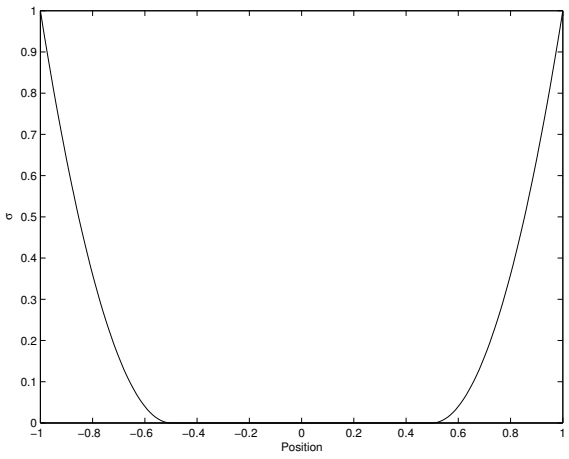


Fig. 1: Variation of σ with respect to position

Question 1 Show that the matrix Σ indeed has a damping effect on the solution. You may assume $\Sigma = \sigma I$ for some $\sigma > 0$ for this analysis. Hint: express the solution as $\mathbf{u}(t) = \sum_{k=0}^{n} \alpha_k \mathbf{v}_k e^{\nu_k t}$ where $\mathbf{v}_k$ are eigenvectors of L and show that $\Re(\nu_k) < 0$.

Answer 1 We write the solution as

$$\mathbf{u}(t) = \sum_{k=0}^{n} \alpha_k \mathbf{u}_k e^{\nu_k t},$$

where $\mathbf{u}_k$ are eigenfunctions of L. This yields

$$\nu_k^2 + 2\sigma\nu_k - \lambda_k c^2 = 0,$$

where λ_k is an eigenvalue of L. We find

$$\nu_k = -\sigma \pm \sqrt{\sigma^2 + \lambda_k c^2}.$$

We know that the eigenvalues of L are negative so when $\sigma^2 + \lambda_k c^2 < 0$ the roots lie in the complex plain with a negative real part σ which causes the solution to damp out exponentially over time.

The sponge layer is perhaps the simplest way to damp unwanted reflections from the boundary. Alternatives are so-called *absorbing boundary conditions* and *perfectly matched layers* (PML). The former imposes a one-way wave-equation at the boundary allowing only outgoing waves. PML boundary conditions are based on a complex coordinate transform in the Fourier domain.

1.2 Time-stepping

A well-known method for integrating the wave-equation is the Leap-Frog (LF) method which is based on central FD approximation of the temporal derivatives:

$$\dot{\mathbf{u}}(n\Delta t) \approx \frac{1}{2\Delta t}\left(\mathbf{u}_{n+1} - \mathbf{u}_{n-1}\right),$$

and

$$\ddot{\mathbf{u}}(n\Delta t) \approx \frac{1}{\Delta t^2}\left(\mathbf{u}_{n+1} - 2\mathbf{u}_n + \mathbf{u}_{n-1}\right),$$

where $\mathbf{u}_n \equiv \mathbf{u}(n\Delta t)$. This leads to

$$\mathbf{u}_{k+1} = 2(I + \Delta t\Sigma)^{-1}\mathbf{u}_k - (I + \Delta t\Sigma)^{-1}(I - \Delta t\Sigma)\mathbf{u}_{k-1} + \Delta t^2 c^2 (I + \Delta t\Sigma)^{-1} L\mathbf{u}_k. \tag{4}$$

Question 2 Analyze the stability of the Leap-Frog method and give a criterion for Δt in terms of h, σ and c. You may assume $\Sigma = \sigma I$ for some $\sigma > 0$ for this analysis. Hint: take $\mathbf{u}_k = G^k\mathbf{v}$, where $\mathbf{v}$ is an eigenvector of L and give conditions for which $|G| < 1$. The eigenvalues of L are all negative and $> -4/h^2$.

Answer 2 We apply a Neumann stability analysis, taking a solution of the form $\mathbf{u}_k = G^k\mathbf{s}$, where $\mathbf{s}$ an eigenvector of L. The method is stable when $|G| \leq 1$. We find

$$\left(G^2 - 2\alpha G + \beta\right)\mathbf{s} = 0,$$

where $\alpha = \left(\frac{1+\Delta t^2 c^2\lambda/2}{1+\Delta t\sigma}\right)$ and $\beta = \frac{1-\Delta t\sigma}{1+\Delta t\sigma}$. The solutions are given by

$$G = \alpha \pm \sqrt{\alpha^2 - \beta^2}.$$

We have stability when $|\alpha| < |\beta| \leq 1$. Since $\sigma, \Delta t$ are both positive we have $|\beta| \leq 1$. Since $\lambda \in [-4/h^2, 0]$ we have stability when $\frac{\Delta t^2 c^2}{h^2} \leq 1 + \Delta t\sigma/2$.

