De Hamming-code

De wiskunde van het fouten verbeteren in digitale gegevens

In het kader van:
(Bij) de Faculteit Wiskunde en Informatica van de TU/e
op bezoek

voorjaar 2007

Inhoudsopgave

1 Wat is het nut van coderen?

Via internet versturen we dagelijks berichten, tekstbestanden, foto's en zelfs films. En dat gebeurt op digitale wijze: wat we versturen zijn rijtjes 0'en en 1'en (bits). Als we bijvoorbeeld het woord *klas* willen versturen, dan zou er in feite

01010010110000010010

verstuurd kunnen worden. Dat komt, simpel gezegd, omdat je in computers op het meest elementaire niveau maar met twee toestanden kunt werken: 'stroom aan' of 'stroom uit'. Misschien kun je je niet zo goed voorstellen wat wiskunde hiermee te maken heeft, maar door het gebruik van wiskunde is er voor gezorgd dat informatie op efficiënte wijze digitaal vastgelegd kan worden. Dit heeft bijvoorbeeld de Nederlandse uitvinding (Philips, met medewerking van de Technische Universiteit Eindhoven) van de CD in de jaren '70 en '80 mogelijk gemaakt. Het vakgebied uit de wiskunde dat hiermee bezig is, is de *coderingstheorie*. Een verwant vakgebied is dat van de *cryptologie* waar men werkt aan het versleutelen van digitale informatie zodat onbevoegden bijvoorbeeld niet je mobiele telefoongesprek kunnen afluisteren. Daar zullen we het niet over hebben.

Wat is nu de rol van wiskunde?

- Wiskunde helpt bij het bedenken hoe je informatie (foto's, tekst, enz.) kunt vastleggen of coderen met 0'en en 1'en (of andere verwante manieren).
- Wiskunde vertelt je hoe je je gegevens zó kunt coderen dat, ook al gaat er iets mis met je digitale informatie, je toch de correcte gegevens of bestanden terug in handen krijgt (decoderen). Daar merk je in de praktijk niets van. Dat verbeteren van fouten zit in de techniek ingebakken. Bovendien zorgt wiskunde ervoor dat deze coderingswijze inclusief de mogelijkheid fouten te verbeteren efficiënt gebeurt. Efficiëntie is van belang om het geheugen en het netwerk niet onnodig te belasten.

Op beide aspecten gaan we in.

Schematische weergave van de rol van coderen

Zonder nog op de wiskunde in te gaan, kun je je de volgende voorstelling maken van wat er gebeurt als je informatie digitaal opslaat en bijvoorbeeld verstuurt. Kijk maar in het volgende schema.

informatie	coderen	verzenden	ontvangen	decoderen	informatie
1101	--→ 1101010	--→ *verstoring*	--→ 1001010	--→ 1101010	--→ 1101

Eerst wordt informatie, bijvoorbeeld een gewoon woord, op een of andere systematische wijze met 0'en en 1'en beschreven. Daarna voegt men op een doordachte wijze extra 0'en en 1'en toe die dienen om later fouten te kunnen verbeteren (in het schema drie 0'en en 1'en). Bij verzending van het rijtje 0'en en 1'en gaat er mogelijk iets mis, zodat de ontvanger een iets ander rijtje 0'en en 1'en krijgt. Maar door de slimme codering is (het apparaat van) de ontvanger in staat de juiste informatie boven water te krijgen.

Decoderen

Het idee achter decoderen zit hem hierin: de gedachte is dat als er iets (maar niet teveel) is fout gegaan met digitale informatie de foute informatie nog veel lijkt op de goede informatie. Om te begrijpen wat je hieraan hebt, moet je maar eens kijken of je ziet welk woord hier bedoeld is:

ondewrijs

Je ziet ongetwijfeld dat het juiste woord *onderwijs* is, omdat het woord *onderwijs* een bestaand woord is dat er het meest op lijkt. Op soortgelijke wijze zul je ook wel niet veel moeite hebben met de volgende zin.

