TU Delft

BSc verslag TECHNISCHE WISKUNDE

“Bayesiaanse dichtheidsschattingen”

(“Bayesian density estimation”)

Phine Schikhof

Technische Universiteit Delft

Begeleider

Dr.ir. F.H. van der Meulen

Overige commissieleden

Dr. N.V. Budko | Drs. E.M. van Elderen

Juli, 2021 | Delft

Voorwoord

Dit verslag is geschreven door een derde jaars Bachelor student van de studie Technische Wiskunde aan de TU Delft. Het is een onderdeel van het Bachelor eindproject.

Dit verslag is geschreven voor mensen die in het eind van hun Bachelor (technische) wiskunde zitten. Er wordt enige kennis van (Bayesiaanse) statistiek en Markovketens verwacht.

Graag bedank ik mijn begeleider Frank van der Meulen voor zijn hulp en advies bij zowel het schrijven van het verslag als bij het bedenken van de inhoud. Ook wil ik graag Hugo Tirion bedanken die mij heeft geholpen met doorlezen van de tekst.

J.C.C. Schikhof
Delft, Juli, 2021

Samenvatting

Voor leken

In de statistiek wordt een model opgesteld voor data, gebruik makend van kansberekening. In het model komen stochastische variabelen voor, wat variabelen zijn waar de waarde bepaald wordt door een kansverdeling. Er zijn twee takken binnen de statistiek, klassiek en Bayesiaans. In de klassieke statistiek wordt alleen de data gemodelleerd door een stochastische variabele en hebben de parameters in het model vaste, eventueel onbekende, waarden.

In de Bayesiaanse statistiek zijn ook de parameters stochastische variabelen. Dit betekent dat er een verschil is in de manier waarin om wordt gegaan met onzekerheid tussen de twee takken. Het voordeel van Bayesiaanse statistiek is dat het mogelijk is met weinig datapunten toch zinnige resultaten te krijgen. Dit komt doordat het het mogelijk is om kennis over de parameters mee te geven in de verdeling van de stochastische variabelen. Met deze kennis is wordt de uitkomst van het model gestuurd. Het sturen kan ook een nadeel zijn omdat incorrecte aannames de uitkomst kunnen beïnvloeden. De mogelijkheid om te sturen wordt kleiner naarmate meer data beschikbaar is.

Om de verdeling van de parameters te schatten, kan het Metropolis-Hastings algoritme gebruikt worden. Dit algoritme maakt een lijst van waarden op basis van hoe waarschijnlijk het is dat bepaalde waarden binnen de verdeling voorkomen. De waarden worden voorgesteld door een voorstel kernel. In dit project worden drie kernels vergeleken, namelijk de *Random Walk*, de Langevin en de Barker kernel.

Het blijkt dat de Random Walk kernel de meeste informatie oplevert per tijdseenheid en dus sneller de verdeling in kaart brengt. Maar wanneer het aantal parameters toeneemt, is juist de Barker kernel handiger omdat deze methode slimmere waarden voorstelt, waardoor minder voorstellen nodig zijn om de verdeling in kaart te brengen.

Voor jaargenoten

Binnen de statistiek bestaan twee verschillende takken, klassieke en Bayesiaanse statistiek. Het verschil tussen Bayesiaanse statistiek en klassieke statistiek is dat binnen de Bayesiaanse statistiek alle onzekerheden in het kansmodel dat de data omschrijft, door middel van een kansverdeling beschreven worden. Hierdoor is niet alleen de data een stochastische variabele, maar de parameters in het model ook. Binnen de klassieke statistiek krijgen de parameters een vaste, eventueel onbekende waarde.

Een probleem definitie bestaat uit een verdeling van de data gegeven de parameters, dit wordt ook wel de aannemelijkheidsfunctie genoemd en een verdeling van de parameters, die de a priori verdeling wordt genoemd. De a priori verdeling kan zo gekozen worden dat deze de kennis van de parameters reflecteert. Vervolgens zijn deze verdelingen met behulp van de regels van voorwaardelijke kansen om te schrijven tot de verdeling van de parameters gegeven de data, wat de a posteriori verdeling genoemd wordt. De a posteriori verdeling bevat een integraal over de parameter ruimte. Deze integraal is vaak niet exact te berekenen, zeker niet wanneer het aantal parameters toeneemt.

Om toch inzicht in de a posteriori verdeling te kunnen krijgen, kan een schatting van de verdeling gemaakt worden met behulp van het Metropolis-Hastings algoritme. Dit algoritme genereert een ergodische Markovketen, wat wil zeggen dat de Markovketen aperiodiek en irreducibel is. Dit algoritme heeft een irreduceerbare Markovketen als input nodig. In dit verslag vergelijk worden drie mogelijke kernels als input voor het Metropolis-Hastings algoritme vergeleken. De drie kernels zijn het Random Walk, het Langevin en Barker voorstel. Deze hebben alle drie verschillende eigenschappen.

Eén van de eigenschappen van de kernels die vergeleken wordt, is de *effective sample size*. Dit is gedefinieerd

als

$$n_{eff} = \frac{n}{1 + 2\sum_{i \neq j} \rho(X_i, X_j)}$$

Het blijkt dat het *Random Walk* voorstel de hoogste *effective sample size* per tijdseenheid heeft. Dat betekent dat het Random Walk voorstel in principe minder trekkingen nodig heeft om de verdeling in kaart te brengen. Het Langevin en Barker voorstel hebben een vergelijkbare maar lagere effective sample size.

Elke kernel bevat een *tuning* parameter die bepaalt hoe ver twee opeenvolgende punten in de Markovketen van elkaar af liggen. Deze afstand mag niet te groot zijn, omdat er dan veel extreme waarden voorkomen in de keten. Te klein is ook niet goed, omdat er dan veel punten nodig zijn om te hele a posteriori verdeling in kaart te brengen. Wanneer het aantal parameters toeneemt, moet de stapgrootte verkleind worden. Het blijkt dat de stapgrootte in het Random Walk voorstel met $\Theta(d^{-1})$ schaalt, waarbij d het aantal dimensies is. Het Barker voorstel schaalt met $\Theta(d^{-1/3})$. Dit wordt geïllustreerd aan de hand van een voorbeeld uit de literatuur.

Inhoudsopgave

1 Introductie 1

2 Bayesiaanse statistiek 3
- 2.1 A priori verdeling kiezen . . . 3

3 Metropolis-Hastings algoritme 7

4 Verschillende kernels 9
- 4.1 Random Walk . . . 9
- 4.2 Barker . . . 9
  - 4.2.1 Oorsprong Barker voorstel . . . 10
- 4.3 Langevin . . . 12
- 4.4 Bereik parameter komt niet overeen met bereik kernel . . . 12
- 4.5 Tuning parameters . . . 13
- 4.6 Aantal iteraties tot invariante verdeling . . . 14

5 Effective sample size 19

6 Bayesiaans schatten van een bivariate kansdichtheid 23

7 Conclusie en Discussie 31

A Code bij alle hoofdstukken 33
- A.1 MH.py . . . 33
- A.2 Data.py . . . 34
- A.3 Posterior.py . . . 35

B Code bij hoofdstuk 2 37

C Code bij hoofdstuk 4 39
- C.1 Code bij paragraaf 4.5 . . . 39
- C.2 Code bij paragraaf 4.6 . . . 42

D Code bij hoofdstuk 5 45

E Code bij hoofdstuk 6 49

Bibliografie 55

1
Introductie

In de late 18de en begin 19de eeuw is de theorie voor Bayesiaanse statistiek ontwikkeld. Toch is Bayesiaanse statistiek pas vrij recent geleden populair geworden. Dit komt omdat toen de theorie bedacht werd, de algoritmen nog niet efficiënt genoeg waren en er natuurlijk geen computers bestonden. Bayesiaanse statistiek vergt namelijk veel rekencapaciteit. Tegenwoordig wordt Bayesiaanse statistiek onder andere gebruikt in machine learning.

Vrij recent is een artikel, zie [2], gepubliceerd waarin een nieuwe voorstel kernel voor het Metropolis-Hastings algoritme wordt geïntroduceerd. Dit voorstel wordt het Barker voorstel genoemd. Het Metropolis-Hastings algoritme genereert een ergodische Markovketen die de verdeling die op een multiplicatieve constante na bekend is.

Het doel van dit verslag is om een aantal eigenschappen van het Random Walk, Langevin en Barker voorstel te vergelijken aan de hand van een aantal voorbeelden. Er wordt gekeken naar het aantal iteraties voordat het Metropolis-Hastings algoritme met deze voorstellen de invariante verdeling bereikt. Ook wordt er gekeken naar de effective sample size. Als laatste worden de Random Walk en Barker kernels gebruikt om een hoger dimensionaal probleem op te lossen. Hierbij wordt gekeken naar hoe de stapgrootte schaalt met het aantal parameters.

Dit verslag is als volgt gestructureerd. In hoofdstuk 2 wordt Bayesiaanse statistiek geïntroduceerd. Ook wordt hier getoond hoe de invloed van slechte aannames afhangt van het aantal datapunten. In hoofdstuk 3 wordt uitgelegd hoe het Metropolis-Hastings algoritme werkt. Vervolgens worden in hoofdstuk 4 de drie verschillende kernels gedefinieerd. Het Barker voorstel wordt afgeleid. Vervolgens wordt er beschreven wat gedaan kan worden wanneer het bereik van de parameters niet overeenkomt met het bereik van de kernel. Daarna volgt een paragraaf over de tuning parameters en de gevolgen van het toepassen van het Metropolis-Hastings algoritme wanneer deze niet goed getuned is. In de laatste paragraaf van dit hoofdstuk wordt gekeken naar het aantal iteraties dat nodig is om de invariante verdeling te bereiken. Daarna volgt hoofdstuk 5 waarin de effective sample size van de drie verschillende kernels berekend wordt. In het laatste hoofdstuk worden de drie kernels toegepast om een bivariate kansdichtheid te schatten. Hierbij wordt gekeken naar hoe de stapgrootte schaalt met het aantal parameters.

2
Bayesiaanse statistiek

Bayesiaanse statistiek is de laatste jaren in populariteit toegenomen. Dit komt vooral doordat er efficiëntere algoritmen zijn ontworpen. De theorie achter Bayesiaanse statistiek is al ouder. Het verschil tussen Bayesiaanse statistiek en de klassieke statistiek zit in de omgang met onzekerheden. In klassieke statistiek wordt de data beschreven aan de hand van een kansverdeling met vaste, eventueel onbekende parameters [6]. In Bayesiaanse statistiek worden alle onzekerheden beschreven aan de hand van een kansverdeling. Zo wordt niet alleen de data beschreven door een stochastische variabele, maar ook de parameters. Aangezien het mogelijk is om kennis over de parameters via de kansverdeling van de parameters mee te geven, kan Bayesiaanse statistiek toegepast worden op situaties waarin weinig data beschikbaar is. Dit kan een voordeel zijn ten opzichte van de klassieke statistiek. Een nadeel van het mee kunnen geven van informatie is dat het hierdoor mogelijk is om de uitkomsten sterk te beïnvloeden met de keuze voor deze verdeling.

Een probleem definitie in de Bayesiaanse statistiek ziet als volgt uit:

$$X \mid \Theta \sim p_{X|\Theta}(x \mid \theta) \tag{2.1}$$

$$\Theta \sim p_{\Theta}(\theta) \tag{2.2}$$

Waarbij X een stochastische variabele is die de data beschrijft en Θ een stochastische variabele die de parameters beschrijft. Zoals te zien in (2.1) heeft een verzameling data gegeven parameters een bepaalde verdeling, die ook wel de aannemelijkheidsfunctie wordt genoemd. De parameters hebben ook een bepaalde verdeling, zie (2.2). Deze verdeling wordt de a priori verdeling genoemd. Uiteindelijk is het de bedoeling om aan de hand van de aannemelijkheidsfunctie en de a priori verdeling de a posteriori verdeling te vinden. Dit is de kansverdeling van de parameters gegeven de data. Aan de hand van de regels voor voorwaardelijke kansen vinden we het volgende.

$$p_{\Theta|X}(\theta \mid x) = \frac{p_{X,\Theta}(x,\theta)}{p_X(x)} = \frac{p_{X|\Theta}(x \mid \theta)p_{\Theta}(\theta)}{p_X(x)} = \frac{p_{X|\Theta}(x \mid \theta)p_{\Theta}(\theta)}{\int p_{X|\Theta}(x \mid \theta)p_{\Theta}(\theta)\mathrm{d}\theta}$$

De integraal $\int p(x \mid \theta)p(\theta)\mathrm{d}\theta$ is meestal niet in gesloten vorm te berekenen omdat θ van hoge dimensies kan zijn. Daarom wordt er gebruik gemaakt van een algoritme uit de klasse van *Markov Chain Monte Carlo* methoden om (afhankelijke) trekkingen uit de a posteriori verdeling te genereren[4]. Deze methoden maken een ergodische Markovketen met als invariante verdeling de a posteriori verdeling. Ergodisch wil zeggen dat de Markovketen aperiodiek en irreducibel is. Een van de methoden om dit te doen is het Metropolis-Hastings algoritme, zie hoofdstuk 3.

2.1. A priori verdeling kiezen

Er zijn verschillende manieren om de a priori verdeling te kiezen. Eén van de manieren is om de verdeling een reflectie te laten zijn van de hoeveelheid informatie die bekend is over de parameters. Mocht er niets bekend zijn over de parameters, dan is de uniforme verdeling het meest voor de hand liggend. Het is toegestaan om een oneigenlijke a priori verdeling te kiezen zolang deze tot een eigenlijke a posteriori verdeling

leidt. Een oneigenlijke verdeling is een verdeling waarvan de integraal over de parameter ruimte tot oneindig integreert, zoals bijvoorbeeld $Unif(\mathbb{R})$ [4]. Soms kan het ook handig zijn om een geconjugeerde a priori verdeling te kiezen. Dit is een a priori verdeling uit een bepaalde familie, bijvoorbeeld de Beta familie, zodat de a posteriori verdeling ook in diezelfde familie zit, eventueel met andere parameters.

Wanneer er veel data beschikbaar is, leiden zowel de klassieke methoden als de Bayesiaanse methoden veelal tot vergelijkbare schatters. Het verschil ontstaat bij weinig data. Bij weinig data hangt de uitkomst van de Bayesiaanse statistiek sterk af van de gekozen a priori verdeling. Wanneer er veel data beschikbaar is, wordt de a priori verdeling minder belangrijk. Dit wordt geïllustreerd in het volgende voorbeeld. In dit voorbeeld wordt een a priori verdeling gekozen die slecht bij de data past. Hierdoor trekt de a priori verdeling de uitkomst als het ware weg van de data. Het volgende schema wordt gebruikt.

$$\begin{aligned} X \mid \Theta &\sim Bin(n,\theta) \\ \Theta \quad &\sim Beta(a,b) \end{aligned}$$

Voor dit schema is het mogelijk om de a posteriori verdeling expliciet uit te rekenen. Dit komt omdat de a priori verdeling uit een geconjugeerde familie komt. De kansdichtheid van $X \mid \Theta$ is gegeven door

$$p_{X|\Theta}(x \mid \theta) = \binom{n}{x} \theta^x (1-\theta)^{n-x}$$

De kansdichtheid van Θ is gegeven door

$$p_\Theta(\theta) = \frac{\theta^{a-1}(1-\theta)^{b-1}}{B(a,b)}$$

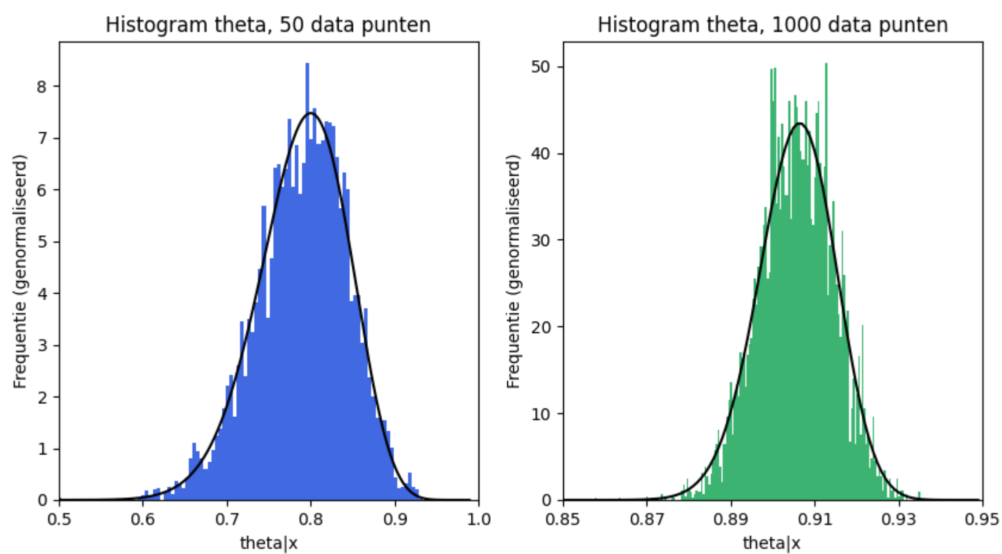
met $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$, de Beta functie, waarbij $\Gamma(t) = \int_0^\infty x^{t-1}e^{-x}\mathrm{d}x$, de Gamma functie.
Voor de a posteriori verdeling geldt dan het volgende

$$p_{\Theta|X}(\theta \mid x) \propto \theta^x(1-\theta)^{n-x}\theta^{a-1}(1-\theta)^{b-1} = \theta^{x+a-1}(1-\theta)^{n-x+b-1}$$

Deze vorm heeft de Beta verdeling ook, dus de a posteriori verdeling is weer een Beta verdeling met parameters $x+a$ en $n-x+b$ [6]. In dit voorbeeld wordt $a=2$ en $b=5$ genomen. De data wordt gegenereerd uit een binomiale verdeling met $p=0.9$.

