

Sekvensiell dekoding av trelliskoder



Jostein Lund

Institutt for Informatikk

Universitetet i Bergen

29. april 2004

Forord

Jeg ønsker å takke min veileder Kjell Jørgen Hole for uvurderlig hjelp og støtte i løpet av mitt hovedfagsarbeid. Han har alltid vært tilgjengelig og har vært til stor hjelp både med sin faglige kompetanse og til å finne frem bøker og publikasjoner som jeg har benyttet i denne oppgaven. Jeg har også dratt stor nytte av hans kunnskaper under utformingen av denne oppgaven.

Videre ønsker jeg å takke mine medstudenter ved Universitetet i Bergen for programmeringsteknisk assistanse og hjelp til å utnytte Unix systemene på universitetet på en effektiv måte.

Retter også en stor takk til min nærmeste familie som har støttet meg gjennom hele prosessen.

Takk!

Innhold

Forord	2
Sammendrag	5
Definisjoner	6
Kapittel 1 - Innledning	10
1.1 Telekommunikasjonen oppstår	10
1.2 Dagens systemer	11
1.3 Digital kommunikasjon.....	11
1.4 Oppgavens innhold	12
Kapittel 2 - Signalkonstellasjoner og støy	14
2.1 Signalkonstellasjonen	14
2.2 Støy	15
2.3 Energi.....	17
2.4 Normalfordelt støy	19
2.5 Kanalkapasitet og Cutoffrate	20
Kapittel 3 - Koding og Sekvensiell dekoding	22
3.1 Konvolusjonskoder	22
3.2 Den frie avstanden	27
3.3 Partisjonering av signalkonstellasjoner.....	28
3.4 Feedback enkodere.....	30
3.5 Koding i halen.....	31
3.6 Dekoding.....	34
3.7 Metrikk.....	35
3.8 Stabel-algoritmen	37
Kapittel 4 - Nye Resultater	44
4.1 Resultater fra kjente koder	44
4.1.1 Tidsbruk	44
4.1.2 Simulering for 8-PSK	46
4.1.3 Simulering for 16-QAM	47
4.2 Resultater fra tilfeldig genererte koder	48
4.2.1 Simulering for 32-QAM	48
4.2.2 Simulering for 64-QAM	49
Kapittel 5 - Oppsummering	52
5.1 Sammendrag	52
5.2 Videre arbeid.....	52
Referanseliste	54

Appendiks - Oversikt over programmet.....	56
Vedlegg 1 - Generering av tilfeldige koder	62
Vedlegg 2 - Simuleringsprogrammet	64
Vedlegg 3 - QAM.h	88

Sammendrag

I denne hovedfagsoppgaven gir jeg en grunnleggende forklaring på hvordan digital kommunikasjon foregår. Jeg gir en innføring i hvordan støy opptrer på en AWGN-kanal og hvordan støyen påvirker informasjonen som sendes over denne kanalen.

Signalkonstellasjoner blir presentert og jeg forklarer hvordan digital informasjon gjøres om til signalpunkter i en signalkonstellasjon. Jeg går også inn på feilkorrigerende koder med vekt på konvolusjonskoder. Det vil bli forklart hvordan informasjonssekvenser blir kodet og hvordan de blir dekodet. Oppgaven vil med andre ord gi en innføring i hvordan man kan kommunisere feilfritt over en kanal med støy.

Til slutt i denne oppgaven presenterer jeg simuleringsresultater for koder som er utarbeidet i [8] og simuleringer for tilfeldig genererte koder. Simuleringsresultatene blir så brukt til å verifisere resultater som er presentert i [3], samt presentere resultater fra simuleringene med de tilfeldig genererte kodene.

Definisjoner

AWGN

Additive White Gaussian Noise. Gaussisk fordelt støy.

BER

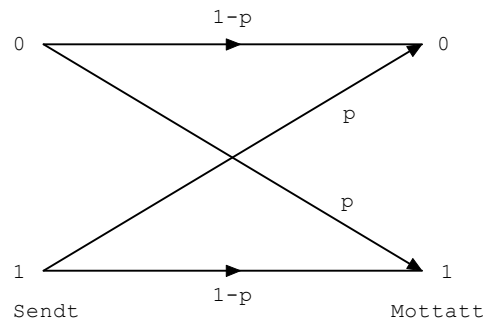
Bit Error Rate. Antall biter som er dekodet feil dividert på totalt antall dekodete biter.

Blokk lengde

Når en informasjonssekvens skal overføres over en kanal deles den ofte opp i blokker. Blokk lengden er antall biter i hver av disse blokkene.

BSC

Binary Symmetric Channel. Figuren viser sannsynligheten p for at en bit som blir overført over kanalen blir feil dekodet og sannsynlighet $1-p$ hvis den blir riktig dekodet.



BSC-Binary Symetric Channel.

BPSK

Binary-Phase-Shift-Keyed. Signalkonstellasjon som inneholder to signalpunkter. Punktene tilsvarer 0 eller 1 i det binære tallsystemet (figur 2.2).

Cutoffrate

Maksimal overføringsrate over en kanal der gjennomsnittlig antall operasjoner pr. dekodet bit er bundet for sekvensiell dekodning.

Dekoder

Endrer kodede data til en informasjonssekvens. Prøver å finne og korrigere feil som kan ha oppstått under overføringen.

Enkoder

Koder en informasjonssekvens før den skal overføres over en kanal. Legger til redundante biter for å sikre feilkorreksjon etter overføringen.

Feedback

I en enkoder med minne der data fra tidligere steg i enkodingen fortsatt befinner seg i enkoderen kan vi snakke om feedback. I en enkoder med feedback vil data fra tidligere steg virke inn på data fra senere steg, altså en tilbakeføringskanal. Biter kan dermed i teorien sirkulere uendelig lenge i enkodren.

Feedforward

I en enkoder med minne der data fra tidligere stegi enkodingen fortsatt befinner seg i enkoderen kan vi snakke om feedforward. Biter som befinner seg i minneblokken beveger seg fremover til en minneblokk lenger frem i enkoderen. Dette kan fortsette helt til bitene forsvinner ut av enkoderen. Bitene kan derfor maksimalt være like mange steg i enkodren som det finnes minneblokker.

Fri avstand

Den minste avstanden mellom to signalsekvenser man kan finne ut fra alle par av signalsekvenser som kan dannes av enkoderer.