Je ewet avst wel wta heir staat.

Deze zin zit vol fouten, maar je hebt al snel in de gaten dat wel bedoeld zal zijn: *Je weet vast wel wat hier staat*. Dat is een zinvolle zin die het 'dichtst' bij de foute zin komt. In de coderingstheorie heeft men dit idee van op elkaar lijken of dicht bij elkaar liggen uitgewerkt in de wereld van `0`'en en `1`'en. Wil je een mini-voorproefje? Stel je eens voor dat we alleen de letters a en b coderen, en wel a met behulp van `000` en b met behulp van `111`. Je verstuurt de letter a, dus de cijferreeks `000`. Maar helaas, onderweg gaat er op atomair niveau iets mis en een `0` wordt veranderd in een `1`, zodat de ontvanger bijvoorbeeld `010` ontvangt. Die ontvanger (en in de praktijk doet de machine dit werk) concludeert dat `010` meer op `000` lijkt dan op `111` en dat dus wel `000` en dus de letter a bedoeld zal zijn.

Als je alleen de letters a en b wilt coderen, kun je dat met één bit doen: een a codeer je met een `0`, een b met een `1`. Maar het net gegeven voorbeeld laat zien dat je soms toch meer bits moet gebruiken dan op het eerste gezicht noodzakelijk lijkt, omdat je dan de deur open houdt om fouten te herstellen.

1 Opgave. (*Waarom soms meer bits nodig zijn*)

a) Codeer a met `0` en b met `1`. Als a (dus `0`) verstuurd wordt en onderweg wordt de `0` veranderd in `1`, wat zal de ontvanger dan concluderen?

b) Nu coderen we a met `00` en b met `11`. We versturen `00`, maar onderweg gaat er iets fout waardoor de ontvanger `10` ontvangt. Wat denk je dat de ontvanger concludeert?

c) Als het goed is zie je nu in dat de codering `000` voor a en `111` voor b (dus met 3 bits) nog niet zo'n gekke keuze is. Er kan dan 'één fout verbeterd worden'.

2 Opgave. (*Decoderen in gewone taal*)
In de net genoemde voorbeelden met woorden en zinnen is er alleen iets mis gegaan met de volgorde van de letters. Maar natuurlijk kunnen ook wel eens de verkeerde letters gebruikt zijn.

a) Decodeer *gescviedenit*.

b) Bij sommige foute woorden kom je in de problemen met decoderen: er zijn meerdere 'juiste' woorden mogelijk. Laat zien dat je *ralen* op meerdere manieren kunt decoderen.

2 Coderen van informatie: het binaire alfabet

2.1 Informatie vastleggen: daar heb je een alfabet voor nodig

Vanwege de elektrotechnische achtergrond van schakelingen is de keuze om informatie vast te leggen met `0`'en en `1`'en wel te begrijpen. Maar als we computers loslaten, zijn er ook andere mogelijkheden. Uit het dagelijkse leven ben je al gewend om met het gewone alfabet te werken om taal weer te geven. En het kan nog anders. Zo werden vroeger (vóór de tijd van de computers) boodschappen elektronisch overgestuurd in Morse-code met lange en korte signalen (die op schrift met korte en lange streepjes worden weergegeven). Getallen beschrijven we in het dagelijkse leven met behulp van het 10-tallig stelsel. De Grieken gebruiken een ander alfabet (α, β, ...) om hun woorden op te schrijven. Russen, Chinezen, Arabieren gebruiken weer andere manieren. Maar wat alle manieren gemeen hebben, is dat er met een bepaald rijtje symbolen een soort woorden gemaakt worden.

In de digitale wereld is het alfabet dat uit een `0` en een `1` bestaat fundamenteel. Daar gaan we verder op in.