Om een schatting van de verdeling van de verdeling van $\Theta \mid X$ te maken, wordt het Metropolis-Hastings algoritme met de Random Walk kernel, zie hoofdstuk 3 en 4, gebruikt. Dit algoritme doet trekkingen uit de a posteriori verdeling. Van deze trekkingen wordt een histogram gemaakt, zodat zichtbaar wordt hoe vaak elke waarde getrokken is. Het histogram toont dan de benadering van de a posteriori verdeling. Voor de implementatie in Python, zie bijlange B.

Zoals te zien is in figuur 2.1 wordt de benadering beter naar mate meer data beschikbaar is. Bij 50 data punten is te zien dat de histogrammen niet rond de eigenlijke waarde van de data gecentreerd zijn. Wanneer er 1000 data punten gebruikt worden, ligt de modus van de verdeling bijna bij 0.9. Dit is dus een betere benadering.



Figuur 2.1: In de figuur zijn de histogrammen te zien die de verdeling van $\Theta \mid X$ benaderen. De zwarte lijn is de exacte a posteriori verdeling. De modus van de verdeling ligt dichter bij de eigenlijke waarde (0.9) met meer data punten.

3 Metropolis-Hastings algoritme

Het Metropolis-Hastings algoritme valt binnen de klasse van Markov Chain Monte Carlo methoden. Dit is een klasse van methoden die een ergodische Markovketen creëert om een kansverdeling te benaderen. Een ergodische Markovketen wil zeggen dat de Markovketen aperiodiek is (het is mogelijk om in hetzelfde punt te blijven voor een bepaalde tijdsduur) en irreducibel (het is mogelijk om elk punt in de toestandsruimte te bereiken vanaf elk willekeurig punt). De kansverdeling waaruit getrokken wordt, moet bekend zijn op een constante na. Dit is precies het geval bij Bayesiaanse statistiek, de a posteriori verdeling is namelijk bekend op $\int p_{X|\Theta}(x \mid \theta)p(\theta)\mathrm{d}\theta$ na.

Voor het Metropolis-Hastings algoritme moet een irreduceerbare Markovketen als input gekozen worden. Deze Markovketen kan gegenereerd worden aan de hand van een Markovkernel. Een kernel is een afbeelding $Q : X \times \mathscr{B} \to [0,1]$ met $(X, \mathscr{A})$ en $(Y, \mathscr{B})$ meetbare ruimten waarvoor geldt:

$$\begin{aligned} &\forall x \in X \quad Q(x,Y) \text{ is een kansverdeling op } (Y,\mathscr{B}) \\ &\forall B \in \mathscr{B} \quad Q(x,B) \text{ is } \mathscr{A}\text{-meetbaar} \end{aligned}$$

Verder geldt, aangenomen dat Q dichtheid q heeft

$$Q(z,A) = \int_A q(z,x)\mathrm{d}x$$

Hieronder volgen de stappen van het Metropolis-Hastings algoritme. $\pi(\cdot)$ is de kansverdeling die benadert wordt. Het Metropolis-Hasting algoritme begint met een initiële waarde x_0.

Stap 1 Gegeven x_n, genereer $y \sim Q(x_n, \cdot)$

Stap 2 Bereken $\alpha(x_n, y) = \min\left\{1, \frac{\pi(y)q(y,x_n)}{\pi(x_n)q(x_n,y)}\right\}$

Stap 3 Accepteer y met kans $\alpha(x_n, y)$: $x_{n+1} = \begin{cases} y & \text{met kans } \alpha(x_n,y) \\ x_n & \text{met kans } 1-\alpha(x_n,y) \end{cases}$

Het idee van het Metropolis-Hastings algoritme is dat er een voorstel wordt gedaan aan de hand van de gekozen kernel. Vervolgens wordt deze waarde geaccepteerd of verworpen op basis van hoe waarschijnlijk deze waarde is ten opzichte van de invariante verdeling, in het geval van Bayesiaanse statistiek de a posteriori verdeling. Wanneer de voorgestelde waarde dichter bij de modus van de kansverdeling ligt, wordt deze nieuwe waarde geaccepteerd. Mocht de voorgestelde waarde juist meer richting de staart van de verdeling liggen dan de huidige waarde, dan wordt deze voorgestelde waarde met een bepaalde kans geaccepteerd. Er zijn oneindig veel keuzes voor een voorstelkernel Q. In hoofdstuk 4 worden voorbeelden van verschillende kernels gegeven.

De vraag is of de ergodische Markovketen die het Metropolis-Hastings algoritme genereert altijd de gewenste invariante verdeling heeft. Aangezien de Markovketen ergodisch is, heeft deze Markovketen een invariante verdeling. Het is genoeg om te laten zien dat de gewenste verdeling samen met de overgangskernel omkeerbaar is. Dan volgt namelijk dat de gewenste verdeling de invariante verdeling is van de Markovketen die bij

de overgangskernel hoort. Omkeerbaar betekent $\pi(x)p(x,y) = \pi(y)p(y,x)$, waarbij$\pi(x)$ en $\pi(y)$ de invariante kansen van x en y zijn en $p(x,y)$ de overgangskans van x naar y. Allereerst berekenen we de overgangskansen van het Metropolis-Hastings algoritme. Daartoe definiëren we eerst

$$
\begin{aligned}
Y \mid X_n = x &\sim Q(x,\cdot) \\
Z \mid Y = y, X_n = x &\sim Ber(\alpha(x,y))
\end{aligned}
$$

met

$$\alpha(x,y) = \min\left\{1, \frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right\}$$

Dan is het mogelijk om X_{n+1} als volgt te schrijven

$$X_{n+1} = ZY + (1-Z)X_n = \psi(Z,Y)$$

$X_{n+1} = x_n$ als $Z = 0$, oftewel als het voorstel verworpen wordt en $X_{n+1} = y$ als $Z = 1$ dan wordt het voorstel geaccepteerd.

Nu kunnen we de kans uitrekenen om van x in een verzameling A terecht te komen.

$$
\begin{aligned}
p(x,A) &= P(X_{n+1} \in A \mid X_n = x) = P(\psi(Z,Y) \in A \mid X_n = x) \\
&= \mathbb{E}_{(Z,Y)}[\mathbb{1}_{\{\psi(Z,Y)\in A\}} \mid X_n = x] \\
&= \int \mathbb{E}_Z[\mathbb{1}_{\{\psi(Z,y)\in A\}} \mid X_n = x] q(x,y) dy \\
&= \int \mathbb{1}_{\{\psi(0,y)\in A\}}(1-\alpha(x,y))q(x,y)dy + \int \mathbb{1}_{\{\psi(1,y)\in A\}}\alpha(x,y)q(x,y)dy \\
&= \mathbb{1}_{x\in A}\int q(x,y)(1-\alpha(x,y)\mathrm{d}y + \mathbb{1}_{y\in A}\int q(x,y)\alpha(x,y)\mathrm{d}y \\
&= \mathbb{1}_{x\in A}\int q(x,y)(1-\alpha(x,y)\mathrm{d}y + \int_A q(x,y)\alpha(x,y)\mathrm{d}y
\end{aligned}
$$

Als $x = y$ dan is het triviaal dat geldt $\pi(x)p(x,y) = \pi(y)p(y,x)$. Als $x \neq y$ dan vinden we $p(x,y)$ door $A = \{y\}$ en $y \neq x$ te nemen

$$p(x,y) = q(x,y)\alpha(x,y)$$

Dan vinden we

$$
\begin{aligned}
\pi(x)p(x,y) &= \pi(x)\cdot q(x,y)\cdot \min\left\{1, \frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right\} \\
&= \min\left\{\pi(x)\cdot q(x,y), \pi(y)\cdot q(y,x)\right\} \\
&= \min\left\{\pi(y)\cdot q(y,x), \pi(x)\cdot q(x,y)\right\} \\
&= \pi(y)\cdot q(y,x)\cdot \min\left\{1, \frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right\} \\
&= \pi(y)p(y,x)
\end{aligned}
$$

Waarbij $\pi(\cdot)$ de gewenste verdeling is. We kunnen hieruit concluderen dat het Metropolis-Hastings algoritme een ergodische Markovketen genegeerd met de gewenste invariante verdeling.

4
Verschillende kernels

De keuze voor de voorstelkernel is op een paar regulariteitsvoorwaarden na helemaal vrij. Daardoor zijn er oneindig veel kernels mogelijk. Sommige kernels zijn echter handiger dan andere. De gekozen kernel heeft namelijk invloed op hoe snel het Metropolis-Hastings algoritme convergeert naar de gewenste kansverdeling. Dit komt doordat sommige kernels sterk afhankelijke voorstellen geven, waardoor een iteratie minder informatie bevat. Verder heeft natuurlijk ook de tijd die het duurt om uit de kernel te trekken invloed op de snelheid van het algoritme. In dit hoofdstuk worden drie mogelijke kernels op een rij gezet.

4.1. Random Walk

Een simpele keuze voor de voorstelkernel is de Random Walk. Deze kernel loopt als het ware blind door de toestandsruimte heen. Het voorstel wordt gegeven door $y = x + \sigma Z$ waarbij Z getrokken wordt uit een symmetrische verdeling, σ is de tuning parameter. In dit verslag wordt een standaard normale verdeling gebruikt. De q waar dit voorstel uit komt, is symmetrisch, hierdoor valt de q weg bij het berekenen van $\alpha(x_n, y)$ [4].

4.2. Barker

Het Barker voorstel is vrij recent ontwikkeld. De voorstelkernel Q wordt in dit geval gegeven door

$$Q(x, \mathrm{d}y) = 2\frac{\mu_\sigma(y-x)}{1+e^{-\nabla \log \pi(x)(y-x)}}\mathrm{d}y$$

waarbij $\mu_\sigma(x) = \sigma^{-d}\mu\left(\frac{x}{\sigma}\right)$ met μ een symmetrische verdeling. In dit project wordt de verdeling $N(0,\sigma^2)$ gebruikt, waarbij σ de tuning parameter is. Verder is d gegeven door het aantal dimensies van θ. Alle operaties worden puntsgewijs toegepast. Op de volgende manier wordt hieruit een voorstel gegenereerd. Allereerst wordt er een waarde $z \sim \mu_\sigma(\cdot)$ getrokken. Dan wordt $p(x,z) = \frac{1}{1+e^{-z\nabla \log \pi(x)}}$ berekent. Nu definiëren we

$$b(x,z) = \begin{cases} 1 & \text{met kans } p(x,z) \\ -1 & \text{met kans } 1-p(x,z) \end{cases}$$

Uiteindelijk is de voorgestelde waarde $y = x + b(x,z)\cdot z$ [2].

Het is niet voor de hand liggend dat deze trekkingen daadwerkelijk uit Q gedaan worden. Daarom volgt hier een bewijs om te laten zien dat de trekkingen uit de gewenste kernel komen. Gedurende het hele bewijs wordt x vast verondersteld. We definiëren eerst

$$\begin{aligned} Z &\sim \mu_\sigma \\ U \mid Z &\sim Ber(p(x,z)) \\ Y &= x + (2U-1)Z \end{aligned}$$

Zij ψ een begrensde meetbare functie. We laten zien dat

$$\mathbb{E}[\psi(Y)] = \int \psi(y)Q(x, \mathrm{d}y)$$

We definiëren $g(u,z) = x + (2u-1)z$. Dan geldt

$$\begin{aligned}
\mathbb{E}[\psi(Y)] &= \mathbb{E}[\psi(g(U,Z))] \\
&= \mathbb{E}_Z[\mathbb{E}_{U|Z}[\psi(g(U,Z))]] \\
&= \mathbb{E}_Z[(1-p(x,Z))\psi(g(0,Z)) + p(x,Z)\psi(g(1,Z))] \\
&= \mathbb{E}_Z[(1-p(x,Z))\psi(x-Z) + p(x,Z)\psi(x+Z)] \\
&= \int (1-p(x,z))\psi(x-z)\mu_\sigma(z)\mathrm{d}z + \int p(x,z)\psi(x+z)\mu_\sigma(z)\mathrm{d}z
\end{aligned}$$

Voor de eerste term geldt $y = x - z$, omdat $u = 0$. Voor de tweede term geldt $y = x + z$, omdat $u = 1$. Verder geldt $\mu_\sigma(z) = \mu_\sigma(-z)$, omdat μ_σ een symmetrische verdeling is. Dan vinden we

$$\begin{aligned}
\mathbb{E}[\psi(Y)] &= \int (1-p(x,x-y))\psi(y)\mu_\sigma(x-y)\mathrm{d}y + \int p(x,y-x)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int (p(x,y-x) - p(x,x-y)+1)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int \left(\frac{1}{1+e^{-(y-x)\nabla\log\pi(x)}} - \frac{1}{1+e^{-(x-y)\nabla\log\pi(x)}} + 1\right)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int \left(\frac{1}{1+e^{-(y-x)\nabla\log\pi(x)}} - \frac{1}{1+e^{(y-x)\nabla\log\pi(x)}} + 1\right)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int \left(\frac{1}{1+e^{-(y-x)\nabla\log\pi(x)}} - \frac{e^{-(y-x)\nabla\log\pi(x)}}{1+e^{-(y-x)\nabla\log\pi(x)}} + \frac{1+e^{-(y-x)\nabla\log\pi(x)}}{1+e^{-(y-x)\nabla\log\pi(x)}}\right)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int \left(\frac{2}{1+e^{-(y-x)\nabla\log\pi(x)}}\right)\psi(y)\mu_\sigma(y-x)\mathrm{d}y \\
&= \int \psi(y)Q(x,\mathrm{d}y)
\end{aligned}$$

4.2.1. Oorsprong Barker voorstel

Continue tijd Markovprocessen kunnen gegenereerd worden met behulp van de infinitesimaal generator. Zij $(X_t)_{t\geq 0}$ een continue tijd Markov proces. Dan is de bijbehorende infinitesimaal generator gedefinieerd als

$$\mathscr{L}f(x) = \lim_{t\downarrow 0} \frac{\mathbb{E}[f(X_t) - f(x)]}{t}$$

voor alle f waarvoor dit limiet bestaat.

In het artikel van Livingstone en Zanella [2], wordt aan de hand van een generator een klasse voorstellen opgesteld. Het voorstel dat hierboven is beschreven is hier een voorbeeld van. Hieronder wordt beschreven hoe deze klasse ontstaat. Allereerst wordt een generator gedefinieerd voor een continu Markovproces op $\mathbb{R}^d$

$$\mathscr{L}f(x) = \int \big[f(y) - f(x)\big] g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) Q(x,\mathrm{d}y)$$

Hierbij is f een functie $f : \mathbb{R}^d \to \mathbb{R}$, $\pi(x)$ een kansverdeling, $Q(x,\mathrm{d}y) := q(x,y)\mathrm{d}y$ is een overgangskernel en voor de *balancing* functie $g : (0,\infty) \to (0,\infty)$ moet $g(t) = t \cdot g(1/t)$ gelden. Deze generator definieert een inhomogeen Poisson proces. Wanneer voor de huidige staat $X_t = x$ geldt, dan wordt de tijd tot de volgende staat bepaald door dit Poisson proces met intensiteit

$$Z(x) := \int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) Q(x,\mathrm{d}y)$$

Dit betekend dat de wachttijd $Exp(Z(x))$ verdeeld is. De volgende staat, y, gezien vanuit de huidige staat, x, wordt getrokken uit de kernel

$$Q^{(g)}(x,\mathrm{d}y) := Z(x)^{-1} g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) Q(x,\mathrm{d}y)$$

De geadjungeerde operator van $\mathscr{L}$ is gegeven door

$$\mathscr{L}^\star h(x) = \int h(y) g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right) q(y,x)\,\mathrm{d}y - h(x)Z(x)$$

Om te laten zien dat $\mathscr{L}^\star$ daadwerkelijk de geadjungeerde operator van $\mathscr{L}$ is, moeten we laten zien dat $<\mathscr{L}f, h> = <f, \mathscr{L}^\star h>$, waarbij $<f,g> = \int f(x)g(x)\,\mathrm{d}x$.

$$\begin{aligned}
<\mathscr{L}f,h> &= \int \mathscr{L}f(x)h(x)\,\mathrm{d}x \\
&= \int\int [f(y)-f(x)]g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)Q(x,\mathrm{d}y)h(x)\,\mathrm{d}x \\
&= \int h(x)\int [f(y)-f(x)]g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)Q(x,\mathrm{d}y)\,\mathrm{d}x \\
&= \int h(x)\left\{\int f(y)g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y - f(x)\int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\right\}\mathrm{d}x \\
&= \int h(x)\int f(y)g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\,\mathrm{d}x - \int h(x)f(x)Z(x)\,\mathrm{d}x
\end{aligned}$$

$$\begin{aligned}
<f,\mathscr{L}^\star h(x)> &= \int f(x)\mathscr{L}^\star h(x)\,\mathrm{d}x \\
&= \int f(x)\left\{\int h(y)g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y - h(x)Z(x)\right\}\mathrm{d}x \\
&= \int f(x)\int h(y)g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y\,\mathrm{d}x - \int f(x)h(x)Z(x)\,\mathrm{d}x
\end{aligned}$$

We moeten dus laten zien dat $\int h(x)\int f(y)g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\,\mathrm{d}x = \int f(x)\int h(y)g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y\,\mathrm{d}x$.