Informasjonssekvens

De opprinnelige dataene som skal overføres over kanalen før de blir delt opp i blokker og kodet.

Inndata

Input. I denne oppgaven mest brukt om data som sendes inn til enkoderen eller dekoderen ved et gitt tidspunkt.

Generatorpolynom

Definerer enkoderen på polynom form. Flere polynomer definerer en enkoder.

Kanalkapasitet

Pålitelig kommunikasjon kan oppnås ved koding når overføringsraten er mindre enn kanalkapasiteten. Kanalkapasiteten måles i informasjonsbiter pr. sekund.

Konstellasjon

En samling av signalpunkter som brukes til overføring av digitale data.

Mapping

Alle punktene i en signalkonstellasjon nummereres. Måten disse punktene nummereres på kalles mappingen av konstellasjonen.

Metrikk

En måte å sammenligne data påvirket av støy med de opprinnelige dataene. Metrikken gir et tall på hvor like de to dataene er.

MLD

Maximum Likelihood Dekoding. Dekoding som dekoder en kodesekvens til den som er mest sannsynlig gitt den mottatte sekvensen. En slik dekodeer må derfor sjekke alle muligheter for å være sikker på å finne den som er mest sannsynlig.

Overhead

Ekstra data som vil svekke ytelsen til systemet, men som er med på å sikre feilkorreksjon.

QAM

Quadrature Amplitude Modulation. En 2-dimensjonal signalkonstellasjon. Punktene i en slik konstellasjon representeres ved to koordinater x og y .

Registerlengde

Constraintlength. Antall minneblokker en enkoder inneholder.

Signalpunkt

Et punkt i en signalkonstellasjon. Konstellasjonene kan ha flere dimensjoner og signalpunktet har da like mange koordinater som antall dimensjoner i konstellasjonen.

Slumptallsgenerator

Et dataprogram som genererer data med en gitt fordeling.

SNR

Signal to Noise Ratio. Angir hvor mye støy som vil påvirke kommunikasjon i det gitte systemet.

Spektral effektivitet

Antall biter pr. sekund overført pr. Hz båndbredde. Måles i bit/sek/Hz

Tilstand til enkoderen

Tilstanden til en enkoder er innholdet til minneblokkene ved et gitt tidspunkt.

Utdata

Output. Data som kommer ut av enkoderen eller dekodeeren.

Kapittel 1

Innledning

1.1 Telekommunikasjonen oppstår

Mennesker har alltid hatt behov for å kommunisere over store avstander. Allerede for flere tusen år siden oppstod de første kommunikasjonssystemene. Fra gammelt av har det vært brukt flere typer signalisering som for eksempel røyk, flammer, trommer, lys, speil, flagg eller andre typer tegn som synes over store avstander. Målet har alltid vært å kommunisere raskere enn man kunne gå eller ri på hest. Denne typen kommunikasjon hadde selvfølgelig store begrensninger. Signalene måtte være enkle koder og strekningen begrenset seg til hvor langt mennesker kunne se eller høre. Utfordringen var å finne en måte hvor man kunne kommunisere lengre enn synet og hørselen. Løsningen var elektrisitet som ble oppdaget midt i det attende århundret [11].

Endelig i det nittende århundret stod den elektriske telegraf klar. Den var et resultat av forskning innen mange områder som elektromagnetisme, generering og lagring av elektrisitet, den industrielle revolusjonen som gjorde det mulig å produsere komponentene og utvikling av koder som omgjorde elektriske impulser til et forståelig språk. Utover på 1830-tallet hadde det blitt utviklet mange enkle elektriske telegrafer, men ingen hadde fått kommersiell utbredelse. Engelskmennene William Cooke og Charles Wheatstone nådde dette målet, men på samme tid kom også amerikaneren Samuel Morse med sin telegraf og det var denne telegraf som fikk størst utbredelse. Morse utviklet også morse-alfabetet som gjorde at man kunne sende all informasjon gjennom en linje.

Den første kommersielle offentlige telegraf linjen ble opprettet i 1845 mellom London og Gosport. Ikke mange år senere var de fleste store byer i England forbundet med telegraf linjer. På 1850 tallet var det strukket telegraf linjer over store deler av Europa og USA. Utfordringen var nå å kunne kommunisere over havene. Allerede i 1850 ble den første undersjøiske telegraf linjen lagt mellom England og Frankrike. Telegraf hadde sin gullalder rundt første verdenskrig. Senere har telefon og radio tatt over mye av kommunikasjonen.

Telefonen var neste skritt i utviklingen av telekommunikasjon. Alexander Graham Bell stod for denne oppfinnelsen i 1876. Telefonen baserte seg på samme teknologi som telegraf, men i stede for elektriske på/av signaler ble det nå overført analoge bølger som ble laget av menneskestemmen. Dette var mye raskere enn å måtte kode og dekode meldinger via telegraf. Når utbredelsen av telefonen ble større oppstod sentralbordene der man manuelt koblet opp en samtale mellom to brukere. Senere ble disse sentralbordene automatisert.

Radioen ble det logiske neste steget innen kommunikasjon. Guglielmo Marconi regnes som radio-kommunikasjonens far. I 1896 overførte han den første meldingen over sitt trådløse system. Etter å ha satt opp en radioforbindelse mellom sydspissen av England og Newfoundland fikk Marconi anerkjennelse verden rundt. Man trodde at avstanden begrenset seg på grunn av jordkrumningen, men en naturlig effekt i jordens atmosfære gjør at lavere radiofrekvenser reflekteres og vandrer langs jordoverflaten. I starten var den trådløse kommunikasjonen begrenset til radio telegrafering, men etter å ha studert radiobølgers egenskaper klarte man i 1920-årene å skape radio-telefonen slik at man også kunne snakke via det nye mediet. På slutten av 1920-tallet ble også kringkastingsradioen brukt til å sende tale ut til folket. Det var en ny måte å spre informasjon til folket på som viser den store fordelen med radiokommunikasjon. Ulempen med denne typen kommunikasjon er at alle kan lytte til meldinger som blir sendt ut, samt at trådløs kommunikasjon er mer utsatt for støy enn de trådbaserte. Løsningen på det første problemet har blitt kryptering av informasjonen. Problemet med støy kan ikke omgås, men ved bruk av feilkorrigerende koder kan man oppnå pålitelig kommunikasjon.