2.2 Het alfabet coderen met rijtjes van vijf `0`'en en `1`'en

De rijtjes `10101` en `00110` zijn twee voorbeelden van rijtjes ter lengte vijf die bestaan uit `0`'en en `1`'en. Op elk van de vijf plaatsen kun je een `0` of een `1` zetten. Dat levert in totaal $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^5 = 32$ verschillende rijtjes op die uit `0`'en en `1`'en bestaan. Met deze 32 rijtjes kun je 32 verschillende symbolen coderen, bijvoorbeeld de 26 letters van het alfabet en nog zes leestekens, zoals *. , ; : -* en *spatie*. De volgende lijst geeft een voorbeeld van zo'n codering. Als je goed kijkt naar de lijst, zie je dat we de rijtjes met `0`'en en `1`'en op een systematische manier aflopen. Dat is in dit voorbeeld verder niet van belang.

`00000`	↔	*a*	`10000`	↔	*q*
`00001`	↔	*b*	`10001`	↔	*r*
`00010`	↔	*c*	`10010`	↔	*s*
`00011`	↔	*d*	`10011`	↔	*t*
`00100`	↔	*e*	`10100`	↔	*u*
`00101`	↔	*f*	`10101`	↔	*v*
`00110`	↔	*g*	`10110`	↔	*w*
`00111`	↔	*h*	`10111`	↔	*x*
`01000`	↔	*i*	`11000`	↔	*y*
`01001`	↔	*j*	`11001`	↔	*z*
`01010`	↔	*k*	`11010`	↔	.
`01011`	↔	*l*	`11011`	↔	,
`01100`	↔	*m*	`11100`	↔	;
`01101`	↔	*n*	`11101`	↔	:
`01110`	↔	*o*	`11110`	↔	-
`01111`	↔	*p*	`11111`	↔	*spatie*

Het woord *adel* codeer je dan als volgt: de *a* vervangen we door het rijtje `00000`, de *d* door `00011`, de *e* door `00100` en de *l* door `01011`. In totaal krijgen we de volgende rij van 20 `0`'en en `1`'en:

`00000000110010001011`

Om een rij van `0`'en en `1`'en om te zetten in een woord of een zin, verdelen we de rij in groepen van vijf en gebruiken de tabel om terug te vertalen naar 'normale' taal. Bijvoorbeeld, als je

`00010011100001100100`

ontvangt, dan splits je in groepen van vijf:

`00010 01110 00011 00100`

en decodeert de rij als het woord *code*.

Fouten kun je niet verbeteren met deze code
Als je het woord *meen* verstuurt, dan verstuur je de digitale code daarvoor, dus

`01100001000010001101`

Gaat er op de vijfde plaats iets fout, dan krijgt de ontvanger de rij bits

`01101001000010001101`,

splitst dit in groepen van 5:

`01101 00100 00100 01101`

en concludeert dat het woord *neen* verstuurd is. Wel een verschil. Als er iets is fout gegaan zal er doorgaans een onzinwoord ontstaan. In dat geval zal de ontvanger wel concluderen dat er iets is misgegaan.

In de volgende paragraaf gaan we bespreken hoe je de codering kunt inrichten om fouten te verbeteren.

3 Decoderen en fouten verbeteren

In gewone woorden zit doorgaans nogal wat overbodige informatie: zoals we zagen, kun je aan een stukje van een (wat langer) woord of aan een foutief geschreven woord vaak al herkennen wat bedoeld is. Het bedoelde woord ligt zogezegd dichtbij het foutieve woord. We zitten hier bij een grondgedachte van de coderingstheorie: zodanig binaire woorden maken dat een 'binair foutje' niet erg is. Dan moeten we wiskundig preciseren wat we bedoelen met binaire woorden die dicht bij elkaar liggen. Hiervoor gebruikt men het begrip *afstand tussen twee binaire woorden*. We beperken ons tot binaire woorden van een vaste lengte.

