Hoofdstuk 1 — Numerical Limitations (waarom numeriek rekenen soms “vals speelt”)

Dit hoofdstuk is de *grondwet* van de hele cursus: elke numerieke methode is een trade-off tussen **benaderingsfout** (truncation) en **rekenfout** (rounding). Als je op het examen moet uitleggen *waarom methode A beter is dan B*, zit het argument bijna altijd hier: **stabiliteit + conditioning + foutopbouw + kost**.

1.1 Approximations in scientific computation

Absolute vs relative error

- **Absolute error**: $$e_{\text{abs}} = x_{\text{approx}} - x_{\text{true}}$$
- **Relative error**: $$e_{\text{rel}} = \frac{x_{\text{approx}} - x_{\text{true}}}{x_{\text{true}}}$$

Interpretatie (examenvriendelijk): als $e_{\text{rel}} \sim 10^{-p}$, dan heb je ongeveer **$p$ correcte significant digits**.

Precision vs accuracy

- **Precision** = hoeveel digits je *opschrijft/bewaart*.
- **Accuracy** = hoeveel digits *effectief correct zijn*. Kernboodschap: meer precision garandeert geen accuracy (bv. een lang getal kan toch totaal naast π zitten).

Truncation error vs rounding error

- **Truncation error**: komt uit de *wiskundige benadering* (afkappen van reeksen, finite differences, discretisatie, …).
- **Rounding error**: komt uit het feit dat computers met **eindige precisie** rekenen en voortdurend afronden.

Belangrijk patroon voor “wanneer welke methode?”
Als je de stapgrootte h kleiner maakt (of N groter, meer iteraties, fijnere grid):

- truncation error daalt meestal,
- rounding error en kost stijgen meestal,
- en soms wordt het resultaat zelfs slechter door foutopstapeling.

Dus: “gewoon fijner nemen” is géén universele win.

1.2 Computer arithmetic

1.2.1 Floating-point number systems (het model achter `float`)

Floating-point lijkt op wetenschappelijke notatie: $$ \pm (d_0 + d_1\beta^{-1} + \dots + d_{p-1}\beta^{-(p-1)}),\beta^{E} $$ met:

- β = base/radix (typisch 2),
- p = precision (aantal “mantissa digits”),
- $E \in [L,U]$ = exponent range.

Normalisatie (zoals in IEEE 754): de leidende digit wordt “vastgezet” (typisch $d_0=1$ voor niet-nul). Daardoor “win” je effectief één bit aan informatie (de bekende *hidden bit* in base 2).

Praktisch in Python:

- `float` is (typisch) **double precision**.
- `numpy` laat ook **single precision** toe (`np.float32`), met minder geheugen maar merkbaar meer rounding error.

1.2.2 Properties (discreet, eindig, en soms gemeen)

(1) Discreet getallenrooster
Tussen twee representabele floats zit een “gat”. Veel decimalen bestaan niet exact in binair.

(2) Overflow/underflow

- Te groot $\to$ `inf` (overflow).
- Te klein $\to$ afronding naar `0.0` (underflow), eventueel via subnormals/denormals.

(3) 0.1 is niet exact Klassiek gevolg: `0.1 + 0.1 + 0.1 != 0.3`.
Dus: `==` is bijna altijd fout bij floats.

(4) Afronding maakt optellen niet-associatief Door rounding geldt vaak: $$(a+b)+c \neq a+(b+c)$$
Concreet gevolg: **somvolgorde** van een lange reeks verandert de uitkomst (en soms merkbaar).

- “Natural order” kan slechter zijn dan “reverse order”.
- Groepering beïnvloedt cancellation en rounding-opstapeling.

> Examenzin die altijd scoort: *Een algoritme kan wiskundig correct zijn, maar numeriek instabiel doordat rounding zich opstapelt afhankelijk van de route die je neemt.*

1.2.3 Good Practices for computer arithmetic (de survival kit)

A. Cancellation (catastrophic cancellation)

- **Vermijd**: subtractie van bijna gelijke getallen.
  Voorbeeldidee: $\sqrt{x+1} - \sqrt{x}$ voor groot x $\to$ twee bijna gelijke grote getallen $\to$ verschil verliest significant digits.
- **Fix**: herschrijf algebraïsch naar een stabiele vorm: $$ \sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1} + \sqrt{x}} $$

B. Addition (small + large)

- **Vermijd**: een heel klein getal optellen bij een enorm groot getal (het kleine kan volledig “verdwijnen” omdat het onder de float-spacing zit).
- **Praktische regel**: sommeer een lijst **van klein naar groot** (of gebruik algoritmes die dit effect beperken).

C. Float-vergelijkingen: altijd met tolerantie

- Absolute tolerantie is belangrijk rond 0.
- Relatieve tolerantie is belangrijk bij grote schalen. In code: `np.isclose` / `math.isclose` i.p.v. `==`.

"Wanneer welke aanpak?" — beslisregels die je echt gebruikt in fysicaproblemen

1) Kies je datatype bewust

- `float64` **(double)**: default voor fysica/numeriek werk $\to$ betere accuracy, meestal nog snel genoeg.
- `float32` **(single)**: nuttig bij grote arrays/GPU/memory pressure, maar verwacht:
  - meer rounding,
  - sneller instabiliteit,
  - grotere gevoeligheid voor somvolgorde/cancellation.

2) Schaal je probleem (units/normalization) vóór je gaat rekenen

Als grootheden extreem groot/klein zijn, krijg je sneller overflow/underflow en slechtere conditioning.

- Rescale variabelen zodat typische waarden $O(1)$ zijn.
- Dit is geen cosmetica: het kan de numerieke betrouwbaarheid drastisch verbeteren.

3) Prefer "numerically stable" formuleringen boven "directe" formuleringen

Zelfde wiskunde, andere route $\to$ andere rounding error.

- Herformuleer uitdrukkingen met subtractie van bijna gelijken.
- Orden sommen slim.
- (Later in de cursus: kies matrixfactorisaties die stabiliteit geven.)

Links naar andere hoofdstukken (de bruggen die examenvragen graag testen)

- **Hoofdstuk 2 (Linear systems)**
  Pivoting en keuze van factorisatie (LU vs Gauss-Jordan vs Cholesky) zijn antwoorden op rounding error, stabiliteit en conditioning.

- **Hoofdstuk 3 (Least squares)**
  "Normal equations" vs "QR" vs "SVD" is een klassieker:
  - normal equations kunnen conditioning verslechteren $\to$ rounding wordt gevaarlijker,
  - QR en SVD zijn duurder maar stabieler.

- **Niet-lineaire oplossingen / optimalisatie / integratie / ODE's**
  Overal zie je hetzelfde thema: kleinere h verlaagt truncation, maar kan rounding/foutopstapeling verhogen; stabiliteit bepaalt of "harder werken" ook echt "beter antwoord" geeft.

Mini-checklist (mondeling examen waardig)

Als je moet verdedigen waarom je aanpak numeriek “goed” is, zeg expliciet:

1. Welke fouten bestaan hier? (truncation + rounding)
2. Waar zit cancellation/ill-conditioning?
3. Welke herformulering / datatype / somvolgorde maakt het stabieler?
4. Wat kost dat (tijd/geheugen) en waarom is die kost het waard?

Hoofdstuk 2 — Systems of Linear Equations

Dit hoofdstuk leert je **lineaire stelsels** oplossen $$ \mathbf{A}\mathbf{x}=\mathbf{b}, $$ en vooral: **hoe je dat doet op een manier die (i) snel is en (ii) numeriek stabiel** is in floating-point (link met H1).

De rode draad (en dit is letterlijk de “type 1”-examenvraag zoals Q1):
transformeer het probleem naar een vorm die je **goedkoop** kan oplossen (triangulair), en doe dat met controle op stabiliteit (pivoting).

2.1 Introduction and Notation

Lineaire modellen in fysica

In de cursus worden o.a. genoemd:

- Newton: $\mathbf{F}=m\mathbf{a}$
- Ohm: $R=U/I$
- Hooke: $\mathbf{F}_s=k\mathbf{x}$

Algemeen: een systeem van m lineaire vergelijkingen met n onbekenden: $$ \begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2\ \vdots\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \end{cases} $$ en in matrixvorm: $$ \mathbf{A}\mathbf{x}=\mathbf{b}, $$ waar $\mathbf{A}\in\mathbb{R}^{m\times n}$ (of $\mathbb{C}^{m\times n}$), $\mathbf{x}\in\mathbb{R}^n$, $\mathbf{b}\in\mathbb{R}^m$.

2.2 Solving Linear Systems

2.2.1 Triangular systems: forward/back substitution

Als je het stelsel kunt omvormen naar een triangulair stelsel, is oplossen goedkoop.

Lower triangular $\mathbf{L}\mathbf{x}=\mathbf{b}$ (forward substitution): $$ x_1=\frac{b_1}{l_{11}},\qquad x_i=\frac{b_i-\sum_{j=1}^{i-1}l_{ij}x_j}{l_{ii}}\quad (i=2,\dots,n). $$

Upper triangular $\mathbf{U}\mathbf{x}=\mathbf{b}$ (back substitution): $$ x_n=\frac{b_n}{u_{nn}},\qquad x_i=\frac{b_i-\sum_{j=i+1}^{n}u_{ij}x_j}{u_{ii}}\quad (i=n-1,\dots,1). $$

Waarom dit belangrijk is: de hele rest van het hoofdstuk gaat over “hoe krijg ik $\mathbf{A}$ naar (een product van) zulke driehoeksmatrices?”

2.2.2 Elementary elimination matrices (Gauss transformations)

Gaussian elimination elimineert systematisch elementen onder de diagonaal via rijoperaties.

Bij stap k kies je een pivot (in de cursus: “the pivot”) en elimineer je a_{ik} voor $i>k$ door rij i te vervangen door: $$ \text{row}_i \leftarrow \text{row}*i + m_i,|text{row}k, |qquad |text{met } m_i=-|frac{a{ik}} {a*{kk}}. $$

Dat kan in matrixvorm met een **elementary elimination matrix** $\mathbf{M}_k$:

- Effect: voeg een veelvoud van rij k toe aan rijen $k+1,\dots,n$ zodat de kolom onder de diagonaal nul wordt.
- In de cursus staat expliciet de vorm: $$ \mathbf{M}_k=\mathbf{I}-\mathbf{m}_k\mathbf{e}_k^\mathsf{T}, $$ waar $\mathbf{m}*k= *[0,|dots,0,m*{k+1},\dots,m_n]^\mathsf{T}$ en $\mathbf{e}_k$ de k-de kolom van $\mathbf{I}$ is.

Belangrijke eigenschap (cursus): $$ \mathbf{M}_k^{-1}=\mathbf{I}+\mathbf{m}_k\mathbf{e}_k^\mathsf{T}. $$ Dus de inverse heeft **dezelfde structuur** maar met **omgekeerde tekens** (in de cursus wordt $\mathbf{M}_k^{-1}$ vaak aangeduid als $\mathbf{L}_k$).

2.2.3 Gaussian elimination $\Rightarrow$ LU factorization

Als je alle eliminatiestappen samenneemt: $$ \mathbf{M}=\mathbf{M}_{n-1}\cdots \mathbf{M}_1, $$ dan wordt $$ \mathbf{M}\mathbf{A}=\mathbf{U}, $$ met $\mathbf{U}$ upper triangular.

Definieer dan: $$ \mathbf{L}=\mathbf{M}^{-1},\qquad \mathbf{U}=\mathbf{M}\mathbf{A}. $$ Dan krijg je de **LU-factorisatie**: $$ \mathbf{A}=\mathbf{L}\mathbf{U}. $$

Waarom LU zo centraal is (examenvraag-stijl):

- Factoriseer één keer (duur),
- Los daarna op met twee triangulaire solves (goedkoop):
    1. $\mathbf{L}\mathbf{y}=\mathbf{b}$ (forward)
    2. $\mathbf{U}\mathbf{x}=\mathbf{y}$ (back)

Extra winst (cursus benadrukt dit): als je meerdere rechterleden hebt (zelfde $\mathbf{A}$, andere $\mathbf{b}$), hergebruik je $\mathbf{L},\mathbf{U}$ en betaal je per extra $\mathbf{b}$ enkel substituties.

2.2.4 Partial pivoting (stabiliteit + vermijden van breakdown)

De cursus geeft 2 problemen bij “naïeve” Gaussian elimination:

1. **Breakdown** als pivot $a_{kk}=0$ (je moet delen door 0).
   Oplossing: wissel rijen zodat je een niet-nul pivot hebt → **pivoting**.

2. **Numerieke instabiliteit** in floating-point: te grote multipliers versterken rounding errors.
   Oplossing: **partial pivoting**: kies in kolom k de entry met **grootste absolute waarde** op/onder de diagonaal als pivot. Dan blijven multipliers in grootte ≤ 1.

Met pivoting verschijnt een permutatiematrix $\mathbf{P}$ en typisch krijg je (notatie zoals in veel software): $$ \mathbf{P}\mathbf{A}=\mathbf{L}\mathbf{U}. $$

De cursus vermeldt ook expliciet een SciPy-conventie: als je $\mathbf{P}$ "in $\mathbf{L}$ absorbeert", kan je schrijven dat $\mathbf{L}\mathbf{U}=\mathbf{A}$ (met een $\mathbf{L}$ die een permutatie van lower-triangular is). Dit is consistent met `scipy.linalg` met optie `permute_l=True`.