$$\begin{aligned}
&\int h(x)\int f(y)g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\,\mathrm{d}x \\
\overset{\text{part. int. naar } y}{=}& \int h(x)\left\{[f(y)\int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y] - \int \frac{\mathrm{d}}{\mathrm{d}y}f(y)\int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\,\mathrm{d}y\right\}\mathrm{d}x \\
=& \int h(x)\left\{f(y)Z(x) - \int \frac{\mathrm{d}}{\mathrm{d}y}f(y)Z(x)\,\mathrm{d}y\right\}\mathrm{d}x \\
=& \int h(x)\left\{f(y)Z(x) - f(y)Z(x)\right\}\mathrm{d}x = 0
\end{aligned}$$

$$\begin{aligned}
&\int f(x)\int h(y)g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y\,\mathrm{d}x \\
\overset{\text{part. int. naar } y}{=}& \int f(x)\left\{[h(y)\int g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y] - \int \frac{\mathrm{d}}{\mathrm{d}y}h(y)\int \left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y\,\mathrm{d}y\right\}\mathrm{d}x = 0
\end{aligned}$$

Want

$$\begin{aligned}
&\int g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right)q(y,x)\,\mathrm{d}y \\
&= \int \frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(y,x)\,\mathrm{d}y = \int \frac{\pi(x)}{\pi(y)}g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y \\
&\overset{\text{part. int. naar } y}{=} \left[\frac{\pi(x)}{\pi(y)}\int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\right] - \int \frac{\mathrm{d}}{\mathrm{d}y}\frac{\pi(x)}{\pi(y)}\int g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right)q(x,y)\,\mathrm{d}y\,\mathrm{d}y \\
&= \frac{\pi(x)}{\pi(y)}Z(x) - \int \frac{\mathrm{d}}{\mathrm{d}y}\frac{\pi(x)}{\pi(y)}Z(x)\,\mathrm{d}y = \frac{\pi(x)}{\pi(y)}Z(x) - \frac{\pi(x)}{\pi(y)}Z(x) = 0
\end{aligned}$$

We hebben nu laten zien dat $\mathscr{L}^\star$ inderdaad de geadjungeerde van $\mathscr{L}$ is. We laten nu zien waarom de geadjungeerde een goed beginpunt vormt voor het maken van een overgangskernel.

$$\begin{aligned}\mathscr{L}^\star\pi(x) &= \int \pi(y) g\left(\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)}\right) q(y,x)\,\mathrm{d}y - \pi(x)Z(x)\\ &= \int \pi(y)\frac{\pi(x)q(x,y)}{\pi(y)q(y,x)} g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) q(y,x)\,\mathrm{d}y - \pi(x)Z(x)\\ &= \int \pi(x) g\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) q(x,y)\,\mathrm{d}y - \pi(x)Z(x)\\ &= \pi(x)Z(x) - \pi(x)Z(x) = 0\end{aligned}$$

Dit impliceert dat π invariant is, zie [2]. Het Barker voorstel ontstaat door $g(t) = \frac{t}{1+t}$ te nemen. Dit zorgt ervoor dat $Z(x) = \frac{1}{2}$.

4.3. Langevin

Een andere mogelijkheid is het Langevin voorstel. Hierbij wordt het voorstel gedaan door $y = x + \frac{1}{2}\sigma\nabla\log\pi(x) + \sqrt{\sigma}Z$ met $Z \sim N(0,1)$ en σ de tuning parameter [4]. In dit geval is Q niet symmetrisch, omdat de voorstellen door de gradiënt als het ware naar het zwaartepunt van de verdeling π worden gestuwd. De q die bij het Langevin voorstel hoort is

$$q(x,y) = \phi\left(\frac{1}{\sqrt{\sigma}}\left(y - x - \frac{1}{2}\sigma\nabla\log\pi(x)\right)\right)$$

met $\phi(x)$ de dichtheid van de standaard normale verdeling, geëvalueerd in x. Dit voorstel is afgeleid van een discrete benadering van Langevin diffusie [3].

Het Langevin voorstel kan ook gemaakt worden uit de algemene formulering van het Barker voorstel. Dit kan door de eerste order Taylor expansie van $\log\pi$ in g te nemen. Dit leidt tot

$$\mathscr{L}f(x) = \int [f(y) - f(x)]\, g\left(e^{\nabla\log\pi(x)(y-x)}\right)\mu_\sigma(y-x)\,\mathrm{d}y$$

Als dan $g(t) = \sqrt{t}$ en voor μ_σ de normale verdeling gekozen worden, ontstaat het Langevin voorstel. Door de notatie lijkt het een ander voorstel. Hieronder wordt aangetoond dat de afleiding via het Barker voorstel en de afleiding via de benadering van Langevin diffusie dezelfde kernel oplevert.

$$\begin{aligned}q^m(x,y) &\propto e^{\frac{1}{2}\nabla\log\pi(x)(y-x)}\mu_\sigma(y-x)\\ &\propto e^{\frac{1}{2\sigma^2}y^2 + \frac{yx}{\sigma^2} + \frac{1}{2}(y-x)\nabla\log\pi(x)}\\ &\propto e^{\frac{1}{2\sigma^2}y^2 + y\left(\frac{x}{\sigma^2} + \frac{1}{2}\nabla\log\pi(x)\right)}\\ &= e^{\frac{1}{2\sigma^2}y^2 + \frac{y}{\sigma^2}\left(x + \frac{1}{2\sigma^2}\nabla\log\pi(x)\right)}\end{aligned}$$

Als $y \sim N(\mu, \sigma^2)$, dan geldt $f_Y(y) \propto e^{-\frac{1}{2\sigma^2}(y-x)^2} \propto e^{-\frac{1}{2\sigma^2}y^2 + \frac{y}{\sigma^2}x}$. Aangezien $q^m(x,y) \propto e^{\frac{1}{2\sigma^2}y^2 + \frac{y}{\sigma^2}\left(x + \frac{1}{2\sigma^2}\nabla\log\pi(x)\right)}$ kunnen we concluderen dat $y \mid x \sim N(x + \frac{1}{2}\sigma^2\nabla\log\pi(x), \sigma^2)$.

4.4. Bereik parameter komt niet overeen met bereik kernel

In sommige problemen kan het voorkomen dat bepaalde waarden van Θ niet passen binnen het probleem. Dit is bijvoorbeeld het geval bij het probleem beschreven in paragraaf 2.1. In dat probleem moet τ positief zijn aangezien het de standaard afwijking van een normale verdeling betreft. Het is niet mogelijk om de negatieve voorstellen van de kernels niet mee te nemen. Dit zorgt namelijk voor een trekking uit een afgekapte kernel wat een verschuiving van de benadering tot gevolg heeft. Een mogelijkheid om dit probleem op te lossen, is het herparametriseren van het probleem. Het Metropolis-Hastings algoritme wordt dan toegepast op $\tilde{\theta}$, waarna de a posteriori verdeling en eventueel de gradiënt in termen van $e^{\tilde{\theta}}$ worden geschreven. Deze aanpak wordt in alle implementaties gebruikt.

Een andere manier om dit probleem op te lossen is door een nieuwe kernel te definiëren. In plaats van $y \mid x$ te trekken, wordt $\log y \mid x$ uit de kernel getrokken. Dan wordt gedefinieerd dat $\log y \mid x \sim N(\log x, \sigma^2)$. Dit geeft $x \mid y \sim e^Z$ met $Z \sim N(\log x, \sigma^2)$. Dus $y \mid x \sim xe^{\sigma} U$ met $U \sim N(0,1)$

4.5. Tuning parameters

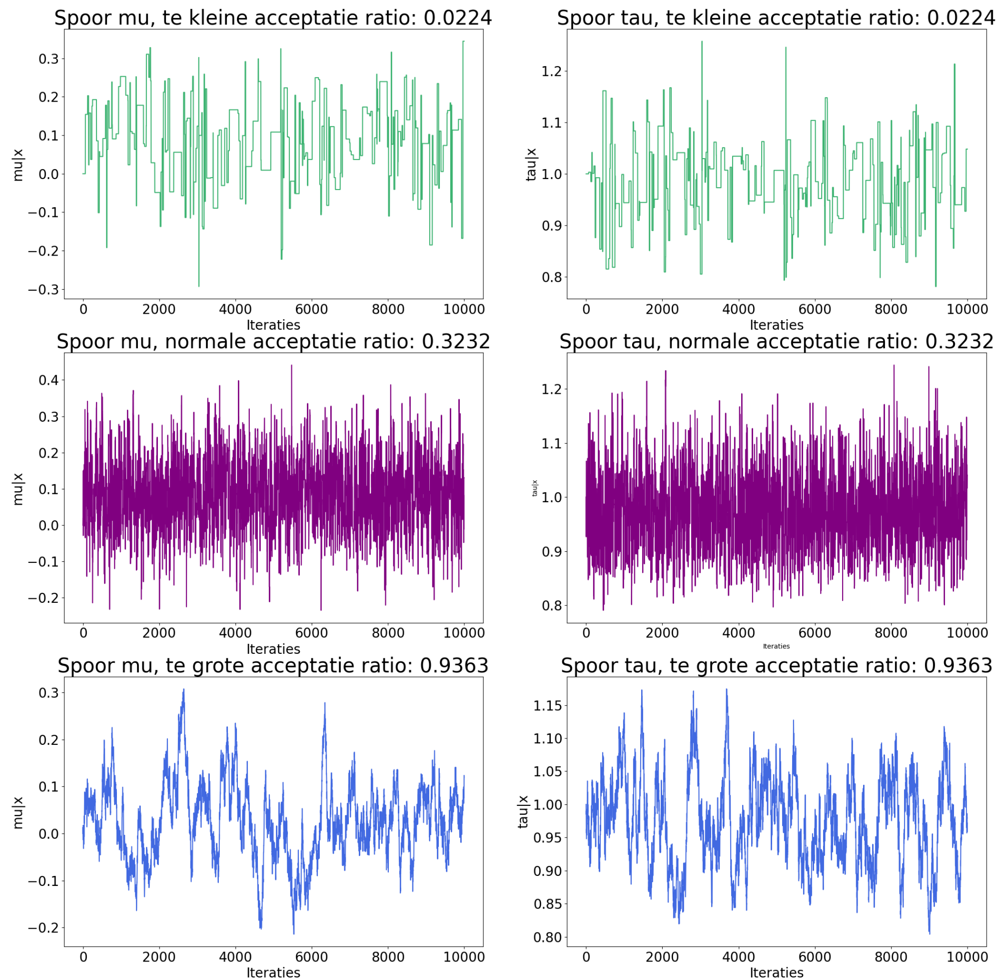
In de kernels komt een tuning parameter voor. Deze parameter heeft invloed op de grootte van het verschil tussen de huidige waarde en de voorgestelde waarde. Deze grootte heeft weer invloed op de acceptatie ratio. Dit is de verhouding tussen het aantal geaccepteerde voorstellen en het totaal aantal voorstellen dat is gedaan. Er wordt over het algemeen geadviseerd om de acceptatie ratio tussen de 0.25 en 0.5 te laten liggen. Een te lage acceptatie kans betekent dat het algoritme vrij extreme waarden voorstelt. Hierdoor duurt het lang voordat de invariante verdeling goed zichtbaar wordt. Ook een te hoge acceptatie kans is niet wenselijk, omdat het algoritme dan waarden voorstelt die dicht bij de huidige waarde liggen. Het gevolg is dat het algoritme veel tijd nodig heeft om de hele verdeling in beeld te brengen.

Hieronder wordt een voorbeeld gegeven van wat er gebeurt als de tuning niet goed is. Daar wordt het volgende probleem voor gebruikt.

$$\begin{aligned} \Theta &= (M, T) \\ X_i \mid \Theta &\sim N(\mu, \tau) \\ M &\sim N(0,1) \\ T &\sim Exp(1) \end{aligned}$$

Waarbij $X = (X_1, \dots, X_n)$. Er worden 100 data punten uit een standaard normale verdeling gebruikt. Om dit probleem te benaderen, wordt de Random Walk kernel gebruikt. Als initiële waarde zijn $\mu = 0$ en $\tau = 1$ genomen. De stapgrootte die voor een te kleine ratio zorgt, is in dit geval 0.8. Dit leidt tot een acceptatie ratio van 0.0234. Voor de normale acceptatie ratio wordt een stapgrootte van 0.15 gebruikt. Dit geeft een acceptatie ratio van 0.331. Dit leidt tot een acceptatie ratio van 0.0234. Voor de te grote acceptatie ratio wordt een stapgrootte van 0.01 gebruikt. Dit leidt tot een acceptatie ratio van 0.938. Voor de implementatie in Python, zie bijlage C.1

In figuur 4.1 zijn de sporen voor de verschillende acceptatie ratio's te zien. Een spoor zijn alle waarden van θ die uit het Metropolis-Hastings algoritme komen, achter elkaar geplot. In het spoor van μ en τ gegenereerd met een te kleine acceptatie ratio is te zien dat het spoor geregeld op een waarde blijft hangen. Er wordt geen nieuwe waarde geaccepteerd, dus wordt de oude waarde herhaald. Wanneer de acceptatie ratio in de goede regio zit, is te zien dat het spoor gecentreerd zit rond de daadwerkelijke data. Ook zijn er geen extreme uitschieters. Verder is er te zien dat het spoor dat gemaakt is met een te grote acceptatie ratio allerlei uitschieters heeft. Er worden dus te veel slechte waarden geaccepteerd.



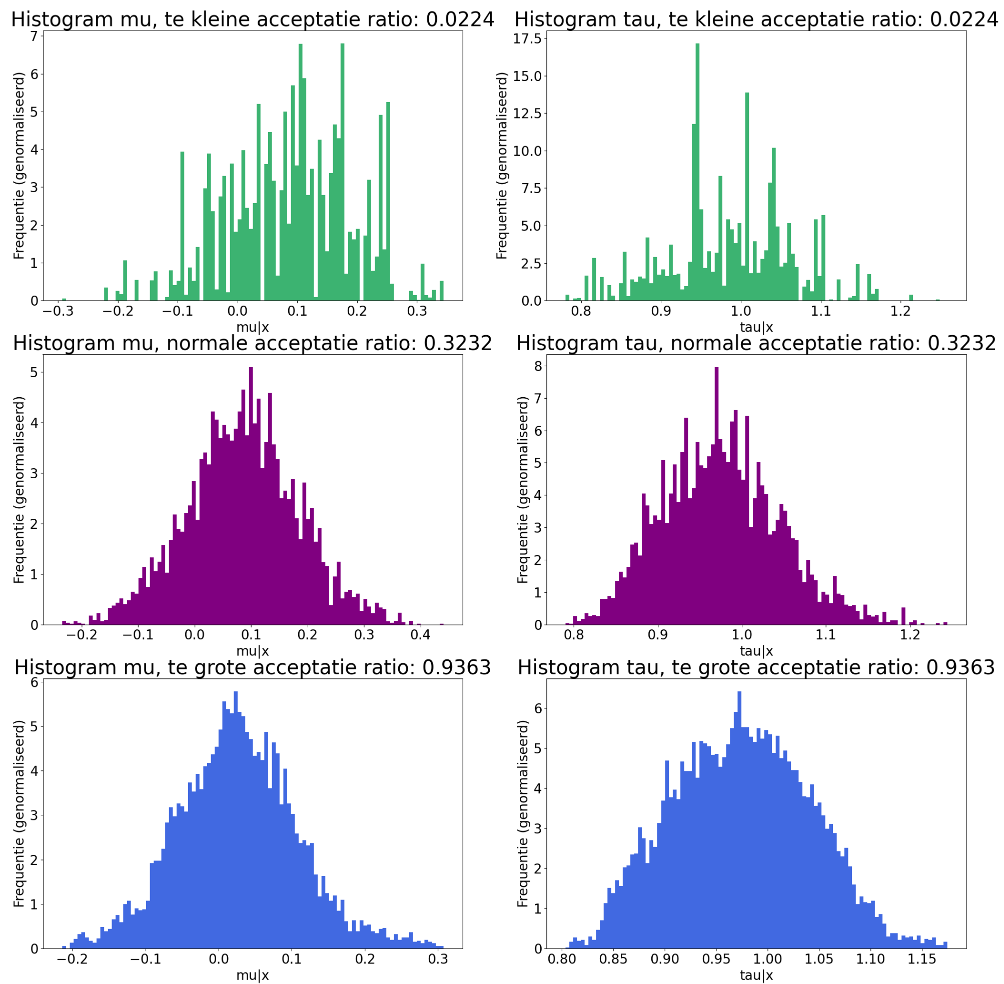
Figuur 4.1: De sporen van hetzelfde probleem met verschillende acceptatie ratio's. De te kleine acceptatie ratio is 0.0234, de normale acceptatie ratio is 0.331 en de te grote acceptatie ratio is 0.938, gemaakt met de Random Walk kernel

Figuur 4.2 laat zien wat er gebeurt met de histogrammen van de parameters met verschillende acceptatie ratio's. Wanneer de acceptatie ratio te klein is, vallen er gaten in de histogrammen, hierdoor is het lastig om te zien welke verdeling een parameter gegeven de data heeft. Met een te grote acceptatie ratio worden de histogrammen eigenlijk te breed. Er zijn namelijk te veel waarden die niet dicht bij de modus liggen geaccepteerd. In figuur 4.3 is de contourplot van dit probleem te zien. Het contourplot laat zien voor welke waarden van μ en τ de simultane verdeling gelijk is aan een constante. Dit betekent