Activity 1 Implement the LF method to compute the solution $\mathbf{u}_k$ for a given $\mathbf{u}_0$, c and $h, \Delta t$. In Matlab you can use `spdiags` to construct L.

2 THE INVERSE PROBLEM

The measurements are modelled as a linear sampling of the solution

$$d_n = \mathbf{p}^T\mathbf{u}_n,$$

where $\mathbf{p}$ is a vector of weights that determine where we sample the solution. We can now pose the inverse problem as a non-linear least-squares problem

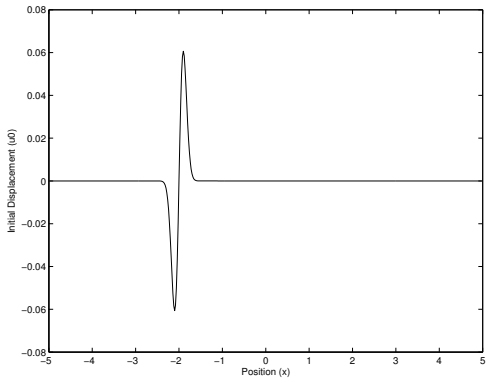
$$\min_c \phi(c),$$

with

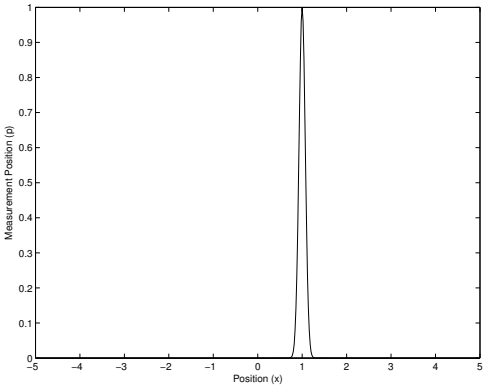
$$\phi(c) = \frac{1}{2}\sum_{n=0}^{N}\left(\mathbf{p}^T\mathbf{u}_n(c) - d_n\right)^2, \tag{5}$$

where $\mathbf{u}_n(c)$ is satisfies (4).

Activity 2 Implement a function that computes the least-squares misfit for a given c, $\mathbf{u}_0$, $\mathbf{p}$ and $\Delta t, \Delta x$. Take $u_0(x) = (x+1)e^{-100(x+1)^2}$ and $p(x) = e^{-100(x-1)^2}$ and generate observations for $c = 1$ en compute the misfit for $c \in [\frac{1}{2}, \frac{3}{2}]$.

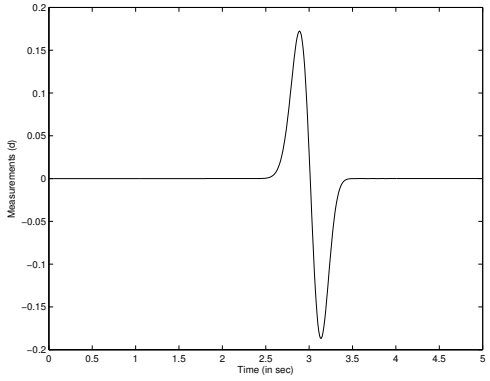


(a) Initial Displacement at $T = 0$

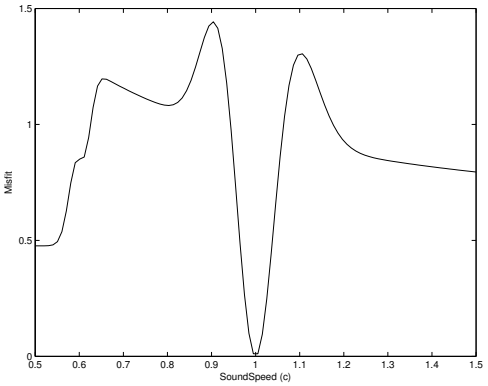


(b) Measurement vector

Fig. 2: Initial displacement and measurement vector for given problem



(a) Measurements by sensor over time



(b) Misfit of data with respect to soundspeed

Fig. 3: Measurements by sensor and misfit of data for given problem

The value for c that minimizes ϕ is said to be the solution of our inverse problem. Although in this case we can find the solution easily by densely sampling the function ϕ, for higher -dimensional inverse problems we typically use an iterative method. A minimizer of ϕ obeys $\phi'(c) = 0$. We can find such a stationary point using Newton's method as follows

$$c_{k+1} = c_k - \frac{\phi'(c_k)}{\phi''(c_k)},$$

starting from some initial guess c_0. To employ this method, we need to compute derivatives of ϕ.