Afstand tussen twee binaire woorden
De *afstand* tussen twee even lange rijtjes `0`'en en `1`'n is het aantal plaatsen waarop de twee verschillen. We geven dat wel aan met de letter d van distance. Bijvoorbeeld: $d(\texttt{100}, \texttt{001}) = 2$ omdat `100` en `001` op precies twee plaatsen verschillen, de eerste en de laatste.

Wat heeft afstand met fouten verbeteren te maken?
Eerder hebben we 32 symbolen (26 letters en 6 leestekens) gecodeerd met behulp van *alle* binaire rijtjes ter lengte 5. Voor het verbeteren van fouten is dat geen verstandige keuze omdat een enkel foutje in een binair stuk meteen een binair rijtje oplevert dat bij een andere letter of ander leesteken hoort. In termen van afstanden: er zijn nogal wat codewoorden die op afstand 1 van elkaar liggen.

De kunst is om een set van codewoorden te gebruiken waarvan elk tweetal gegarandeerd op zekere afstand, bijvoorbeeld 3 of meer, van elkaar ligt. Denken we aan die rijtjes van 5, en gebruiken we minder codewoorden, dan is de prijs sowieso dat we niet elke letter of elk

leesteken er meer mee kunnen coderen. Willen we toch alle 32 symbolen kunnen coderen, dan zullen we langere rijtjes moeten gebruiken. Wiskunde is belangrijk bij het systematisch vinden van binaire rijtjes die zekere minimumafstand tot elkaar hebben. Op zo'n constructie gaan we later in.

Als elk tweetal codewoorden eens afstand minstens 3 had...
Denk je eens in dat alle gebruikte codewoorden op afstand minstens 3 van elkaar liggen. Als er één foutje ontstaat in zo'n codewoord, zeg c, dan heeft het resulterende binaire woord, zeg c', afstand 1 tot het codewoord. Het veranderde woord is dus geen codewoord meer. De afstand van c' tot elk codewoord verschillend van c is nu minstens 2.

3 Opgave. (*Over afstanden tussen binaire woorden*)
Kun je dit beredeneren?

a) Gegeven zijn drie binaire rijtjes van dezelfde lengte: a, b en c. Gegeven is $d(a,b) = 2$, $d(a,c) = 10$. Ligt b dichter bij a of bij c?

b) Laat a, b en c drie binaire rijtjes zijn van dezelfde lengte. Veronderstel dat a en b op k plaatsen verschillen, en b en c op ℓ plaatsen. Op hoeveel plaatsen kunnen a en c dan maximaal verschillen? Kun je je resultaat uitdrukken in termen van een ongelijkheid waarin $d(a,b)$, $d(b,c)$ en $d(a,c)$ voorkomen?

Er is dus een uniek codewoord, namelijk c, waar c' het dichtst bij ligt. Bij ontvangst van c' zal een ontvanger dit decoderen als c.

4 Opgave. (*Twee fouten decoderen*)
Codeer a als `00000` en b als `11111`. Wat is de afstand tussen deze twee codewoorden? Laat zien dat je nu *twee* fouten kunt corrigeren.

4 Waar is nu die wiskunde om codes te maken?

4.1 Er zit wiskunde in die 0'en en 1'en: binair rekenen

Je vraagt je inmiddels misschien af waar nu de wiskunde om de hoek komt kijken bij het (de)coderen. Die wiskunde nu speelt voornamelijk een rol bij het construeren van sets van codewoorden met gegarandeerde minimumafstand. Maar dan moet je wel meer kunnen met die 0'n en 1'n, en wel rekenen! Optellen en vermenigvuldigen dus. Juist die rekenstructuur heeft wiskundigen in staat gesteld coderingssystemen te ontwerpen die je anders niet zou hebben kunnen vinden.