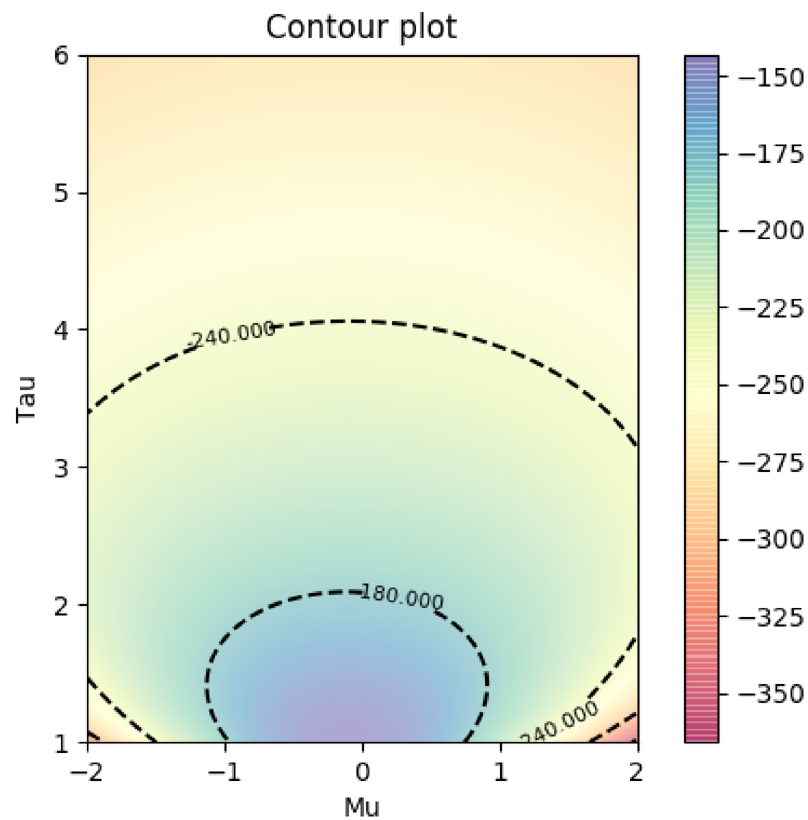
$$\{\mu, \tau : p_M(\mu) p_T(\tau) p_{X|(M,T)} = c\}$$

4.6. Aantal iteraties tot invariante verdeling

De drie bovengenoemde kernels hebben verschillende aantallen iteraties nodig om de invariante verdeling te bereiken. Dit is goed te zien wanneer de initiële waarden ver van de werkelijke waarde gekozen worden. Het duurt daardoor langer om de invariante verdeling te bereiken. Om dit te illustreren wordt hetzelfde probleem gebruikt als in paragraaf 4.5. In de implementatie van het Langevin en Barker voorstel wordt *automatic*



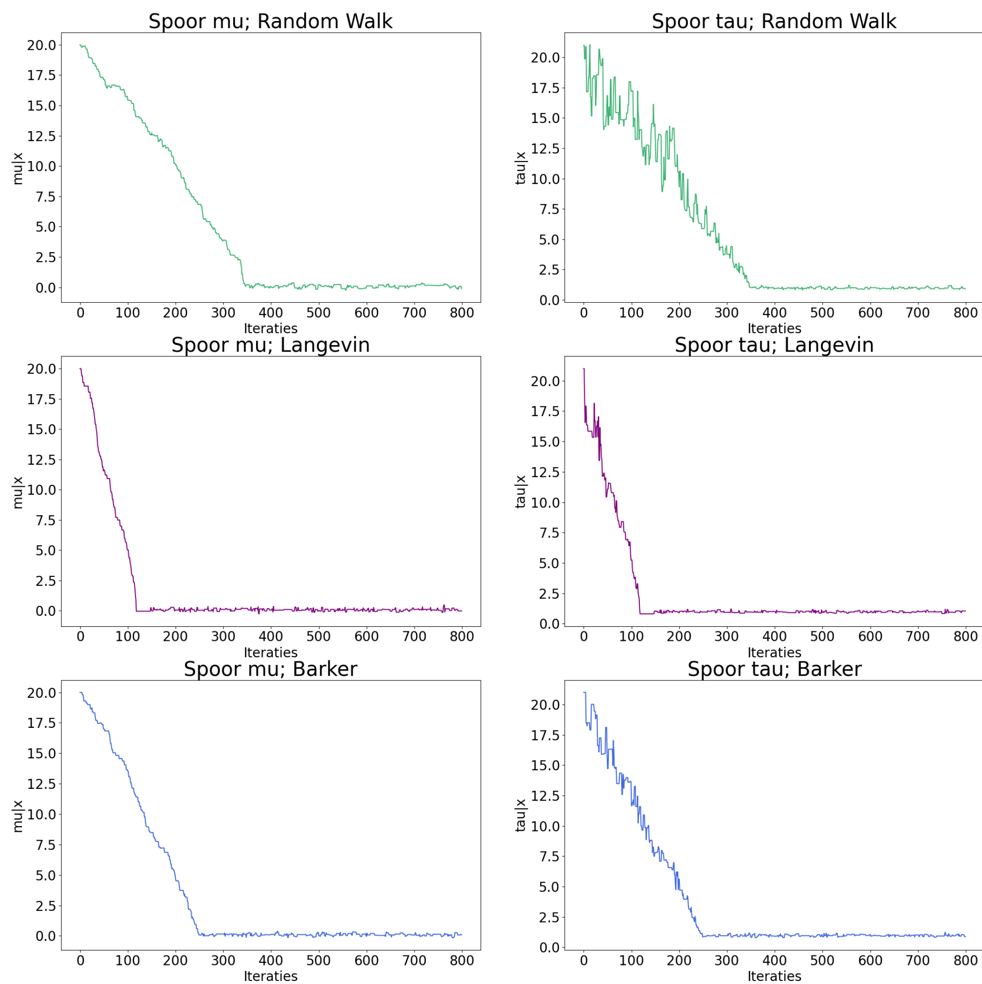
Figuur 4.2: De histogrammen van hetzelfde probleem met verschillende acceptatie ratio's. De te kleine acceptatie ratio is 0.0234, de normale acceptatie ratio is 0.331 en de te grote acceptatie ratio is 0.938, gemaakt met de Random Walk kernel



Figuur 4.3: Contourplot van de a posteriori verdeling op een multiplicative constante na. De a posteriori verdeling hangt af van de data $(X_1, \ldots X_n)$.

differentiation gebruikt om de gradiënt te berekenen. Voor de implementatie in Python, zie bijlage C.2. Als initiële waarde voor μ is 20 gekozen en voor τ is 21 gekozen. Deze waarden liggen ver af van de modus van de verdeling van μ en τ. Voor het Random Walk voorstel is 0.18 voor de tuning parameter gekozen. Dit leidt tot een acceptatie ratio van 3,8. De stapgrootte van het Langevin voorstel is op 0.025 gezet wat leidt tot een acceptatie ratio van 3,387. Voor het Barker voorstel is een stapgrootte van 0.25 gebruikt, wat leidt tot een acceptatie ratio van 3,488.

In figuur 4.4 is te zien dat het Random Walk voorstel het langzaamste convergeert. Het Langevin voorstel convergeert het snelst. Het Barker voorstel ligt ertussen in, maar dichterbij het Langevin voorstel.



Figuur 4.4: Convergentiesnelheden voor de verschillende kernels.

5

Effective sample size

Er zijn, zoals eerder gezegd, oneindig veel keuzes voor voorstelkernels. Om iets te kunnen zeggen over hoe effectief de verschillende voorstelkernels zijn, kan de effective sample size gebruikt worden.

Het Metropolis-Hastings algoritme produceert voor elke dimensie van Θ van keten van lengte N, het aantal trekkingen dat gedaan wordt. Deze ketens vormen een benadering van π. Voor de effective sample size is het belangrijk dat er een *burn-in* van deze trekkingen afgehaald wordt. Dit zijn meestal de eerste 1000 iteraties. Deze eerste trekkingen zijn namelijk sterk afhankelijk van de gekozen beginwaarde. De lengte van de keten die overblijft is n. De effective sample size wordt berekend per dimensie van Θ.

De effective sample size wordt berekend door $n_{eff} = \frac{\frac{1}{n}\sum_{i=1}^{n}\text{var}(X_i)}{\text{var}(\overline{X}_n)}$. Het wordt aangenomen dat $\text{var}(X_i) = \text{var}(X_j)$ als $i \neq j$. Vervolgens kan $\text{var}(\overline{X}_n)$ herschreven worden als

$$
\begin{aligned}
\text{var}(\overline{X}_n) &= \text{var}\left(\frac{X_1 + \ldots + X_n}{n}\right) \\
&= \frac{1}{n^2}\left(\sum_{i=1}^{n}\text{var}(X_i) + 2\sum_{i\neq j}\text{cov}(X_i, X_j)\right) \\
&= \frac{1}{n^2}\left(\sum_{i=1}^{n}\text{var}(X_i) + 2\sum_{i\neq j}\sqrt{\text{var}(X_i)\text{var}(X_j)}\rho(X_i, X_j)\right) \\
&= \frac{1}{n^2}\left(\sum_{i=1}^{n}\text{var}(X_i) + 2\sum_{i\neq j}\text{var}(X_i)\rho(X_i, X_j)\right) \\
&= \frac{1}{n^2}\left(\left(1 + 2\sum_{i\neq j}\rho(X_i, X_j)\right)\sum_{i=1}^{n}\text{var}(X_i)\right)
\end{aligned}
$$

Dus vinden we

$$n_{eff} = \frac{n}{1 + 2\sum_{i\neq j}\rho(X_i, X_j)}$$

Aangezien het niet mogelijk is om $\rho(X_i, X_j)$ te berekenen, moet n_{eff} benaderd worden. Dit wordt hier gedaan aan de hand van de definities gegeven in artikel [5]. Voor het vergelijken wordt effective sample size door de *CPU time* en de *system time* die het duurt om het Metropolis-Hastings algoritme te runnen gedeeld. Hierdoor is een tijdrovend programma niet beter dan een snel programma met een vergelijkbare effective sample size.

Wanneer een benadering van een verdeling of schatter wordt gemaakt, zijn er meer trekkingen nodig wanneer de trekkingen afhankelijk zijn. De effective sample size is een benadering van het aantal onafhankelijke trekkingen dat nodig zou zijn om een even goed beeld te krijgen van de verdeling als deze door de afhankelijke trekkingen benaderd wordt. Dus als een kernel een hoge effective sample size heeft, heeft het Metropolis-Hastings algoritme minder trekkingen nodig om de verdeling in kaart te brengen dan een kernel met een lage

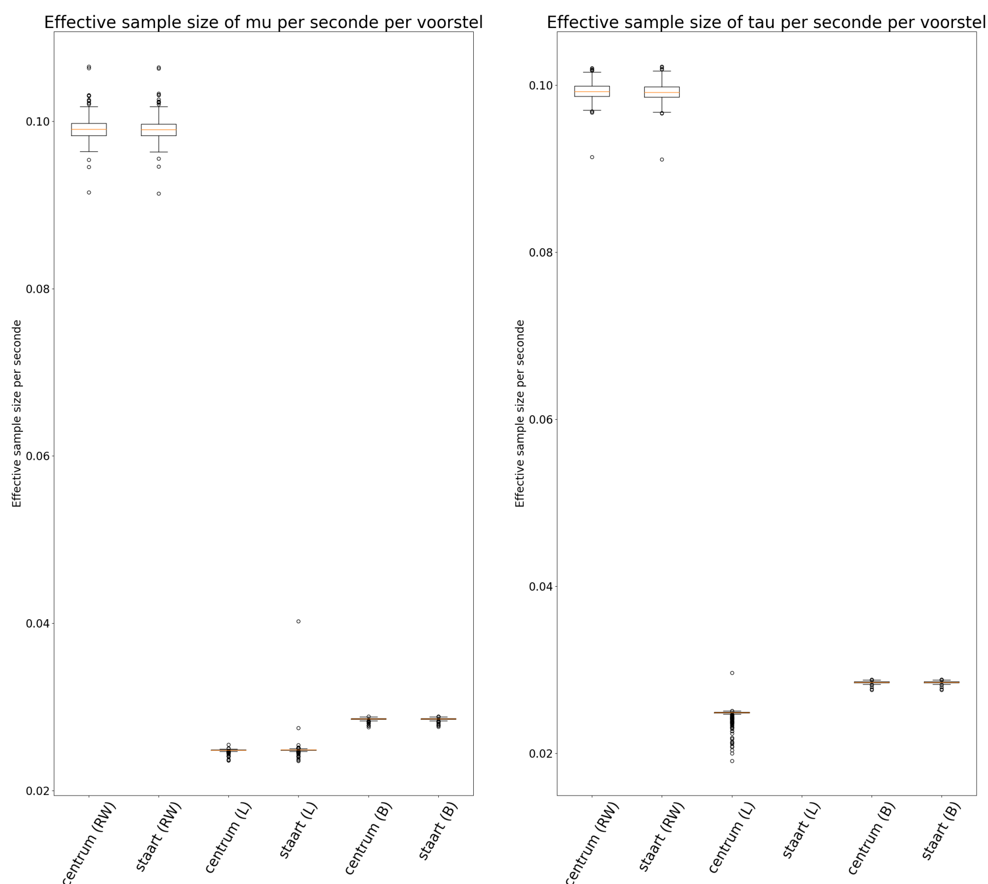
effective sample size.

Om de verschillende kernels met elkaar te vergelijken wordt gebruik gemaakt van een Monte Carlo schema. Hierin wordt het Metropolis-Hastings algoritme een aantal keren uitgevoerd. Voor elke keer dat het uitgevoerd wordt, wordt ook de effective sample size berekend. Het aantal iteraties van het Metropolis-Hastings algoritme is zo gekozen dat het algoritme de tijd heeft gehad om te convergeren, namelijk 10^4 iteraties.

Dit schema wordt op het probleem uit paragraaf 4.5 uitgevoerd. Dat is het volgende probleem.

$$\begin{aligned} \Theta &= (M, T) \\ X_i \mid \Theta &\sim N(\mu, \tau) \\ M &\sim N(0,1) \\ T &\sim Exp(1) \end{aligned}$$

Waarbij $X = (X_1, \ldots, X_n)$. Er worden 100 datapunten uit een standaard normale verdeling gebruikt. De effective sample size wordt berekent voor initiële waarden die in de modus van de verdeling liggen, namelijk $\mu = 0$ en $\tau = 1$ en voor initiële waarden die meer in de staart van de verdeling liggen, namelijk $\mu = 5$ en $\tau = 6$. De gemiddelde acceptatie ratio van het Random Walk voorstel komt met een stapgrootte van 0.15 uit op 0.33. Het Langevin voorstel heeft een acceptatie ratio van 0.36 met stapgrootte 0.022. Als laatste heeft het Barker voorstel met stapgrootte 0.25 een acceptatie ratio van 0.32.



Figuur 5.1: Voor verschillende initiële waarden en verschillende kernels de bijbehorende effective sample size. Het beginpunt van *center* komt overeen met $\mu = 0$ en $\tau = 1$, *tail* begint bij $\mu = 5$ en $\tau = 6$.

Zoals te zien is in figuur 5.1 is de effective sample size van het Random Walk voorstel het grootst. Van alle beschreven kernels in hoofdstuk 4 is dit de methode die de kortste rekentijd nodig heeft. Dit zou kunnen bijdragen aan een hogere effective sample size. Verder lijkt het beginpunt weinig effect te hebben op de effective sample size. Het Langevin voorstel heeft de kleinste effective sample size.

Voor het berekenen van de gradiënt in het Langevin en Barker voorstel, wordt gebruik gemaakt van automatic differentiation, omdat de gradiënt niet in gesloten vorm te berekenen is. Om te illustreren hoeveel extra tijd het kost om de gradiënt te moeten berekenen met automatic differentation wordt de a posteriori verdeling 100 keer geëvalueerd en wordt de gradiënt 100 keer berekend met automatic differentation in Python. Deze 100 *runs* zijn getimed en daar is het gemiddelde van genomen. Dit is 1000 keer uitgevoerd en van deze 1000 runs is het minimum genomen. Dit geeft een idee van de daadwerkelijke tijd die de computer nodig heeft om de stap één keer uit te voeren. Het resultaat is dat het evalueren van de a posteriori verdeling 0.040 ms duurt en de automatic differentiation 0.115 ms. Als de gradiënt niet in gesloten vorm uitgerekend kan worden, duurt de uitvoering van een programma met 10^4 iteraties 0.75 seconden langer.

6 Bayesiaans schatten van een bivariate kansdichtheid

Wanneer het aantal dimensies in het probleem groeit, moet de stapgrootte in de kernel kleiner worden om de acceptatie ratio in de goede range te houden. Bij het Random Walk voorstel schaalt het aantal stappen met $\Theta(d^{-1})$. Ter herinnering: $f(n) = \Theta(g(n))$ als er constanten c_1 en c_2 bestaan zodanig dat $0 \leq f(n) \leq c_1 g(n)$ en $0 \leq c_2 g(n) \leq f(n)$ voor alle $n \geq n_0$. Voor het Barker voorstel schaalt het aantal stappen met $\Theta(d^{-1/3})$, waarbij d het aantal dimensies is [2]. Om dit verschil te illustreren, worden de twee voorstellen toegepast op een vereenvoudigde versie van het probleem dat in [1] wordt beschreven. In het originele probleem wordt er gebruik gemaakt van een *inspection time*, deze wordt achterwege gelaten.

In het vereenvoudigde model wordt op een bepaalde tijd X een waarde Y geobserveerd met $0 \leq X \leq 1$ en $0 \leq Y \leq 2$. Er worden n van dit soort combinaties geobserveerd. Uiteindelijk zijn we geïnteresseerd in de simultane verdeling van de set $\mathfrak{D}_n = \{(X_i, Y_i), i = 1, \ldots, n\}$.