Som nevnt er det kun lavere frekvenser som kan vandre med jordkrumningen. Frekvenser brukt for TV signaler beveger seg i en rett bane og forsvinner dermed ut i verdensrommet. Satellittkommunikasjon er derfor tatt i bruk. Satellitten kan brukes som et speil i rommet, og sende signalene tilbake til jorden. Dermed kan også høyere frekvenser benyttes til kommunikasjon over store avstander.

1.2 Dagens systemer

I dag er trådløse kommunikasjonssystemer en del av vår hverdag. Kravene til ytelse og pålitelighet øker i takt med den hurtige utviklingen på området. Nå er det ikke lenger kun beskjeder fra person til person, men store data mengder med eksempelvis musikk eller video som skal overføres. I 1993 kom det digitale mobiltelefonnettet GSM på banen, og mobiltelefonen har etter hvert blitt allemannseie. Internetttilgang til trådløse enheter har også blitt et krav.

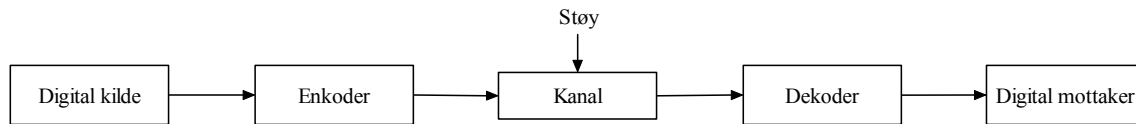
Den siste tiden har trådløse datanettverk (WLAN) blitt innført for å erstatte de gamle kabelbaserte (LAN) nettverkene både på arbeidsplasser og i hjemmet. Det har etter hver blitt mange konkurrenter på feltet. Mest kjent av dagens systemer er kanskje Wi-Fi (IEEE 802.11b) og HIPERLAN/2. Flere standarder er på vei inn i markedet. Målet er hele tiden å øke båndbredden og den spektrale effektiviteten [16].

1.3 Digital kommunikasjon

Informasjonsteori er læren om overføring av informasjon mellom en sender og en mottager. Claude Shannon publiserte i 1948 "The Mathematical Theory Of Communication" [12] som danner grunnlaget for den moderne informasjonsteorien. Som nevnt er støy et problem når vi skal overføre informasjon over en kanal. På 1950- og 1960-tallet arbeidet man med å utvikle teorier for effektive enkodere og dekodere. I løpet

av 1970-årene kom de første bøkene om feilkorrigerende koder og teoriene ble i større grad tatt i bruk i praksis.

Hensikten med feilkorrigerende koder er å finne og korrigere feil som oppstår når man overfører informasjon over en kanal.



Figur 1.1: Modell av et digitalt kommunikasjonssystem.

Figur 1.1 viser en modell av et digitalt kommunikasjonssystem. Den digitale kilden inneholder de dataene som skal sendes. Enkoderen koder om signalet slik at det kan sendes over kanalen. Kanalen er det mediet som signalet overføres gjennom, for eksempel en kobberkabel i et trådbasert nett, eller eter i et trådløst nett. Dekoderen tar i mot signalet og omdanner dette til informasjon som er forståelig for den digitale mottakeren.

Når feilkorrigerende koder benyttes legges det til redundant informasjon i enkoderen som gjør at dekodeeren senere kan oppdage og korrigere evt. feil som har oppstått i overføringen. For å overføre digital informasjon over en kanal, må signalene først gjøres om til analoge signaler. Dette gjøres også i enkoderen. Denne prosessen kalles modulasjon. Dekoderen må før den korrigerer feil demodulere signalet til digital informasjon igjen.

1.4 Oppgavens innhold

I denne oppgaven går jeg inn på kommunikasjonsprosessen i trådløse nettverk. På grunn av liten CPU-kapasitet i mange trådløse enheter er det viktig å bruke så få resurser som mulig til koding og dekoding. *Sekvensiell dekoding av trelliskoder* er godt egnet til dette da denne typen dekoding inneholder forholdsvis få operasjoner pr. dekodet bit i forhold til andre typer dekoding. Trådløse enheter drives ofte av et batteri og ved færre beregninger vil også batteriet vare lenger. Denne typen dekoding kan også brukes ved adaptiv koding og modulasjon. I et slikt adaptivt system vil koden man benytter tilpasse seg *signal til støy forholdet* (SNR) slik at man hele tiden benytter en godt egnet kode til kommunikasjon over kanalen [17]. Dette vil øke ytelsen i forhold til systemer som bruker samme kode uavhengig av SNR. Jeg vil ikke gå nærmere inn på dette i denne oppgaven.

I kapittel 2 gir jeg en innføring i signalkonstellasjoner med hovedvekt på *Quadrature Amplitude Modulation (QAM)-konstellasjoner*. Her blir det også forklart hvordan støy opptrer på en *Additive White Gaussian Noise (AWGN)-kanal* og hvordan støyen påvirker informasjon som sendes over en slik kanal. Begreper som *kanalkapasitet* og *Cutoffrate* defineres også i dette kapitlet.

I kapittel 3 vil jeg gå nærmere inn på koding og sekvensiell dekoding. Det beskrives hvordan en informasjonssekvens blir kodet og hvordan den kodede bitsekvensen gjøres om til QAM-symboler som skal overføres over kanalen. Ved bruk av den sekvensielle dekodingsalgoritmen kalt stabelalgoritmen viser jeg hvordan en kodet bitsekvens med AWGN dekodes til informasjonsbiter igjen.

Kapittel 4 viser egne simuleringresultater for noen kjente trelliskoder og for noen tilfeldig genererte trelliskoder. *BER* (Bit Error Rate) fra simuleringene er avhengig av parametere som registerlengden, konstellasjonstype, blokk lengde og SNR. Jeg har gjort simuleringer der jeg har brukt ulike parametere og funnet ut hvordan dette påvirker BER. Simuleringsresultatene sammenlignes med tidligere kjente resultater og teoretiske øvre skranker. Her gis også en del forklaringer til resultatene.

