Reed–solomon-koder

1 Innledning og motivasjon

Har du noen gang tenkt på hvorfor en DVD eller CD kan spilles av selv om den har små riper? Hvordan kan datamaskinen fortsatt lese informasjonen riktig når noen deler av platen er skadet? Hemmeligheten ligger i *feilrettende koder*. Disse kodene legger til ekstra (redundant) informasjon til dataene slik at selv om deler av dataene blir ødelagt, kan de originale dataene rekonstrueres. Et av de mest brukte systemene for slik feilretting er *Reed–Solomon-koder*. I dette papiret skal vi fra grunnen av utforske hva Reed–Solomon-koder er, hvordan de fungerer, og hvorfor de er så nyttige i praksis.

For å forstå Reed–Solomon-koder trenger vi først litt bakgrunn i matematikken som ligger bak. Spesielt må vi lære om *Galois-kropper*, som er endelige matematiske strukturer der vi kan utføre operasjoner som addisjon, subtraksjon, multiplikasjon og divisjon (utenom divisjon på null) – kort sagt alt vi kan gjøre med vanlige tall. Vi vil også se på polynomer over slike strukturer, og et viktig verktøy kalt Lagrange-interpolasjon som lar oss finne igjen et polynom ut fra verdiene det tar. Til slutt knytter vi dette sammen og forklarer Reed–Solomon-kodene, og vi ser på et praktisk eksempel med to figurer, Pipp og Pepp, samt noen moderne anvendelser (for eksempel hvordan QR-koder og DVD-er bruker dette).

2 Galois-kropper: $\mathrm{GF}(p)$ og $\mathrm{GF}(p^m)$ med eksempler

En *Galois-kropp* (ofte skrevet som GF, fra engelsk *Galois Field*) er en *endelig kropp*, det vil si et system med et begrenset antall elementer der man kan legge sammen, trekke fra, gange og dele (som sagt, dele forutsatt at det ikke er deling på null). Det enkleste eksempelet er heltallsaritmetikk modulo et primtall. For eksempel, hvis vi ser på heltall modulo 7, så har vi bare syv elementer: $\{0, 1, 2, 3, 4, 5, 6\}$. Vi definerer addisjon og multiplikasjon som vanlig, men hver gang vi får et resultat som er større enn 6 eller negativt, så *tar vi resten* etter deling på 7 (det vi kaller modulo-aritmetikk). Denne strukturen kalles GF(7) fordi den har 7 elementer. Alle ikke-null elementer har en invers under multiplikasjon; for eksempel i GF(7) er $3 \cdot 5 = 15 \equiv 1 \pmod 7$, så 3 og 5 er hverandres multiplikative invers (5 er «det samme som» 3^{-1} i

denne kroppen). Tilsvarende er $2 \cdot 4 = 8 \equiv 1 \pmod 7$, så $2^{-1} = 4$ i GF(7). Dermed oppfyller disse 7 tallene alle egenskapene til en kropp. Generelt er GF(p) (for p et primtall) kroppen bestående av heltallene $0, 1, \ldots, p-1$ med operasjoner definert modulo p.

Men hva om vi ønsker oss en kropp med p^m elementer, hvor $m > 1$? Hvis p^m ikke er et primtall, kan vi ikke lenger bruke en enkel modulo-aritmetikk på heltall. Løsningen er å konstruere nye elementer ved hjelp av polynomer. Uten å gå for dypt inn i teori kan vi forklare idéen slik: For å få GF(p^m), kan vi starte med GF(p) og introdusere et polynom av grad m som ikke lar seg faktorisere (akkurat som et primtall er et heltall som ikke kan faktoriseres i mindre faktorer). Dette polynomet bruker vi til å definere et nytt tall α som er en rot av polynomet. Et konkret eksempel: For å lage GF(2^3), altså en kropp med 8 elementer, kan vi ta utgangspunkt i GF(2) = $\{0, 1\}$ (tall mod 2) og polynomet $f(x) = x^3 + x + 1$. Dette polynomet har ingen røtter i GF(2) (man kan sjekke at $f(0) = 1$ og $f(1) = 1 + 1 + 1 = 1$, ingen gir 0), og dermed fungerer det som vår «primfaktor». Vi innfører et symbol α og sier at α er en rot av $f(x)$, altså at $\alpha^3 + \alpha + 1 = 0$. Fra dette følger at $\alpha^3 = \alpha + 1$ innenfor den nye strukturen. Nå kan *hvert* element i GF(8) skrives som $a + b\alpha + c\alpha^2$, der $a, b, c \in \{0, 1\}$. Totalt finnes $2^3 = 8$ slike kombinasjoner (inkludert $0 + 0\alpha + 0\alpha^2 = 0$). Vi definerer addisjon ved å addere koeffisientene (mod 2) i tilsvarende uttrykk. For eksempel, $(1 + \alpha) + (\alpha + \alpha^2) = 1 + 0\alpha + \alpha^2$ (fordi $\alpha + \alpha = 0$ i GF(2) aritmetikk). Multiplikasjon defineres ved å multiplisere som polynomer og så bruke relasjonen $\alpha^3 = \alpha + 1$ for å redusere resultatet tilbake til formen $a + b\alpha + c\alpha^2$. For eksempel, $\alpha \cdot (\alpha^2 + 1) = \alpha^3 + \alpha = (\alpha + 1) + \alpha = 1$ (siden α^3 byttes ut med $\alpha + 1$). Det viser seg at denne konstruksjonen gir oss en fullverdig kropp med 8 elementer, GF(8).