De data in dit probleem wordt als volgt gegenereerd

$$(X, Y) \sim \begin{cases} (U, V) & \text{met kans } 0.3 \\ (1 - U, V) & \text{met kans } 0.7 \end{cases}$$

Waarbij (U, V) de volgende verdeling heeft $f(u, v) = \frac{3}{8}(u^2 + v)\mathbb{1}_{[0,1]\times[0,2]}(u, v)$. De rechthoek $[0, 1] \times [0, 2]$ wordt opgedeeld in een aantal vakjes, a_n. Deze vakjes vormen een partitie, C. N_i is een vector met het aantal data punten in alles vakjes.

Om de simultane verdeling te vinden, definiëren we de aannemelijkheidsfunctie f op basis van $\mathfrak{D}_n$ als $\prod p_i^{N_i}$ met $p_i = S(\bar{U}z\sqrt{\tau})$. Waarbij $S(x_1, \ldots, x_p) = \frac{e^{(x_1, \ldots, x_p)}}{\sum_{i=1}^{p} e_i^x}$, de softmax functie. $\bar{U}$ wordt als volgt gevonden:
De partitie C kan omgezet worden in een graaf. Hierbij representeert elk vakje een knooppunt. Knooppunten zijn verbonden als de vakjes horizontale of verticale buren zijn. Van deze graaf is het mogelijk om een $a_n \times a_n$ Laplace matrix te maken. Deze is gedefinieerd door

$$L_{i,j} = \begin{cases} \text{aantal buren knooppunt } i & \text{als } i = j \\ -1 & \text{als } i \neq j \text{ en knooppunt } i \text{ en } j \text{ buren zijn} \\ 0 & \text{anders} \end{cases}$$

Neem vervolgens $\Upsilon = L + a_n^{-2} I$. Dan is $\bar{U} = U^{-1}$ met U de Cholesky ontbinding van Υ.

De a priori verdelingen kiezen we als volgt

$$T \sim Exp(1)$$
$$Z \sim N_{a_n}(0, I_{a_n})$$

Waarbij τ de hoeveelheid *smoothing* tussen verschillende vakjes in de partitie bepaald. En $S(Z)$ representeert de waarde van de a posteriori verdeling in elk vakje.

In [1] is besloten om dit probleem met een combinatie van het Gibbs algoritme en het Metropolis-Hastings algoritme op te lossen. Dit wordt overgenomen. Dit levert het volgende algoritme op. Het algoritme begint met een initiële waarde voor τ en z.

Stap 1 Stel τ^0 voor aan de hand van $q(\cdot,\tau)$
Stap 2 Accepteer of verwerp τ^0 aan de hand van de Metropolis-Hastings acceptatie voorwaarde
Stap 3 Stel z^0 voor aan de hand van het *Random Walk*, Langevin of Barker voorstel
Stap 4 Accepteer of verwerp z^0 aan de hand van de Metropolis-Hastings acceptatie voorwaarde

Hierbij wordt uit $q(\cdot,\tau)$ getrokken door $\tau^0 = \tau e^{\eta U}$ met $U \sim N(0,1)$ en η de tuning parameter. Wanneer het Random Walk voorstel gebruikt wordt om z voor te stellen, moet een net iets andere variant gebruikt worden omdat het klassieke Random Walk voorstel problemen kan ondervinden wanneer het aantal dimensies richting oneindig gaat. Het voorstel wordt dan $z^0 = \rho z + \sqrt{1-\rho^2}\,w$ met $w \sim N_{a_n}(0, I_{a_n})$ en $\rho \in [0,1)$ de *tuning* parameter. De reden om het voorstellen en accepteren van τ en z los van elkaar te doen, is dat het schalen zo makkelijker is omdat dit voor τ en z apart gedaan kan worden [6].

Voor zowel het Barker voorstel is de gradiënt naar z van het logaritme van a posteriori verdeling nodig. Voor dit probleem is het mogelijk om de gradiënt exact te berekenen. De snelste manier hiervoor is om de gradiënt in gesloten vorm af te leiden en direct te laten berekenen. Het is mij alleen niet gelukt om dit foutloos uit te rekenen en te implementeren. Daarom heb ik besloten om de gradiënt te laten berekenen met automatic differentiation. Door het hoge aantal dimensies van z kost automatic differentiation veel meer tijd dan als het gelukt was om de gradiënt exact uit te rekenen. Het gevolg is dat het niet mogelijk is het aantal dimensies ver op te voeren. Eén run duurt namelijk lang en om de tuning goed te krijgen zijn meerdere runs nodig.

Voor elke implementatie worden 10^4 data punten gebruikt die gegenereerd worden aan de hand van bovenstaand schema. Verder worden er 10^6 iteraties van het algoritme uitgevoerd. Elke kernel is op twee verschillende gird verfijningen toegepast. Hierdoor is het mogelijk om het schalen van de stapgrootte met het aantal dimensies in kaart te brengen. De eerste verfijning is twee bins in de x richting en vier bins in de y richting. De tweede verfijning is vijf bins in de x richting en tien bins in de y richting. Dit zorgt ervoor dat alle vakjes in het grid vierkant zijn. In figuur 6.1 is de data te zien die genegeerd is aan de hand van bovenstaand schema in de vorm van een hittekaart. Voor de volledige implementatie zie bijlage A en E

Tabel 6.1: Per voorstel en per verfijning de stapgrootte van τ en z met de bijbehorende acceptatie ratio.

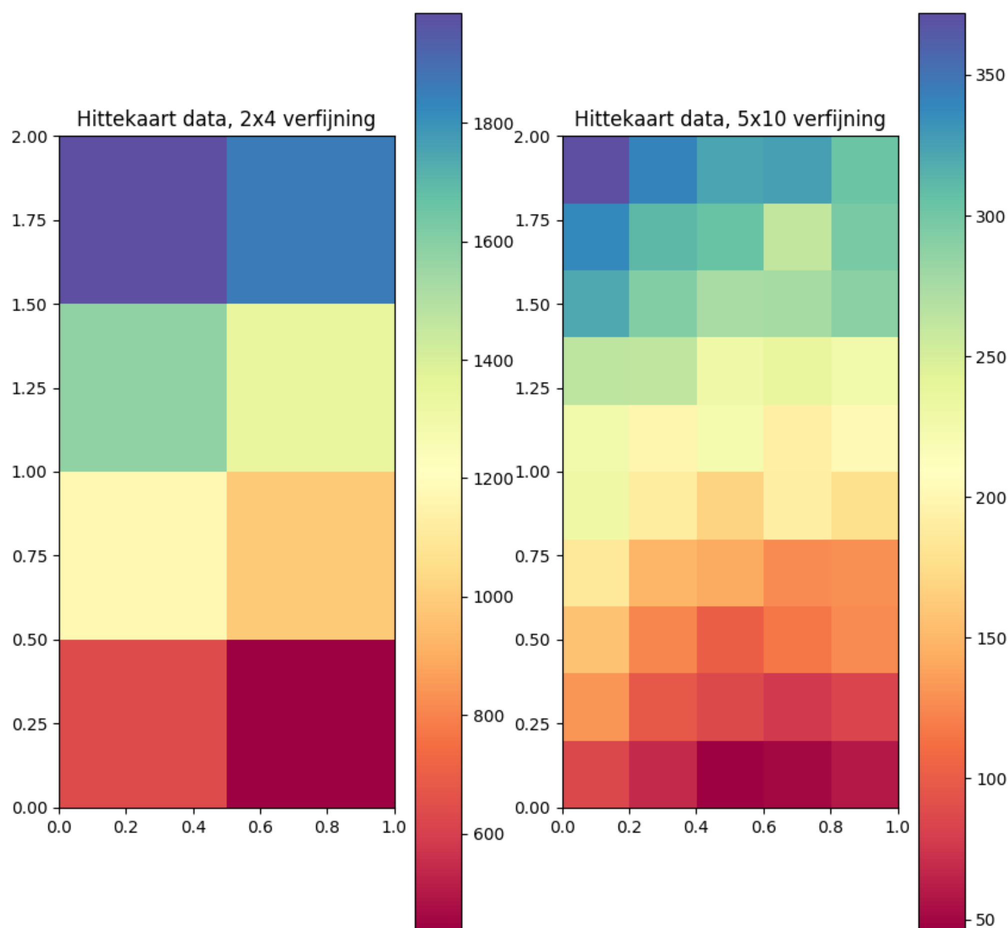
	2×4				5×10			
	Stap τ	Stap z	Acc ratio τ	Acc ratio z	Stap τ	Stap z	Acc ratio τ	Acc ratio z
Random Walk	0.17	0.9999	0.342679	0.334233	0.17	0.99999	0.356301	0.447344
Barker	0.15	0.015	0.379635	0.39002	0.15	0.0055	0.385657	0.469541

In tabel 6.1 is te per verfijning te zien welke stapgroottes van τ en z een bepaalde acceptatie ratio opleveren. Zoals verwacht, is de stapgrootte van z kleiner wanneer het aantal dimensies van z toeneemt. Deze stapgroottes leveren de traceplots in figuren 6.2 en 6.3 op. De plots zien er vrij rommelig uit. Toch is de benadering van de a posteriori verdeling wel goed zoals te zien is in figuur 6.4. Deze plot laat het gemiddelde van de softmax functie toegepast op de benadering van z zien.

Volgens het artikel van Livingstone en Zanella [2] moet gelden dat de stapgrootte van het Random Walk voorstel met $\Theta(d^{-1}$ groeit waarbij d het aantal dimensies is. In dit voorbeeld gaan we van 8 naar 50 dimensies. Dus verwachten we dat de stapgrootte

$$\frac{\frac{1}{8}}{\frac{1}{50}} = \frac{50}{8} = 6,25$$

keer zo groot wordt. In dit voorbeeld is een variant van het Random Walk voorstel gebruikt. Hierdoor is het niet mogelijk om meteen de stapgroottes door elkaar te delen. Om deze stapgroottes toch met de schaling voor het Random Walk voorstel uit paragraaf 4.1 te kunnen vergelijking nemen we voor de stapgrootte van het



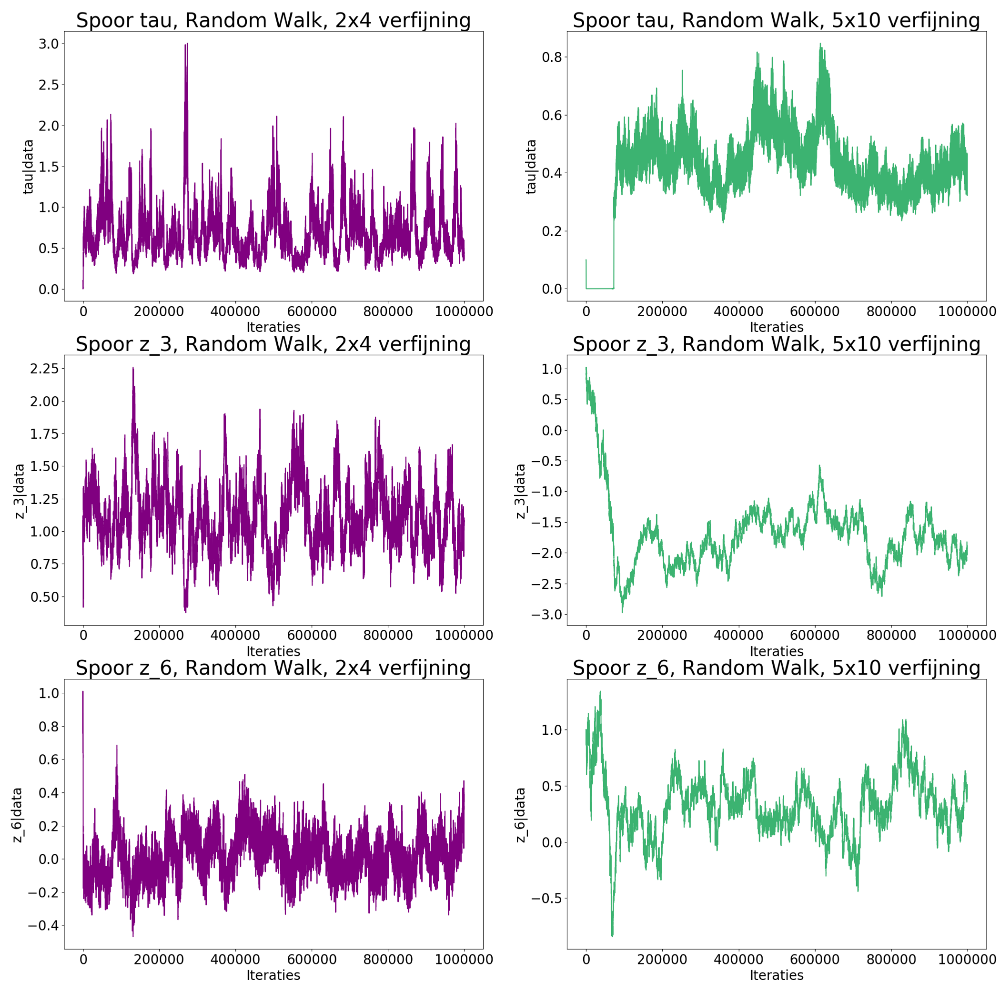
Figuur 6.1: Hittekaart van de 10^4 data punten voor zowel de 2×4 als de 5×10 verfijning.

Random Walk voorstel uit paragraaf 4.1 $\sqrt{1-\rho^2}$. Het verschil tussen de twee voorstellen is daardoor alleen nog dat z vermenigvuldigd wordt met een getal heel dicht bij 1 in plaats van dat z zelf genomen wordt. Ik ga ervan uit dat dit verschil niet veel invloed heeft op de stapgrootte. We vinden dan

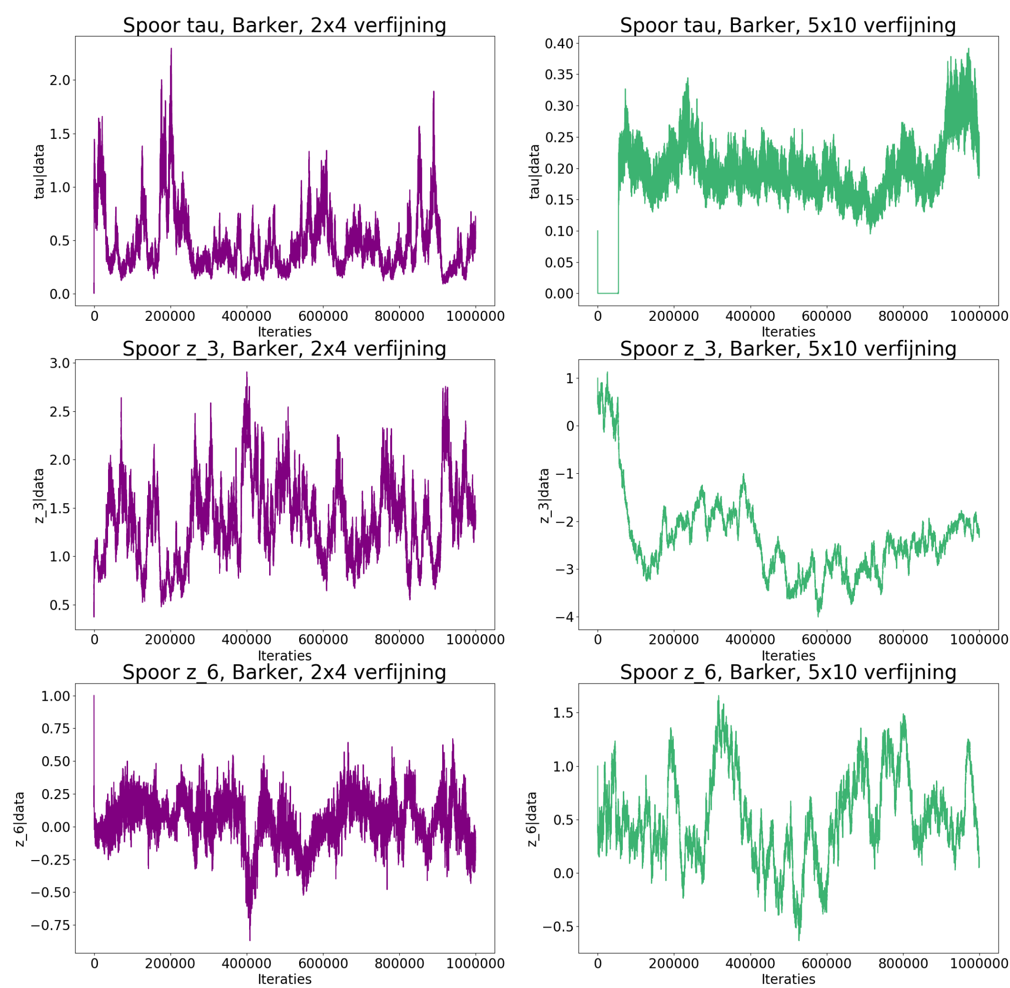
$$\frac{\sqrt{1-(0.9999)^2}}{\sqrt{1-(0.99999)^2}} = \frac{0.01414}{0.004472} = 3,162 \tag{6.1}$$

Door deze berekening lijkt het alsof de stapgrootte in dit geval minder hard groeit dan verwacht. Maar daarbij wordt dan geen rekening gehouden met het verschil in acceptatie ratio's. Zoals te zien in tabel 6.1 is de acceptatie ratio voor het hoger dimensionale geval hoger dan die voor het lager dimensionale geval. Om de acceptatie ratio van het hoger dimensionale geval dichtbij die van het lager dimensionale geval te krijgen moeten de voorstellen in elke iteratie dichter bij de huidige waarde komen te liggen. Dit betekent dat $\sqrt{1-\rho^2}$ kleiner moet worden. Hierdoor wordt de noemer in vergelijking 6.1 kleiner, dus wordt de verhouding groter. Wat betekent groei van de stapgrootte in werkelijkheid dichter bij 6,25 ligt dan nu het geval lijkt.