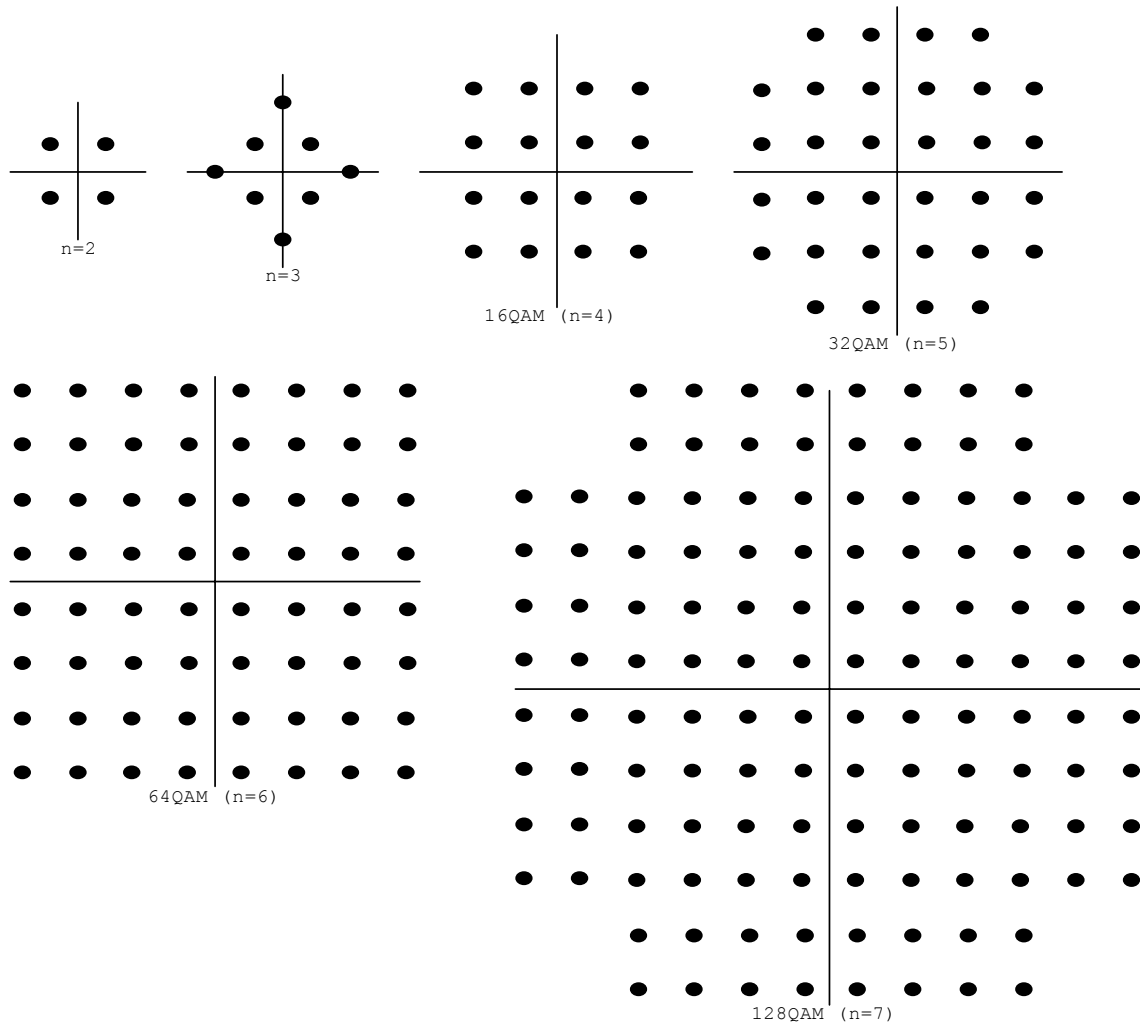
I kapittel 5 kommer en kort oppsummering og noen forslag til videre arbeid som kan bygge på denne oppgaven.

Kapittel 2

Signalkonstellasjoner og støy

Dette kapittelet vil først gi en kort innføring i signalkonstellasjoner. Videre forklares det hvordan QAM-signalpunkter påvirkes av AWGN når de overføres over en AWGN-kanal og hvordan man kan generere slik støy ved simuleringer. Det gis også en forklaring på energibruk ved å sende et signalpunkt over en kanal. Til slutt defineres cutoffraten for en båndbegrenset AWGN-kanal.

2.1 Signalkonstellasjoner

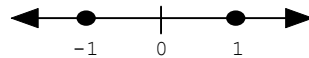


Figur 2.1: Rektangulære 2-dimensjonale konstellasjoner [5].

QAM-signaler er vanligvis representert i 2-dimensjonale konstellasjoner. En konstellasjon med 2^n punkter kan sende n biter pr. symbol. Typiske konstellasjoner vil

inneholde 4, 8, 16, 32, osv. punkter. Når n er et partall, vil vi kunne lage rektangulære konstellasjoner. F. eks 16 ($4 * 4$) og 64 ($8 * 8$). Hvis n er et oddetall, vil vi få en såkalt kryss konstellasjon. Denne får man ved å fjerne punkter fra en rektangulær konstellasjon. F. eks en konstellasjon med 32 punkter vil være en $6 * 6$ rektangulær hvor man fjerner et punkt i hvert hjørne for å få 32 punkter. For en konstellasjon med 128 punkter vil man måtte fjerne 4 punkter i hvert hjørne. Se figur 2.1 for eksempler på ulike konstellasjoner. Hvert punkt i signalkonstellasjonen blir gitt koordinater ut i fra aksene i figuren.

Den *spektrale effektiviteten* til et kommunikasjonssystem er definert som antall overførte informasjonsbiter pr. sekund pr. Hz båndbredde. En BPSK (Binary Phase Shift Keyed) – konstellasjon er en 1-dimensjonal signalkonstellasjon med kun 2 signalpunkter. Figur 2.2 viser en slik konstellasjon. For et system som bruker BPSK vil man kun trenge 1 bit for å representere et symbol. Den spektrale effektiviteten til dette systemet blir 1 bit/sek/Hz.



Figur 2.2: BPSK 1-dimensjonal signal konstellasjon.

I tilfeller der dataraten (biter/s) overstiger det tilgjengelige båndbredden (Hz) kan man øke den spektrale effektiviteten ved å øke størrelsen på konstellasjonen. Det vil fortsatt bli overført like mange symboler pr. sekund pr. Hz, men antall mulig verdier for hvert symbol blir større. Ved bruk av 16-QAM vil man derfor få en spektral effektivitet på 4 bits/sek/Hz. Ved å øke den spektrale effektiviteten trengs det også større gjennomsnittlig energi for å opprettholde den samme minimumsavstanden mellom symbolene som tidligere. Ved å benytte en 2-dimensjonal konstellasjon vil man spare energi i forhold til å benytte en 1-dimensjonal konstellasjon med samme antall punkter.