Praktisch besluit (waarom het "beter" is): partial pivoting maakt LU **veel robuuster** zonder de orde van de kost te veranderen.

2.2.5 Gauss–Jordan elimination (waarom meestal niet)

Je kunt ook elimineren tot een **diagonale** matrix (of zelfs $\mathbf{I}$) door *ook boven* de diagonaal weg te werken: dat is Gauss–Jordan.

Cursusboodschap:

- Ja, het kan.
- Maar de extra kost levert meestal niet genoeg voordeel op voor het oplossen van $\mathbf{A}\mathbf{x}=\mathbf{b}$, zeker niet t.o.v. LU + substitutie.

2.3 Special types of linear systems

2.3.1 Symmetric positive definite (SPD) $\Rightarrow$ Cholesky

Als $\mathbf{A}$ **symmetric positive definite** is, bestaat: $$ \mathbf{A}=\mathbf{L}\mathbf{L}^\mathsf{T} $$ (Cholesky factorization).

De cursus somt expliciet voordelen op:

- De n wortels zijn van **positieve** getallen → algoritme is goed-gedefinieerd
- **Geen pivoting** nodig
- Je gebruikt enkel de **lower triangle** van $\mathbf{A}$ (minder opslag)
- Kost: ongeveer $n^3/6$ multiplications (en vergelijkbaar aantal additions)

Conclusie (cursus): Cholesky is ongeveer **half zoveel werk en opslag** als algemene LU.

Wanneer kies je Cholesky?
Als je uit de fysica/matrixstructuur kunt argumenteren dat $\mathbf{A}$ SPD is (bv. bepaalde energie/Hessiaan-achtige matrices, normal equations later in H3).

2.3.2 Computational complexity (kostargumenten die je moet kunnen verwoorden)

De cursus geeft de typische flop-counts:

- LU-factorisatie van $n\times n$: ongeveer $n^3/3$ flops
- Volledige matrix-inversie: ongeveer n^3 flops (dus ~3× duurder)
- Oplossen met forward + backward substitution na LU: ongeveer n^2 flops (voor grote n verwaarloosbaar t.o.v. factorisatie)
- Cramer's rule: "astronomisch duur"

Belangrijkste praktijkregel (cursus zegt dit letterlijk in spirit):
Bereken $\mathbf{A}^{-1}$ bijna nooit expliciet; los $\mathbf{A}\mathbf{x}=\mathbf{b}$ op via factorisatie + substitutie (sneller én nauwkeuriger).

2.4 Sensitivity and Conditioning

Hier leer je het verschil tussen:

- "ik heb een $\mathbf{x}$ uitgerekend"
- "mijn $\mathbf{x}$ is betrouwbaar"

2.4.1 Vector norms (p-normen zoals in de cursus)

Voor $p>0$: $$ |\mathbf{x}|*p=|*left(|*sum{i=1}^n |x_i|^p\right)^{1/p}. $$

Belangrijke gevallen:

- $p=1$ (Manhattan): $$|\mathbf{x}|*1=|*sum{i=1}^n |x_i|$$
- $p=2$ (Euclidisch): $$|\mathbf{x}|*2=|*left(|*sum{i=1}^n |x_i|^2\right)^{1/2}$$
- $p=\infty$: $$|\mathbf{x}|_\infty=\max_i |x_i|$$

2.4.2 Matrix norms (induced norms die SciPy ook gebruikt)

In de cursus worden typisch gebruikt:

- 1-norm (max kolomsom): $$|\mathbf{A}|*1=|*max_j |*sum_i |*a{ij}|$$
- ∞-norm (max rijsom): $$|\mathbf{A}||*infty=|*max_i |*sum_j |*a{ij}|$$

Sleutel-eigenschap voor foutbounds: $$ |\mathbf{A}\mathbf{x}|\le |\mathbf{A}|;|\mathbf{x}|. $$

2.4.3 Condition number (cursusnotatie met cond)

Voor een gekozen norm: $$ \mathrm{cond}(\mathbf{A})=|\mathbf{A}|;|\mathbf{A}^{-1}|. $$ Specifiek bv.: $$ \mathrm{cond}|*infty(|*mathbf{A})=/|*mathbf{A}/|\infty;|\mathbf{A}^{-1}|_\infty. $$

Interpretatie: hoe hard kan het probleem kleine inputfouten **amplifiëren**?

2.4.4 Error estimation (perturbatie in $\mathbf{b}$)

Neem: $$ \mathbf{A}\mathbf{x}=\mathbf{b},\qquad \mathbf{A}\mathbf{x}'=\mathbf{b}+\Delta\mathbf{b}, \qquad \Delta\mathbf{x}=\mathbf{x}'-\mathbf{x}. $$ Dan: $$ \mathbf{A}\Delta\mathbf{x}=\Delta\mathbf{b} \quad\Rightarrow\quad \Delta\mathbf{x}=\mathbf{A}^{-1}\Delta\mathbf{b}. $$ Normeren geeft de standaard bound (idee zoals in de cursus): $$ \frac{|\Delta\mathbf{x}|}{|\mathbf{x}|} \le \mathrm{cond}(\mathbf{A}); \frac{|\Delta\mathbf{b}|}{|\mathbf{b}|}. $$

Interpretatie (examenvriendelijk): bij grote $\mathrm{cond}(\mathbf{A})$ kan een klein meetfoutje in $\mathbf{b}$ tot een grote fout in $\mathbf{x}$ leiden.

2.4.5 Residual (wat controleer je in de praktijk?)

Als je numeriek $\mathbf{x}{|text{num}}$ krijgt, definieer: $$ |mathbf{r}=|mathbf{A}|mathbf{x}{\text{num}}-\mathbf{b}. $$ Dan: $$ \mathbf{x}{|text{num}}-|mathbf{x}=-|mathbf{A}^{-1}|mathbf{r} |quad|Rightarrow|quad /|mathbf{x}{\text{num}}-\mathbf{x}|\le |\mathbf{A}^{-1}|;|\mathbf{r}|. $$

Belangrijke nuance: een kleine residual betekent pas "goede oplossing" als $\mathbf{A}$ niet slecht geconditioneerd is.

2.5 Software (SciPy/NumPy zoals in de cursus)

- Solve (algemeen): `scipy.linalg.solve(A, b)`
- LU (met pivoting): `scipy.linalg.lu(A)` of efficiënter `lu_factor` + `lu_solve`
- Cholesky (SPD): `scipy.linalg.cholesky(A, lower=True)` en dan twee substituties
- Normen/residuals: `scipy.linalg.norm(...)`

"Welke methode wanneer?" (de mapping fysica-probleem → methode)

1. **Algemeen dense $n\times n$ stelsel**
    - Kies: LU met partial pivoting (standaard in `solve`)
    - Waarom: robuust; kost $\sim n^3/3$ flops
2. **Zelfde $\mathbf{A}$, veel verschillende $\mathbf{b}$**
    - Kies: één keer LU-factorisatie, daarna herhaald `lu_solve`
    - Waarom: factorisatie duur, solve per $\mathbf{b}$ goedkoop ($\sim n^2$)
3. **$\mathbf{A}$ is symmetric positive definite**
    - Kies: Cholesky $\mathbf{A}=\mathbf{L}\mathbf{L}^\mathsf{T}$
    - Waarom: geen pivoting, halve opslag, $\sim n^3/6$ multiplications
4. **Iemand wil $\mathbf{A}^{-1}$ expliciet**
    - Meestal: niet doen
    - Waarom (cursus): $\sim n^3$ flops en minder nauwkeurig dan factorisatie + substitutie
5. **Je vertrouwt de oplossing niet**
    - Check: residual $\mathbf{r}$ én een idee van $\mathrm{cond}(\mathbf{A})$
    - Waarom: kleine residual is niet voldoende bij slechte conditioning

---

Links naar andere notebooks (zoals "type 1"-vragen)

- **Naar H1 Numerical limitations:** pivoting en “stabiele factorisaties” zijn een direct antwoord op rounding-error amplificatie.
- **Naar H3 Linear least squares:** SPD en Cholesky komen terug via normal equations; en de keuze LU vs QR vs SVD is precies “kost vs stabiliteit”.
- **Naar eigenwaarden/SVD later:** conditioning en (bijna) singulariteit worden daar “structureel zichtbaar” via singular values.

Hoofdstuk 3 — Linear Least Squares

In dit hoofdstuk los je problemen op van het type

- **overdetermined**: $m>n$ (meer vergelijkingen dan onbekenden), typisch bij data/metingen,
- waarbij het stelsel $\mathbf{A}\mathbf{x}=\mathbf{b}$ **geen exacte oplossing** heeft.

Je zoekt dan $\mathbf{x}$ die de **residual** klein maakt: $$ \mathbf{r}(\mathbf{x})=\mathbf{b}-\mathbf{A}\mathbf{x}, \qquad \min_{\mathbf{x}} |\mathbf{r}(\mathbf{x})|_2. $$

De examenfocus is bijna altijd: **welke methode wanneer en waarom** (kost vs stabiliteit), plus de connectie met H2 (factorisaties) en H4 (SVD/eigen).

3.1 Wat is een least squares probleem?

Overdetermined: $m>n$

$\mathbf{A}\in\mathbb{R}^{m\times n}$, $\mathbf{b}\in\mathbb{R}^{m}$, $\mathbf{x}\in\mathbb{R}^{n}$ met $m>n$.

Het LS-probleem is: $$ \min_{\mathbf{x}\in\mathbb{R}^n} |\mathbf{b}-\mathbf{A}\mathbf{x}|_2^2. $$

Waarom de **kwadraat**-norm? Omdat

- het wiskundig glad is (afleidbaar),
- en fysisch vaak overeenkomt met “energie”/“error energy”.

Geometrische interpretatie (zeer examenvriendelijk)

De kolommen van $\mathbf{A}$ spannen een subruimte op: $\mathcal{C}(\mathbf{A})$ (column space).

Least squares kiest $\hat{\mathbf{x}}$ zodat $\mathbf{A}\hat{\mathbf{x}}$ de **orthogonale projectie** is van $\mathbf{b}$ op $\mathcal{C}(\mathbf{A})$.

Dus in de optimum geldt: $$ \mathbf{r}=\mathbf{b}-\mathbf{A}\hat{\mathbf{x}} \perp \mathcal{C}(\mathbf{A}) \quad\Longleftrightarrow\quad \mathbf{A}^\mathsf{T}\mathbf{r}=\mathbf{0}. $$

3.2 Normal equations (klassiek, goedkoop, maar minder stabiel)

Uit $\mathbf{A}^\mathsf{T}\mathbf{r}=\mathbf{0}$ volgt: $$ \mathbf{A}^\mathsf{T}(\mathbf{b}-\mathbf{A}\hat{\mathbf{x}})=\mathbf{0} \quad\Rightarrow\quad (\mathbf{A}^\mathsf{T}\mathbf{A})\hat{\mathbf{x}}=\mathbf{A}^\mathsf{T}\mathbf{b}. $$

Dit heet de **normal equations**.

Wanneer werkt dit mooi?

Als $\mathbf{A}$ **full column rank** heeft (rank n), dan is $\mathbf{A}^\mathsf{T}\mathbf{A}$:

- symmetric,
- **positive definite**, dus je kunt **Cholesky** gebruiken (link met H2): $$ \mathbf{A}^\mathsf{T}\mathbf{A}=\mathbf{L}\mathbf{L}^\mathsf{T}. $$

Waarom is dit numeriek riskanter?

Belangrijk inzicht (cursusklassieker):

- In 2-norm geldt typisch: $$ \mathrm{cond}_2(\mathbf{A}^\mathsf{T}\mathbf{A})=\mathrm{cond}_2(\mathbf{A})^2. $$

Dus je **condition number kwadrateert** → rounding errors worden veel sterker versterkt dan bij methodes die rechtstreeks met $\mathbf{A}$ werken.

Samengevat

- **Pro**: relatief goedkoop (zeker als $m\gg n$) en eenvoudig; Cholesky is snel.
- **Con**: kan slecht zijn bij ill-conditioned $\mathbf{A}$; gevaarlijk bij bijna-rank-deficient.

3.3 QR-factorisatie (standaard “goede keuze” voor LS)

Je factoriseert: $$ \mathbf{A}=\mathbf{Q}\mathbf{R}, $$ waar

- $\mathbf{Q}\in\mathbb{R}^{m\times n}$ kolom-orthonormaal is (in “thin QR”),
- $\mathbf{R}\in\mathbb{R}^{n\times n}$ upper triangular.

Dan wordt: $$ |\mathbf{b}-\mathbf{A}\mathbf{x}|_2

|\mathbf{b}-\mathbf{Q}\mathbf{R}\mathbf{x}|_2. $$

Omdat $\mathbf{Q}$ orthonormaal is, behoudt het 2-normen onder transformatie met $\mathbf{Q}^\mathsf{T}$ (isometrie). Je krijgt: $$ \min_{\mathbf{x}}|\mathbf{Q}^\mathsf{T}\mathbf{b}-\mathbf{R}\mathbf{x}|_2. $$

In het full-rank geval los je gewoon het triangulaire stelsel: $$ \mathbf{R}\hat{\mathbf{x}}=\mathbf{Q}^\mathsf{T}\mathbf{b} $$ met back substitution.