På lignende vis finnes det en (unik) kropp med p^m elementer for alle primtall p og positive heltall m. Noen ganger skriver vi $\mathbb{F}_{p^m}$ for å betegne en slik endelig kropp (vanlig notasjon i matematikk). For eksempel er $\mathbb{F}_8$ det samme som GF(2^3) i eksempelet over, og $\mathbb{F}_9$ (9 elementer) kan konstrueres fra GF(3) med et irreduserbart polynom av grad 2. I dette papiret vil vi primært bruke små eksempler som GF(7) eller GF(8) for å illustrere konseptene.

2.1 Konkret eksempel av en Galois-krop

La oss se på et konkret eksempel for å forstå hvordan en Galois-kropp fungerer. Vi vil konstruere kroppen GF(2^3), som har $2^3 = 8$ elementer. Dette er en utvidelse av den enkleste Galois-kroppen, GF(2), som består av elementene $\{0, 1\}$ med addisjon og multiplikasjon modulo 2.

2.2 Konstruksjon av $\mathrm{GF}(2^3)$

For å konstruere $\mathrm{GF}(2^3)$, trenger vi et irreduktibelt polynom av grad 3 over $\mathrm{GF}(2)$. Et slikt polynom er:

$$f(x) = x^3 + x + 1$$

Dette polynomet har ingen røtter i $\mathrm{GF}(2)$, og derfor kan vi bruke det til å definere en utvidelse. Vi introduserer et symbol α som er en rot av $f(x)$, det vil si at:

$$\alpha^3 + \alpha + 1 = 0 \quad \Rightarrow \quad \alpha^3 = \alpha + 1$$

2.3 Elementene i $\mathrm{GF}(2^3)$

Hvert element i $\mathrm{GF}(2^3)$ kan uttrykkes som et polynom av grad mindre enn 3 med koeffisienter i $\mathrm{GF}(2)$. Det vil si, hvert element har formen:

$$a_0 + a_1\alpha + a_2\alpha^2 \quad \text{der } a_i \in \{0, 1\}$$

Det finnes $2^3 = 8$ slike kombinasjoner. Vi kan liste dem opp i en tabell:

Binær	**Polynomform**	**Potenser av α**
000	0	—
001	1	α^0
010	α	α^1
011	$\alpha + 1$	α^3
100	α^2	α^2
101	$\alpha^2 + 1$	α^6
110	$\alpha^2 + \alpha$	α^4
111	$\alpha^2 + \alpha + 1$	α^5

2.4 Addisjon og multiplikasjon i $\mathrm{GF}(2^3)$

Addisjon utføres ved å legge sammen koeffisientene modulo 2 (dvs. som en XOR-operasjon). For eksempel:

$$(\alpha^2 + \alpha) + (\alpha + 1) = \alpha^2 + (\alpha + \alpha) + 1 = \alpha^2 + 0 + 1 = \alpha^2 + 1$$

Multiplikasjon utføres ved å multiplisere polynomene og deretter redusere resultatet modulo $f(x) = x^3 + x + 1$. For eksempel:

$$\alpha \cdot (\alpha + 1) = \alpha^2 + \alpha$$

$$\alpha^2 \cdot \alpha = \alpha^3 = \alpha + 1$$

3 Polynomer over endelige kropper

Et *polynom* er et matematisk uttrykk som kombinerer koeffisienter og en variabel x i potenser, for eksempel:

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

der $a_0, a_1, \ldots, a_n$ er koeffisienter fra en gitt mengde tall.

Når vi arbeider over en *endelig kropp* (også kjent som en Galois-kropp), betyr det at både koeffisientene og variabelen x tar verdier fra en endelig mengde med elementer, for eksempel $\mathrm{GF}(7) = \{0, 1, 2, 3, 4, 5, 6\}$.

I slike kropper utføres alle aritmetiske operasjoner (addisjon, subtraksjon, multiplikasjon og divisjon) med moduloregning. For eksempel, i $\mathrm{GF}(7)$ er $3 + 5 = 1$, siden $3 + 5 = 8$, og $8 \mod 7 = 1$.

Hvorfor er dette viktig?

Å forstå polynomer over endelige kropper er avgjørende fordi:

- **Entydighet:** Et polynom av grad mindre enn k er entydig bestemt av sine verdier i k distinkte punkter. Dette er en grunnleggende egenskap som brukes i kodeteori.
- **Feilretting:** I Reed–Solomon-koder representerer meldingen et polynom, og den sendte koden består av polynomets verdier i ulike punkter. Hvis noen av disse verdiene blir feil under overføring, kan vi bruke interpolasjon (som Lagrange-interpolasjon) til å rekonstruere det opprinnelige polynomet, og dermed rette feilene.