2.1 Computing derivatives

We have

$$\phi'(c) = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{p}^T \mathbf{u}_n}{\partial c} \right) r_n = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p} r_n,$$

where $r_n = \left(\mathbf{p}^T \mathbf{u}_n(c) - d_n\right)$ denotes the residual. Differentiation of the recursion relation (4) yields

$$\frac{\partial \mathbf{u}_{n+1}}{\partial c} = 2\frac{\partial \mathbf{u}_n}{\partial c} - \frac{\partial \mathbf{u}_{n-1}}{\partial c} + \Delta t^2 c^2 L \frac{\partial \mathbf{u}_n}{\partial c} + 2\Delta t^2 c L \mathbf{u}_n.$$

Although we could use this relation to obtain the quantities $\frac{\partial \mathbf{u}_{n+1}}{\partial c}$ needed to compute the derivative, it is more convenient to re-organize the computations. The reason for this is that in this form we would need to repeat the computation for each parameter.

Question 3 Show that the derivative can be expressed as

$$\phi'(c) = \sum_{n=1}^{N-1} 2c\, \mathbf{u}_n^T L^T \mathbf{v}_{n+1},$$

where $\mathbf{v}_n$ obeys the following recursion relation

$$\mathbf{v}_{n-1} = 2\mathbf{v}_n - \mathbf{v}_{n+1} + \Delta t^2 c^2 L^T \mathbf{v}_n + \Delta t^2 \mathbf{p} r_{n-1},$$

with $\mathbf{v}_N = \Delta t^2 \mathbf{p} r_N$ and $\mathbf{v}_{N-1} = 2\mathbf{v}_N + \Delta t^2 c^2 L^T \mathbf{v}_N + \Delta t^2 \mathbf{p} r_{N-1}$.

Answer 3 We use the method of Lagrange multiplier to find the derivative of $\phi(c)$. The Lagrange function is given by $\Lambda(\mathbf{u}, c, \mathbf{v}) = \frac{1}{2}(\mathbf{p}^T\mathbf{u}-\mathbf{d})^2 - \mathbf{v}^T(A(c)\mathbf{u}-\mathbf{f})$ where $(A(c)\mathbf{u}-\mathbf{f})$ is a wave equation, $\mathbf{u} = [u_0, u_1, \ldots, u_N]^T$ and $\mathbf{f} = [u(0,x), 0, \ldots, 0]^T$. Matrix $A(c)$ can be obtained from equation 4 and is given as

$$A(c) = \begin{pmatrix} I & & & & & \\ -I & I & & & & \\ P & Q & R & & & \\ & & \ddots & & & \\ & & & P & Q & R \end{pmatrix},$$

where $P = (I - \Delta t\Sigma)/\Delta t^2, Q = -(2I + \Delta t^2 c^2 L)/\Delta t^2$ and $R = (I + \Delta t\Sigma)/\Delta t^2$. For finding the minimum of $\phi = \frac{1}{2}(\mathbf{p}^T\mathbf{u} - \mathbf{d})^2$, the gradient of Λ must be considered, i.e., $\nabla_{\mathbf{u},c,\mathbf{v}}\Lambda(\mathbf{u}, c, \mathbf{v}) = [0, \phi'(c), 0]^T$. This gives

$$\frac{\partial \Lambda}{\partial \mathbf{u}} = \mathbf{p}.(\mathbf{p}^T\mathbf{u} - \mathbf{d}) - A(c)^T\mathbf{v} = 0 \tag{6}$$

$$\frac{\partial \Lambda}{\partial c} = -\left(\frac{\partial A(c)\mathbf{u}}{\partial c}\right)^T \mathbf{v} = \phi'(c) \tag{7}$$

$$\frac{\partial \Lambda}{\partial \mathbf{v}} = -(A(c)\mathbf{u} - \mathbf{f}) = 0 \tag{8}$$

From equation 6, we can calculate v and from equation 7, we get $\phi'(c)$.