Omdat er maar twee symbolen zijn, is eenvoudig uitgelegd hoe je optelt en vermenigvuldigt:

Optelling:	$0+0=0$	$0+1=1$	$1+0=1$	$\underline{1+1=0}$
Vermenigvuldiging:	$0\cdot 0=0$	$0\cdot 1=0$	$1\cdot 0=0$	$1\cdot 1=1$

De meest bijzondere rekenregel is onderstreept: $1+1=0$. Dit rekensysteem(pje) is natuurlijk anders dan wat je gewend bent. Je kunt er bijvoorbeeld als volgt aan denken. Het gaat eigenlijk om rekenregels voor de even en oneven getallen. De even getallen zijn de getallen $\ldots, -4, -2, 0, 2, 4, 6, \ldots$; de oneven getallen zijn de getallen $\ldots -5, -3, -1, 1, 3, 5, 7, \ldots$. Waarschijnlijk weet je wel dat als je twee even getallen optelt, je weer een even getal krijgt. En als je een oneven en een even getal optelt krijg je een oneven getal. En als je twee oneven

getallen optelt krijg je een even getal. Geven we de even getallen met een `0` weer en de oneven getallen met een `1`, dan ontstaan de genoemde rekenregels. Hier is een klein tabelletje.

$+$	`0`	`1`
`0`	`0`	`1`
`1`	`1`	`0`

$+$	even	oneven
even	even	oneven
oneven	oneven	even

En voor de vermenigvuldiging zien de tabelletjes er zo uit:

$\cdot$	`0`	`1`
`0`	`0`	`0`
`1`	`0`	`1`

$\cdot$	even	oneven
even	even	even
oneven	even	oneven

Het is werkelijk een heel eenvoudig rekensysteem. Hier zijn wat voorbeelden van berekeningen.

a) $(\mathtt{1}+\mathtt{1})\cdot\mathtt{1}=\mathtt{0}\cdot\mathtt{1}=\mathtt{0}$.

b) $\mathtt{1}+\mathtt{1}+\mathtt{1}+\mathtt{1}+\mathtt{1}=\mathtt{1}$.

Misschien valt je op dat we het niet over delen en aftrekken hebben gehad. Kun je aan de rekenregels zien waarom die twee operaties overbodig zijn?

5 Opgave. (*Rekenen met* `0`*'n en* `1`*'n*)
Bereken in het binaire rekensysteem.

a) $(\mathtt{1}+\mathtt{1}+\mathtt{1})\cdot(\mathtt{1}+\mathtt{1}+\mathtt{1}+\mathtt{1})$.

b) $\mathtt{1}+(\mathtt{1}+\mathtt{1})\cdot\mathtt{1}+\mathtt{1}$.

Dit rekensysteem heeft wel vele ingewikkelde en nuttige varianten. Vooral in de vakgebieden algebra en getaltheorie van de wiskunde kom je ze tegen. En bij toepassingen in de coderingstheorie en cryptologie.

4.2 Een vergelijking voor een controlebit

Een hele lichte bescherming tegen foutjes is het toevoegen van een zogenaamd *controlebit* aan de codewoorden. In ons voorbeeld waarin we 32 letters en leestekens codeerden met alle binaire woorden van lengte 5, kan dit op de volgende manier: we voegen aan elk codewoord een `0` of een `1` toe op zo'n manier dat het totaal aantal enen even is. Dus `00100` verlengen we tot `001001`, terwijl we `01010` verlengen tot `010100`.

In ons binair rekensysteem telt een even aantal enen op tot `0` (elk groepje van twee enen telt op tot `0` enz.). Bijvoorbeeld: $\mathtt{1}+\mathtt{1}+\mathtt{1}+\mathtt{1}=\mathtt{0}$. Een oneven groepje enen telt op tot `1`. Als we een rijtje van vijf bits hebben, en die noteren we met de letters x, y, z, u, v, dan moeten we blijkbaar de zesde bit w zó kiezen dat $x+y+z+u+v+w=\mathtt{0}$ in ons binaire systeem. Als we aan beide kanten w optellen, dan krijgen we

$$x+y+z+u+v+w+w=w.$$

Omdat $w+w=\mathtt{0}$ of w nu `0` of `1` is, vereenvoudigt deze regel tot

$$w=x+y+z+u+v.$$

Kortom, voor het zesde bit neem je gewoon de som van de eerste vijf bits. Het resultaat is een set van 32 codewoorden waarvan de som van de bits *altijd* gelijk is aan `0`.