Hoe bouw je QR numeriek stabiel?

De cursus benadrukt typisch **Householder transformations** (reflecties):

- ze zijn orthogonaal,
- ze zijn numeriek zeer stabiel,
- ze elimineren kolom per kolom tot je $\mathbf{R}$ hebt.

(Alternatief: Givens rotations, vooral handig als je sparseness wil behouden, maar Householder is de default in dense LS.)

Waarom QR beter is dan normal equations?

- QR vermijdt $\mathbf{A}^\mathsf{T}\mathbf{A}$ en dus het kwadrateren van de conditioning.
- Orthogonale transformaties zijn “rounding-friendly”.

3.4 SVD (duurste, maar “gouden standaard” voor diagnose en rank issues)

Singular value decomposition: $$ \mathbf{A}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}, $$ met

- $\mathbf{U}\in\mathbb{R}^{m\times m}$ orthogonaal,
- $\mathbf{V}\in\mathbb{R}^{n\times n}$ orthogonaal,
- $\mathbf{\Sigma}$ diagonaal (singular values $\sigma_1\ge \sigma_2\ge\dots\ge 0$).

Least squares oplossing via pseudoinverse

De (minimum-norm) LS-oplossing kan geschreven worden met de pseudoinverse: $$ \hat{\mathbf{x}}=\mathbf{A}^+\mathbf{b}

\mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^\mathsf{T}\mathbf{b}, $$ waar $\mathbf{\Sigma}^+$ de inverse neemt van niet-nul singular values (en 0 laat staan).

Waarom SVD zo nuttig?

- Detecteert **rank deficiency**: kleine σ_i betekenen “bijna afhankelijk”.
- Geeft de beste numerieke controle over ill-conditioning.
- Maakt regularisatie (zoals truncated SVD) conceptueel makkelijk.

Nadeel

- Kostbaar (grootste constante factoren). Gebruik SVD wanneer je *moet* (stabiliteit/diagnose), niet standaard voor alles.

3.5 Rank bepalen (rechtstreeks link met voorbeeldvraag Q6)

Je zoekt een **numerieke rank**: hoeveel richtingen zijn “significant” boven floating-point noise?

Methode 1: SVD (meest robuust)

- Compute $\sigma_1\ge\dots\ge\sigma_n$.

- Kies tolerance, bv. $\sigma_i > \tau$ met τ typisch gekoppeld aan machine precision en schaal (in de cursus vaak “relative threshold”).
- Rank = aantal singular values boven de threshold.

Voordeel: zeer betrouwbaar.
Nadeel: duur.

Methode 2: Rank-revealing QR (QR met pivoting)

Je doet kolompivoting: $$ \mathbf{A}\mathbf{P}=\mathbf{Q}\mathbf{R}, $$ waar $\mathbf{P}$ kolommen herschikt zodat diagonaal van $\mathbf{R}$ afneemt. Dan lees je rank af uit de grootte van $|r_{ii}|$ (met tolerance).

Voordeel: goedkoper dan SVD, vaak “goed genoeg”.
Nadeel: minder robuust dan SVD in lastige gevallen.

3.6 Kostvergelijking (rechtstreeks link met voorbeeldvraag Q8)

Voor $\mathbf{A}\in\mathbb{R}^{m\times n}$ met $m>n$:

1. **Normal equations**: vorm $\mathbf{A}^\mathsf{T}\mathbf{A}$ en $\mathbf{A}^\mathsf{T}\mathbf{b}$ en los op (Cholesky)

- typisch goedkoopste in flops,
- maar slechtste in stabiliteit (conditioning kwadrateert).

2. **QR via Householder**

- iets duurder dan normal equations,
- veel stabieler.

3. **SVD**

- duurst,
- meest robuust (zeker bij rank deficiency / ill-conditioning).

Dus voor $\mathbf{A}$ van grootte $2n\times n$ (zoals in de voorbeeldvraag): $$ \text{Normal equations (min work)} ;<; \text{QR (middel)} ;<; \text{SVD (max work)}. $$

3.7 “Welke methode wanneer?” (de beslisboom die je mondeling moet kunnen verdedigen)

Kies normal equations + Cholesky als…

- je $\mathbf{A}$ redelijk **goed geconditioneerd** is,
- je puur snelheid wil,
- je weet/verwacht dat rank issues geen rol spelen,
- en je liefst $\mathbf{A}^\mathsf{T}\mathbf{A}$ toch al nodig hebt (maar wees eerlijk over stabiliteit).

Argument: goedkoop; SPD $\Rightarrow$ Cholesky snel (link H2).

Kies QR (Householder) als default voor LS als…

- je een “normale” LS-fit doet op meetdata,
- je stabiliteit wil zonder extreme kost,
- je geen expliciete rank deficiency verwacht.

Argument: orthogonale transformaties zijn stabiel; geen conditioning-kwadratering.

Kies SVD als…

- je rank deficiency vermoedt (multicollineariteit, bijna lineair afhankelijke kolommen),
- je condition number groot is,
- je een betrouwbare numerieke rank wil,
- je een diagnose/regularisatie nodig hebt.

Argument: singular values geven meteen inzicht + beste numerieke robuustheid.

Links naar andere hoofdstukken / notebooks (type-1 examenvragen)

Link met Hoofdstuk 2 (LU vs QR, precies zoals voorbeeldvraag Q3)

- **LU** is de standaard voor square $\mathbf{A}\mathbf{x}=\mathbf{b}$ (direct solve).
- **QR** is de standaard voor least squares omdat je de LS-structuur benut en stabiliteit wint.
- Je *kan* een square systeem ook met QR oplossen: $\mathbf{A}=\mathbf{Q}\mathbf{R}\Rightarrow \mathbf{R}\mathbf{x}=\mathbf{Q}^\mathsf{T}\mathbf{b}$.
    - Voordeel: betere numerieke stabiliteit in sommige gevallen (orthogonale transformaties).
    - Nadeel: meestal duurder dan LU voor pure square solve.

Link met Hoofdstuk 1 (numerical limitations)

Normal equations zijn het klassieke voorbeeld waar je *een wiskundig nette stap* zet die numeriek slecht is omdat cond kwadrateert.

Link met eigenwaarden/SVD notebook

SVD is de “rank/conditioning lens” en verklaart waarom sommige LS-problemen inherent gevoelig zijn.

Link met optimization notebook

Least squares is een speciaal geval van convex optimization: $$ \min_{\mathbf{x}} |\mathbf{b}-\mathbf{A}\mathbf{x}|_2^2. $$ Later zie je dezelfde ideeën terug (gradients/Hessians, condition numbers, step choices).

Hoofdstuk 4 — Eigenvalue Problems

In dit hoofdstuk draait alles rond het eigenprobleem $$ \mathbf{A}\mathbf{x}=\lambda \mathbf{x}, $$ waar λ een **eigenwaarde** is en $\mathbf{x}\neq \mathbf{0}$ een bijhorende **eigenvector**.

De examenkern is (zoals altijd in deze cursus): **welke methode kies je wanneer, en waarom (kost + stabiliteit + doel: 1 eigenwaarde of allemaal)?**
En: hoe dit terugkoppelt naar H2 (lineaire stelsels oplossen) en H3 (SVD/conditioning/rank).

4.1 Introductie, concept en nuttige eigenschappen

4.1.1 Characteristic polynomial (theorie, maar numeriek meestal NIET doen)

Uit $\mathbf{A}\mathbf{x}=\lambda \mathbf{x}$ volgt $$ (\mathbf{A}-\lambda \mathbf{I})\mathbf{x}=\mathbf{0}. $$ Een niet-triviale oplossing bestaat enkel als $$ \det(\mathbf{A}-\lambda \mathbf{I})=0. $$ De polynoom $$ p(\lambda)=\det(\mathbf{A}-\lambda \mathbf{I}) $$ heet de **characteristic polynomial**; zijn wortels zijn de eigenwaarden.

Belangrijk (cursus zegt dit expliciet): eigenwaarden vinden via de wortels van $p(\lambda)$ is *geen goede numerieke strategie* voor matrices van niet-triviale grootte, o.a. omdat

- de coëfficiënten van $p(\lambda)$ extreem gevoelig kunnen zijn voor kleine perturbaties in $\mathbf{A}$,
- rounding errors in $p(\lambda)$ de wortels volledig kunnen verpesten,
- wortels van een hoge-graads polynoom vinden zelf al een lastig numeriek probleem is.

Conclusie: in praktijk gebruik je iteratieve methodes (power/inverse/Rayleigh/QR) of library-routines.

4.1.2 Properties and transformations (dit stuurt de algoritmes)

De cursus geeft een lijst “eigenwaarden blijven hetzelfde, of transformeren voorspelbaar”:

- **Symmetric/Hermitian:**
  Als $\mathbf{A}$ symmetric/Hermitian is, dan zijn alle eigenwaarden **reëel**.

- **Shift:** als $\mathbf{A}\mathbf{x}=\lambda\mathbf{x}$ en σ is een scalair: $$ (\mathbf{A}-\sigma\mathbf{I})\mathbf{x}=(\lambda-\sigma)\mathbf{x}. $$ Eigenwaarden schuiven met σ, eigenvectoren blijven hetzelfde.

- **Inversion:** $\mathbf{A}^{-1}$ heeft dezelfde eigenvectoren, eigenwaarden worden $$ \lambda \mapsto \frac{1}{\lambda}. $$

- **Powers:** $\mathbf{A}^k$ heeft dezelfde eigenvectoren, eigenwaarden worden $$ \lambda \mapsto \lambda^k. $$

- **Polynomials:** voor een polynoom $p(t)$: $$ p(\mathbf{A})\mathbf{x}=p(\lambda)\mathbf{x}. $$ Dus eigenwaarden transformeren als $\lambda\mapsto p(\lambda)$, eigenvectoren blijven die van $\mathbf{A}$.

- **Similarity:** $\mathbf{B}$ is similar aan $\mathbf{A}$ als er een invertibele $\mathbf{T}$ bestaat zodat $$ \mathbf{B}=\mathbf{T}^{-1}\mathbf{A}\mathbf{T}. $$ Dan hebben $\mathbf{A}$ en $\mathbf{B}$ **dezelfde eigenwaarden**, en eigenvectoren worden systematisch “meegetransformeerd”.

Waarom dit belangrijk is: QR-iteratie en veel “matrix-reducties” zijn gebouwd op similarity-transformaties (eigenwaarden blijven identiek, maar de matrix wordt eenvoudiger).

4.2 Eigenwaarden en eigenvectoren berekenen (methodes)

4.2.1 Power iteration (dominante eigenwaarde)

Doel: schat de **dominante** eigenwaarde (grootste modulus) en eigenvector.

Idee:

- Kies een willekeurige $\mathbf{x}_0\neq \mathbf{0}$,
- herhaal $$ \mathbf{x}*{k}=|*mathbf{A}|*mathbf{x}*{k-1}, $$
- normaliseer elke stap (bv. met $|\cdot|_\infty$ of $|\cdot|_2$) om overflow/onderflow te vermijden.

Waarom dit werkt (cursusproof in woorden): schrijf $$ \mathbf{x}*0=|*sum{j=1}^n \alpha_j \mathbf{v}_j, $$ met $\mathbf{v}_j$ eigenvectoren. Dan $$ \mathbf{x}_k=\mathbf{A}^k\mathbf{x}*0=|*sum{j=1}^n \alpha_j \lambda_j^k \mathbf{v}_j. $$ Als er één unieke eigenwaarde λ_1 is met maximale modulus en $\alpha_1\neq 0$, dan domineert die term en convergeert de richting van $\mathbf{x}_k$ naar $\mathbf{v}_1$.

Convergentiesnelheid (intuïtie): $$ \text{foutfactor} \sim \left|\frac{\lambda_2}{\lambda_1}\right|^k. $$ Dus traag als $|\lambda_2|\approx|\lambda_1|$.

Wanneer gebruiken?

- Je wil alleen de grootste eigenwaarde/eigenvector.
- Matrix is groot en je wil iets heel simpels.
- Je aanvaardt lineaire (soms trage) convergentie.

Wanneer niet?

- Je wil meerdere eigenwaarden.
- Dominante eigenwaarde is niet uniek of spectrum ligt “dicht opeen”.

4.2.2 Inverse iteration (kleinste eigenwaarde, of eigenwaarde dicht bij een shift)

De cursus: inverse iteration convergeert naar de eigenvector van de **grootste eigenwaarde van $\mathbf{A}^{-1}$**, dus naar de eigenvector van de **kleinste** eigenwaarde van $\mathbf{A}$.