Et konkret eksempel

La oss si at vi har en melding som vi representerer som et polynom $f(x)$ av grad mindre enn k. Vi evaluerer dette polynomet i n forskjellige punkter i en endelig kropp $\mathrm{GF}(q)$ og sender disse verdiene. Hvis opptil $(n - k)/2$ av disse verdiene blir feil under overføring, kan vi fortsatt rekonstruere $f(x)$ nøyaktig ved hjelp av interpolasjon.

Oppsummering

Polynomer over endelige kropper gir en strukturert og pålitelig måte å representere og manipulere data på, spesielt i sammenhenger der feil kan oppstå under overføring eller lagring. De er grunnlaget for mange feilrettingskoder, inkludert Reed–Solomon-koder, som brukes i alt fra CD-er og DVD-er til QR-koder og satellittkommunikasjon.

4 Lagrange-interpolasjon (fullstendig matematisk bevis)

Lagrange-interpolasjon er hovedtingen du skal ha tatt med deg etter å ha lest papiret, og er så å si hele ryggmargen til reed-solomon koder. Jeg ønsker derfor å gå veldig i detalj her, vi starter med å bryte ned komponentene i slutt formelen

$$P(x) \;=\; \sum_{i=0}^{n} y_i \underbrace{\prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}}_{\text{dette er } L_i(x)}.$$

- *Produkttegnet* $\prod_{\substack{j=0 \\ j\neq i}}^{n}$ betyr at vi *ganger* sammen et sett med faktorer for alle j fra 0 til n, bortsett fra $j = i$. Grunnen til at «$j \neq i$» står der, er at vi *ikke* vil ha en brøk hvor nevneren blir null (da $x_i - x_i = 0$).

- Hver brøk ser slik ut:
$$\frac{x - x_j}{x_i - x_j}.$$
- *Telleren* $(x - x_j)$ gjør at hele produktet blir 0 hvis $x = x_j$ (og $j \neq i$). - *Nevneren* $(x_i - x_j)$ er bare en konstant (for hvert j) som «normaliserer» produktet slik at når $x = x_i$, blir hver faktor $\frac{x_i - x_j}{x_i - x_j} = 1$.

- Summen $\sum_{i=0}^{n} y_i \cdot L_i(x)$ betyr at vi bygger polynomet $P(x)$ ved å «skru på» riktig y_i akkurat der $x = x_i$. Alle de andre $L_j(x_i)$ med $j \neq i$ blir 0.

- På denne måten sikrer vi at $P(x)$ «treffer» (x_i, y_i) for hvert i, uten å måtte finne en komplisert løsning. Formelen er «ferdiglaget» for å oppfylle betingelsene.

Dette er hovedidéen bak Lagrange-formelen: - Hvert *Lagrange-basispolynom* $L_i(x)$ er 1 ved x_i, men 0 ved alle andre x_j. - Når du multipliserer $L_i(x)$ med y_i og summerer for alle i, får du et polynom som tar verdien y_i ved x_i og samtidig nuller ut resten.

La oss anta at vi har $n + 1$ distinkte punkter

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n),$$

der alle x_i er forskjellige. Lagrange-interpolasjon gir oss en eksplisitt formel for et polynom

$$P(x)$$

av grad *høyst* n som går gjennom alle disse punktene. Formelen er:

$$P(x) \;=\; \sum_{i=0}^{n} y_i \cdot \underbrace{\prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}}_{L_i(x)}.$$

Her kalles hvert $L_i(x)$ for et *Lagrange-basispolynom* og har spesielle egenskaper som vi straks skal bevise.

4.1 Konstruksjon og idé

Idéen bak konstruksjonen er:

- Bygg et polynom $L_i(x)$ som er lik 1 når $x = x_i$, og 0 når $x = x_j$ for alle $j \neq i$.
- Multipliser deretter $L_i(x)$ med y_i, slik at summen $\sum y_i\, L_i(x)$ blir y_i ved $x = x_i$, men bidrar med 0 ved alle x_j der $j \neq i$.

Dermed sikrer hvert $L_i(x)$ at «riktig» y_i «slås på» ved $x = x_i$, og «slås av» ved alle andre x_j. Når vi legger sammen alle slike termer, får vi et polynom som *nøyaktig* treffer de oppgitte punktene.

4.2 Bevis for at $P(x_i) = y_i$ P(x)=y

Vi viser først at $P(x_i) = y_i$ for hvert i. Se på $L_i(x)$:

$$L_i(x) \;=\; \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

1. Hvis $x = x_i$, får vi

$$L_i(x_i) = \prod_{j\neq i} \frac{x_i - x_j}{x_i - x_j} = \prod_{j\neq i} 1 = 1.$$

2. Hvis $x = x_k$ for $k \neq i$, da blir *en* av faktorene i produktet

$$(x - x_j) = x_k - x_k = 0,$$

slik at $L_i(x_k) = 0$ for alle $k \neq i$.