Het nut van dit zesde bit zit 'm in het volgende. Als er één fout gemaakt wordt, dan is er dus een `0` in een `1` veranderd of een `1` in een `0`. De som van de bits is in beide gevallen niet meer gelijk aan `0`, maar gelijk aan `1`. We spreken wel van een *pariteitscheck*, een check op het even zijn of oneven zijn van het aantal eentjes in een binair woord. Je concludeert nu dat er iets is fout gegaan. Helaas kun je niet aangeven wat er fout is gegaan. Je kunt wel detecteren dat er iets is fout gegaan, maar je kunt de fout niet verbeteren.

6 Opgave. Je kunt ook rijtjes `0`'en en `1`'en optellen. Voor wat komen gaat is het even verstandiger de rijtjes zo op te schrijven:

$$(\mathtt{0},\mathtt{1},\mathtt{0},\mathtt{1},\mathtt{0},\mathtt{1}) \text{ in plaats van } \mathtt{010101},$$

de cijfers gescheiden door komma's en het hele rijtje ingeklemd tussen haken.

(a) Definieer een pleksgewijze optelling: wat wordt

$$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n)$$

(de som van twee rijtjes ter lengte n)? Geef een paar voorbeelden.

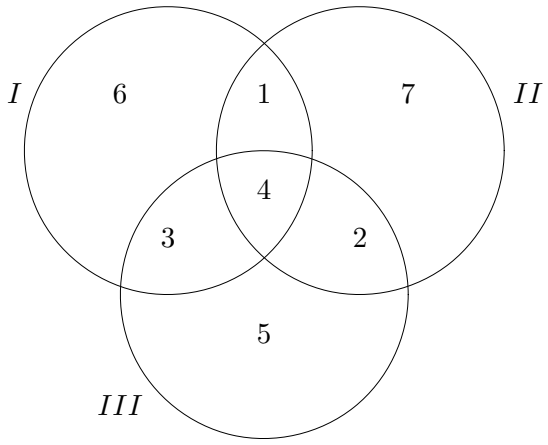
(b) We gaan geen rijtjes met elkaar vermenigvuldigen, maar enkel een rijtje met een van de getallen `0` of `1`. Wat zou een redelijke definitie zijn van

$$\mathtt{0} \cdot (x_1, x_2, \ldots, x_n), \quad \mathtt{1} \cdot (x_1, x_2, \ldots, x_n)?$$

5 De Hamming-code

5.1 *Hamming-code met cirkels.*
De Hamming-code is een code die drie extra bits (drie controlebits) toevoegt aan de rijtjes van vier `0`'en en `1`'en. Zo ontstaat een code met 16 ($= 2^4$) codewoorden waarin elk tweetal codewoorden op minstens drie plaatsen van elkaar verschilt. De code is dus 1-fout herstellend: dat wil zeggen, als in zo'n codewoord op één plaats een foutje ontstaat, dan is die fout te vinden en te verbeteren.

Aan de hand van het volgende schema leggen we uit hoe je de drie extra bits moet toevoegen. We illustreren dit aan de hand van het rijtje `1011`.