2.2 Støy

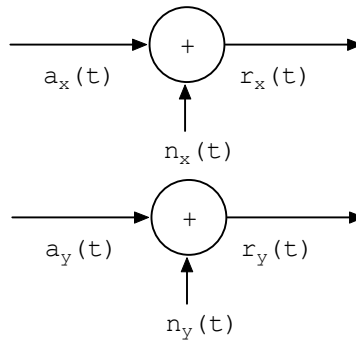
En vanlig form for støy i et kommunikasjonssystem er AWGN. Hvis det sendte signalet er $a(t)$ ved tidspunkt t vil det mottatte signalet $r(t)$ være [1, s. 6]:

$$r(t) = a(t) + n(t)$$

der $n(t)$ er støyen i systemet. Den stokastiske variabelen $n(t)$ er normalfordelt med forventning 0 og varians σ^2 .

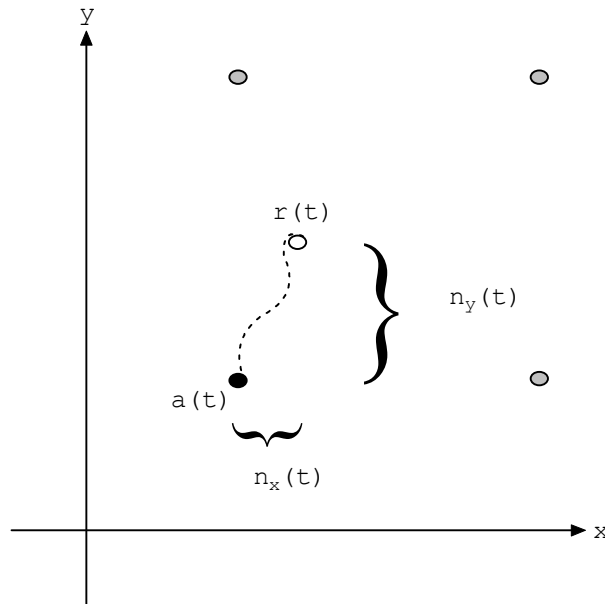
$$n(t) \sim N(0, \sigma^2)$$

Ved bruk av 2-dimensjonale QAM konstellasjoner vil støyen bli lagt til som vist i figur 2.3



Figur 2.3: Støy lagt til i x og y retning ved tidspunkt t .

Hvert punkt i konstellasjonen blir representert ved to koordinater (a_x, a_y) . For å gjøre notasjonen enklere er t utelatt i dette uttrykket. Når disse koordinatene blir overført over en kanal vil det bli påvirket av AWGN i x - og y -retning. Punktet vil derfor vandre i koordinatsystemet når det blir lagt til støy (se figur 2.4). Jo mer støy, jo mer vil punktet vandre. MLD (Maximum Likelihood Dekoding) dekker en kodesekvens til den som er mest sannsynlig gitt den mottatte sekvensen. Ved MLD vil punktet bli dekodet til det punktet som ligger nærmest det mottatte punktet. Det vil si at hvis punktet vandrer tilstrekkelig langt, vil punktet ligge nærmere et annet punkt i konstellasjonen enn det opprinnelig sendte punktet, og vi vil få en feil ved dekoding.



Figur 2.4: $a(t)$ vandrer i koordinatsystemet.

2.3 Energi

Når et signalpunkt skal sendes over en kanal trengs det en viss energi. Energien E som trengs for å sende et punkt blir større jo lenger punktet ligger fra origo. Et punkt i origo er definert til å ha 0 i energi. Dette er årsaken til at konstellasjoner aldri inneholder punkter som ligger i origo. P.g.a. dette er det mest hensiktsmessig å gi punktene odde koordinater med en avstand på 2 mellom hvert punkt i både x - og y -retning.

For å kunne beregne variansen σ^2 trenger vi den gjennomsnittlige energien E_s for en konstellasjon. Signal til støy forholdet er definert som [3]

$$SNR = \frac{E_s}{2\sigma^2}$$

Dette gir

$$\sigma = \sqrt{\frac{E_s}{2 \cdot SNR}}$$

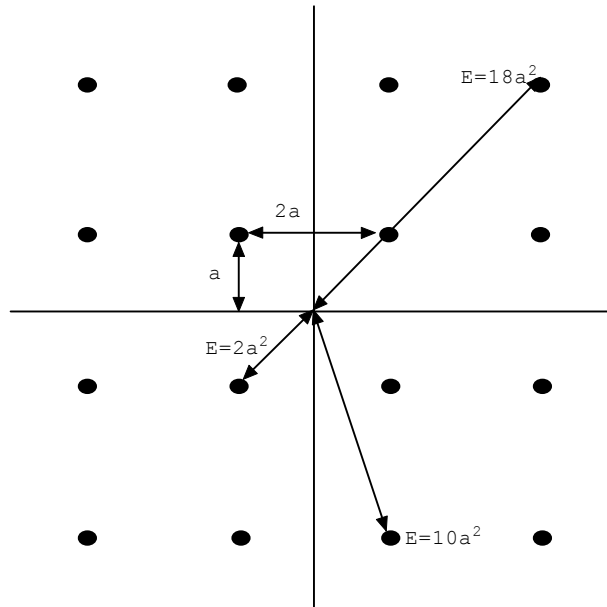
$SNR_{dB} = 10 \cdot \log_{10} SNR$ er signal til støyforholdet oppgitt i desibel (dB). Legg merke til at $SNR = 10^{SNR_{dB}/10}$. Hvis vi har en 2-dimensional QAM konstellasjon med N punkter representerer vi et symbol i denne konstellasjonen ved $a^i, i = 0, 1, \dots, N-1$. Sannsynligheten for at et punkt blir valgt er $Q(i)$. Den gjennomsnittlige energien E_s er da definert som [3]

$$E_s = \sum_{i=0}^{N-1} Q(i) \|a^i\|^2$$

Vi setter at alle punkter i konstellasjonen har like stor sannsynlighet for å bli valgt. Dette gir $Q(i) = 1/N$. Splitter vi $\|a^i\|^2$ opp i x - og y -retning får vi $\|a^i\|^2 = (a_x^i)^2 + (a_y^i)^2$. Innsatt i formelen for E_s får vi da

$$E_s = \frac{1}{N} \sum_{n=0}^{N-1} ((a_x^i)^2 + (a_y^i)^2)$$

Eksempel:



Figur 2.5: 16-QAM med punkter i avstand $2a$ fra hverandre i x - og y -retning.