Nå ser vi på $P(x)$ i det generelle tilfellet:

$$P(x) = \sum_{i=0}^{n} y_i\, L_i(x).$$

Verdien til $P(x)$ ved $x = x_k$ blir:

$$P(x_k) = \sum_{i=0}^{n} y_i \, L_i(x_k).$$

I denne summen er $L_k(x_k) = 1$ og alle andre $L_i(x_k)$ (for $i \neq k$) er 0. Dermed får vi

$$P(x_k) = y_k \cdot 1 + \sum_{i \neq k} y_i \cdot 0 \;=\; y_k.$$

Altså oppfyller $P(x)$ interpolasjonsbetingelsen

$$P(x_i) = y_i \quad \text{for alle } \; i.$$

Visuell takeaway

Tenk deg at hvert $L_i(x)$ er som en fleksibel stang. Vi bøyer stangen slik at den treffer høyden y_i akkurat ved x_i, og lar den samtidig gå gjennom null ved alle de andre punktene x_j. Når vi legger sammen alle disse bøyningene fra hver stang, får vi én samlet kurve som treffer alle punktene perfekt. Dette er den geometriske intuisjonen bak Lagrange interpolasjon.

4.3 Unikhet: Hvorfor finnes det bare ett slikt polynom av grad $\leq n$<= n?

Vi må også vise at *hvis* et polynom av grad *høyst* n går gjennom disse $n+1$ ulike punktene, så *må* det være identisk med $P(x)$ over. Beviset bruker et elegant argument med nullpunkter:

- Anta at det finnes to polynomer, $P_1(x)$ og $P_2(x)$, av grad $\leq n$ som begge går gjennom de $n+1$ punktene (x_i, y_i).
- Da har vi $P_1(x_i) = y_i = P_2(x_i)$ for alle $i = 0, 1, \ldots, n$.
- Se på differansen
  $$D(x) = P_1(x) - P_2(x).$$
  Dette er også et polynom, og $\deg(D) \leq n$. Men vi vet at
  $$D(x_i) = P_1(x_i) - P_2(x_i) = 0 \quad \text{for alle } i = 0, 1, \ldots, n.$$
  Altså har $D(x)$ minst $n+1$ nullpunkter.
- Et ikke-trivielt polynom av grad $\leq n$ kan ha *høyst* n nullpunkter (med mindre det er identisk null). Her har $D(x)$ hele $n+1$ nullpunkter. Den *eneste* muligheten er at $D(x)$ er identisk lik nullpolynomet, dvs. $D(x) = 0$ for alle x.

- Dermed $P_1(x) = P_2(x)$ identisk.

Vi har dermed vist både *eksistens* (via formelen for $P(x)$) og *unikhet* (via differanse-argumentet). Dette fullfører beviset for at det *finnes nøyaktig ett* polynom av grad høyst n som interpolerer de $n + 1$ punktene.

4.4 Hvorfor er Lagrange-interpolasjon nødvendig for Reed–Solomon-koder?

Reed–Solomon-koder benytter nettopp denne egenskapen: Hvis man har et polynom $f(x)$ av grad $< k$, kan man *kode* data ved å beregne $f(x)$ i flere punkter $x_1, x_2, \ldots, x_n$. Man sender så de evaluerte punktene $(x_i, f(x_i))$.

Skulle noen av punktene bli ødelagt (feil data i overføringen), kan mottakeren *gjenfinne* det opprinnelige polynomet $f(x)$ gjennom interpolasjon, så lenge ikke for mange punkter er feil (en mer avansert interpolasjonsmetode benyttes når punktene ikke er helt «perfekte», men idéen er den samme).

- *Hovedidé:* Man kan gjenfinne $f(x)$ fordi (i) polynomet har lav grad (maks $k - 1$), og (ii) selv med noen feil, finnes algoritmer (f.eks. Berlekamp–Welch) som identifiserer hvilke punkter som er gale og rekonstruerer det rette $f(x)$.

- *Hvor kommer Lagrange inn?* I selve kjernebeviset for at slike koder fungerer er vi avhengige av at *et polynom av grad $< k$ er entydig bestemt av k korrekte datapunkter*. Lagrange-interpolasjon er den mest direkte måten å se akkurat dette på.

Dermed er Lagrange-interpolasjon *nødvendig* for å forstå og formelt vise hvorfor Reed–Solomon-koder *virker* som de gjør:

Ingen to forskjellige polynomer av grad $< k$ kan gi like evalueringspunkter i k distinkte x-verdier.

Dette er selve fundamentet i at man kan «rulle tilbake» (rekonstruere) meldingspolynomet til tross for en viss andel feil i transmisjonen. Så istedetfor å sende hele koden 3 ganger hvor da byte forholdet på redundans blir 1 til 3 kan vi heller bare sende meldingspolynomet flere ganger for rekonstruksjon

5 Feilrettingskapasitet i Reed-Solomon-Koder

En av de mest imponerende egenskapene ved Reed-Solomon-koder er deres evne til å oppdage og korrigere feil i dataoverføringer. Dette er spesielt viktig i miljøer med støy eller fysisk skade (for eksempel riper på en CD). Men hvor mange feil kan en Reed-Solomon-kode faktisk rette?