Praktisch doe je niet expliciet $\mathbf{A}^{-1}$, je lost per iteratie een lineair stelsel op: $$ \mathbf{A}\mathbf{y}*k=|*mathbf{x}*{k-1},\qquad \mathbf{x}_k=\frac{\mathbf{y}_k}{|\mathbf{y}_k|}. $$

Shifted inverse iteration (superbelangrijk): wil je een eigenwaarde nabij σ? Gebruik $$ (\mathbf{A}-\sigma\mathbf{I})\mathbf{y}*k=|*mathbf{x}*{k-1}. $$ Dan convergeer je naar de eigenvector van de eigenwaarde van $\mathbf{A}$ die het **dichtst bij σ** ligt.

Opmerking uit de cursus: bij shifted inverse iteration krijg je in feite de inverse van de **geshiftte** eigenwaarde; om terug naar de eigenwaarde van $\mathbf{A}$ te gaan:

- neem het reciproke,
- tel σ terug erbij.

Kost/implementatie-inzicht (link met H2):

- Als σ vast blijft: factoriseer $\mathbf{A}-\sigma\mathbf{I}$ één keer (LU) en hergebruik in elke iteratie.
- Dus “duur + veel goedkope solves” (zoals H2).

Wanneer gebruiken?

- Je wil een eigenwaarde/eigenvector **in een specifiek gebied** van het spectrum.
- Je hebt een redelijke shift (bv. uit fysisch inzicht, of uit een ruwe schatting).

4.2.3 Rayleigh quotient iteration (snelle convergentie met slimme shift)

De cursus linkt dit aan een LS-probleem: $$ \mathbf{x},\lambda \cong \mathbf{A}\mathbf{x}. $$ De beste LS-schatting van λ is de **Rayleigh quotient**: $$ \rho(\mathbf{x})=\frac{\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}}{\mathbf{x}^\mathsf{T}\mathbf{x}} \quad (\text{voor reële } \mathbf{A}, \mathbf{x}). $$

Rayleigh quotient iteration gebruikt deze als shift:

1. $\sigma_k=\rho(\mathbf{x}_k)$
2. los $$ (\mathbf{A}-\sigma_k\mathbf{I})\mathbf{y}_k=\mathbf{x}_k $$
3. normaliseer: $\mathbf{x}_{k+1}=\mathbf{y}_k/|\mathbf{y}_k|$

Waarom dit “episch goed” kan zijn:

- Als je al een redelijke eigenvector-guess hebt, gaat dit vaak veel sneller dan gewone power/inverse.
- (Klassieke extra kennis: voor symmetric matrices heb je vaak zeer snelle, zelfs kubische convergentie wanneer je dichtbij zit.)

Wanneer gebruiken?

- Je wil heel snel “finetunen” naar een eigenpaar.
- Je hebt al een goede startvector (bv. van inverse iteration of fysische mode-vorm).

4.2.4 Deflation (meerdere eigenwaarden na elkaar, maar met valkuilen)

Deflation probeert na het vinden van $(\lambda_1,\mathbf{x}_1)$ een nieuwe matrix te maken waarin die eigenwaarde “verwijderd” is.

De cursus toont de constructie met een vector $\mathbf{u}_1$ zodat $$ \mathbf{u}_1^\mathsf{T}\mathbf{x}*1=|lambda_1, $$ en dan $$ |mathbf{A}*{\text{deflated}}=\mathbf{A}-\mathbf{x}_1\mathbf{u}_1^\mathsf{T}. $$

Daarna kan je opnieuw power iteration doen om de “volgende” eigenwaarde te vinden.

Cursus-waarschuwing: deflation wordt snel

- omslachtig,
- numeriek minder accuraat,
- en in de praktijk gebruik je betere methodes om veel eigenwaarden te vinden.

Wanneer toch nuttig?

- Klein probleem, didactisch.
- Je wil “een paar” eigenwaarden en accepteert dat je later inverse iteration met shifts nodig hebt.

4.2.5 QR iteration (workhorse voor alle eigenwaarden)

De cursus noemt dit “in practice, the fastest and most used method” voor alle eigenwaarden (dense case).

QR-iteratie definieert een reeks:

1. QR-factorisatie: $$ \mathbf{A}_m=\mathbf{Q}_m\mathbf{R}_m $$
2. Update: $$ \mathbf{A}_{m+1}=\mathbf{R}_m\mathbf{Q}_m $$

Cruciale eigenschap (waarom eigenwaarden behouden blijven): Omdat $\mathbf{Q}*m$ orthogonaal is, $$ |mathbf{A}* {m+1}=\mathbf{R}_m\mathbf{Q}_m

---

\mathbf{Q}_m^\mathsf{T}\mathbf{A}_m\mathbf{Q}*m, $$ dus $|mathbf{A}*{m+1}$ is **similar** aan $\mathbf{A}_m$ → eigenwaarden blijven dezelfde.

De iteratie convergeert naar een (quasi-)triangulaire vorm; de diagonaal convergeert naar de eigenwaarden.

Shifts in QR iteration (versnellen): Neem bv. als shift de rechtsonder entry: $$ \mu_m=(\mathbf{A}*m)*{nn}. $$ Doe QR op $\mathbf{A}*m-|mu_m|mathbf{I}$ en zet daarna shift terug: $$ |mathbf{A}* {m+1}=\mathbf{R}_m\mathbf{Q}_m+\mu_m\mathbf{I}. $$ Cursus: dit laat off-diagonale elementen sneller naar 0 gaan.

Wanneer gebruiken?

- Je wil **alle** eigenwaarden (en eventueel eigenvectoren) van een dense matrix.
- Je wil een beproefde, robuuste methode (library).

4.3 Singular Value Decomposition berekenen (koppeling met eigenwaarden)

De cursus herhaalt: $$ \mathbf{A}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathsf{T}, $$ met $\mathbf{U}$ orthogonaal ($m\times m$), $\mathbf{V}$ orthogonaal ($n\times n$), en $\mathbf{\Sigma}$ diagonaal ($m\times n$) met $\sigma_i\ge 0$.

Belangrijke link (ook in H3):

- σ_i^2 zijn eigenwaarden van $\mathbf{A}^\mathsf{T}\mathbf{A}$,

- right singular vectors $\mathbf{v}_i$ zijn eigenvectoren van $\mathbf{A}^\mathsf{T}\mathbf{A}$,
- left singular vectors $\mathbf{u}_i$ zijn eigenvectoren van $\mathbf{A}\mathbf{A}^\mathsf{T}$, en $$ \mathbf{A}\mathbf{v}_i=\sigma_i \mathbf{u}_i. $$

In het symmetrische voorbeeld uit de cursus geldt zelfs $\mathbf{U}=\mathbf{V}$.

Wanneer denk je “SVD” i.p.v. “eigen”?

- Niet-vierkante matrix.
- Rank/conditioning/least-squares diagnose (H3).
- Je wil robuustheid i.p.v. pure snelheid.

4.4 Software (SciPy) — wat kies je wanneer?

Cursusboodschap (belangrijk): QR-iteratie was lang de standaard; nieuwere methodes (divide-and-conquer, RRR) zijn vaak sneller voor **alle eigenvectoren**, maar QR heeft een lange “reliability track record”. In SciPy is `linalg.eig` de meest algemene methode.

Praktisch:

- `scipy.linalg.eig`
  Algemeen eigenprobleem (complex mogelijk), gebruikt QR-achtige aanpak.
- Voor matrices met speciale structuur zijn er snellere routines (cursus geeft o.a. tridiagonal symmetric varianten).
- Voor SVD:
  - `scipy.linalg.svd`
  - `scipy.linalg.svdvals`

Vuistregels

- Symmetric/Hermitian? Gebruik de symmetric/Hermitian routines (sneller + reële eigenwaarden + stabieler).
- Tridiagonal symmetric? Gebruik tridiagonal routine (nog sneller).
- Algemeen? `eig`.

4.5 Physics Example — spring-and-mass system (waarom eigenwaarden fysisch tellen)

De cursus zet een gekoppeld veer-massa systeem om naar een matrixvorm.

Voor twee massa’s met veren krijg je (in tweede orde) iets van de vorm: $$ \ddot{\mathbf{x}}=\mathbf{A}\mathbf{x}, $$ met $\mathbf{x}=[x_1,x_2]^\mathsf{T}$ en een matrix $\mathbf{A}$ die afhangt van k en M_1,M_2 (in de cursus staat expliciet een 2×2 matrix met termen zoals $-2k/M_1$, k/M_1, etc.).

De eigenwaarden λ komen uit $$ \det(\mathbf{A}-\lambda \mathbf{I})=0. $$

Fysische interpretatie (normale modes):

- Voor een stabiel veer-systeem verwacht je oscillaties met ω.
- Typisch geldt dan $\lambda=-\omega^2$ (teken hangt af van hoe $\mathbf{A}$ exact gedefinieerd is).
- Eigenvectoren geven de **mode-vormen** (hoe massa's relatief bewegen).
- Eigenwaarden geven de **frequenties** (via $\omega=\sqrt{-\lambda}$ als $\lambda<0$).

Dit is precies waarom eigenproblemen zo vaak opduiken in fysica: *coupled linear dynamics = modes = eigenvectors*.

"Welke methode wanneer?" — examengerichte beslisregels

Je wil één dominante eigenwaarde/eigenvector

- **Power iteration**
- Waarom: goedkoop per iteratie (matrix-vector product), eenvoudig.
- Valkuil: traag als spectrum dicht is; faalt als dominante eigenwaarde niet uniek is.

Je wil de kleinste eigenwaarde of eentje dicht bij een schatting σ

- **(Shifted) inverse iteration**
- Waarom: door $(\mathbf{A}-\sigma\mathbf{I})^{-1}$ wordt "dichtst bij σ" dominant.
- Kost: per iteratie een solve; met vaste shift kan je LU hergebruiken.

Je hebt al een goede eigenvector-guess en wil razendsnel convergeren

- **Rayleigh quotient iteration**
- Waarom: shift wordt automatisch "best mogelijke" eigenwaarde-schatting voor de huidige vector.

Je wil meerdere eigenwaarden, maar het probleem is klein/onderwijs

- **Deflation** (met gezond wantrouwen)
- Waarom: conceptueel oké, maar wordt numeriek snel minder mooi.

Je wil alle eigenwaarden (dense matrix, standaard library-werk)

- **QR iteration / `scipy.linalg.eig`**
- Waarom: robuust, standaard, convergent naar (quasi-)triangulaire vorm waarvan de diagonaal de eigenwaarden geeft.
- Gebruik shifts voor versnelling (bibliotheken doen dit slim).

Links met andere hoofdstukken (type-1 examenvragen)

- **Link met H2 (lineaire stelsels):** inverse/Rayleigh iteraties bestaan uit "herhaald een lineair stelsel oplossen" → LU/Cholesky keuzes bepalen runtime en stabiliteit.
- **Link met H3 (least squares & SVD):** singular values via eigenwaarden van $\mathbf{A}^\mathsf{T}\mathbf{A}$; rank/conditioning komt rechtstreeks uit SVD.
- **Link met H1 (numerical limitations):** "characteristic polynomial roots" is het poster-child van een wiskundig correcte maar numeriek slechte route.

Hoofdstuk 5 — Nonlinear equations

We willen een vergelijking oplossen van de vorm $$ f(x)=0 $$ (of in meerdere dimensies: $\mathbf{f}(\mathbf{x})=\mathbf{0}$). Het grote verschil met lineaire stelsels is dat:

- er **meerdere** oplossingen kunnen zijn (of geen),
- het **vinden** van een oplossing vaak iteratief gaat,
- stabiliteit/stopcriteria niet triviaal zijn.

De examenkern (zie voorbeeldvragen): **hoe verbeteren methodes elkaar conceptueel** en **wanneer kies je wat**.

5.1 Introduction

Niet-lineaire nulpunten duiken overal op in fysica:

- evenwichtscondities (krachtbalans, momentbalans),
- energie-minima (zelfde als optimalisatie: $\nabla \phi(\mathbf{x})=\mathbf{0}$),
- impliciete tijdstappen (ODE/PDE impliciet $\Rightarrow$ telkens een nonlineair solve-probleem).

5.2 Number of solutions

Bracketing (1D)

Als f continu is en je vindt $a<b$ met $$ f(a),f(b)<0, $$ dan bestaat er minstens één wortel in (a,b) (Intermediate Value Theorem).

Waarom dit goud waard is: bracketing-methodes zijn *robuust* (ze garanderen een wortel zolang de continuïteit aanneemt).

Meerdere wortels

- Als f niet-monotoon is, kunnen er meerdere nulpunten zijn.
- In de praktijk: scan/plot om intervallen te vinden waar f van teken verandert.

5.3 Sensitivity (conditioning van een wortel)

Neem een eenvoudige wortel x^* met $f(x^*)=0$ en $f'(x^*)\neq 0$. Een kleine perturbatie δf in de functie kan een wortelshift δx geven met (eerste orde): $$ 0 = f(x^*+\delta x)+\delta f \approx f(x^*)+f'(x^*)\delta x + \delta f \quad\Rightarrow\quad \delta x \approx -\frac{\delta f}{f'(x^*)}. $$

Interpretatie:

- Als $|f'(x^*)|$ klein is (platte functie), is de wortel **slecht geconditioneerd**: minieme fouten → grote δx.

- Bij een meervoudige wortel (typisch $f'(x^*)=0$) wordt het nog gevoeliger én convergentie wordt trager.