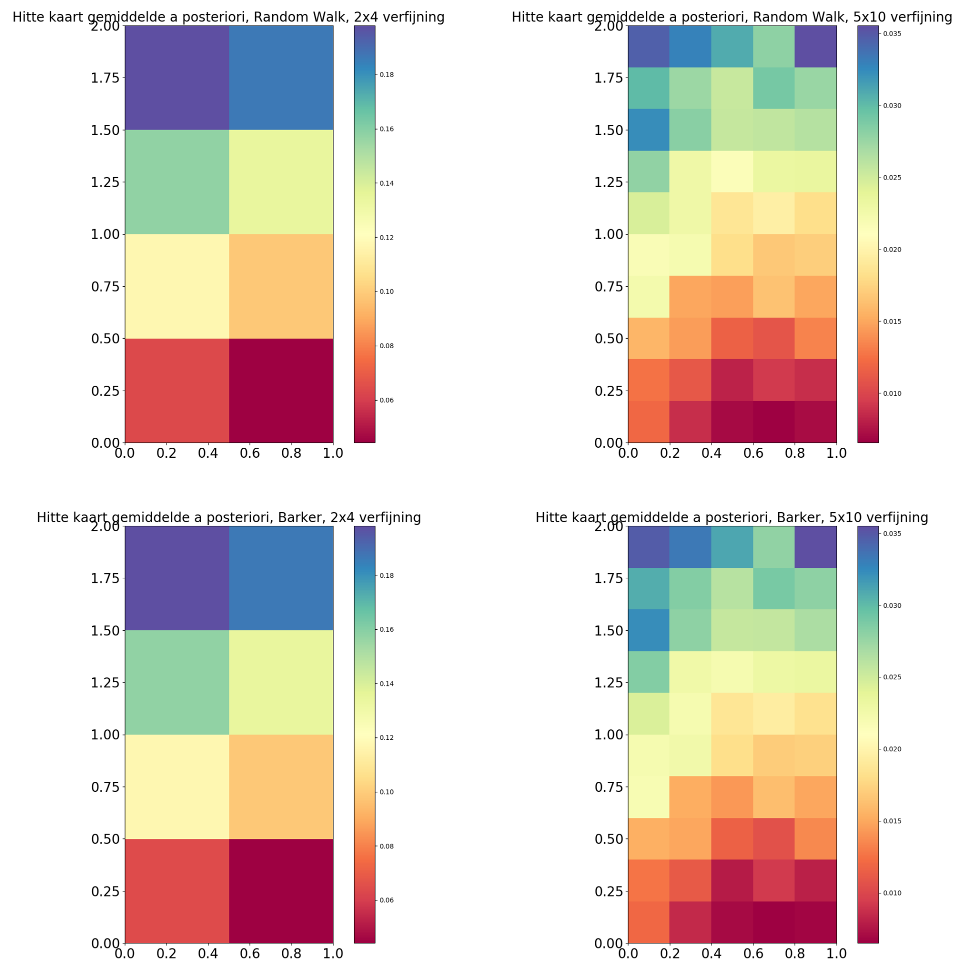
Voor het Barker voorstel kunnen we een vergelijkbare berekening doen. Alleen geldt voor deze voorstellen



Figuur 6.2: De traceplots van τ en van z_3 met middelpunt (0.2; 1.25) voor de lagere dimensionale verfijning en middelpunt (0.5; 1.875) voor de hogere dimensionale verfijning. z_6 heeft als middelpunt in de lager dimensionale verfijning (0.8; 0.75) en (0.5; 1.625) voor de hoger dimensionale verfijning. Deze traceplots zijn gemaakt met behulp van de Random Walk kernel.



Figuur 6.3: De traceplots van τ en van z_3 met middelpunt (0.2; 1.25) voor de lagere dimensionale verfijning en middelpunt (0.5; 1.875) voor de hogere dimensionale verfijning. z_6 heeft als middelpunt in de lager dimensionale verfijning (0.8; 0.75) en (0.5; 1.625) voor de hoger dimensionale verfijning. Deze traceplots zijn gemaakt aan de hand van het Barker voorstel.



Figuur 6.4: De hittekaart van het gemiddelde van de a posteriori verdeling

dat de stapgrootte met $\Theta(d^{-1/3})$ groeit. Hierdoor verwachten we dat de stapgrootte met een factor

$$\frac{8^{-1/3}}{50^{-1/3}} = 1,84$$

groeit. Voor het Barker voorstel vinden we dat de stapgrootte groeit met

$$\frac{0.015}{0.0055} = 2,7273$$

Dit lijkt op het eerste oog groter dan de verwachting. Maar ook hier geldt dat de acceptatie ratio van het hoger dimensionale geval hoger is dan die van het lager dimensionale geval. Om dit dichter bij elkaar te brengen zou de stapgrootte van z in het hoger dimensionale geval groter moeten worden, zodat de afstand tussen de voorstellen groter is. Hiertoe wordt de noemer groter, waardoor de verhouding kleiner wordt en dus meer richting 1,84 gaat.

Hieruit kunnen we concluderen dat het erop lijkt dat de schaling die in het artikel van Livingstone en Zanlla gegeven wordt, inderdaad klopt. Toch betekent dit niet dat het Langevin of Barker voorstel dan altijd beter zijn in een hoger dimensionaal probleem. Dat komt omdat zowel het Langevin als het Barker voorstel de gradiënt van de target functie nodig hebben. Wanneer deze exact te berekenen is, zijn het Langevin en Barker voorstel handiger dan het Random Walk voorstel. Maar wanneer deze gradiënt berekend moet worden met bijvoorbeeld automatic differentation dan kan het zijn dan de betere schaling van de stapgrootte wanneer het aantal dimensies toeneemt, niet meer opweegt tegen de extra tijd die het kost om de gradiënt te berekenen.

7 Conclusie en Discussie

In dit project is gekeken naar verschillende eigenschappen van het Barker voorstel. Dit voorstel is toegepast binnen het Metropolis-Hastings algoritme. Ook is dit voorstel vergeleken met het Random Walk voorstel en het Langevin voorstel. Er is gekeken naar het verschil in effective sample size, wat gedefinieerd wordt door

$$n_{eff} = \frac{n}{1 + 2\sum_{i \neq j} \rho(X_i, X_j)}$$

Verder zijn het Random Walk voorstel en het Barker voorstel toegepast op een hoger dimensionaal probleem om in kaart te brengen hoe de stapgrootte binnen de kernels schaalt met het toenemen van het aantal dimensies van de parameter.

De conclusie is dat het Random Walk voorstel de hoogste effective sample size heeft. Het Langevin en Barker voorstel hebben een vergelijkbare en lagere effective sameple size. Dit is een voordeel van het Random Walk voorstel. Maar de strapgrootte van het Random Walk voorstel schaalt met $\Theta(d^{-1})$, met d het aantal dimensies van de parameter. Terwijl de stapgrooote van het Barker voorstel met $\Theta(d^{-1/3})$ schaalt. Dit betekent dat in principe het Barker voorstel handiger is wanneer het aantal dimensies toeneemt. Dit geldt vooral wanneer de gradiënt van de target verdeling exact uit te rekenen is. Wanneer de gradiënt niet exact te berekenen is, zou het kunnen dat de tijdwinst van een grotere stapgrootte niet op weegt tegen de extra tijd die het kost om te gradiënt te berekenen met bijvoorbeeld automatic differentiation.

Voor vervolgonderzoek zou het mogelijk zijn om te kijken waar de overgang ligt tussen het voordeel van de voordelige schaling van de stapgrootte van het Barker voorstel tegen het nadeel van de gradiënt te moeten berekenen met automatic differentiation. Ook kan er gekeken worden naar de andere eigenschappen die genoemd worden in het artikel van Livingstone en Zanella [2], zoals *robustness to tuning*. Verder zou het interessant kunnen zijn om te kijken naar de eigenschappen van het Barker voorstel wanneer een andere symmetrische verdeling dan de normale verdeling wordt gekozen voor μ_σ.

A

Code bij alle hoofdstukken

In deze bijlage staat de code die als basis dient voor alle andere code. In veel bestanden wordt de inhoud van deze bestanden geïmporteerd.

A.1. MH.py

```
import autograd.numpy as np
from operator import setitem
from autograd import grad

from Posterior import *

def Prw(x, stepsize):
    return x + stepsize*np.random.normal(0, 1, len(x))

def Qrw(x, y, stepsize):
    return 0

def Pl(x, stepsize, grad_x):
    return x + 1/2*stepsize*grad_x + np.sqrt(stepsize)*np.random.normal(0, 1, len(x))

def Ql(x, y, stepsize, grad_x):
    Arg = 1/np.sqrt(stepsize)*(y-x-1/2*stepsize*grad_x)
    return lognorm(np.array([0, 1]), Arg)

def p(x, z, grad_x):
    denominator = 1 + np.exp(-z*grad_x)
    return 1/denominator

def b(x, z, grad_x):
    return np.where(p(x, z, grad_x)>np.random.uniform(0, 1, len(x)), 1, -1)

def Pb(x, z, grad_x):
    return (x + b(x, z, grad_x)*z)#.astype(float)

def Qb(x, y, grad_x):
    return np.sum(np.log(1 + np.exp(grad_x*(x-y))))

def Ppos(x, stepsize):
    return x*np.exp(stepsize*np.random.normal(0, 1, len(x)))

def Qpos(x):
    return np.sum(np.log(x))



#Functie met als input het aantal iteraties, het logaritme van de te benaderen functie,
#de voorstel kernel, de stapgrootte, een initiele waarde, het aantal parameters dat
                                             positief moet zijn (deze moeten aan het
                                             eind van de array staan) en de data
```

```
#Output: array of arrays van Theta en de acceptatie rate.
def MH(n, target, method, stepsize, theta, n_pos, data):
    Theta = np.zeros([len(theta), n])
    theta_prop = np.zeros([len(theta)])
    theta_prop_pos = np.zeros([len(theta)])
    theta_pos = np.concatenate((theta[:n_pos], np.exp(theta[n_pos:])))

    Theta[:, 0] = theta_pos
    ll = target(theta_pos, data)
    acc = 0

    if method == "Ql" or method == "Qb":
        grad_theta_prop = np.zeros([len(theta)])
        grad_theta = grad(target, 0)(theta_pos, data)
    for i in range(1, n):
        if method == "Qrw":
            setitem(theta_prop, np.array(range(0, len(theta))), Prw(theta, stepsize))
            theta_prop_pos = np.concatenate((theta_prop[:n_pos], np.exp(theta_prop[
                                            n_pos:])))
        if method == "Ql":
            setitem(theta_prop, np.array(range(0, len(theta))), Pl(theta, stepsize,
                                            grad_theta))
            theta_prop_pos = np.concatenate((theta_prop[:n_pos], np.exp(theta_prop[
                                            n_pos:])))
            setitem(grad_theta_prop, np.array(range(0, len(grad_theta))), grad(target,
                                            0)(theta_prop_pos, data))
        if method == "Qb":
            z = np.random.normal(0, stepsize, len(theta))
            setitem(theta_prop, np.array(range(0, len(theta))), Pb(theta, z, grad_theta
                                            ))
            theta_prop_pos = np.concatenate((theta_prop[:n_pos], np.exp(theta_prop[
                                            n_pos:])))
            setitem(grad_theta_prop, np.array(range(0, len(grad_theta))), grad(target,
                                            0)(theta_prop_pos, data))
        ll_prop = target(theta_prop_pos, data)
        if method == "Qrw":
            a = np.log(np.random.uniform())
            if  a < ll_prop - ll:
                setitem(theta, np.array(range(0, len(theta))), theta_prop)
                setitem(theta_pos, np.array(range(0, len(theta))), theta_prop_pos)
                ll = ll_prop
                acc += 1
        elif method == "Ql":
            if np.log(np.random.uniform()) < ll_prop + Ql(theta_prop, theta, stepsize,
                                            grad_theta_prop) - (ll + Ql(
                                            theta, theta_prop, stepsize,
                                            grad_theta)):
                setitem(theta, np.array(range(0, len(theta))), theta_prop)
                setitem(theta_pos, np.array(range(0, len(theta))), theta_prop_pos)
                ll = ll_prop
                setitem(grad_theta, np.array(range(0, len(grad_theta))),
                                                grad_theta_prop)
                acc += 1
        elif method == "Qb":
            if np.log(np.random.uniform()) < ll_prop + Qb(theta, theta_prop, grad_theta
                                            ) - (ll + Qb(theta_prop, theta,
                                             grad_theta_prop)):
                setitem(theta, np.array(range(0, len(theta))), theta_prop)
                setitem(theta_pos, np.array(range(0, len(theta))), theta_prop_pos)
                ll = ll_prop
                setitem(grad_theta, np.array(range(0, len(grad_theta))),
                                                grad_theta_prop)
                acc += 1
        Theta[:, i] = theta_pos
    return Theta, acc/n
```

A.2. Data.py

```
import autograd.numpy as np
```

```
np.random.seed(10)

def normdata(mu, sigma, n):
    return np.random.normal(mu, sigma, n)

def bindata(p, n):
    return np.random.binomial(n, p)

def X_marg_inv(t):
    numerator = (np.sqrt(4*t**2+1)+2*t)**(2/3)-1
    denominator = (np.sqrt(4*t**2+1)+2*t)**(1/3)
    return numerator/denominator

def Y_cond_inv(s, u):
    return np.sqrt(u**4 + 4*u**2*s + 4*s) - u**2

def markmodeldata(n, x_bins, y_bins):
    D = np.zeros([n, 2])
    N = np.zeros([y_bins, x_bins])
    for i in range(0, n):
        r = np.random.uniform()
        u = X_marg_inv(r)
        s = np.random.uniform()
        v = Y_cond_inv(s, u)
        if np.random.uniform() < 0.3:
            D[i, :] = np.array([u, v])
        else:
            D[i, :] = np.array([1-u, v])
        N[int(np.floor(D[i][1]*(y_bins/2))), int(np.floor(D[i][0]*x_bins))] += 1
    return D, N.reshape(1, x_bins*y_bins)[0]

def unifmarkdata(n, x_bins, y_bins):
    D = np.zeros([n, 2])
    N = np.zeros([y_bins, x_bins])
    for i in range(0, n):
        D[i, :] = np.array([np.random.uniform(0, 1), np.random.uniform(0, 2)])
        N[int(np.floor(D[i][1]*(y_bins/2))), int(np.floor(D[i][0]*x_bins))] += 1
    return D, N.reshape(1, x_bins*y_bins)[0]
```

A.3. Posterior.py

```
import autograd.numpy as np
import autograd.scipy as scipy
from autograd import grad

def lognorm(theta, x):
    return np.sum(scipy.stats.norm.logpdf(x, theta[0], theta[1]))

def logexp(theta, x):
    return np.sum(np.log(theta*np.exp(-theta*x)))

def logtarget(theta, x):
    return lognorm(theta, x) + logexp(1, theta[1]) + lognorm(np.array([0, delta]),
                                                    theta[0])

delta = 1

a = 2
b = 5

def logt10(theta, x):
    return np.sum(scipy.stats.beta.logpdf(theta, a+x, b+10-x))

def logt50(theta, x):
    return np.sum(scipy.stats.beta.logpdf(theta, a+x, b+50-x))

def logt100(theta, x):
    return np.sum(scipy.stats.beta.logpdf(theta, a+x, b+100-x))
```

```
def logt1000(theta, x):
    return np.sum(scipy.stats.beta.logpdf(theta, a+x, b+1000-x))

def matrixU(x_bins, y_bins):
    #L is een x_bins*y_bins bij x_bins*y_bins matrix.
    a = np.ones((1, x_bins*y_bins-x_bins))[0]
    b = np.ones((1, x_bins*y_bins-1))[0]
    Adj = np.diag(b, -1) + np.diag(b, 1) + np.diag(a, -x_bins) + np.diag(a, x_bins)
    for i in range(1, x_bins*y_bins):
        if i%x_bins == 0:
            Adj[i, i-1] = 0
            Adj[i-1, i] = 0
    Deg = np.diag((Adj.sum(axis=1, keepdims=True)).reshape(1, x_bins*y_bins)[0], 0)
    L = Deg-Adj
    Upsilon = L + (x_bins*y_bins)**(-2) * np.identity(x_bins*y_bins)
    return np.linalg.inv(np.linalg.cholesky(Upsilon))

def logstdmultinorm(x):
    #return np.sum(scipy.stats.norm.logpdf(x, 0, 1))
    Mean = np.zeros(len(x))
    Cov = np.identity(len(x))
    return np.sum(scipy.stats.multivariate_normal.logpdf(x, Mean, Cov))

def logsoftmax(x):
    return x-np.log(np.sum(np.exp(x)))

def softmax(x):
    return np.exp(x)/np.sum(np.exp(x))

#N is het aantal waarnemingen per cel in een vector van lengte x_bins*y_bins
def logpostmarkmodel(z, tau, N, U):
    A = np.dot(U, z)
    logsoft = logsoftmax(np.dot(U, z)*np.sqrt(tau))
    LogsoftSum = np.sum(N*logsoft)
    LogExp = logexp(1, tau)
    LogStdNormal = logstdmultinorm(z)
    return LogsoftSum + LogExp + LogStdNormal

#Poging tot uitrekenen gradient
def logpostmarkmodel_grad(z, tau, N, U):
    A = U * np.sqrt(tau)
    c = np.sum(np.exp(np.dot(A, z)))
    Arg1 = c*A * np.array([np.exp(np.dot(A, z))]).transpose()
    Arg2 = np.dot(np.array([np.exp(np.dot(A, z))]).transpose(), (A*np.array([np.exp(np.
                                            dot(A, z))]).transpose()).sum(axis=0,
                                            keepdims=True))
    Grad_log_pi = ((Arg1-Arg2)/c**2) * (np.array([1/softmax(np.dot(A, z))]).transpose()
                                            )
    return (Grad_log_pi * (np.array([N]).transpose())).sum(axis=0, keepdims=True)[:, 0]
                                             - z
```