$$A(c)^T\mathbf{v} = \mathbf{p}.(\mathbf{p}^T\mathbf{u} - \mathbf{d}) \tag{9}$$

$$\begin{pmatrix} I & -I & P^T & & & & \\ & I & Q^T & & & & \\ & & R^T & & & & \\ & & & \ddots & & & \\ & & & & R^T & Q^T & P^T \\ & & & & & R^T & Q^T \\ & & & & & & R^T \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{N-2} \\ v_{N-1} \\ v_N \end{pmatrix} = \begin{pmatrix} \mathbf{p}r_0 \\ \mathbf{p}r_1 \\ \mathbf{p}r_2 \\ \vdots \\ \mathbf{p}r_{N-2} \\ \mathbf{p}r_{N-1} \\ \mathbf{p}r_N \end{pmatrix} \tag{10}$$

The expressions for v_k are obtained (given in question 3) by solving equation 10. The derivative of $\phi(c)$ is calculated as follows:

$$\begin{aligned} \phi'(c) &= -\left(\frac{\partial A(c)u}{\partial c}\right)^T v \\ &= \sum_{n=1}^{N-1} \left(\frac{\partial (c^2 L\mathbf{u}_n)}{\partial c}\right)^T \mathbf{v}_{n+1} \\ &= \sum_{n=1}^{N-1} 2c\,\mathbf{u}_n^T L^T \mathbf{v}_{n+1} \end{aligned}$$

Question 4 Compute the second derivative of ϕ.

Answer 4 From normal method, second derivative of ϕ is

$$\phi''(c) = \sum_{n=0}^{N} \frac{\partial \left(\left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p} r_n \right)}{\partial c} = \sum_{n=0}^{N} \left(\frac{\partial \mathbf{u}_n}{\partial c} \right)^T \mathbf{p}\mathbf{p}^T \left(\frac{\partial \mathbf{u}_n}{\partial c} \right) + \left(\frac{\partial^2 \mathbf{u}_n}{\partial c^2} \right)^T \mathbf{p} r_n \tag{11}$$

where

$$\frac{\partial^2 \mathbf{u}_{n+1}}{\partial c^2} = 2\frac{\partial^2 \mathbf{u}_n}{\partial c^2} - \frac{\partial^2 \mathbf{u}_{n-1}}{\partial c^2} + \Delta t^2 c^2 L \frac{\partial^2 \mathbf{u}_n}{\partial c^2} + 4\Delta t^2 c L \frac{\partial \mathbf{u}_n}{\partial c} + 2\Delta t^2 L \mathbf{u}_n. \tag{12}$$

By adjoint-state method, again the Lagrange function is used to compute.

$$\left[\nabla^2_{\mathbf{u},c,\mathbf{v}} \Lambda(\mathbf{u}, c, \mathbf{v}) \right] (\partial \mathbf{u}, \partial c, \partial \mathbf{v})^T = (0, -\partial \phi'(c), 0)^T$$

The hessian of Lagrange function $\Lambda(\mathbf{u}, c, \mathbf{v})$ is given as:

$$\nabla^2_{\mathbf{u},c,\mathbf{v}} \Lambda(\mathbf{u}, c, \mathbf{v}) = \begin{pmatrix} \mathbf{p}\mathbf{p}^T & \mathbf{v}^T \left(\frac{\partial A(c)}{\partial c} \right) & A(c)^T \\ \left(\frac{\partial A(c)}{\partial c} \right)^T \mathbf{v} & \mathbf{v}^T \left(\frac{\partial^2 A(c)\mathbf{u}}{\partial c^2} \right) & \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right)^T \\ A(c) & \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right) & 0 \end{pmatrix} \tag{13}$$

Hence, we obtain $\partial \mathbf{u}$ and $\partial \mathbf{v}$ in terms of ∂c as follows:

$$\partial \mathbf{u} = -A(c)^{-1} \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right) \partial c$$

$$\partial \mathbf{v} = \left(A(c)^T \right)^{-1} \left[\mathbf{p}\mathbf{p}^T A(c)^{-1} \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right) - \mathbf{v}^T \left(\frac{\partial A(c)}{\partial c} \right) \right].$$

Big expression for $\phi''(c)$ is as follows:

$$\begin{aligned} \phi''(c) = &- \left(\frac{\partial A(c)}{\partial c} \right)^T A(c)^{-1} \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right) \mathbf{v} + \mathbf{v}^T \left(\frac{\partial^2 A(c)\mathbf{u}}{\partial c^2} \right) \\ &+ \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right)^T \left(A(c)^T \right)^{-1} \mathbf{p}\mathbf{p}^T A(c)^{-1} \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right) - \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right)^T \left(A(c)^T \right)^{-1} \mathbf{v}^T \left(\frac{\partial A(c)}{\partial c} \right) \end{aligned}$$