Link met H1/H2: zelfs een perfecte iteratiemethode kan niet “beter zijn dan de conditioning” van het probleem.

5.4 Convergence rates and stopping criteria

Convergentie-orde

Je meet vaak: $$ |e_{k+1}| \approx C |e_k|^p, \quad\text{met } e_k=x_k-x^*. $$

- $p=1$: lineair
- $p=2$: kwadratisch (Newton)
- $p\approx 1.618$: secant

Stopcriteria (praktisch én examenvriendelijk)

Gebruik zelden maar één criterium; combineer typisch:

- **Interval** klein: $|b-a|\le \text{tol}$
- **Stap** klein: $|x_{k+1}-x_k|\le \text{tol}(1+|x_{k+1}|)$
- **Residual** klein: $|f(x_k)|\le \text{tol}$

Let op: kleine residual garandeert geen kleine fout als $f'(x^*)$ klein is (conditioning).

5.5 Solving nonlinear equations in one dimension

(A) Bisection (bracketing, altijd veilig)

Start met $[a,b]$ met $f(a)f(b)<0$. Neem $$ m=a+\frac{b-a}{2}. $$ Kies het subinterval waar het teken verandert.

Waarom die lijn exact zo staat (zoals in examenvraag Q7):
$m=a+(b-a)/2$ vermijdt kleine “numerieke” verrassingen en is de standaard mid-point (zelfde als $(a+b)/2$ maar explicieter in floating-point redenering).

- Convergentie: lineair, fout halveert per iteratie
- Pro: gegarandeerde convergentie (bij continuïteit)
- Con: traag

Wanneer gebruiken?

- Als je zekerheid wil (fysica-probleem waar “geen gekke dingen” mogen gebeuren).
- Als f evalueren goedkoop is.
- Als je initieel enkel een bracket hebt, geen goede startwaarde.

(B) Secant method (open method, sneller, geen afgeleide nodig)

Gebruik twee startpunten x_{k-1},x_k en benader f' door een secant: $$ x_{k+1} = x_k - f(x_k)\frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})}. $$

- Convergentie: superlineair ($p\approx 1.618$) bij “mooie” wortels
- Pro: sneller dan bisection; geen f'
- Con: geen garantie op behoud van bracket; kan divergeren

Conceptuele verbetering t.o.v. bisection (zoals Q2.1):

- bisection gebruikt alleen tekeninfo; secant gebruikt *lokaal lineair model* van f → grotere, “slimmere” stappen.

(C) Newton’s method (afgeleide, heel snel als je dichtbij zit)

Iteratie: $$ x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}. $$

- Convergentie: kwadratisch bij eenvoudige wortel en goede start
- Pro: extreem efficiënt als f' beschikbaar is en start goed is
- Con: vraagt f'; kan divergeren of naar verkeerde wortel schieten; faalt bij $f'(x_k)\approx 0$

Conceptuele verbetering t.o.v. secant (zoals Q2.2):

- secant benadert f'; Newton gebruikt de echte afgeleide → sneller en betrouwbaarder *als* f' goed berekenbaar is.

(D) Inverse interpolation / inverse quadratic interpolation (IQI)

In plaats van $f(x)$ te interpoleren en nul te zoeken, interpoleer je **x als functie van f**: $$ x \approx q(f). $$ Dan is de wortel meteen $x^*\approx q(0)$.

- Pro: vaak sneller dan secant; gebruikt geen afgeleiden
- Con: minder robuust; implementatie complexer

Waarom IQI voor nulpunten, maar “gewone” interpolatie voor minima (link met voorbeeldvraag Q4):

- Bij een nulpunt is $f=0$ het *doel* → het is logisch om rechtstreeks $x(f=0)$ te schatten.
- Bij een minimum heb je typisch $f'(x)=0$ of je wil x minimaliseren → “parabool-fit” in x-ruimte is natuurlijker (je modelleert $f(x)$ als parabool en neemt het toppunt).

(E) Hybride topkeuze: Brent-achtige methodes

In de praktijk wil je:

- de robuustheid van bracketing,
- de snelheid van secant/IQI.

Brent’s methode combineert bracketing + secant + IQI en valt terug op bisection als het gevaarlijk wordt.

Wanneer kiezen?

- Default in productie: “zo goed als altijd werkt” zonder veel tuning.

5.6 Systems of nonlinear equations

We lossen: $$ \mathbf{f}(\mathbf{x})=\mathbf{0},\qquad \mathbf{f}:\mathbb{R}^n\to\mathbb{R}^n. $$

Newton voor systemen

Lineariseer rond $\mathbf{x}_k$: $$ \mathbf{f}(\mathbf{x}_k+\Delta\mathbf{x})\approx \mathbf{f}(\mathbf{x}_k)+\mathbf{J}(\mathbf{x}*k)|Delta|mathbf{x}, $$ waar $|mathbf{J}$ de Jacobiaan is: $$ J*{ij}(\mathbf{x})=\frac{\partial f_i}{\partial x_j}. $$

Stel $\mathbf{f}(\mathbf{x}_{k+1})=\mathbf{0}$: $$ \mathbf{J}(\mathbf{x}_k)\Delta\mathbf{x}=-\mathbf{f}(\mathbf{x}*k),|qquad |mathbf{x}*{k+1}=\mathbf{x}_k+\Delta\mathbf{x}. $$

Link met H2: elke Newton-stap is een lineair stelsel solve. Keuze LU/Cholesky/sparse bepaalt je totale kost.

Praktische stabilisatie

- **Damping/line search:** $\mathbf{x}_{k+1}=\mathbf{x}_k+\alpha\Delta\mathbf{x}$ met $0<\alpha\le 1$ om overshoots te vermijden.
- **Goede start** is essentieel (Newton is lokaal).

Welke methode wanneer?

- **Je hebt alleen een bracket en wil zekerheid:** bisection of Brent.
- **Je hebt geen afgeleide maar wil sneller:** secant of Brent.
- **Je hebt f' (of Jacobiaan) en een goede start:** Newton.
- **Je wil “best of both worlds” zonder te veel gedoe:** Brent (1D) of gedempte Newton (multi-D).

Links met andere hoofdstukken

- H2: Newton voor systemen = herhaald lineaire stelsels oplossen.
- H6: optimalisatie gebruikt vaak solves van $\nabla \phi(\mathbf{x})=\mathbf{0}$ → nonlineair.
- H9/H10: impliciete ODE/PDE-stappen geven nonlineaire systemen.

Hoofdstuk 6 — Optimization

Optimalisatie: vind $\mathbf{x}$ die een functie minimaliseert/maximaliseert: $$ \min_{\mathbf{x}} \ \phi(\mathbf{x}) \quad\text{of}\quad \max_{\mathbf{x}} \ \phi(\mathbf{x}). $$

Examenkern:

- link met H5: optimaliteit $\Rightarrow \nabla\phi(\mathbf{x})=\mathbf{0}$ is een nonlineair stelsel,
- link met H3: least squares is een speciaal geval,

- vooral: **wanneer welke methode** (kost, afgeleiden, convexiteit, dimension, constraints).

6.1 Introduction

Optimalisatie komt in fysica vaak als:

- energie-minimum (stabiel evenwicht),
- parameterfitting (model ↔ data),
- control/trajectory problems (project).

6.2 Optimality conditions

1D

- Stationair punt: $\phi'(x^*)=0$.
- Minimum: $\phi''(x^*)>0$ (lokaal).

Multi-D

- Stationair punt: $\nabla \phi(\mathbf{x}^*)=\mathbf{0}$.
- Hessiaan: $$ \mathbf{H}(\mathbf{x})=\nabla^2 \phi(\mathbf{x}) $$ Minimum als $\mathbf{H}$ positief definiet is.

Link met H5: “solve $\nabla\phi=0$” is exact een nonlineair systeem.

6.3 Optimization in one dimension

(A) Bracketing van een minimum

Je zoekt een interval (a,b,c) met $a<b<c$ en $\phi(b)<\phi(a),\phi(c)$.

(B) Golden section search (robust, geen afgeleiden)

Gebruikt een vaste ratio zodat je één functie-evaluatie hergebruikt per iteratie.

- Pro: gegarandeerde krimp van interval; enkel ϕ nodig
- Con: lineaire convergentie; relatief veel evaluaties

(C) Parabolische interpolatie (sneller, maar kan misgaan)

Fit een parabool door drie punten en neem het toppunt.

- Pro: vaak snel bij “gladde” minima
- Con: kan instabiel zijn zonder safeguards

(D) Brent voor minima

Combineert golden section (veilig) met paraboolstappen (snel). **Dit is vaak de default** als je 1D zonder afgeleiden doet.

6.4 Multidimensional unconstrained optimization

Hier is de keuze vooral: heb je afgeleiden (grad/Hessiaan) en hoe duur is een functie-evaluatie?

(A) Direct search: Nelder–Mead (zoals examenvraag Q5)

Werkt met een simplex in $\mathbb{R}^n$ (n+1 punten) en gebruikt:

- reflectie,
- expansie,
- contractie,
- shrink.

Waarom convergeert het vaak traag?

- Het gebruikt enkel functiewaarden (geen richtinginfo),
- simplex-deformaties kunnen “slenteren” door vlakke valleien,
- geen echte curvature-informatie.

Hoe verbeteren?

- Gebruik gradient-based methodes (BFGS, CG, Newton),
- of combineer: Nelder–Mead om “in de buurt” te komen, daarna BFGS/Newton.

Wanneer toch nuttig?

- Als ϕ noisy/niet-differentieerbaar is,
- als gradients niet beschikbaar of onbetrouwbaar zijn,
- als dimension klein is en evaluaties goedkoop zijn.

(B) Gradient descent (goedkoop per iteratie, kan veel iteraties vragen)

Stap: $$ \mathbf{x}_{k+1}=\mathbf{x}_k - \alpha_k \nabla \phi(\mathbf{x}_k). $$

- Pro: simpel; enkel gradient nodig
- Con: sterk afhankelijk van step size; traag bij slechte conditioning (langgerekte dalen)

Link met H3/H1: conditioning van Hessiaan bepaalt hoe “scheef” het dal is → hoe traag GD is.

(C) Newton’s method (snel, maar duur per iteratie)

Newton-stap: $$ \mathbf{H}(\mathbf{x}_k)\Delta\mathbf{x}=-\nabla\phi(\mathbf{x}*k), |qquad |mathbf{x}* {k+1}=\mathbf{x}_k+\Delta\mathbf{x}. $$

- Pro: zeer snel dichtbij minimum (curvature correct)
- Con: Hessiaan bouwen/inverteren is duur; solve kost typisch $\mathcal{O}(n^3)$ (dense); kan naar saddle gaan als Hessiaan niet PD is

Praktisch vaak met:

- line search of trust region,
- Hessiaan-regularisatie.

(D) Quasi-Newton (BFGS / L-BFGS) — vaak de “sweet spot”

BFGS bouwt een benadering $\mathbf{B}_k\approx \mathbf{H}^{-1}$ uit gradient-info.

- Pro: bijna Newton-snelheid zonder Hessiaan; zeer populair
- L-BFGS: geheugen-slim voor grote n

(E) Conjugate Gradient (CG) voor grote problemen

Voor kwadratische doelen (of bijna) en grote n:

- Pro: lage memory; goede scaling
- Con: minder robuust buiten convex/kwadratisch gebied

6.5 Non-linear Least Squares (koppeling met H3)

Doel: $$ \min_{\mathbf{x}} \ |\mathbf{r}(\mathbf{x})|_2^2, \qquad \mathbf{r}:\mathbb{R}^n\to\mathbb{R}^m. $$ Dit is de niet-lineaire versie van H3.

(A) Gauss–Newton

Lineariseer residual: $$ \mathbf{r}(\mathbf{x}+\Delta\mathbf{x})\approx \mathbf{r}(\mathbf{x})+\mathbf{J}(\mathbf{x})\Delta\mathbf{x}. $$ Minimaliseer dan de lineaire LS: $$ \min_{\Delta\mathbf{x}} \ |\mathbf{r}(\mathbf{x})+\mathbf{J}\Delta\mathbf{x}|_2^2. $$ Leidt tot normal equations: $$ (\mathbf{J}^\mathsf{T}\mathbf{J})\Delta\mathbf{x}=-\mathbf{J}^\mathsf{T}\mathbf{r}. $$

Link met H3: dit is exact “LS in elke iteratie” → QR/SVD argumenten blijven gelden.

(B) Levenberg–Marquardt (demping/regularisatie)

Voegt een term toe: $$ (\mathbf{J}^\mathsf{T}\mathbf{J}+\lambda \mathbf{I})\Delta\mathbf{x}=-\mathbf{J}^\mathsf{T}\mathbf{r} $$ om stabiliteit te verbeteren (vooral als $\mathbf{J}^\mathsf{T}\mathbf{J}$ slecht geconditioneerd is).

6.6 Constrained optimization

Constraints kunnen zijn:

- bounds: $l_i\le x_i\le u_i$,
- equality: $g(\mathbf{x})=0$,
- inequality: $h(\mathbf{x})\le 0$.