5.1 Kodens parametre

En Reed-solomon-kode spesifierses vanligvig som en (n, k)-kode der:

- k er antall medlingssymboler (det vi ønsker å sende),
- n er totalt antall symbolske verdier som sendes (inkludert redundans),
- $n - k$ er antall redundante symbolske verdier lagt til for feilretting

5.2 Et vanlig tilfelle: $(255, 223)$ over $\mathrm{GF}(2^8)$

Et svært vanlig oppset er $(255, 223)$, altså 223 meldingssymboler og 32 redundans symboler. Denne koden bruker $\mathrm{GF}(2^8)$, altså en Galois-kropp med 256 elementer. Dette oppsettet er utbredt fordi det passer ekstremt godt med hvordan datamaskiner behandler og lagrer informasjon:

- Hver verdi i $\mathrm{GF}(2^8)$ passer perfekt i én byte (8 bits), som er den grunnleggende enheten i digital lagring.
- 256 mulige symboler gir fleksibilitet, men i reed-solomon bruker vi maks 255 (alle unntatt null-elementet som evalueringspunkt)
- $(255, 223)$ gir et balansert forhold mellom feilretting of effektivitet. 223 byte data + 32 byte redundans = god robusthet og lav kostnad med tanke på redundans budsjettet
- Dette formatet støttes bredt i maskinvare og programvare (CD, DVD, QR-koder, DAB, satellitt, Wi-Fi).

Kort sagt: $(255, 223)$ over $\mathrm{GF}(2^8)$ er standard fordi det er både byte-vennlig, rask og robust i praksis

5.3 Feilrettingskapasitet

Reed-solomon-koder kan rette opptil:

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor \text{ symbolfeil}$$

I vårt eksempel med $(255, 223)$: $t = \left\lfloor \frac{255-223}{2} \right\rfloor = \left\lfloor \frac{32}{2} \right\rfloor = 16$

Det betyr at hvis opptil 16 av de 255 symbolene blir feil under overføring (eller lesing), kan mottakeren fortsatt gjenopprette hele meldingen på 223 symboler

5.4 Hvorfor 255?

Når vi bruker $\mathrm{GF}(2^8)$, så finnes det nøyaktig 256 forskjellige symboler, men vi ekskluderer ofte null-elementet som evalueringspunkt i kodeing. Derfor brukes maks $n = 255$ for antall symbolske verdier i én RS-blokk.

5.5 Minimumavstand og MDS-koder

Reed-solomon-koder er det vi kaller *MDS-koder* (Maximum Distance Seperable). Det betyr at når den høyeste mulige minimumavstanden d gitt ved:

$$d = n - k + 1$$

Minimumavstanden forteller hvor mange symboler som må være forskjellige før to meldinger blir betraktet som ulike til punktet hvor man ender opp med å gjette hvilken data som er riktig (som ikke er en situasjon du vil finne deg i). Og hvis man overstiger korrigerings terskelen vil det være flere gyldige kodeord som matematisk fungerer fint for konstruksjonen. Denne avstanden bestemmer også hvor mange feil vi kan oppdage og rette.

- **Rette:** opptil $\left\lfloor \frac{d-1}{2} \right\rfloor$ symbolfeil.
- **Oppdage:** opptil $d - 1$ symbolfeil

I $(255, 223)$-eksempelet har vi:

$$d = 255 - 223 + 1 = 33$$

$$\Rightarrow t = \left\lfloor \frac{33 - 1}{2} \right\rfloor = 16$$

5.6 Hvorfor er Reed-Solomon-Koder MDS-koder?

At minimumavstanden faktisk er $d = n - k + 1$ kommer fra hvordan Reed-Solomon-koder er bygd opp:

- Hver melding representeres som et polynom $f(x)$ med grad mindre enn k, og vi lager et kodeord ved å evaluere $f(x)$ i n ulike punkter.
- Hvis to meldinger er forskjellige, får vi to forskjellige polynomer. Da er forskjellen mellom dem også et polynom av grad mindre enn k.
- Et slikt polynom kan være null i *høyst* $k - 1$ punkter (ellers måtte det vært null overalt, og da har vi bare en haug like polynomer).
- Det betyr at to ulike meldinger kan være like i maks $k - 1$ posisjoner, og må være forskjellige i minst:

$$n - (k - 1)) = n - k + 1$$

Derfor har Reed-Solomon-koder maksimal mulig avstand mellom kodeord, og dette er prinsippet bak *MDS-koder*

6 Avsending og mottakelse av Reed-Solomon-Koder

Så langt har vi implisert at vi "sender et polynom", men dette er ikke helt presist. I praksis sender vi ikke selve polynomet (altså koeffisientene), men en *kodeord-vektor* som består av polynomets verdier i ulike punkter

Fra polynom til kodeord

Gitt en melding som vi representerer ved et polynom $f(x)$ med grad mindre enn k, velger vi n distinkte punkter $x_1, x_2, \ldots, x_n$ i en Galois-kropp GF(q). Deretter evaluerer vi polynomet i disse punktene og sender følgende vektor:

$$(f(x_1), f(x_2), \ldots, f(x_n)$$

Denne vektoren kalles kodeordet. Det er disse verdiene som blir sendt gjennom en kanal eller lagret fysisk, for eksempel på en DVD.

Hvorfor ikke bare sende koeffisientene direkte?

Hvis vi bare sendte koeffisientene til $f(x)$ (altså selve meldingen), har vi nå ingen måte å sjekke eller rette opp feil. Ved å sende polynomets verdier i flere punkter, får vi rom for å oppdage og korrigere feil - fordi polynomet har struktur som vi kan utnytte

7 Dekoding: Hvordan finner vi tilbake til meldingen?

Etter at mottakeren eller avleser har fått en vektor med data - noen riktige, noen kanskje feil - ønsker vi å rekonstruere det opprinnelige polynomet $f(x)$ som representerer meldingen.