Figur 2.5 viser en 16-QAM konstellasjon med avstandene fra de ulike punktene til origo. Energien til punktene nærmest får vi fra pythagoras $E = a^2 + a^2 = 2a^2$. På samme måte får vi at energien til punktene som ligger nest lengst fra origo ligger

$$\begin{aligned} E &= (3a)^2 + a^2 \\ &= 10a^2 \end{aligned}$$

og at punktene som ligger lengst fra origo får

$$\begin{aligned} E &= (3a)^2 + (3a)^2 \\ &= 18a^2. \end{aligned}$$

Vi ser at energien til hvert enkelt punkt er det man summerer over i formelen for E_s . Vi har 4 punkter med energi $2a^2$, 8 punkter med energi $10a^2$ og 4 punkter med energi $18a^2$. Dette kan man sette inn i uttrykket for E_s .

Den gjennomsnittlige energien til konstellasjonen blir da:

$$\begin{aligned} E_s &= \frac{4 \times 2a^2 + 8 \times 10a^2 + 4 \times 18a^2}{16} \\ &= 10a^2 \end{aligned}$$

2.4 Normalfordelt støy

For å simulere kommunikasjon over en AWGN kanal, må man generere støyen som skal legges på signalene. Dette har jeg gjort på følgende måte.

En slumptallsgenerator har generert uniformt fordelte heltall $X_n \in \{0,1,\dots,s-1\}, n \geq 0$. Der s er et stort heltall. Uttrykket:

$$U_n = (X_n + 1) / s$$

gir uniformt fordelte reelle tall i intervallet $(0,1]$. Vi setter U_n og U_{n+1} som variable i funksjonene [13]

$$\begin{aligned} V_n &= \sqrt{-2 \ln(U_n)} \sigma \sin(2\pi U_{n+1}) \\ V_{n+1} &= \sqrt{-2 \ln(U_n)} \sigma \cos(2\pi U_{n+1}) \end{aligned}$$

V_n og V_{n+1} er da normalfordelt med forventning 0 og varians σ^2 .

Når et symbol a^i blir sendt over en kanal med støy, vil støyen bli lagt til symbolet. V_n legges til i x-retning og V_{n+1} legges til i y-retning.

$$\begin{aligned} r_x^i &= a_x^i + V_n \\ r_y^i &= a_y^i + V_{n+1} \end{aligned}$$

Eksempel:

La oss si at vi bruker 16-QAM ved kommunikasjon på en trådløs kanal og ønsker å sende punktet (1,-3) over kanalen. Kanalen er påvirket av AWGN med signal til støy forhold lik 12dB.

Ved bruk av 16-QAM vil innsetting i uttrykket for E_s gi $E_s = 10$. Setter vi E_s og SNR i uttrykket for σ får vi

$$\sigma = \sqrt{\frac{10}{2 \cdot 10^{12/10}}} \approx 0,5617$$

La oss si at slumptallsgeneratoren gir tallene 11623 og 24752. Videre lar vi $s=32768$. Dette gir

$$U_n = 11623 + 1/32768 \approx 0.3547, \text{ og } U_{n+1} = 24752 + 1/32768 \approx 0.7554$$

$$Y_n = (\sqrt{-2 \cdot \ln(0.3547)}) \cdot 0.5617 \cdot \sin(2 \cdot \pi \cdot 0.7554) \approx -0.8082$$

$$Y_{n+1} = (\sqrt{-2 \cdot \ln(0.3547)}) \cdot 0.5617 \cdot \cos(2 \cdot \pi \cdot 0.7554) \approx 0.0274$$

Vi har $a_x^i = 1$ og $a_y^i = -3$. Vi får

$$r_{x_{mottatt}}^i = 1 + (-0.8082) = 0.1918, \text{ og } r_{y_{mottatt}}^i = -3 + 0.0274 = -2.9726$$

Vi ser at når man sender punktet (1,-3) og legger til støy vil man motta punktet (0.1918,-2.9726). Punktet har altså vandret litt mot origo, men ved MLD dekoding ville dette punktet fortsatt bli dekodet til (1,-3) som er det nærmeste punktet, og vi ville ikke fått noen feil i overføringen.

2.5 Kanalkapasitet og Cutoffrate

Det er to parametere som definerer begrensningene til yteevnen til et digitalt kommunikasjonssystem. Dette er kanalkapasiteten C og cutoffraten R_0 . Shannon har vist at pålitelig kommunikasjon kan oppnås ved koding når overføringsraten er mindre enn kanalkapasiteten. C kan bare nås med uendelig stor kompleksitet i kodingen. R_0 tar hensyn til kompleksiteten og forteller oss den maksimale raten for pålitelig overføring med en fornuftig kompleksitet. Cutoffraten R_0 til en båndbegrenset AWGN [3] er:

$$R_0 = (\log_2 e) \left[1 + \frac{E_s}{2N_0} - \sqrt{1 + \left(\frac{E_s}{2N_0} \right)^2} \right] + \log_2 \left[\frac{1}{2} \left(1 + \sqrt{1 + \left(\frac{E_s}{2N_0} \right)^2} \right) \right]$$

og måles i biter/T der T er tiden det tar å sende et symbol. Her er E_s den gjennomsnittlige inndata energien og N_0 er den spektrale effektiviteten til støyen. R_0 kan bare nås med Gaussisk fordelte inndata.

I [3] defineres også R_0^* for diskrete inndata og kontinuerlig utdata for en båndbegrenset AWGN kanal.

$$R_0^* = 2 \log_2 N - \log_2 \left\{ \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \exp \left[- \frac{(a_x^i - a_x^j)^2 + (a_y^i - a_y^j)^2}{8\sigma^2} \right] \right\}$$

der a_x^i og a_y^i er x- og y-koordinatene for et 2-dimensjonalt signalpunkt a^i .

Man vil trenge større SNR for å oppnå R_0^* lik en gitt R_0 . Forskjellen i SNR oppstår på grunn av ulike inndata fordelinger. Legg merke til at R_0 kun er en funksjon av SNR, mens R_0^* avhenger av en spesifikk signalkonstellasjon.