Conceptueel:

- Lagrange multipliers / KKT-voorwaarden (voor inzicht),

- numeriek: penalty/barrier of gespecialiseerde algoritmes (SLSQP, trust-constr, etc.).

Welke methode wanneer?

- **Geen afgeleiden / noisy functie / klein n:** Nelder–Mead.
- **Glad + gradients beschikbaar + medium n:** (L-)BFGS is vaak top.
- **Zeer klein n + Hessiaan beschikbaar:** Newton (met line search/trust region).
- **Niet-lineaire LS (model fit):** Gauss–Newton / Levenberg–Marquardt.
- **Constraints:** kies een constrained solver (bounds: L-BFGS-B; algemene constraints: SLSQP of trust-constr).

Links met andere hoofdstukken

- H5: optimaliteit = nulpunten van $\nabla\phi$.
- H3: (non-)linear least squares als kernsubroutine.
- H2: Newton/Gauss–Newton stappen lossen lineaire systemen op; factorisatie-keuzes bepalen runtime.

Hoofdstuk 7 — Interpolation

Interpolatie: gegeven discrete data (x_i,y_i), construeer een functie $p(x)$ zodat $$ p(x_i)=y_i \quad \text{voor alle } i. $$

Examenkern:

- **global polynomial** vs **piecewise**,
- numerieke stabiliteit (Vandermonde is gevaarlijk),
- wanneer gebruik je interpolatie vs least squares (H3),
- en waarom splines vaak winnen bij veel punten.

7.1 Introduction

Interpolatie is nuttig als je:

- waarden wil “tussenin” schatten,
- een gladde curve nodig hebt voor verdere analyse (bv. integratie, differentiatie, root finding).

Let op: interpolatie forceert exact door meetpunten → bij ruis is dat vaak een slecht idee (dan wil je H3: regression/least squares).

7.2 Polynomial interpolation of discrete data

7.2.1 Lagrange-form

Voor $n+1$ punten: $$ p(x)=\sum_{i=0}^n y_i \ell_i(x), \qquad \ell_i(x)=\prod_{\substack{j=0\ j\neq i}}^n \frac{x-x_j}{x_i-x_j}. $$

- Pro: conceptueel helder
- Con: duur/instabiel als je het naïef evalueert

7.2.2 Newton-form + divided differences

Newton-vorm: $$ p(x)=a_0+a_1(x-x_0)+a_2(x-x_0)(x-x_1)+\cdots $$ Coeffs a_k via divided differences.

- Pro: makkelijk uitbreidbaar met nieuwe punten; efficiënte evaluatie (Horner-achtig)
- Con: nog steeds global polynomial (Runge-risico)

7.2.3 Vandermonde-systeem (waarom je dit meestal NIET wil)

Je kan ook oplossen uit: $$ \mathbf{V}\mathbf{c}=\mathbf{y}, \qquad V_{ij}=x_i^j. $$ Maar $\mathbf{V}$ is vaak extreem slecht geconditioneerd → numeriek ellende (link H1/H2/H3).

Cursusboodschap: doe dit alleen als didactiek; in echte code: nee.

7.2.4 Interpolation error + Runge phenomenon

Foutterm (als f glad genoeg is): $$ f(x)-p(x)=\frac{f^{(n+1)}(\xi)}{(n+1)!}\prod_{i=0}^n (x-x_i) $$ voor een ξ tussen de nodes.

Runge: bij gelijk verdeelde nodes kan het global polynomial wild oscilleren aan de randen. Fix: **Chebyshev nodes** (clusteren naar de randen) → veel stabielere global interpolatie.

7.2.5 Barycentric interpolation (de numeriek stabiele manier)

Barycentric formule evalueert Lagrange-interpolatie stabiel en snel. Cursuspunt: dit is de praktische manier als je toch global polynomial wil.

7.3 Piecewise polynomial interpolation

Waarom piecewise?

Bij veel punten is één polynoom van hoge graad meestal een ramp (oscillaties, conditioning). Piecewise houdt de graad laag en stabiliteit hoog.

7.3.1 Piecewise linear

Snel, robuust, maar niet glad in afgeleide.

7.3.2 Cubic splines

Cubic spline: op elk interval een cubisch polynoom met voorwaarden op continuïteit:

- p, p', p'' continu op knooppunten. Je hebt extra randvoorwaarden (bv. “natural spline”: tweede afgeleide 0 aan de randen).

- Pro: zeer glad; meestal “default” voor gladde interpolatie
- Con: kan overshoots geven bij monotone data

7.3.3 Shape-preserving (PCHIP)

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) behoudt monotoniciteit (minder overshoot).

- Pro: goed voor data die fysisch monotone moeten blijven (bv. dichtheid, cumulatieven)
- Con: iets minder “glad” dan klassieke cubic spline (maar meestal een feature)

7.4 Software (SciPy)

Typische tools:

- `scipy.interpolate.interp1d` (linear/quadratic/cubic, eenvoudig)
- `scipy.interpolate.CubicSpline`
- `scipy.interpolate.PchipInterpolator`
- `scipy.interpolate.BarycentricInterpolator`

Welke methode wanneer?

- **Kleine set punten, zeer gladde functie, je wil hoge orde:** barycentric / Chebyshev nodes (global).
- **Veel punten, je wil stabiel en glad:** cubic spline.
- **Monotone fysische data (geen overshoot toegestaan):** PCHIP.
- **Ruis aanwezig / je wil niet exact door punten:** geen interpolatie → H3 least squares fit.

Links met andere hoofdstukken

- H3: interpolatie vs regression (interpolatie forceert exact; LS filtert ruis).
- H8: integratie/differentiatie van geïnterpoleerde data (splines zijn handig).
- H11: Fourier-achtige interpolatie voor periodieke signalen (FFT-wereld).

Hoofdstuk 8 — Numerical Integration and Differentiation

Doel:

- integraal: $$ I=\int_a^b f(x),dx $$
- afgeleide: $$ f'(x) $$

Examenkern:

- foutorde vs kost,
- (on)stabiliteit van differentiatie,

- adaptiviteit,
- Richardson extrapolation als “order booster”.

8.1 Integration

Integratie is meestal numeriek **vriendelijker** dan differentiatie (differentiatie versterkt ruis).

8.2 Existence, Uniqueness, Conditioning

Voor integralen:

- bestaan/uniqueness is zelden het probleem (voor nette f),
- conditioning hangt vaak af van hoe “wild” of “singulier” f is, en van cancellation.

8.3 Numerical Quadrature

We bouwen een benadering: $$ \int_a^b f(x),dx \approx \sum_{i=0}^N w_i f(x_i). $$

8.3.1 Newton–Cotes (equispaced)

- Rectangle/left/right: lage orde
- Midpoint: betere orde
- Trapezoidal rule
- Simpson’s rule (parabool-fit)

Orders (typisch in de cursus):

- midpoint en trapezoid: fout $O(h^2)$
- Simpson: fout $O(h^4)$ waar h de stapgrootte is.

Trade-off:

- hogere orde = minder stappen voor dezelfde truncation error,
- maar bij te klein h kan rounding meespelen (link H1).

8.3.2 Composite rules

Splits $[a,b]$ in subintervallen en pas de regel herhaald toe.

- Werkt goed voor “redelijk gladde” f.

8.3.3 Gaussian quadrature (hoog rendement voor gladde f)

Kies nodes/weights zodat de regel exact is voor polynomen tot een hoge graad.

- Pro: extreem efficiënt als f glad is
- Con: minder plug-and-play als f singular/oscillerend is (maar er bestaan varianten)

8.3.4 Adaptive quadrature

Pas het interval dynamisch aan: verfijn waar f moeilijk is.

- Pro: efficiënt bij lokale pieken/singulariteit
- Con: overhead/complexiteit; maar vaak de beste “algemene” keuze

---

8.4 Other integration problems

Improper integrals / singular integrands

Strategieën:

- variabeletransformatie om singulariteit te verzwakken,
- splits interval rond de singulariteit,
- adaptive methods.

Oscillerende integralen

Standaard Newton–Cotes kan falen. Mogelijke routes:

- speciale oscillatory quadrature,
- herformuleren (Fourier/FFT-ideeën bij periodieke structuren).

Multi-dimensional integrals

Deterministische quadrature explodeert in kost met dimensie (“curse of dimensionality”). → Monte Carlo (H12) wordt dan aantrekkelijk.

8.5 Numerical Differentiation

Finite differences

Forward: $$ f'(x)\approx \frac{f(x+h)-f(x)}{h} \quad \text{(truncation } O(h)\text{)} $$ Central: $$ f'(x)\approx \frac{f(x+h)-f(x-h)}{2h} \quad \text{(truncation } O(h^2)\text{)} $$

Cruciale trade-off: truncation vs rounding

- truncation daalt als $h\to 0$,
- maar rounding/cancellation stijgt omdat je bijna gelijke getallen aftrekt: $f(x+h)-f(x)$.

Gevolg: er bestaat een **optimale** h; “zo klein mogelijk” is fout (link H1).

Differentiatie van ruis

Ruis wordt versterkt door afgeleiden → vaak eerst smoothen of een model fitten (H3/H7) en dáárvan differentiëren.

8.6 Richardson Extrapolation

Idee: combineer twee benaderingen met stap h en $h/2$ om lagere-orde fout weg te elimineren.

Voor een methode met fout: $$ A(h)=A + C h^p + O(h^{p+1}), $$ dan: $$ A \approx \frac{2^p A(h/2)-A(h)}{2^p-1}. $$

Gebruik:

- upgrade trapezoid → Romberg-achtige integratie,
- upgrade differentieformules.

8.7 SciPy

- `scipy.integrate.quad` (adaptief, 1D)
- `scipy.integrate.quadrature` / `fixed_quad` (Gauss)
- voor ODE-integratie: zie H9 (andere wereld)

Welke methode wanneer?

Integratie

- **Algemene 1D integralen, weinig gedoe:** adaptief (`quad`-stijl).
- **Zeer glad, hoge nauwkeurigheid met weinig evaluaties:** Gaussian quadrature.
- **Simpele integralen, je wil controle/educatief:** composite trapezoid/Simpson.
- **Hoge dimensie:** ga naar H12 (Monte Carlo).

Differentiatie

- **Je hebt een analytisch model of fit:** differentieer het model (best).
- **Je hebt zuivere data (weinig ruis):** central differences + Richardson.
- **Ruis aanwezig:** vermijd directe differentiatie; smooth/fit eerst.

Links met andere hoofdstukken

- H7: splines/fit als "voorbewerking" voor differentiatie.
- H9/H10: discretisaties gebruiken exact deze finite differences; stabiliteit hangt af van stapgroottes.
- H12: multi-D integratie → Monte Carlo.

Hoofdstuk 9 — Ordinary Differential Equations (ODEs)

We lossen een initial value problem (IVP): $$ \mathbf{y}'(t)=\mathbf{f}(t,\mathbf{y}(t)), \qquad \mathbf{y}(t_0)=\mathbf{y}_0, $$ waar $\mathbf{y}(t)$ een vector is.

Examenkern:

- discretisatie + local/global truncation error,
- stiff vs non-stiff,
- expliciet vs impliciet,

- adaptieve step size,
- link met H5/H2 (impliciet → nonlineair + lineaire solves).

9.1 Introduction and useful concepts

- IVP: startwaarde gegeven, evolutie vooruit.
- BVP: randvoorwaarden op twee punten (zie 9.3).

Numerieke methode maakt een rooster: $$ t_n=t_0+n h, $$ en benadert $\mathbf{y}(t_n)\approx \mathbf{y}_n$.

9.2 Numerically solving ODE's

Discrete variable methods

We vervangen de differentiaalvergelijking door update-regels: $$ \mathbf{y}_{n+1} = \mathbf{y}_n + h,\Phi(t_n,\mathbf{y}_n,h), $$ waar Φ de methode-specifieke "slope-combinatie" is.

9.2.1 Euler methods

Forward Euler (expliciet): $$ \mathbf{y}_{n+1}=\mathbf{y}_n + h,\mathbf{f}(t_n,\mathbf{y}_n). $$

- Pro: goedkoop, simpel
- Con: lage orde, kan instabiel zijn

Backward Euler (impliciet): $$ \mathbf{y}*{n+1}=|*mathbf{y}n + h,|*mathbf{f}(t{n+1},|*mathbf{y}*{n+1}). $$

- Pro: veel stabieler, goed voor stiff problemen
- Con: vereist per stap het oplossen van een nonlineair probleem (H5); vaak Newton + lineaire solves (H2)

9.2.2 Runge–Kutta (RK) methods (expliciet, hogere orde)

Algemeen RK gebruikt meerdere "stages" $\mathbf{k}_i$ en combineert die.

RK4 (klassieker): hoge nauwkeurigheid per stap, maar vaste stapgrootte (tenzij je error-estimates toevoegt).

9.2.3 Adaptive step size (embedded RK)

In de praktijk wil je controle op fout:

- compute twee oplossingen van verschillende orde in dezelfde stap,
- schat error,
- pas h aan.

Waarom dit vaak wint:

- je investeert rekenwerk waar het nodig is (snelle variatie),
- je spaart werk waar het niet nodig is.