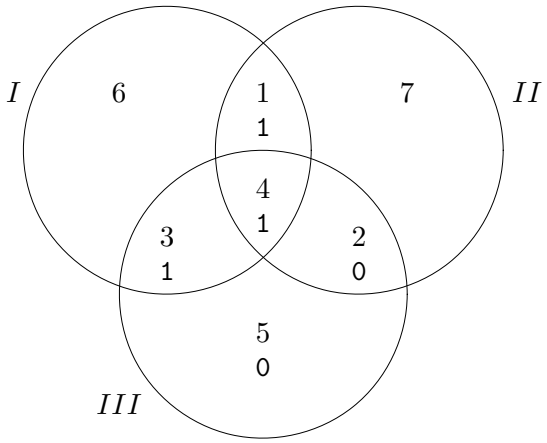


Teken drie cirkels I, II, III en nummer de zeven ontstane binnengebieden als in de figuur. In elk gebied komt een `0` of een `1` van het codewoord te staan en de regel is dat elk van de drie cirkels een even aantal `1`'en moet bevatten. Hoe maak je nu met dit schema uit een rijtje van vier symbolen een rijtje van zeven symbolen?

- Schrijf het eerste symbool in gebied 1, het tweede in gebied 2, het derde in gebied 3 en het vierde in gebied 4.
- Bepaal de drie laatste symbolen met de regel:

in elk van de drie cirkels staat een even aantal `1`'en.

Om te bepalen wat er op plaats 5 moet komen, kijken we naar cirkel III: er staan twee `1`'en en één `0`. Er moet dus op plaats 5 een `0` komen te staan. Deze `0` is al in de volgende figuur aangegeven.



7 Opgave. De zesde en zevende plek zijn nog niet ingevuld met een `0` of `1`. Doe dit zelf. Wat wordt dus het hele codewoord van 7 bits?

De Hamming-code: nu algebraïsch

Net hebben we de Hamming-code besproken aan de hand van een plaatje. Nu kijken we meer algebraïsch. Als we de eerste vier bits aangeven met x_1, x_2, x_3, x_4, dan kunnen we de 5de, 6de en 7de bit (aangegeven met x_5, x_6 en x_7) bepalen op de volgende manier. Om x_5 te bepalen, gebruiken we cirkel III. De som van de bits moet `0` zijn, dus $x_2 + x_3 + x_4 + x_5 = \texttt{0}$. Links en rechts x_5 optellen levert

$$x_5 = x_2 + x_3 + x_4.$$

Net zo vinden we $x_6 = x_1 + x_3 + x_4$ (met cirkel I) en $x_7 = x_1 + x_2 + x_4$ (met cirkel II). Samen:

$$\begin{aligned} x_5 &= x_2 + x_3 + x_4 \\ x_6 &= x_1 + x_3 + x_4 \\ x_7 &= x_1 + x_2 + x_4 \end{aligned}$$

Met deze drie vergelijkingen kun je dus het 5de, 6de en 7de bit bepalen.

De Hamming-code is lineair: sommen van codewoorden zijn codewoorden

Aan deze vergelijkingen kunnen we een belangrijke eigenschap van de Hamming-code aflezen, namelijk dat de code *lineair* is: sommen van codewoorden zijn ook weer codewoorden. In symbolen: als $(x_1, \ldots, x_7)$ en $(y_1, \ldots, y_7)$ codewoorden zijn, dan is de som $(x_1+y_1, \ldots, x_7+y_7)$ ook een codewoord. Om dit na te gaan, moeten we controleren of deze som aan de drie vergelijkingen voldoet. Dus het 5de bit hiervan moet de som zijn van het 2de, 3de en 4de bit, het 6de bit de som van het 1ste, 3de en 4de, enz. Het 5de bit van de som is $x_5 + y_5$. De 2de, 3de en 4de bit zijn achtereenvolgens $x_2 + y_2$, $x_3 + y_3$ en $x_4 + y_4$. Tel $x_5 = x_2 + x_3 + x_4$ en $y_5 = y_2 + y_3 + y_4$ op en herschik een beetje:

$$x_5 + y_5 = (x_2 + x_3 + x_4) + (y_2 + y_3 + y_4) = (x_2 + y_2) + (x_3 + y_3) + (x_4 + y_4),$$

zoals gewenst. In een schema:

5de bit	2de bit	3de bit	4de bit
x_5	x_2	x_3	x_4
y_5	y_2	y_3	y_4
$x_5 + y_5$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$

Voor het 6de en het 7de bit verloopt de controle net zo. Loop die zelf eens na.