Husk at i praksis er disse dataene symboler fra ovennevnte Galois-kropp, ofte GF(2^8), som betyr at hvert symbol kan tolkes som en byte - altså en rekke med 8 biter: enere og nullere. En DVD, QR-kode eller nettverksprotokoll arbeider i bunn og grunn med slike bitstrenger, og polynomene vi bruker virker direkte på disse verdiene.

To typer dekoding

1. Ingen feil: Hvis vi vet at alle verdiene er riktige, går vi rett over på Lagrange-interpolasjon til å finne tilbake til $f(x)$. Vi trenger her kun k punkter, siden et polynom av grad $< k$ er entydig bestemt av sine verdier i k distinkte punkter.

2. Feil i noen punkter: Hvis noen av verdiene er feil - men vi ikke vet hvilke - må vi bruke en mer avansert metode for å rette feil. En kjent metode er *Berlekamp-Welch-algoritmen*.

Hvordan fungerer feilretting

Idéen bak dekoding med feil er å finne to polynomer:

- Et **feilpolynom** $E(x)$ som er null der vi *ikke* har feil.
- Et polynom $Q(x) = E(x) \cdot f(x)$ som kombinerer feilkoden og meldingen.

Vi vet da at:

$$Q(x_i) = E(x_i) \cdot y_i$$

for alle i, der y_i er en mottatt verdi (som for all del kan være feil). Dette gir oss et stort system av ligninger som vi kan løse for å finne $E(x)$ og $Q(x)$. Når vi har disse, kan vi gjenvinne meldingen:

$$f(x) = \frac{Q(x)}{E(x)}$$

Intuisjonen bak

Selv om noen symboler (dvs. noen av en og null byte verdiene) er feil, har vi fortsatt nok korrekt informasjon - fordi vi har sendt mer enn opprinnelig nødvendig. Feilene oppfører seg som bråk"i dataene, og brukt konstruksjonen forsøker å finne en løsning som passer best mulig med strukturen til det gitte polynomet.

Dette fungerer fordi polynomer har svært strenge regler: hvis et lavgradspolynom passer perfekt med mange nok punkter, er det *det eneste* mulige polynomet. Dette gjør at vi kan «gjette» hvilke punkter som ikke passer inn - og dermed finne både hvilke verdier som er feil og hva som egentlig ble sendt

8 Eksempel: Pipp sender ”Hei!” til Pepp

La oss se på et konkret eksempel med små tall for å vise hvordan Reed–Solomon-koder fungerer i praksis — og hvorfor de er så nyttige når ting går galt. Vi møter Pipp og Pepp:

> **Pipp** bor i en vanlig leilighet på Majorstuen.
> **Pepp** bor i et hus laget av utarmet U-238, som stadig forårsaker *bitflips* i all elektronikk.

Pipp ønsker å sende meldingen ”Hei!” til Pepp. Hver bokstav konverteres til ASCII og deretter tolkes som tall i GF(7) (altså modulo 7), for enkelhets skyld. Dette er en forenkling av virkeligheten, men nok til å demonstrere poenget.

Trinn 1: Pipp forbereder meldingen

Meldingen `"Hei!"` har følgende ASCII-verdier:

Tegn	ASCII-verdi
H	72
e	101
i	105
!	33

Vi oversetter så disse til elementer i GF(7) ved å ta resten modulo 7:

$$72 \mod 7 = 2, \quad 101 \mod 7 = 3, \quad 105 \mod 7 = 0, \quad 33 \mod 7 = 5$$

Dermed representeres meldingen som:

$$\texttt{"Hei!"} \quad \Rightarrow \quad (2,\ 3,\ 0,\ 5)$$

Vi tolker dette som koeffisientene til et polynom $f(x)$ av grad mindre enn 4:

$$f(x) = 2 + 3x + 0x^2 + 5x^3 = 2 + 3x + 5x^3$$

Trinn 2: Pipp koder meldingen

Pipp evaluerer polynomet i seks forskjellige punkter for å få med redundans. Det gir oss en $(n, k) = (6, 4)$-kode:

$$x = 0, 1, 2, 3, 4, 5$$

x	Beregning	$f(x) \mod 7$
0	$2 + 0 + 0 = 2$	2
1	$2 + 3 + 5 = 10$	$10 \mod 7 = 3$
2	$2 + 6 + 5 \cdot 8 = 2 + 6 + 40 = 48$	$48 \mod 7 = 6$
3	$2 + 9 + 5 \cdot 27 = 2 + 9 + 135 = 146$	$146 \mod 7 = 6$
4	$2 + 12 + 5 \cdot 64 = 2 + 12 + 320 = 334$	$334 \mod 7 = 5$
5	$2 + 15 + 5 \cdot 125 = 2 + 15 + 625 = 642$	$642 \mod 7 = 5$

Pipp sender da følgende kodeord:

$$(2,\ 3,\ 6,\ 6,\ 5,\ 5)$$

Hvor mange feil tåler denne koden?