9.2.4 Stability en stiffness (het echte “wanneer welke solver” criterium)

Een probleem is **stiff** als expliciete methodes een extreem klein h moeten nemen voor stabiliteit, zelfs als de oplossing zelf niet zo snel varieert.

- **Non-stiff:** expliciete RK-methodes zijn vaak ideaal.
- **Stiff:** impliciete methodes (Backward Euler, BDF, Radau) winnen gigantisch.

Kernboodschap: stapgrootte wordt bij stiff problemen bepaald door stabiliteit, niet door nauwkeurigheid.

9.2.5 Impliciete methodes in de praktijk (link met H5/H2)

Een impliciete stap vraagt: $$ \mathbf{y}*{n+1}-|*mathbf{y}n - h,|*mathbf{f}(t{n+1},|*mathbf{y}*{n+1})=\mathbf{0}. $$ Dit is een nonlineair systeem in $\mathbf{y}_{n+1}$. Newton geeft: $$ \mathbf{J},\Delta \mathbf{y}=-\mathbf{g}(\mathbf{y}), $$ waar $\mathbf{J}$ een Jacobiaan is en je lineaire solves nodig hebt (H2).

9.3 Boundary Value Problems (BVP’s) for ODE’s

BVP: je kent voorwaarden aan twee uiteinden, bv. $$ y(a)=\alpha,\qquad y(b)=\beta. $$

(A) Shooting method

Maak van BVP een IVP met onbekende start-slope s en zoek s zodat de eindvoorwaarde klopt. Dat is root finding (H5): $$ F(s)=y(b;s)-\beta=0. $$

- Pro: conceptueel simpel
- Con: kan instabiel zijn bij gevoelige problemen (slechte conditioning)

(B) Finite difference / collocation

Discretiseer en los een (groot) algebraïsch systeem op, vaak sparse.

- Pro: robuuster voor lastige BVP’s
- Con: implementatie zwaarder, maar SciPy heeft tooling (`solve_bvp`)

SciPy (typische mapping)

- `scipy.integrate.solve_ivp`
  - non-stiff: `RK45` / `DOP853`
  - stiff: `Radau` / `BDF`
- `scipy.integrate.solve_bvp` voor BVP

Welke methode wanneer?

- **Snelle, gladde dynamica, niet stiff:** expliciete RK + adaptieve stap.
- **Stiff (veel tijdschalen):** impliciet (`BDF`/`Radau`).
- **BVP met eenvoudige fysica:** shooting + root finding (H5).
- **BVP moeilijk/sensitief:** finite difference/collocation (`solve_bvp`).

Links met andere hoofdstukken

- H5: impliciete stappen en shooting vragen root finding.
- H2: Newton-stappen vragen lineaire solves (LU/sparse).
- H10: PDE time-stepping reduceert vaak tot ODE-systemen (method of lines).

Hoofdstuk 10 — Partial Differential Equations (PDEs)

PDE's beschrijven velden $u(\mathbf{x},t)$ met ruimte én tijd. Numeriek komt het altijd neer op:

1. discretiseer ruimte (finite differences / finite elements / spectral),
2. kies een tijdstapper (expliciet of impliciet),
3. controleer stabiliteit (CFL / stiffness),
4. los per stap een lineair of nonlineair stelsel op.

De examenkern in Py4Sci-stijl: **mapping PDE → discretisatie → type stelsel → geschikte solver**, plus **kost vs stabiliteit**.

10.1 Classificatie (waarom je dit meteen wil weten)

Voor 1D voorbeelden:

Elliptisch (stationair)

Laplace/Poisson: $$ u_{xx} = f(x) \quad\text{of}\quad \nabla^2 u = f. $$

- Geen tijd.
- Discretisatie $\Rightarrow$ **lineair stelsel** $\mathbf{A}\mathbf{u}=\mathbf{b}$ (vaak SPD en sparse).

Parabolisch (diffusie)

Heat equation: $$ u_t = \kappa u_{xx}. $$

- Eén tijdafgeleide, tweede orde in ruimte.
- Expliciete methodes hebben **strenge stabiliteitslimiet**: $\Delta t = O((\Delta x)^2)$.

Hyperbolisch (golven/transport)

Wave equation: $$ u_{tt} = c^2 u_{xx} $$ of advection: $$ u_t + c u_x = 0. $$

- Propagatie; stabiliteit bepaalt meestal $\Delta t = O(\Delta x)$ (CFL).

Waarom dit examenrelevant is: het vertelt je meteen of een probleem “stiff” wordt (parabolisch vaak wel) en dus impliciete methodes verdient.

10.2 Ruimtediscretisatie (finite differences als default)

Rooster in 1D: $$ x_i = a + i\Delta x,\qquad i=0,\dots,N. $$

10.2.1 Centrale stencil’s (zoals in integratie/differentiatie hoofdstuk)

Eerste afgeleide: $$ u_x(x_i)\approx \frac{u_{i+1}-u_{i-1}}{2\Delta x}. $$ Tweede afgeleide: $$ u_{xx}(x_i)\approx \frac{u_{i+1}-2u_i+u_{i-1}}{(\Delta x)^2}. $$

10.2.2 Boundary conditions (BCs) als “extra vergelijkingen”

- Dirichlet: $u(a)=\alpha$, $u(b)=\beta$ (fixeer randwaarden).
- Neumann: $u_x(a)=\gamma$ (randstencil of ghost points).
- Robin: combinatie.

BC’s bepalen hoe je de eerste/laatste rijen van je matrix $\mathbf{A}$ opbouwt.

10.3 Stationaire PDE’s: Poisson/Laplace $\Rightarrow$ sparse linear system

Voor Poisson in 1D: $$ -u_{xx}=f(x), $$ met centrale stencil krijg je: $$ \frac{-u_{i+1}+2u_i-u_{i-1}}{(\Delta x)^2}=f_i. $$

Dit levert een tridiagonale (sparse) matrix $\mathbf{A}$: $$ \mathbf{A}\mathbf{u}=\mathbf{b}. $$

10.3.1 Welke solver?

- Kleine N (dense of klein sparse): directe solve.
- Grote N (sparse): `spsolve` of iteratief:
  - SPD: Conjugate Gradient (CG),
  - niet-SPD: GMRES/BiCGSTAB (afhankelijk van structuur).

Waarom: sparse opslag + sparse matrix-vector products zijn $O(N)$ i.p.v. $O(N^2)$.

10.3.2 Conditioning en grid

Conditioning van discretisatie-operators wordt slechter als Δx kleiner wordt (meer roosterpunten). Dit beïnvloedt:

- aantal iteraties bij iteratieve solvers,
- gevoeligheid voor rounding.

10.4 Time-dependent PDE’s: method of lines

Discretiseer eerst ruimte. Dan krijg je een ODE-systeem: $$ \mathbf{u}'(t)=\mathbf{F}(t,\mathbf{u}(t)). $$ Daarna gebruik je ODE-methodes (H9) op $\mathbf{u}(t)$.

Dit is een sleutelbrug tussen PDE en ODE: PDE numeriek oplossen = “grote ODE” + (sparse) lineaire algebra.

10.5 Tijdstepping: expliciet vs impliciet

10.5.1 Heat equation (parabolisch): waarom expliciet vaak faalt

Neem FTCS: $$ u_i^{n+1}=u_i^n + \mu\left(u_{i+1}^n-2u_i^n+u_{i-1}^n\right), \qquad \mu=\kappa\frac{\Delta t}{(\Delta x)^2}. $$

Stabiliteit vereist typisch: $$ \mu \le \frac{1}{2}\quad \text{(1D klassieke bound)}. $$

Interpretatie: bij kleinere Δx moet Δt kwadratisch kleiner → veel stappen → duur.

Impliciet (Backward Euler)

$$ u_i^{n+1}=u_i^n + \mu\left(u_{i+1}^{n+1}-2u_i^{n+1}+u_{i-1}^{n+1}\right). $$ In matrixvorm: $$ (\mathbf{I}-\mu\mathbf{L})\mathbf{u}^{n+1}=\mathbf{u}^n. $$

- Pro: veel stabieler → grotere Δt mogelijk.
- Con: per stap een sparse solve (maar dat is vaak goedkoper dan miljoenen expliciete stappen).

Crank–Nicolson (CN)

Gemiddelde van expliciet/impliciet: $$ (\mathbf{I}-\tfrac{\mu}{2}\mathbf{L})\mathbf{u}^{n+1}

(\mathbf{I}+\tfrac{\mu}{2}\mathbf{L})\mathbf{u}^{n}. $$

- Vaak 2e orde in tijd en stabieler dan expliciet.
- Kan oscillaties geven bij ruwe data als je te grote Δt neemt (numeriek “ringing”).

10.5.2 Hyperbolisch (advection/waves): CFL is de baas

Voor advection $u_t+cu_x=0$ is een typische CFL-conditie: $$ \frac{|c|\Delta t}{\Delta x}\le C_{\max}. $$

Fysische betekenis: informatie beweegt met snelheid c; per tijdstap mag een “feature” niet meer dan ongeveer één cel opschuiven, anders mis je de dynamica en wordt het instabiel.

Schema-keuze is cruciaal:

- upwind schema’s zijn stabieler (maar numeriek diffuus),
- centrale schema’s kunnen oscilleren zonder extra stabilisatie.

10.6 Welke methode wanneer? (examengerichte beslisregels)

Stationair elliptisch (Poisson/Laplace)

- Bouw sparse $\mathbf{A}$, solve $\mathbf{A}\mathbf{u}=\mathbf{b}$.
- SPD? CG (eventueel met preconditioner) of directe factorisatie als klein genoeg.

Diffusie (heat)

- Kleine problemen: expliciet kan, maar check $\Delta t \sim (\Delta x)^2$.
- Grotere of fijne grids: impliciet/CN wint bijna altijd.

Advection/waves

- Expliciet met CFL is vaak oké en efficiënt.
- Let op numerieke dispersie/oscillaties → kies passend schema (upwind/flux limiters in geavanceerde setting).

10.7 Typische valkuilen (die in “insights” gevraagd worden)

1. **Je kiest Δt op nauwkeurigheid, maar stabiliteit is strenger** (diffusie).
2. **Circulaire artefacten** als je periodieke BC per ongeluk “meeneemt”.
3. **Sparse vs dense**: een PDE-matrix dense behandelen is dood door geheugen/kost.
4. **BC-implementatie**: vaak de bron van bugs (eerste/laatste rij fout).

SciPy mapping (praktisch)

- `scipy.sparse` voor operators
- `scipy.sparse.linalg.spsolve`, `cg`, `gmres`, …
- method of lines: gebruik `solve_ivp` met RHS die sparse operator toepast

---

Links met andere hoofdstukken

- H8: finite differences + truncation/rounding.
- H9: method of lines = groot ODE-probleem.
- H2: impliciete stappen = lineaire solves (sparse LU/CG).
- H4: eigenwaarden van discretisatie-operator sturen stabiliteit (stiffness).

Hoofdstuk 11 — Fast Fourier Transform (FFT)

FFT is een snelle manier om de DFT te berekenen. Het is niet alleen “spectra plotten”; het is een **reken-truc** om dingen te versnellen die anders $O(N^2)$ kosten (convolutie/correlatie) en een **analyse-tool** (frequenties, filtering, aliasing).

Examenkern:

- DFT/IDFT + interpretatie
- FFT-kost: $O(N\log N)$
- spectral leakage + windows

- convolution theorem + zero-padding (circulair vs lineair!)
- DFT vs DCT: wanneer welke

11.1 DFT: definitie en interpretatie

Voor samples x_n, $n=0,\dots,N-1$:

DFT: $$ X_k=\sum_{n=0}^{N-1} x_n,e^{-2\pi i kn/N}. $$

Inverse: $$ x_n=\frac{1}{N}\sum_{k=0}^{N-1} X_k,e^{2\pi i kn/N}. $$

Frequentieschaling

Als sample rate f_s (samples per seconde), dan correspondeert bin k met frequentie: $$ f_k=\frac{k}{N}f_s $$ (en bij real signals is er symmetrie zodat je vaak `rfft` gebruikt).

Periodiciteitsassumptie

DFT behandelt je data als één periode van een periodiek signaal met periode N.

- Als $x_0\neq x_{N-1}$ in "trend", creëer je een rand-discontinuïteit.
- Dat veroorzaakt leakage (zie 11.4).

11.2 FFT: waarom het snel is

Direct DFT: $O(N^2)$.

FFT (Cooley–Tukey): splits in even/odd indices en hergebruik kleinere DFT's:

- $O(N\log N)$.

Praktische impact:

- $N=10^6$ is haalbaar met FFT, onmogelijk met directe DFT.

11.3 Real FFT en efficiëntie

Voor reële x_n geldt conjugate symmetry: $$ X_{N-k}=\overline{X_k}. $$ Dus je kan de helft opslaan/berekenen:

- `rfft` en `irfft` zijn sneller en zuiniger.

11.4 Spectral leakage, resolution, windows

11.4.1 Wat is leakage?

Als je een zuivere sinus hebt met frequentie die niet exact op een bin valt, dan "lekt" de energie naar andere bins.

Intuïtie: je neemt een eindig venster van een oneindig signaal. Dat is een vermenigvuldiging met een window in tijd → in frequentie is dat een convolutie met de Fourier transform van de window. Dus je spectrum wordt uitgesmeerd.