Kapittel 3

Koding og Sekvensiell dekoding

Dette kapittelet vil først gi en innføring i konvolusjonskoder og hvordan man koder informasjonssekvenser ved hjelp av disse. Enkoderen som gjør kodejobben er definert ved et generatorpolynom. Her gis en grundig forklaring på hvordan et slikt polynom kan definere et skjematisk oppsett av en enkoder. Flere måter å representere kodesekvensene blir vist. Det er ønskelig å oppnå en størst mulig minimumsavstand mellom symboler i en signalkonstellasjon. Forklaringer på hvordan dette gjøres kommer i dette kapittelet.

Etter at hele kodingsprosessen er beskrevet vil dekodingen bli presentert. Stabelalgoritmen til sekvensiell dekoding blir satt i fokus. For å kunne sammenligne i hvor stor grad en bitsekvens er lik en annen benytter algoritmen en metrikk som angir et tall for nettopp dette. Hvordan denne beregningen gjøres blir beskrevet i detalj. Til slutt i kapittelet vil det bli gitt et stort eksempel som går gjennom hele prosessen med koding og overføring av den kodede sekvensen over en kanal med støy og dekoding.

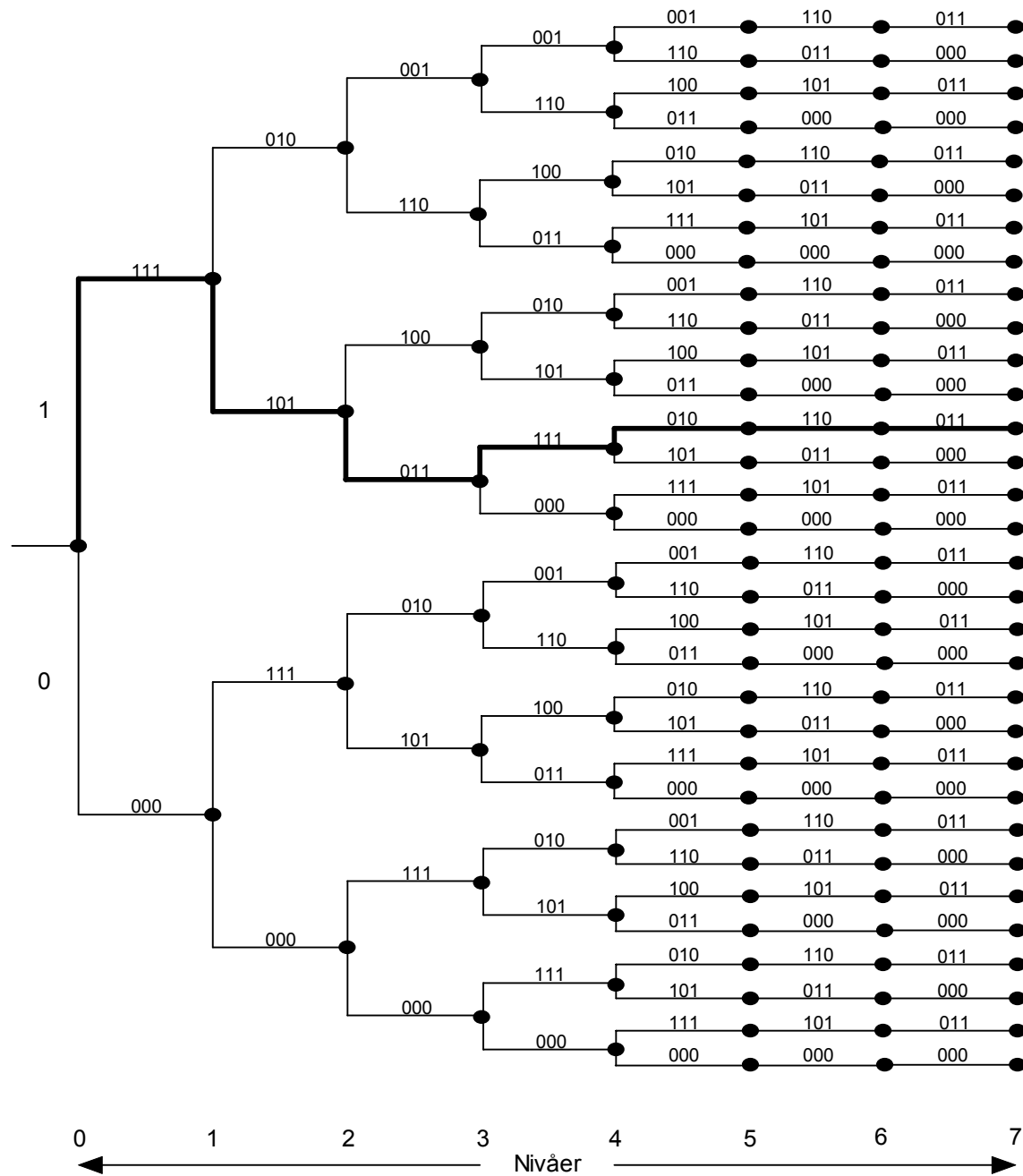
3.1 Konvolusjonskoder

Kapittelet bygger på [1], [2] og [3].

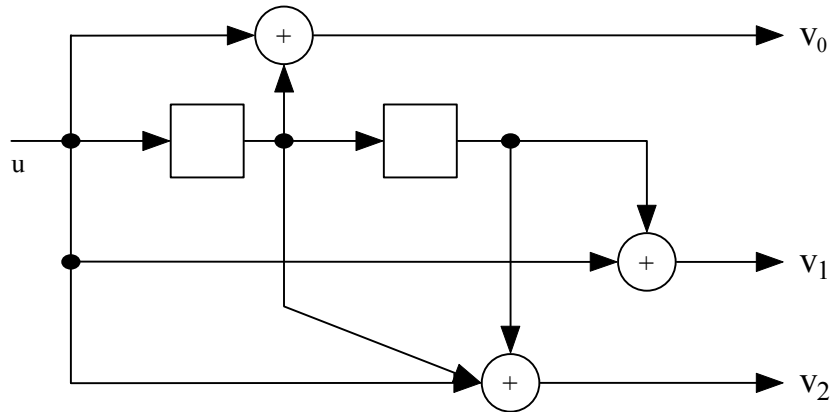
For en (n,k,m) -konvolusjonskode gjelder det :