B

Code bij hoofdstuk 2

De onderstaande code produceert drie .txt bestanden om figuur 2.1 te maken.

```
import autograd.numpy as np

from Data import *
from Posterior import *
from MH import *

#Genereer data
data50 = bindata(0.9, 50)
data1000 =  bindata(0.9, 1000)

#Run MH algoritme
Theta50, acc50 = MH(10000, logt50, "Qrw", 0.17, np.array([0.5]), 1, data50)
Theta1000, acc1000 = MH(10000, logt1000, "Qrw", 0.03, np.array([0.5]), 1, data1000)

#Stop Theta in array
Outputarray = np.zeros([2, 10000])
Outputarray[:1, :] = Theta50
Outputarray[1:, :] = Theta1000

#Stop acceptatie ratio in array
Outputarray_acc = np.zeros([2, 1])
Outputarray_acc[:1, :] = acc50
Outputarray_acc[1:, :] = acc1000

#Stop data in array
Outputarray_data = np.zeros([2, 1])
Outputarray_data[:1, :] = data50
Outputarray_data[1:, :] = data1000

#Sla Theta op
output = open("Output_slechte_prior_Theta.txt", "w")
for row in Outputarray:
    np.savetxt(output, row)
output.close()

#Sla acceptatie ratio op
output = open("Output_slechte_prior_acc.txt", "w")
for row in Outputarray_acc:
    np.savetxt(output, row)
output.close()

#Sla data op
output = open("Output_slechte_prior_data.txt", "w")
for row in Outputarray_data:
    np.savetxt(output, row)
output.close()
```

De volgende code maakt van de drie .txt bestanden het histogram in figuur 2.1.

```
import autograd.numpy as np
import matplotlib.pyplot as plt

import autograd.scipy as scipy

#Laad Theta
Output = np.loadtxt("Output_slechte_prior_Theta.txt").reshape(2, 10000)
Theta50 = Output[0]
Theta1000 = Output[1]

#Laad data
Output = np.loadtxt("Output_slechte_prior_data.txt").reshape(2, 1)
Data50 = Output[0]
Data1000 = Output[1]

#Laad acceptatie ratio
Output = np.loadtxt("Output_slechte_prior_acc.txt").reshape(2, 1)
Acc50 = Output[0]
Acc1000 = Output[1]

#Definieer range x-as
x50 = np.arange(0.5, 0.99, 0.001)
x1000 = np.arange(0.85, 0.95, 0.001)

#Begin figuur
plt.figure(figsize=[10,5])
#Twee subplots naast elkaar, schrijf in eerste subplot
plt.subplot(1, 2, 1)
#Histogram van Theta met 50 data punten
plt.hist(Theta50, bins=100, density=True, color = 'royalblue')
#Definieer lengte x-as
plt.xlim([0.5, 1])
#Plot kansverdeling 50 data punten
plt.plot(x50, scipy.stats.beta.pdf(x50, 2+Data50, 5+50-Data50), color = 'black')
#Creer titel
plt.title('Histogram theta, 50 data punten')
#Creer titel x-as
plt.xlabel('theta|x')
#Creer titel y-as
plt.ylabel('Frequentie (genormaliseerd)')

#Schrijf in tweede subplot
plt.subplot(1, 2, 2)
#Histogram van Theta met 1000 data punten
plt.hist(Theta1000, bins=1000, density=True, color = 'mediumseagreen')
#Definieer lengte x-as
plt.xlim([0.85, 0.95])
#Specificeer markeringen x-as
plt.xticks(np.arange(0.85, 0.96, 0.02))
#Plot kansverdeling 1000 data punten
plt.plot(x1000, scipy.stats.beta.pdf(x1000, 2+Data1000, 5+1000-Data1000), color = '
                                            black')
#Creer titel
plt.title('Histogram theta, 1000 data punten')
#Creer titel x-as
plt.xlabel('theta|x')
#Creer titel y-as
plt.ylabel('Frequentie (genormaliseerd)')

#Sla figuur op
plt.savefig("Histogram_slechte_prior")
```

C

Code bij hoofdstuk 4

Deze appendix bevat de code die gebruikt is om de plaatjes in hoofdstuk 4 te maken.

C.1. Code bij paragraaf 4.5

De onderstaande code genereert twee .txt bestanden met daarin de benodigde data voor het maken van figuur 4.1.

```
import autograd.numpy as np

from Data import *
from MH import *

#Genereer 100 punten uit een standaard normale verdeling
data = normdata(0, 1, 100)

#Specificeer aantal iteraties
n = 10000

#Voer Metropolis-Hastings algoritme uit met n itaties, op de functie logtarget, met
                                            begin punten (0, 1). Dit model heeft   n
                                            variabele die positief is.
Theta_klein, acc_klein = MH(n, logtarget, "Qrw", 0.8, np.array([0., np.log(1)]), 1,
                                            data)
Theta_normaal, acc_normaal = MH(n, logtarget, "Qrw", 0.15, np.array([0., np.log(1)]), 1
                                            , data)
Theta_groot, acc_groot = MH(n, logtarget, "Qrw", 0.01, np.array([0., np.log(1)]), 1,
                                            data)

#Stop de Theta's gezamelijk in een array
Outputarray = np.zeros([6, n])
Outputarray[:2, :] = Theta_klein
Outputarray[2:4, :] = Theta_normaal
Outputarray[4:, :] = Theta_groot

#Sla de array met Theta erin op
output = open("Output_slechte_tuning_Theta.txt", "w")
for row in Outputarray:
    np.savetxt(output, row)
output.close()

#Sla de acceptatie ratio's op
output1 = open("Output_slechte_tuning_acc_rate.txt", "w")
np.savetxt(output1, np.array([acc_klein]))
np.savetxt(output1, np.array([acc_normaal]))
np.savetxt(output1, np.array([acc_groot]))
output1.close()
```

De volgende code maakt de traceplots in figuur 4.1.

```
import autograd.numpy as np
import matplotlib.pyplot as plt

#Laad Theta en de acceptatie ratio's
Output = np.loadtxt("Output_slechte_tuning_Theta.txt").reshape(6, 10000)
Acc = np.loadtxt("Output_slechte_tuning_acc_rate.txt").reshape(3)

#Begin figuur
plt.figure(figsize=[25,25])
#Creer 6 subplots, drie onder elkaar, twee naast elkaar. Schrijf in eerste subplot
plt.subplot(3, 2, 1)
#Plot een lijn van Mu met te kleine acceptatie ratio
plt.plot(Output[0], color = 'mediumseagreen')
#Creer titel
plt.title('Spoor mu, te kleine acceptatie ratio: ' + str(Acc[0]), fontsize = 30)
#Creer x-as label
plt.xlabel('Iteraties', fontsize = 20)
#Creer y-as label
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 2)
#Plot een lijn van Tau met te kleine acceptatie ratio
plt.plot(Output[1], color = 'mediumseagreen')
plt.title('Spoor tau, te kleine acceptatie ratio: ' + str(Acc[0]), fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 3)
#Plot een lijn van Mu met normale acceptatie ratio
plt.plot(Output[2], color = 'purple')
plt.title('Spoor mu, normale acceptatie ratio: ' + str(Acc[1]), fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 4)
#Plot een lijn van Tau met normale acceptatie ratio
plt.plot(Output[3], color = 'purple')
plt.title('Spoor tau, normale acceptatie ratio: ' + str(Acc[1]), fontsize = 30)
plt.xlabel('Iteraties')
plt.ylabel('tau|x')
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 5)
#Plot een lijn van Mu met te grote acceptatie ratio
plt.plot(Output[4], color = 'royalblue')
plt.title('Spoor mu, te grote acceptatie ratio: ' + str(Acc[2]), fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 6)
#Plot een lijn van Tau met te grote acceptatie ratio
plt.plot(Output[5], color = 'royalblue')
plt.title('Spoor tau, te grote acceptatie ratio: ' + str(Acc[2]), fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.savefig("Traceplot_slechte_tuning")
```

De onderstaande code maakt de histogrammen in figuur 4.2

```
import autograd.numpy as np
import matplotlib.pyplot as plt

import autograd.scipy as scipy

#Laad Theta
Output = np.loadtxt("Output_slechte_tuning_Theta.txt").reshape(6, 10000)

#Laad acceptatie ratio
Acc = np.loadtxt("Output_slechte_tuning_acc_rate.txt").reshape(3)

#Begin figuur
plt.figure(figsize=[25,25])
#Twee subplots naast elkaar, schrijf in eerste subplot
plt.subplot(3, 2, 1)
#Histogram van Mu met te kleine acceptatie ratio
plt.hist(Output[0], bins=100, density=True, color = 'mediumseagreen')
#Creer titel
plt.title('Histogram mu, te kleine acceptatie ratio: ' + str(Acc[0]), fontsize = 30)
#Creer titel x-as
plt.xlabel('mu|x', fontsize = 20)
#Creer titel y-as
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in tweede subplot
plt.subplot(3, 2, 2)
#Histogram van Tau met te kleine acceptatie ratio
plt.hist(Output[1], bins=100, density=True, color = 'mediumseagreen')
plt.title('Histogram tau, te kleine acceptatie ratio: ' + str(Acc[0]), fontsize = 30)
plt.xlabel('tau|x', fontsize = 20)
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in derde subplot
plt.subplot(3, 2, 3)
#Histogram van Mu met normale acceptatie ratio
plt.hist(Output[2], bins=100, density=True, color = 'purple')
plt.title('Histogram mu, normale acceptatie ratio: ' + str(Acc[1]), fontsize = 30)
plt.xlabel('mu|x', fontsize = 20)
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in vierde subplot
plt.subplot(3, 2, 4)
#Histogram van Tau met normale acceptatie ratio
plt.hist(Output[3], bins=100, density=True, color = 'purple')
plt.title('Histogram tau, normale acceptatie ratio: ' + str(Acc[1]), fontsize = 30)
plt.xlabel('tau|x', fontsize = 20)
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in vijfde subplot
plt.subplot(3, 2, 5)
#Histogram van Mu met te grote acceptatie ratio
plt.hist(Output[4], bins=100, density=True, color = 'royalblue')
plt.title('Histogram mu, te grote acceptatie ratio: ' + str(Acc[2]), fontsize = 30)
plt.xlabel('mu|x', fontsize = 20)
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in tweede subplot
plt.subplot(3, 2, 6)
#Histogram van Tau met te grote acceptatie ratio
plt.hist(Output[5], bins=100, density=True, color = 'royalblue')
plt.title('Histogram tau, te grote acceptatie ratio: ' + str(Acc[2]), fontsize = 30)
```

```
plt.xlabel('tau|x', fontsize = 20)
plt.ylabel('Frequentie (genormaliseerd)', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Sla figuur op
plt.savefig("Histogram_slechte tuning")
```

De volgende code maakt het contourplot in paragraaf 4.5.

```
import autograd.numpy as np
import matplotlib.pyplot as plt

from Posterior_Dummy import *
from Data import *

#Genereer 100 punten uit een standaard normale verdeling
data = normdata(0, 1, 100)

#Definieer stapgrootte
h = 0.01
#Definieer de range van Mu
m = np.arange(-2.0, 2.0, h)
#Definieer de range van Tau
t = np.arange(1., 6.0, h)
#Creer een meshgrid
M, T = np.meshgrid(m, t)

#Aantal rijen van Mu
nrows = np.shape(M)[0]
#Aantal kolommen van Mu
ncols = np.shape(M)[1]
#Maak een lege array met het aantal rijen en kolommen gelijk aan Mu
Z = np.empty([nrows, ncols])
#Vul Z met de juiste waarden
for i in range(0, nrows):
    for j in range(0, ncols):
        Z[i, j] = logtarget(np.array([M[i, j], T[i, j]]))

#Maak een contourplot
contours = plt.contour(M, T, Z, 3,colors = 'black')
#Label in de lijnen
plt.clabel(contours, inline=True, fontsize=8)
#Kleurovergang
plt.imshow(Z, extent=[-2, 2, 1, 6], origin='lower', cmap='Spectral', alpha=0.5)
#Legenda voor de kleuren
plt.colorbar()
#Label x-as
plt.xlabel("Mu")
#Label y-as
plt.ylabel("Tau")
#Speficiceer titel
plt.title("Contour plot")
#Sla figuur op
plt.savefig("Contourplot_normaal")
```

C.2. Code bij paragraaf 4.6

De onderstaande code genereert twee .txt bestanden met daarin de benodigde data voor het maken van figuur 4.4

```
import autograd.numpy as np

from Data import *
from MH import *

#Genereer 100 punten uit een standaard normale verdeling
data = normdata(0, 1, 100)
```

```
#Specificeer aantal iteraties
n = 800

#Voer Metropolis-Hastings algoritme uit met n iteraties, op de functie logtarget, met
                                            begin punten (20, 21). Dit model heeft
                                              n   variabele die positief is.
Theta_rw, acc_rw = MH(n, logtarget, "Qrw", 0.18, np.array([20., np.log(21)]), 1, data)
Theta_l, acc_l = MH(n, logtarget, "Ql", 0.025, np.array([20., np.log(21)]), 1, data)
Theta_b, acc_b = MH(n, logtarget, "Qb", 0.25, np.array([20., np.log(21)]), 1, data)

#Stop de Theta's gezamelijk in een array
Outputarray = np.zeros([6, n])
Outputarray[:2, :] = Theta_rw
Outputarray[2:4, :] = Theta_l
Outputarray[4:, :] = Theta_b

#Sla de array met Theta erin op
output = open("Output_convergentie_snelheid_Theta.txt", "w")
for row in Outputarray:
    np.savetxt(output, row)
output.close()

#Sla de acceptatie ratio's op
output1 = open("Output_convergentie_snelheid_acc_rate.txt", "w")
np.savetxt(output1, np.array([acc_rw]))
np.savetxt(output1, np.array([acc_l]))
np.savetxt(output1, np.array([acc_b]))
output1.close()
```

De volgende code maakt de traceplots in figuur 4.4

```
import autograd.numpy as np
import matplotlib.pyplot as plt

#Laad Theta en de acceptatie ratio's
Output = np.loadtxt("Output_convergentie_snelheid_Theta.txt").reshape(6, 800)
Acc = np.loadtxt("Output_convergentie_snelheid_acc_rate.txt").reshape(3)

#Begin figuur
plt.figure(figsize=[25,25])
#Creer 6 subplots, drie onder elkaar, twee naast elkaar. Schrijf in eerste subplot
plt.subplot(3, 2, 1)
#Plot een lijn van Mu gegenereerd door Random Walk
plt.plot(Output[0], color = 'mediumseagreen')
#Creer titel
plt.title('Spoor mu; Random Walk', fontsize = 30)
#Creer x-as label
plt.xlabel('Iteraties', fontsize = 20)
#Creer y-as label
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in tweede subplot
plt.subplot(3, 2, 2)
#Plot een lijn van Tau gegenereerd door Random Walk
plt.plot(Output[1], color = 'mediumseagreen')
plt.title('Spoor tau; Random Walk', fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in derde subplot
plt.subplot(3, 2, 3)
#Plot een lijn van Mu gegenereerd door Langevin
plt.plot(Output[2], color = 'purple')
plt.title('Spoor mu; Langevin', fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
```

```
plt.yticks(fontsize = 20)

#Schrijf in vierde subplot
plt.subplot(3, 2, 4)
#Plot een lijn van Tau gegenereerd door Langevin
plt.plot(Output[3], color = 'purple')
plt.title('Spoor tau; Langevin', fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in vijfde subplot
plt.subplot(3, 2, 5)
#Plot een lijn van Mu gegenereerd door Barker
plt.plot(Output[4], color = 'royalblue')
plt.title('Spoor mu; Barker', fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('mu|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#Schrijf in zesde subplot
plt.subplot(3, 2, 6)
#Plot een lijn van Mu gegenereerd door Barker
plt.plot(Output[5], color = 'royalblue')
plt.title('Spoor tau; Barker', fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|x', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.savefig("Traceplot_convergentie_snelheid")
```