Vi har her:

- $n = 6$, $k = 4$
- Minimumsavstand: $d = n - k + 1 = 3$
- Feil som kan rettes (ukjente posisjoner):

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor = \left\lfloor \frac{2}{2} \right\rfloor = 1$$

Konklusjon: Koden kan:

- **Rette opptil én feil** – selv om vi ikke vet hvor feilen sitter
- **Oppdage opptil to feil**

Trinn 3: Pepp mottar – og strålingen slår til

Pepp mottar:

$$(2,\ 3,\ 6,\ 1,\ 5,\ 5)$$

Verdien ved $x = 3$ er blitt forvrengt: den skal være 6, men har blitt 1 - som forventet av uran huset

Trinn 4: Pepp dekoder

Pepp vet at meldingen kommer fra et polynom av grad < 4, og at han har mottatt 6 verdier. Men han vet *ikke* hvilke av dem som er riktige eller gale.

Siden feilen er på et ukjent sted, kan ikke Pepp bare plukke fire verdier og gjøre Lagrange-interpolasjon — han risikerer å velge en feil verdi.

Løsning: Berlekamp–Welch-algoritmen

Før vi kjører brute-force-søk i Python, bruker vi følgende valideringsregler i GF(7):

$$Q(x_i) \equiv E(x_i)\, y_i \pmod 7 \quad \text{for alle } i.$$

Det betyr at for hver kandidat av polynomene

$$E(x) = e_0 + e_1 x, \quad Q(x) = q_0 + q_1 x + q_2 x^2 + q_3 x^3 + q_4 x^4,$$

tester vi om likningen over holder for samtlige mottatte punkter (x_i, y_i). Når vi finner en kombinasjon som passer alle punktene, har vi identifisert:

- *"Feilpolynomet"* $E(x)$ – roten gir posisjonen til feilen.
- *"Sammenkoblingspolynomet"* $Q(x)$ – slik at $Q = E \cdot f$.

Deretter rekonstrueres det opprinnelige meldingspolynomet ved

$$f(x) = \frac{Q(x)}{E(x)} \pmod 7.$$

```
import sympy as sp

# Definer feltet GF(7)
p = 7
x = sp.symbols('x')

# Ukjente koeffisienter E(x)=e0+e1*x, Q(x)=q0+...+q4*x**4
e0, e1, q0, q1, q2, q3, q4 = sp.symbols('e0 e1 q0 q1 q2 q3 q4')

# Mottatte punkter (x_i, y_i)
points = [(0, 2), (1, 3), (2, 6), (3, 1), (4, 5), (5, 5)]
solutions = []

# Bruteforce-søk i GF(7)
for e0_val in range(p):
    e1_val = 1
    for a in range(p):
        for b in range(p):
```

```
            for c in range(p):
                for d in range(p):
                    for e_val in range(p):
                        q_vals = (a, b, c, d, e_val)
                        if all(
                            (a + b*xi + c*xi**2 + d*xi**3 + e_val*xi**4) % p
                            == ((e0_val + e1_val*xi) * yi) % p
                            for xi, yi in points
                        ):
                            solutions.append((e0_val, e1_val) + q_vals)

# Hent første løsning
e0_val, e1_val, a, b, c, d, e_val = solutions[0]

# Bygg polynomer i GF(7)
E_poly = sp.Poly(e0_val + e1_val*x, x, domain=sp.GF(p))
Q_poly = sp.Poly(a + b*x + c*x**2 + d*x**3 + e_val*x**4, x, domain=sp.GF(p))

# Del Q på E for å få f, med koeffisienter mod 7
quotient, _ = Q_poly.div(E_poly)
coeffs = [int(ci % p) for ci in quotient.all_coeffs()]
f = sum(coeffs[i] * x**(len(coeffs)-i-1) for i in range(len(coeffs)))

print("E(x) =", E_poly.as_expr())
print("Q(x) =", Q_poly.as_expr())
print("f(x) =", f)
```

Resultater fra kodeavsnittet Kjøring av Python-scriptet gir følgende output i konsollen:

```
PS C:\R2dex> & C:/Python311/python.exe c:/R2dex/R2dex/del2/Misc/rando.py
E(x) = x - 3
Q(x) = -2*x**4 - x**3 + 3*x**2 + 1
f(x) = 5*x**3 + 3*x + 2
PS C:\R2dex>
```

Her er tolkningen av hvert polynom:

- $E(x) = x - 3$: feilpolynomet, som identifiserer at det finnes en enkelt feil i posisjon $x = 3$.
- $Q(x) = -2x^4 - x^3 + 3x^2 + 1$: kombinert polynom $E(x)\, f(x)$, satt opp slik at $Q(x_i) = E(x_i)\, y_i$ for alle mottatte punkter.
- $f(x) = 5x^3 + 3x + 2$: det opprinnelige meldingspolynomet rekonstruert modulo 7.

Dette har avgjørende betydning fordi vi nå både har lokalisert hvilken mottaksverdi som var feil ($x = 3$) og gjenfunnet det korrekte meldingspolynomet uten å vise alle mellomregninger. Berlekamp-Welch-algoritmen fortjener et helt eget skriv, og vi kommer derfor ikke til å bevise den ytterligere i dette papiret.