11.4.2 Windowing (Hann/Hamming/Blackman/...)

Je vermenigvuldigt x_n met een window w_n die naar 0 gaat aan de randen.

- Pro: veel minder leakage (side lobes omlaag)
- Con:
  - main lobe breder → slechtere frequentieresolutie
  - amplitude-bias (je moet vaak "window correction" doen als amplitudes exact moeten)

Wanneer windowen?

- Bijna altijd bij real-world data die niet exact periodiek is in het sample-interval.
- Als je vooral frequenties wil detecteren i.p.v. perfecte amplitudes.

Wanneer niet?

- Als je signaal exact periodic in het interval is (zeldzaam), dan is boxcar oké en geeft hoogste resolutie.

11.5 Aliasing (Nyquist)

Als je sample rate f_s is, dan is de Nyquist frequentie: $$ f_N=\frac{f_s}{2}. $$ Frequenties boven f_N folden terug (aliasing). FFT kan dat niet "fixen"; dat is een sampling probleem.

Examenvriendelijk inzicht: aliasing is geen numerieke fout van FFT; het is informatieverlies door sampling.

11.6 Convolution en correlation met FFT (de grote speed-up)

Discrete lineaire convolutie: $$ (y*g)*n*=|*sum_m y_m g*{n-m}. $$

DFT-convolution theorem: $$ \mathrm{DFT}(y*g)=\mathrm{DFT}(y),\mathrm{DFT}(g). $$

Circulair vs lineair (de valkuil)

FFT op lengte N geeft **circulaire convolutie**:

- indices wrappen modulo N.

Voor lineaire convolutie moet je **zero-padding** doen tot minstens: $$ N_{\text{pad}} \ge N_y + N_g - 1. $$

Typische "insights"-vraag na coding:

- Waarom zag ik wrap-around/artefacten?
  → omdat je circulair deed zonder padding.

Correlatie

Correlatie lijkt op convolutie maar met een omkering/conjugatie: $$ (r_{xy})*k = |sum_n x_n |overline{y}{n-k}}. $$ FFT-truc blijft: FFT's + puntgewijs product + inverse FFT.

11.7 DFT vs DCT (wanneer welke, zoals jullie vaak vragen)

DFT

- basisfuncties: complexe exponenten
- impliciet periodiek: "wrap-around"

DCT (Discrete Cosine Transform)

DCT komt overeen met een **even (reflective) extension** van je data:

- minder rand-discontinuïteit als je signaal niet-periodiek is,
- vaak compacter spectrum voor gladde niet-periodieke functies.

Wanneer is DCT beter?

- data op $[0,L]$ met "reflectie"-achtig randgedrag,
- compressie (energie in lage modes),
- PDE's met Neumann-achtige BC's (cosinus-basis natuurlijk).

Wanneer DFT beter?

- echt periodieke signals,
- complexe fase-informatie,
- convolution/correlation in periodieke settings.

11.8 Welke methode wanneer? (beslisregels)

- **Spectrale analyse (frequenties vinden):** FFT (`rfft` voor real).
- **Filtering / smoothing / band-pass:** FFT + filtermasker (maar let op windowing & edge effects).
- **Snelle convolutie met lange kernel:** FFT + zero-padding.
- **Niet-periodieke, gladde data op een interval:** DCT vaak beter dan DFT.
- **Als randvoorwaarden periodic zijn (PDE/spectraal):** DFT/FFT is natuurlijk.

11.9 Typische valkuilen (die punten kosten)

1. Geen zero-padding → circulaire artefacten.
2. Geen window → leakage en "spookfrequenties".
3. Verkeerde frequentie-as (fftfreq) → verkeerde interpretatie.
4. Aliasing verwarren met leakage.
5. Vergeten dat FFT output complex is: magnitude $|X_k|$, phase $\arg(X_k)$.

SciPy mapping

- `scipy.fft.fft`, `ifft`
- `scipy.fft.rfft`, `irfft`
- `scipy.fft.fftfreq`, `rfftfreq`
- `scipy.fft.next_fast_len` (snelle padding lengte)
- `scipy.fft.dct`, `idct` (voor DCT)

Links met andere hoofdstukken

- H7/H8: interpolatie/differentiatie kunnen spectraal (FFT) bij periodieke problemen.
- H10: spectrale methodes en snelle Poisson-solvers bij periodieke BC.
- H1: interpretatieproblemen (leakage/aliasing) domineren; numerieke rounding is zelden de bottleneck.

Hoofdstuk 12 — Monte Carlo

Monte Carlo (MC) methodes lossen numerieke problemen op via steekproeven. Het “superpower”-argument:

- de fout schaalt typisch als $1/\sqrt{N}$, bijna onafhankelijk van dimensie,
- dus bij hoge dimensie (waar deterministische quadrature explodeert) is MC vaak de enige praktische optie.

Tegelijk: $1/\sqrt{N}$ is traag. Het examen draait dus om:

1. **wanneer MC de juiste hamer is**,
2. **hoe je error en betrouwbaarheid kwantificeert**,
3. **hoe je variance verlaagt** (variance reduction),
4. **wat MCMC doet en wanneer je het nodig hebt**.

12.1 Randomness in computation: PRNG en reproducibility

Pseudo-random number generators (PRNG)

Computers genereren geen “echte” random getallen, maar pseudo-random:

- deterministisch algoritme,
- lijkt statistisch random (als de generator goed is),
- volledig bepaald door een **seed**.

Waarom dit exam-relevant is: reproduceerbaarheid.

- Zelfde seed $\Rightarrow$ zelfde resultaten (essentieel in wetenschappelijke code).
- Andere seed $\Rightarrow$ statistisch equivalente resultaten (als de generator goed is).

Praktische good practice (zoals in moderne NumPy/SciPy)

- Maak één generator-object.
- Vermijd “global state” (zoals `np.random.seed(...)` overal).

12.2 Monte Carlo integratie: de basis-estimator

We willen een integraal over een domein D: $$ I=\int_D f(\mathbf{x}),d\mathbf{x}. $$

12.2.1 Uniform sampling op een domein

Als je uniform $\mathbf{x}_i$ in D trekt en $V=\mathrm{vol}(D)$: $$ \hat{I}*N = V ,|frac{1}{N}|sum*{i=1}^N f(\mathbf{x}_i). $$

Unbiasedness (intuïtief): $$ \mathbb{E}[\hat{I}_N]=I $$ als je echt uniform samplet.

12.2.2 Foutschatting (standard error)

Definieer de steekproefgemiddelde: $$ \bar{f}=\frac{1}{N}\sum_{i=1}^N f(\mathbf{x}*i), $$ *en steekproefvariantie:* $$ s_f^2 = |frac{1}{N-1}|sum*{i=1}^N (f(\mathbf{x}_i)-\bar{f})^2. $$ Dan is de standaardfout op I: $$ \mathrm{SE}(\hat{I}_N) \approx V,\frac{s_f}{\sqrt{N}}. $$

Cruciale interpretatie:

- Als je error 10× kleiner wil, heb je ~100× meer samples nodig.
- Daardoor is variance reduction vaak belangrijker dan “gewoon meer samples”.

12.2.3 Confidence intervals (CI)

Voor groot N (CLT) is: $$ \hat{I}_N \approx \mathcal{N}(I,\mathrm{SE}^2). $$ Een (ongeveer) 95% interval: $$ \hat{I}_N \pm 1.96,\mathrm{SE}. $$

Wat je mondeling moet kunnen zeggen:

- Dit is asymptotisch (werkt beter voor grote N),
- Bij heavy tails of kleine N kan dit misleidend zijn.

12.3 Curse of dimensionality: waarom MC soms wint

Deterministische quadrature in d dimensies:

- nodes groeien vaak exponentieel met d.

MC:

- error $\sim 1/\sqrt{N}$, niet exponentieel afhankelijk van d.

Beslisregel (exam-waardig):

- $d\le 3$ en f glad: deterministisch (H8) wint vaak.
- $d\gg 1$: MC is vaak de enige haalbare.

12.4 Variance reduction: sneller dan $1/\sqrt{N}$ in de praktijk

Het doel is niet “andere asymptotiek” (meestal blijft $1/\sqrt{N}$), maar een veel kleinere constante: Var omlaag.

12.4.1 Importance sampling

Kies een pdf $p(\mathbf{x})$ die lijkt op waar $|f|$ groot is. Dan: $$ I=\int_D f(\mathbf{x}),d\mathbf{x} =\int_D \frac{f(\mathbf{x})}{p(\mathbf{x})} p(\mathbf{x}),d\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^N \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}, \quad \mathbf{x}_i\sim p. $$

Wanneer gebruiken?

- als f “spiky” is of lokaal groot (anders mis je de piek met uniform sampling),
- bij zeldzame events.

Valkuilen:

- als p te klein is waar f groot is, krijg je enorme gewichten $\frac{f}{p}$ → variance explodeert.
- je moet p kunnen samplen én evalueren.

12.4.2 Stratified sampling

Splits D in strata D_j met volumes V_j. Sample in elk stratum apart en combineer: $$ \hat{I}=\sum_j V_j \bar{f}_j. $$

Wanneer gebruiken?

- als f sterk varieert per regio,
- als je “garandeert” dat elke regio vertegenwoordigd is.

12.4.3 Control variates

Vind g met bekende integraal $G=\int g$ en sterke correlatie met f. Neem estimator: $$ \hat{I} = \hat{I}_f - c(\hat{I}_g - G). $$ Kies c optimaal via covariantie/variantie (in praktijk vaak geschat uit samples).

Wanneer gebruiken?

- als je een benaderende “goed integrerende” functie kent die op f lijkt.

12.4.4 Antithetic variates (symmetrie benutten)

Sample in paren $(\mathbf{x},\tilde{\mathbf{x}})$ die negatief correleren (bv. u en $1-u$ bij uniforme variabelen).

- vermindert variance als f monotone/structureel is.

12.5 Monte Carlo voor andere taken

12.5.1 Monte Carlo voor π / geometrische probabiliteit

Klassiek: area/volume ratio's. Dit is vooral didactisch om unbiasedness en $1/\sqrt{N}$ te tonen.

12.5.2 Random walks en diffusion

Random walks linken MC aan fysische processen (diffusie, Brownse beweging).

12.6 Markov Chain Monte Carlo (MCMC): sampelen uit moeilijke verdelingen

Soms wil je geen integraal rechtstreeks, maar samples uit een verdeling $\pi(\mathbf{x})$ (bv. posterior in Bayes).

Als direct sampelen niet lukt, gebruik je een Markov chain met stationaire verdeling π.

12.6.1 Metropolis–Hastings

- huidige state: $\mathbf{x}$
- voorstel (proposal): $\mathbf{x}'\sim q(\mathbf{x}'|\mathbf{x})$
- acceptatie: $$ \alpha(\mathbf{x}\to \mathbf{x}')= \min\left(1, \frac{\pi(\mathbf{x}'),q(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x}),q(\mathbf{x}'|\mathbf{x})} \right). $$

Voor symmetrische proposal (random walk): $$ \alpha= \min\left(1,\frac{\pi(\mathbf{x}')}{\pi(\mathbf{x})}\right). $$

12.6.2 Burn-in, mixing, autocorrelation

MCMC-samples zijn **gecorreleerd**.

- burn-in: eerste stuk weggooien (chain nog niet in stationaire regime),
- autocorrelatie betekent dat "effectieve N" kleiner is dan het aantal stappen,
- tuning van proposal-stepsize beïnvloedt acceptance rate en mixing.

Examengerichte insight:

- Te kleine stappen: hoge acceptatie maar langzaam exploreren.
- Te grote stappen: lage acceptatie, chain "stuck". Je wil een "sweet spot" (afhankelijk van dimensie/proposal).

12.7 Welke methode wanneer?

Gebruik deterministische quadrature (H8) als...

- dimensie laag is,
- f glad is,
- je hoge precisie wil met weinig evaluaties.

Gebruik plain Monte Carlo als...

- dimensie hoog is,

- domein complex is,
- je snel een ruwe schatting + foutbalk wil.

Gebruik variance reduction als…

- plain MC te traag convergeert,
- f spiky / rare-event / sterk variabel is,
- je structuur kunt exploiteren (symmetrie, bekende vergelijkbare g).

Gebruik MCMC als…

- je samples uit $\pi(\mathbf{x})$ nodig hebt en direct sampling moeilijk is,
- je Bayesiaanse inferentie doet,
- je integralen wil als expectations onder π: $$ \mathbb{E}_{\pi}[h(\mathbf{x})] \approx \frac{1}{N}\sum_{i=1}^N h(\mathbf{x}_i), \quad \mathbf{x}_i\sim \pi. $$

Links met andere hoofdstukken

- H8: multi-D integratie → MC is het natuurlijke alternatief voor curse of dimensionality.
- H1: errorbars, rounding, reproducibility (seed) zijn essentieel.
- H3/H6: stochastic sampling vs fitting/optimisation; control variates lijken op “modelfout compenseren”.
- Statistische interpretatie: CLT, variantie, confidence intervals (wordt vaak in “insights” gevraagd na coding).