We can ignore the double derivative (small changes) and negative terms. Hence, the only term remaining is $\left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right)^T \left(A(c)^T \right)^{-1} \mathbf{p}\mathbf{p}^T A(c)^{-1} \left(\frac{\partial A(c)\mathbf{u}}{\partial c} \right)$. Hence,

$$\phi''(c) = \sum_{n=1}^{N-1} 2c\, \mathbf{u}_n^T L^T \mathbf{w}_{n+1} \tag{14}$$

where

$$\mathbf{w}_{n-1} = 2\mathbf{w}_n - \mathbf{w}_{n+1} + \Delta t^2 c^2 L^T \mathbf{w}_n + \Delta t^2 \mathbf{p}\mathbf{p}^T \mathbf{b_{n-1}},$$

with $\mathbf{w}_N = \Delta t^2 \mathbf{p}\mathbf{p}^T \mathbf{b_N}$ and $\mathbf{w}_{N-1} = 2\mathbf{w}_N + \Delta t^2 c^2 L^T \mathbf{w}_N + \Delta t^2 \mathbf{p}\mathbf{p}^T \mathbf{b_{N-1}}$. and

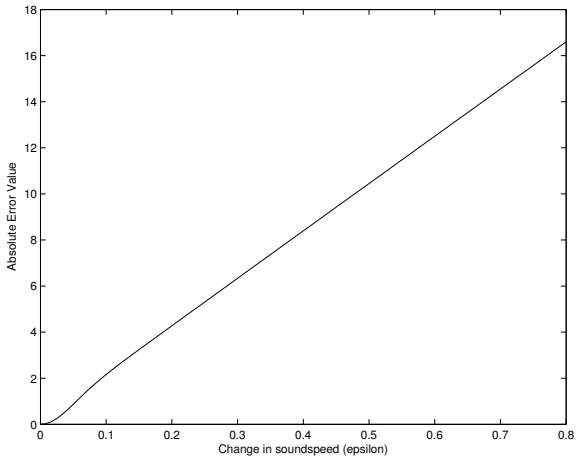
$$\mathbf{b}_{n+1} = 2\mathbf{b}_n - \mathbf{b}_{n-1} + \Delta t^2 c^2 L^T \mathbf{b}_n + 2c\Delta t^2 L \mathbf{u_n}$$

with $\mathbf{b}_0 = \mathbf{b}_1 = 0$.

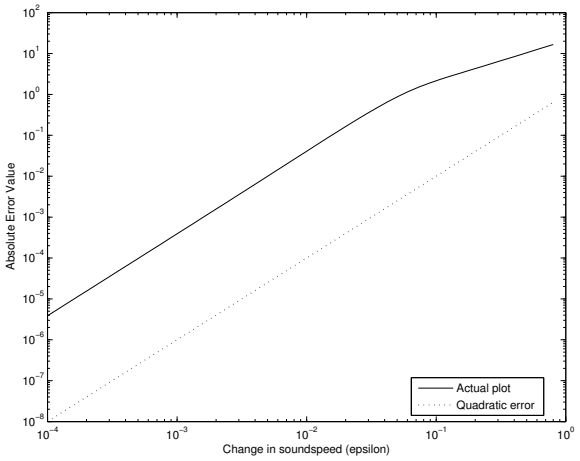
Activity 3 Implement the expressions for the first and second derivative and have your function from activity 2 return the value and first and second derivative of ϕ for a given c, $\mathbf{u}_0$, $\mathbf{p}$ and Δt, Δx. Test the first derivative by checking numerically that the quantity

$$\left\| \phi(c + \epsilon) - \phi(c) - \epsilon \phi'(c) \right\|,$$

decays as $\mathcal{O}(\epsilon^2)$ as $\epsilon \downarrow 0$.