8 Opgave. Gegeven zijn twee codewoorden die afstand 4 hebben (dus op precies plaatsen verschillen). Op hoeveel plaatsen heeft hun som dan een een `1` staan?

De minimumafstand van de Hamming-code is 3

De vergelijkingen en de lineariteitseigenschap kunnen je ook helpen inzien dat de afstand tussen elk tweetal codewoorden (d.w.z. oplossingen van de vergelijkingen) minstens 3 is. Veronderstel namelijk maar dat er twee woorden zijn waartussen de afstand minder dan 3 is, dus 1 of 2. De som van de twee codewoorden is dan ook een codewoord zoals we zagen, maar met precies één `1` of precies twee `1`'n (op precies de plek(ken) waar de twee codewoorden verschillen). Kijk nu naar de vergelijkingen, maar herschreven als hieronder. Is er bijvoorbeeld één `1` in het codewoord en wel op plaats 3, dan levert invullen in de eerste twee vergelijkingen $\mathtt{1} = \mathtt{0}$. Dat kan dus niet. Net zo als er een `1` op precies één andere plaats voorkomt.

$$\begin{array}{rcrcrcrcrcrcr} & & x_2 & + & x_3 & + & x_4 & + & x_5 & & & & = \mathtt{0} \\ x_1 & + & & + & x_3 & + & x_4 & + & & + & x_6 & & = \mathtt{0} \\ x_1 & + & x_2 & + & & + & x_4 & + & & & & + & x_7 = \mathtt{0} \end{array}$$

Als er een `1` op precies twee plaatsen voorkomt, dan kun je op soorgelijke wijze concluderen dat niet aan de vergelijkingen is voldaan. Kijk maar eens wat er misgaat als $x_3 = x_5 = \mathtt{1}$.

De vergelijkingen kunnen je helpen bij het beter begrijpen van de Hamming-code, maar dan moet je je wel wat meer verdiepen in de wereld van `0`'n en `1`'n.

9 Opgave. We bekijken nu een code die aan een k-tal informatiebits twee controlebits toevoegt. Laat zien dat er in zo'n code dan altijd twee codewoorden zitten op afstand ≤ 2, dus dat deze code nooit minimumafstand 3 kan hebben. Concludeer dat de Hamming-code de meest efficiënte code is om in rijtjes van 4 bits 1 fout te kunnen verbeteren.

Alles goed en wel: maar hoe verzon men de Hamming-code?

Het voert te ver om in kort bestek uit de doeken te doen hoe je zo'n code als de Hamming-code kunt verzinnen. Het heeft iets met vectoren te maken, maar ook iets met het tweetallig stelsel. Als je de coëfficiënten van de variabelen in de drie vergelijkingen (gelezen van boven naar beneden) verzamelt en groepeert in drietallen, krijg je:

$$\texttt{011}, \quad \texttt{101}, \quad \texttt{110}, \quad \texttt{111}, \quad \texttt{100}, \quad \texttt{010}, \quad \texttt{001}.$$

Bij de variabele x_1 lees je in de vergelijkingen dus af: `0` (eerste vergelijking), `1` (tweede vergelijking), `1` (derde vergelijking). Hier staan in het tweetallig stelsel de (gewone decimale) getallen 1, 2, 3, 4, 5, 6, 7 uitgeschreven. Zo staat `101` voor $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$.

10 Opgave. Schrijf de getallen 1, 2, 3, 4, 5, 6, 7 in het tweetallig stelsel.

Codes: tot slot

In CD-spelers combineert men op listige manier codes waarvan de lengte van de codewoorden 28 respectievelijk 32 is en die een veel grotere foutverbeterende capaciteit hebben. Op een muziek-CD kunnen wel een half miljoen fouten staan terwijl je daar bij het afspelen niets van merkt.