Validering: Lagrange-interpolasjon over riktige punkter

De fire feilfrie punktene er

$$(0, 2),\ (1, 3),\ (2, 6),\ (4, 5).$$

Lagrange-formelen gir

$$f(x) = \sum_{i=0}^{3} y_i\, \ell_i(x), \qquad \ell_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \quad \text{(alle beregnet i GF(7))}.$$

Basis-polynomer

$$\ell_0(x) = \frac{(x-1)(x-2)(x-4)}{(0-1)(0-2)(0-4)} = \frac{(x-1)(x-2)(x-4)}{-1 \cdot (-2) \cdot (-4)} = 6\,(x-1)(x-2)(x-4) = 6x^3 + 1,$$

$$\ell_1(x) = \frac{x(x-2)(x-4)}{1 \cdot (-1) \cdot (-3)} = 5\,x(x-2)(x-4) = 5x^3 + 5x^2 + 5x,$$

$$\ell_2(x) = \frac{x(x-1)(x-4)}{2 \cdot 1 \cdot (-2)} = 5\,x(x-1)(x-4) = 5x^3 + 3x^2 + 6x,$$

$$\ell_3(x) = \frac{x(x-1)(x-2)}{4 \cdot 3 \cdot 2} = 5\,x(x-1)(x-2) = 5x^3 + 6x^2 + 3x.$$

Summasjon

$$\begin{aligned} f(x) &= 2\,\ell_0(x) + 3\,\ell_1(x) + 6\,\ell_2(x) + 5\,\ell_3(x) \\ &= 2(6x^3+1) + 3(5x^3+5x^2+5x) + 6(5x^3+3x^2+6x) + 5(5x^3+6x^2+3x) \\ &\equiv (5+1+2+4)\,x^3 \;+\; (0+1+4+2)\,x^2 \;+\; (0+1+1+1)\,x \;+\; 2 \\ &\equiv 5x^3+3x+2 \pmod 7, \end{aligned}$$

som bekrefter

$$f(x) = 2 + 3x + 5x^3.$$

Gjenoppretting av meldingen

Nå som Pepp har polynomet, kan han evaluere det i $x = 0, 1, 2, 3$ for å finne den opprinnelige meldingen:

x	$f(x)$	**"Gjetting" av ASCII**
0	2	$7 \cdot 10 + 2 = 72$ `= H`
1	3	$7 \cdot 14 + 3 = 101$ `= e`
2	6	$7 \cdot 15 + 0 = 105$ `= i`
3	5	$7 \cdot 4 + 5 = 33$ `= !`

Merk om "gjetting"

I dette eksemplet gir $f(x)$ kun *resten* når ASCII-verdien deles på 7. Vi må derfor velge en *kvotient* q i uttrykket

$$\text{ASCII} = 7 \cdot q + \text{rest}$$

slik at resultatet blir en ASCII-kode som gir mening i meldingen.
I et ekte Reed–Solomon-system (som bruker $\text{GF}(2^8)$), er ikke denne "gjettingen" av ASCII nødvendig. Da vil hver symbolverdi direkte tilsvare én byte (0–255), og meldingen kan gjenopprettes eksakt og entydig. Vi har brukt $\text{GF}(7)$ her kun for å gjøre eksemplet forståelig og lett å følge på papir.

Ferdig! Pepp har gjenopprettet meldingen `"Hei!"`, til tross for bitfeil og stråling i huset sitt.

Referanser

[1] CS Theory Toolkit, *Reed–Solomon Codes || @ CMU || Lecture 11d of CS Theory Toolkit*, YouTube. Tilgjengelig fra: https://www.youtube.com/watch?v=1pQJkt7-R4Q

[2] Blelloch, Guy, *Reed–Solomon Codes*, Carnegie Mellon University. Tilgjengelig fra: https://www.cs.cmu.edu/guyb/realworld/reedsolomon/reed$_s$olomon$_c$odes.html

[3] Verbeure, Tom, *Reed–Solomon Error Correcting Codes from the Bottom Up*, Publisert 7. august 2022. Tilgjengelig fra: https://tomverbeure.github.io/2022/08/07/Reed-Solomon.html

[4] Kværnø, Anne (modifisert av André Massing), *Polynomial Interpolation: Lagrange Interpolation*, NTNU, TMA4125 – Vår 2021. Tilgjengelig fra: https://www.math.ntnu.no/emner/TMA4125/2021v/lectures/LagrangeInterpolation.pdf

[5] Guth, Larry, *The Berlekamp–Welch Algorithm*, MIT – The Polynomial Method, Lecture 2. Tilgjengelig fra: https://math.mit.edu/ lguth/PolyMethod/lect2.pdf

[6] Wikipedia contributors. *Reed–Solomon error correction.* (2024). Hentet 26. mai 2025 fra: https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction

[7] Wikipedia contributors. *Berlekamp-Welch algorithm* (2012). Hentet 26. mai 2025 fra: https://en.wikipedia.org/wiki/Berlekamp-Welch$_a$lgorithm

[8] Khandani, A. *Digital Communication Systems.* Rutgers University. Hentet 26. mai 2025 fra: https://content.sakai.rutgers.edu/access/content/user/ak892/Digital/Communicatio