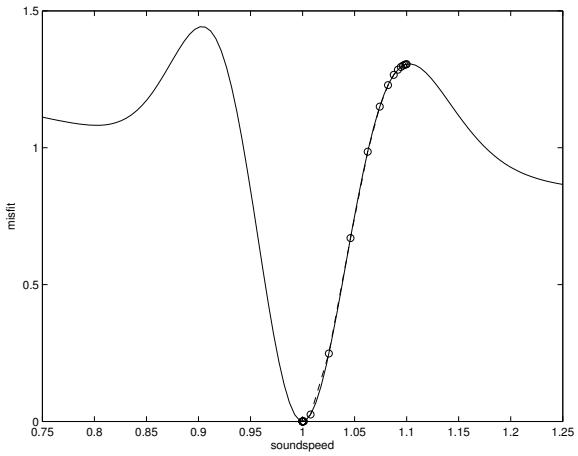
(a) Error Variation with ϵ (Normal Plot)



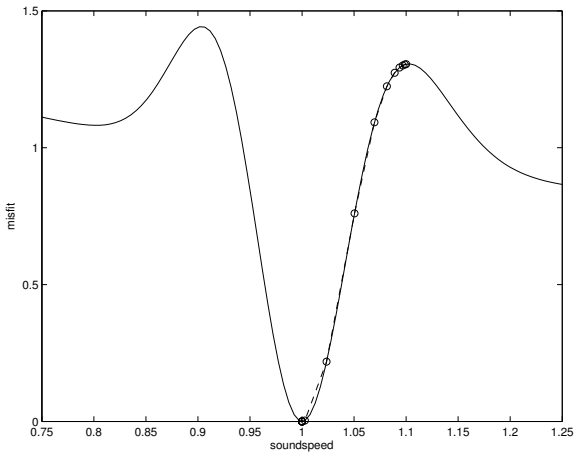
(b) Error Variation with ϵ (Logarithmic Plot)

Fig. 4: Error Variation of computed derivative of ϕ around soundspeed of 1.05

Activity 4 Solve an inverse problem using a standard library for optimization (with derivatives). Try different initial guesses for the soundspeed. What happens if the initial guess is too far from the true soundspeed? Can you explain this?



(a) Gradient Descent Method($\alpha = 0.001$)



(b) Newton method

Fig. 5: Finding optimum value of c through different methods (initial value of 1.1)

3 APPLICATION

If the velocity is spatially dependent, we can express the ODE as

$$\ddot{\mathbf{u}}(t) = C^2 L\mathbf{u}(t),$$

where C is a diagonal matrix containing the soundspeed at each gridpoint. We can derive the partial derivatives $\frac{\partial \phi}{\partial c_k}$ in a similar fashion as before. However, this would require us to compute $\frac{\partial \mathbf{u}_{n+1}}{\partial c_k}$ for each parameter separately. In this case it is more efficient to express the derivative as

$$\frac{\partial \phi}{\partial c_k}(\mathbf{c}) = \sum_{n=0}^{N} \mathbf{u}_n^T \left(\frac{\partial C^2}{\partial c_k} \right)^T \mathbf{v}_{n+1},$$

where $\mathbf{v}_n$ obeys the following (backward) recursion relation

$$\mathbf{v}_{n-1} = 2\mathbf{v}_n - \mathbf{v}_{n+1} + \Delta t^2 C^2 L^T \mathbf{v}_n + \Delta t^2 \mathbf{p} r_n,$$

with $\mathbf{v}_N = \Delta t^2 \mathbf{p} r_N$ and $\mathbf{v}_{N-1} = 2\mathbf{v}_N - \Delta t^2 C^2 L^T \mathbf{v}_N + \Delta t^2 \mathbf{p} r_N$. This is referred to as the *adjoint state approach*. With this expression we can re-use the computed $\mathbf{v}$ to compute all partial derivatives.

Challenge 1 Repeat the exercise for a 2-D wave-equation on a rectangular domain with spatially varying parameters $c(x)$. Instead of taking only a single measurement for a single initial condition, we now need multiple measurements $d_{ij}(t) = \mathbf{p}_i^T \mathbf{u}^{(j)}(t)$ where $\mathbf{u}^{(j)}$ is the solution of (3) with $\mathbf{u}(0) = \mathbf{u}_0^{(j)}$. The optimization problem can no longer be solved using a simple 1D Newton iteration. Many standard packages are available for high-dimensional optimization problems. The typical input for such codes is a function that provides the function value and gradient and possibly a function that computes Hessian-vector products for a given vector of parameters. In Matlab you can use for example `fminunc`.

Challenge 3 Higher order methods in space or extensions to irregular domains can be easily implemented by replacing the matrix L by an appropriate pseudo-spectral or finite element matrix. Many open-source packages are available for doing this. In Matlab for example you can use the `MATLAB Differentiation Matrix Suite` or ...