D
Code bij hoofdstuk 5

De onderstaande code berekent de effective sample size.

```
import autograd.numpy as np
from time import process_time

from Data import *
from MH import *
from Posterior import *

#Input is een array of arrays; elke rij is een parameter in het model
#Output is de Effective Sample Size

def Variogram(Theta, t):
    return ((Theta[:, t:] - Theta[:, :-t])**2).sum(axis=1, keepdims=True)/(len(Theta[0]
                                    )-t)

def ESS(n, n_burn, logtarget, Q, stepsize, startval, n_pos, data):
    start = process_time()
    Theta, acc_rate = MH(n + n_burn, logtarget, Q, stepsize, startval, n_pos, data)
    end = process_time()
    #Remove the first n_burn values of each parameter.
    Theta = np.delete(Theta, slice(0, n_burn), 1)
    W = ((Theta-Theta.mean(axis=1, keepdims=True))**2).sum(axis=1, keepdims=True)/(len(
                                    Theta[0])-1)
    #B = len(Theta[0])*(Theta.mean(axis=1, keepdims=True)-Theta.mean())**2

    #Var = (len(Theta[0])-1)/len(Theta[0]) * W + 1/len(Theta[0]) * B
    ##B=0
    Var = (len(Theta[0])-1)/len(Theta[0]) * W #= np.array([var_mu, var_tau]) --> niet
                                    verwerkt in rho.

    #rho = np.ones(len(Theta[0]))
    rho = np.ones([len(Theta), len(Theta[0])])

    for i in range(0, len(Theta)):
        t=1
        negative_autocorr = False
        while not negative_autocorr and (t<len(Theta[0])):
            rho[i, t] = 1-Variogram(Theta, t)[i]/(2*Var[i])
            if not t%2:
                negative_autocorr = sum(rho[i][t-1:t+1])<0
            t+=1

    return len(Theta[0])/(1+2*rho.sum(axis=1, keepdims=True))/(end-start), acc_rate


def f(mu_data, sigma_data, n_data, N_MCMC, n_burn, logtarget, Q, stepsize, n_pos,
                                startval):
    data = normdata(mu_data, sigma_data, n_data)
    return ESS(N_MCMC, n_burn, logtarget, Q, stepsize, startval, n_pos, data)
```

De onderstaande code roept het bestand ESS.py aan om de een .txt bestand te creëren met de effective sample size van de drie kernels per parameter.

```
import autograd.numpy as np

from ESS import *

#Definieer initi le waarden verder van centrum verdeling
tail = np.array([5., 6.])
#Definieer initi le waarden in centrum verdeling
center = np.array([0., 1.])
#Creer array met de startwaarden
startvals = [center, tail]

#Het aantal keer dat de effective sample size berekent wordt per startwaarde
B = 500

#Stapgrootte per methode
stepsize_rw = 0.15
stepsize_l = 0.022
stepsize_b = 0.25

##Outputarray: 1ste rij: Ess_mu_center, 2de rij: Ess_tau_center, 3de rij:
                                             acc_rate_center
##4de rij: Ess_mu_tail, 5de rij: Ess_tau_tail, 6de rij: acc_rate_tail
Outputarray = np.zeros([6, B])

#Bereken effective sample size 500 keer per startwaarde
for s in range(len(startvals)):
    for b in range(B):
        Ess, acc_rate = f(0, 1, 100, 10000, 500, logtarget, "Ql", stepsize_l, 1,
                                                     startvals[s])
        Outputarray[3*s, b] = Ess[0]
        Outputarray[3*s+1, b] = Ess[1]
        Outputarray[3*s+2, b] = acc_rate
        if b%5==0:
            print(b)

#Sla de output op
output = open("Output_l_ess+acc_rate.txt", "w")
for row in Outputarray:
    np.savetxt(output, row)
output.close()
```

De onderstaande code maakt de boxplot te zien in figuur 5.1

```
import autograd.numpy as np
import matplotlib.pyplot as plt

#Laad effective sample size en acceptatie ratios
OutputRW = np.loadtxt("Output_rw_ess+acc_rate.txt").reshape(6, 500)
OutputL = np.loadtxt("Output_l_ess+acc_rate.txt").reshape(6, 500)
OutputB = np.loadtxt("Output_b_ess+acc_rate.txt").reshape(6, 500)

#Creer lijsten waarin de output gesorteerd is
Ess_mu = [OutputRW[0], OutputRW[1], np.where(np.isnan(OutputL[0]), np.nanmean(OutputL[0
                                             ]), OutputL[0]), np.where(np.isnan(OutputL[
                                             1]), np.nanmean(OutputL[1]), OutputL[1]),
                                             OutputB[0], OutputB[1]]
Ess_tau = [OutputRW[3], OutputRW[4], OutputL[3], OutputL[4], OutputB[3], OutputB[4]]
Acc = [OutputRW[2], OutputRW[5], OutputL[2], OutputL[5], OutputB[2], OutputB[5]]

#Maak het figuur
plt.figure(figsize=[30,25])
#Maak twee subfiguren, schrijf in het eerste
plt.subplot(1, 2, 1)
#Maak een boxplot voor de effective sample size van mu
plt.boxplot(Ess_mu)
#Definieer de titel
plt.title('Effective sample size of mu per seconde per voorstel', fontsize=30)
#Specificeer de juiste namen van de markeringen van de x-as
```

```
plt.xticks([1, 2, 3, 4, 5, 6], ['centrum (RW)', 'staart (RW)', 'centrum (L)', 'staart (
                                            L)', 'centrum (B)', 'staart (B)'], fontsize
                                             = 25, rotation = 60)
plt.yticks(fontsize = 20)
#Specificeer de titel van de y-as
plt.ylabel('Effective sample size per seconde', fontsize = 20)

#Schrijf in tweede subfiguur
plt.subplot(1, 2, 2)
#Maak een boxplot voor de effective sample size van tau
plt.boxplot(Ess_tau)
plt.title('Effective sample size of tau per seconde per voorstel', fontsize = 30)
plt.xticks([1, 2, 3, 4, 5, 6], ['centrum (RW)', 'staart (RW)', 'centrum (L)', 'staart (
                                            L)', 'centrum (B)', 'staart (B)'], fontsize
                                             = 25, rotation = 60)
plt.yticks(fontsize = 20)
plt.ylabel('Effective sample size per seconde', fontsize = 20)

#plt.show()
plt.savefig("Boxplot_ESS")
```

E

Code bij hoofdstuk 6

De volgende code is de gebruikte implementatie van het Gibbs algoritme.

```
import autograd.numpy as np
from operator import setitem
from autograd import grad

from Posterior import *

def Prw(x, stepsize):
    Mean = np.zeros(len(x))
    Cov = np.identity(len(x))
    return stepsize*x + np.sqrt(1-stepsize**2)*np.random.multivariate_normal(Mean, Cov,
                                                                           1)

def Pl(x, stepsize, grad_x):
    return x + 1/2*stepsize*grad_x + np.sqrt(stepsize)*np.random.normal(0, 1, len(x))

def Ql(x, y, stepsize, grad_x):
    Arg = 1/np.sqrt(stepsize)*(y-x-1/2*stepsize*grad_x)
    return lognorm(np.array([0, 1]), Arg)

def p(x, z, grad_x):
    denominator = 1 + np.exp(-z*grad_x)
    return 1/denominator

def b(x, z, grad_x):
    return np.where(p(x, z, grad_x)>np.random.uniform(0, 1, len(x)), 1, -1)

def Pb(x, z, grad_x):
    return (x + b(x, z, grad_x)*z).astype(float)

def Qb(x, y, grad_x):
    return np.sum(np.log(1 + np.exp(grad_x*(x-y))))

def Ppos(x, stepsize):
    return x*np.exp(stepsize*np.random.normal(0, 1))

def Qpos(x):
    return np.sum(np.log(x))



def Gibbs(n, target, method, stepsize_z, stepsize_tau, z, tau, N, U):
    Unif = np.log(np.random.uniform(0, 1, n))
    NormT = np.random.normal(0, 1, n)
    NormZ = np.random.normal(0, stepsize_z, n*len(z))

    Tau = np.zeros(n)
    Tau[0] = tau
    Z = np.zeros([len(z), n])
```

```
    Z[:, 0] = z

    Theta = np.zeros([len(z), n-1])

    ll = target(z, tau, N, U)

    acc_tau = 0
    acc_z = 0

    z_prop = np.zeros([len(z)])

    if method == "Ql" or method == "Qb":
        grad_z_prop = np.zeros([len(z)])
        grad_z = grad(target, 0)(z, tau, N, U)
    for i in range(1, n):
        tau_prop = Ppos(tau, stepsize_tau)
        ll_prop = target(z, tau_prop, N, U)
        if np.log(np.random.uniform()) < ll_prop + Qpos(tau) - (ll + Qpos(tau_prop)):
            tau = tau_prop
            ll = ll_prop
            acc_tau += 1
        Tau[i] = tau

        if method == "Qrw":
            setitem(z_prop, np.array(range(0, len(z))), Prw(z, stepsize_z))
        elif method == "Ql":
            setitem(z_prop, np.array(range(0, len(z))), Pl(z, stepsize_z, grad_z))
            setitem(grad_z_prop, np.array(range(0, len(z))), grad(target, 0)(z_prop,
                                                        tau, N, U))
        elif method == "Qb":
            setitem(z_prop, np.array(range(0, len(z))), Pb(z, np.random.normal(0,
                                                        stepsize_z, len(z)), grad_z))
            setitem(grad_z_prop, np.array(range(0, len(z))), grad(target, 0)(z_prop,
                                                        tau, N, U))
        ll_prop = target(z_prop, tau, N, U)
        if method == "Qrw":
            if np.log(np.random.uniform()) < ll_prop - ll:
                setitem(z, np.array(range(0, len(z))), z_prop)
                ll = ll_prop
                acc_z += 1
        elif method == "Ql":
            if np.log(np.random.uniform()) < ll_prop + Ql(z_prop, z, stepsize_z,
                                                        grad_z_prop) - (ll + Ql(z,
                                                        z_prop, stepsize_z, grad_z)):
                setitem(z, np.array(range(0, len(z))), z_prop)
                ll = ll_prop
                setitem(grad_z, np.array(range(0, len(z))), grad_z_prop)
                acc_z += 1
        elif method == "Qb":
            if np.log(np.random.uniform()) < ll_prop + Qb(z, z_prop, grad_z) - (ll + Qb
                                                        (z_prop, z, grad_z_prop)):
                setitem(z, np.array(range(0, len(z))), z_prop)
                ll = ll_prop
                setitem(grad_z, np.array(range(0, len(grad_z))), grad_z_prop)
                acc_z += 1
        Z[:, i] = z
        Theta[:, i-1] = softmax(np.dot(U, z)*np.sqrt(tau))
    return Theta, Tau, Z, acc_tau/n, acc_z/n
```

De onderstaande code maakt twee .txt bestanden. De eerste bevat de waarden van τ en z. De tweede bevat de gemiddelden van a posteriori verdeling.

```
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt

from Data import *
from Posterior import *
from Gibbs import *

x_bins = 2
```

```
y_bins = 4
tau = 0.1

D, N = markmodeldata(10000, x_bins, y_bins)
U = matrixU(x_bins, y_bins)

Zet = np.ones(x_bins*y_bins)

np.random.seed()

Stepsize_tau = 0.17
Stepsize_z = 0.9999

n_iters = 1000000

Theta, Tau, Z, acc_tau, acc_z = Gibbs(n_iters, logpostmarkmodel, "Qrw", Stepsize_z,
                                        Stepsize_tau, Zet, tau, N, U)

Outputarray = np.zeros([x_bins*y_bins+1, n_iters])
Outputarray[0, :] = Tau
Outputarray[1:, :] = Z

Thetaoutput = open("Theta_Qrw_posterior_data_10000_iters_100000_2x4.txt", "w")
for row in Theta:
    np.savetxt(Thetaoutput, row)
Thetaoutput.close()

output = open("Tau_and_Z_Qrw_posterior_data_10000_iters_100000_2x4.txt", "w")
for row in Outputarray:
    np.savetxt(output, row)
output.close()
```

De onderstaande code genereert de figuren in hoofdstuk 6

```
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt

from Data import *

D8, N8 = markmodeldata(10000, 2, 4)
D50, N50 = markmodeldata(10000, 5, 10)

n_iters = 1000000

OutputRW8 = np.loadtxt("Tau_and_Z_Qrw_posterior_data_10000_iters_100000_2x4.txt").
                                        reshape(9, n_iters)
ThetaRW8 = np.loadtxt("Theta_Qrw_posterior_data_10000_iters_100000_2x4.txt").reshape(8,
                                         n_iters-1)
OutputRW50 = np.loadtxt("Tau_and_Z_Qrw_posterior_data_10000_iters_100000_5x10.txt").
                                        reshape(51, n_iters)
ThetaRW50 = np.loadtxt("Theta_Qrw_posterior_data_10000_iters_100000_5x10.txt").reshape(
                                        50, n_iters-1)

OutputB8 = np.loadtxt("Tau_and_Z_Qb_posterior_data_10000_iters_100000_2x4.txt").reshape
                                        (9, n_iters)
ThetaB8 = np.loadtxt("Theta_Qb_posterior_data_10000_iters_100000_2x4.txt").reshape(8,
                                        n_iters-1)
OutputB50 = np.loadtxt("Tau_and_Z_Qb_posterior_data_10000_iters_100000_5x10.txt").
                                        reshape(51, n_iters)
ThetaB50 = np.loadtxt("Theta_Qb_posterior_data_10000_iters_100000_5x10.txt").reshape(50
                                        , n_iters-1)

#Plot data
N8_data = N8.reshape(4, 2)
N50_data = N50.reshape(10, 5)

plt.figure(figsize=[10,10])
plt.subplot(1, 2, 1)
plt.imshow(N8_data, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
```

```
plt.title("Hittekaart data, 2x4 verfijning")

plt.subplot(1, 2, 2)
plt.imshow(N50_data, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
plt.title("Hittekaart data, 5x10 verfijning")

#plt.show()
plt.savefig("Hittekaart_data_10000_iters_1000000_compgrad")

plt.close()

#Plot gemiddelde a posteriori
TRW8 = (ThetaRW8.mean(axis=1, keepdims=True)).reshape(4, 2)
TRW50 = (ThetaRW50.mean(axis=1, keepdims=True)).reshape(10, 5)

TB8 = (ThetaB8.mean(axis=1, keepdims=True)).reshape(4, 2)
TB50 = (ThetaB50.mean(axis=1, keepdims=True)).reshape(10, 5)

plt.figure(figsize=[25,25])
plt.subplot(2, 2, 1)
plt.imshow(TRW8, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
plt.title("Hitte kaart gemiddelde a posteriori, Random Walk, 2x4 verfijning", fontsize
                                            = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(2, 2, 2)
plt.imshow(TRW50, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
plt.title("Hitte kaart gemiddelde a posteriori, Random Walk, 5x10 verfijning", fontsize
                                             = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(2, 2, 3)
plt.imshow(TB8, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
plt.title("Hitte kaart gemiddelde a posteriori, Barker, 2x4 verfijning", fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(2, 2, 4)
plt.imshow(TB50, extent=[0, 1, 0, 2], origin='lower', cmap='Spectral')
plt.colorbar()
plt.title("Hitte kaart gemiddelde a posteriori, Barker, 5x10 verfijning", fontsize = 20
                                            )
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#plt.show()
plt.savefig("Hittekaart_posterior_mean_data_10000_iters_1000000_compgrad")
plt.close()

#Traceplot RW
plt.figure(figsize=[25,25])
plt.subplot(3, 2, 1)
plt.plot(OutputRW8[0], color = 'purple')
plt.title("Spoor tau, Random Walk, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 3)
plt.plot(OutputRW8[3], color = 'purple')
plt.title("Spoor z_3, Random Walk, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_3|data', fontsize = 20)
plt.xticks(fontsize = 20)
```

```
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 5)
plt.plot(OutputRW8[6], color = 'purple')
plt.title("Spoor z_6, Random Walk, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_6|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 2)
plt.plot(OutputRW50[0], color = 'mediumseagreen')
plt.title("Spoor tau, Random Walk, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 4)
plt.plot(OutputRW50[3], color = 'mediumseagreen')
plt.title("Spoor z_3, Random Walk, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_3|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 6)
plt.plot(OutputRW50[6], color = 'mediumseagreen')
plt.title("Spoor z_6, Random Walk, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_6|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#plt.show()
plt.savefig("Traceplots_Qrw_posterior_data_10000_iters_1000000_compgrad")
plt.close()

#Traceplot B
plt.figure(figsize=[25,25])
plt.subplot(3, 2, 1)
plt.plot(OutputB8[0], color = 'purple')
plt.title("Spoor tau, Barker, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 3)
plt.plot(OutputB8[3], color = 'purple')
plt.title("Spoor z_3, Barker, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_3|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 5)
plt.plot(OutputB8[6], color = 'purple')
plt.title("Spoor z_6, Barker, 2x4 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_6|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 2)
plt.plot(OutputB50[0], color = 'mediumseagreen')
plt.title("Spoor tau, Barker, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('tau|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)
```

```
plt.subplot(3, 2, 4)
plt.plot(OutputB50[3], color = 'mediumseagreen')
plt.title("Spoor z_3, Barker, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_3|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

plt.subplot(3, 2, 6)
plt.plot(OutputB50[6], color = 'mediumseagreen')
plt.title("Spoor z_6, Barker, 5x10 verfijning", fontsize = 30)
plt.xlabel('Iteraties', fontsize = 20)
plt.ylabel('z_6|data', fontsize = 20)
plt.xticks(fontsize = 20)
plt.yticks(fontsize = 20)

#plt.show()
plt.savefig("Traceplots_Qb_posterior_data_10000_iters_1000000_compgrad")
```

Bibliografie

[1] Geurt Jongbloed, Frank van der Meulen, and Lixue Pang. Bayesian nonparametric estimation in the current status continuous mark model. *arXiv preprint arXiv:1911.10387*, 2019.

[2] Samuel Livingstone and G. Zanella. The barker proposal: combining robustness and efficiency in gradient-based mcmc. *arXiv: Computation*, 2019.

[3] Tristan Marshall and Gareth Roberts. An adaptive approach to langevin mcmc. *Statistics and Computing*, 22(5):1041–1057, 2012.

[4] Frank van der Meulen. Statistical inference, lecture notes for the course wi4455, Augustus 2020.

[5] Jack Walton. Speed up Python code with Numpy: an example case. `https://jwalton.info/Efficient-effective-sample-size-python/`, 2018. [Online; accessed 26-05-2021].

[6] G.A. Young and R.L. Smith. *Essentials of Statistical Inference*. Cambridge University Press, 2005.