- n – Antall utdata biter ved et gitt tidspunkt
- k – Antall inndata biter ved et gitt tidspunkt
- m – Antall minneblokker i enkoderen
- L – Lengden til informasjonssekvensen målt i biter

For en (n,k,m) -kode er det vanlig å representere de 2^{kL} kodeordene med lengde $N=n(L+m)$ i et kodetre. Hver informasjonssekvens er en sti gjennom treet med lengde kL biter. Treet inneholder $L+m+1$ nivåer. Startpunktet er første nivå. Så kommer L nivåer med informasjonsbiter og til slutt m nivåer for at enkoderen skal returneres til tilstand 0. Tilstanden til enkoderen beskriver innholdet i minneblokkene. 0 tilstanden vil si at alle minneblokkene i enkoderen skal inneholde 0. Dette gjøres ved å kun gi 0 som inndata til enkoderen. Ved å gjøre dette m ganger, vil alle minneblokkene nullstilles og enkoderen er tilbake i nulltilstanden. Disse siste m nivåene i treet blir kalt halen til treet. Grunnen til at det legges til en hale er for å sikre feilkorrigering i de siste bitene i sekvensen. Figur 3.1 viser kodetreet til en $(3,1,2)$ -kode og Figur 3.2 viser konvolusjonsenkoderen som genererer dette treet. Raten til koden er antall inndata biter dividert med antall utdata biter. Raten til koden i eksemplet er $1/3$.



Figur 3.1: Kodetreet til en (3,1,2)-kode med L=5 ([1], s. 352).



Figur 3.2: Enkoder til en (3,1,2)-kode.

Informasjonssekvensen $u=10011$ er markert i kodetreet i figur 3.1. Vi kan se at dette tilsvarer den kodede sekvensen $v=111\ 101\ 011\ 111\ 010$. Vi ser her bort i fra de to siste nivåene da disse ikke inngår i informasjonssekvensen men kun brukes til å føre enkoderen tilbake til nulltilstanden. Disse tilleggsbitene vil medføre overhead i koden. Det er vist at en hale av legde m er nødvendig for å opprettholde god ytelse for "feedforward" enkodere [10]. I denne typen enkodere vil alle biter vandre fremover i enkoderen og forsvinne ut etter en viss tid. Enkoderen i figur 3.2 er en "feedforward" enkoder.

Generatorpolynomet for denne enkoderen er $G(D) = [D + 1, D^2 + 1, D^2 + D + 1] = [3, 5, 7]$. Generatorpolynomet skrives ofte på oktal form. Den oktale formen tar utgangspunkt i den binære som i dette tilfellet vil være $G(D)=[011, 101, 111]$. Hvert desimale tall tilsvarer 3 biter binært.

På generell form kan vi skrive. $G(D) = [g_m^{(1)} g_{m-1}^{(1)} \dots g_0^{(1)}, g_m^{(2)} g_{m-1}^{(2)} \dots g_0^{(2)}, \dots, g_m^{(n)} g_{m-1}^{(n)} \dots g_0^{(n)}]$. Der n er antall polynomer og m er antall minneblokker. Man ser ut i fra generator polynomet at 1 gir $g_0^{(x)} = 1$, D gir $g_1^{(x)} = 1$, D^2 gir $g_2^{(x)} = 1$ o.s.v. der x angir hvilket polynom som omtales. For å sette polynomet over på oktal form, setter vi inn desimale tall i stede for de binære.

Her er et litt større eksempel:

$$G(D)=[D + 1, D^4, D^4 + D^3 + D + 1]$$

Vi får dette binære polynomet som så overføres til oktal form

$$G(D) = [\underbrace{00011}_0 \underbrace{10000}_3, \underbrace{10000}_2 \underbrace{0}_0, \underbrace{11011}_3 \underbrace{11}_3]$$

På oktal form får vi:

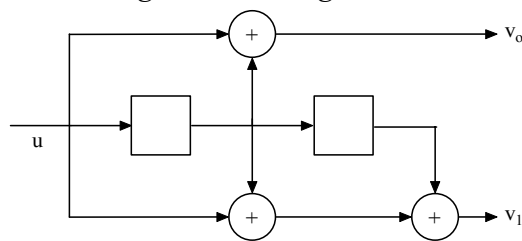
$$G(D)=[03, 20, 33]$$

I kodetreet i figur 3.1 øker antall noder eksponentielt med lengden på datasekvensen. Derfor finnes det også en annen måte å representere koden på. Vi kan fjerne en del redundant informasjon. Ved å flette sammen deler av treet som tar utgangspunkt i samme

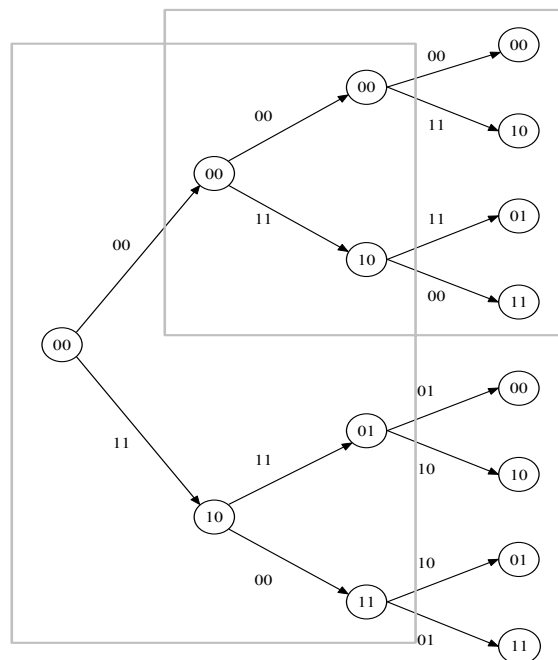
tilstand kan vi redusere antall noder til 2^m . En slik representasjon kalles et trellis diagram.

Eksempel

Vi har en (2,1,2) konvolusjons enkoder med rate $\frac{1}{2}$. Enkoderen er vist i figur 3.3. På samme måte som vist tidligere vil denne enkoderen gi kodetreet i figur 3.4. En gren oppover tilsvarende inndata 0 og en gren nedover tilsvarende inndata 1. Tallene langs stiene viser utdata og tallene i hver node viser tilstanden til enkoderen. Vi kan se at det ligger mye redundant informasjon i kodetreet. Innholdet i de to boksene i figur 3.4 er eksakt det samme. Det er dette vi ønsker å unngå i trellis diagrammet.



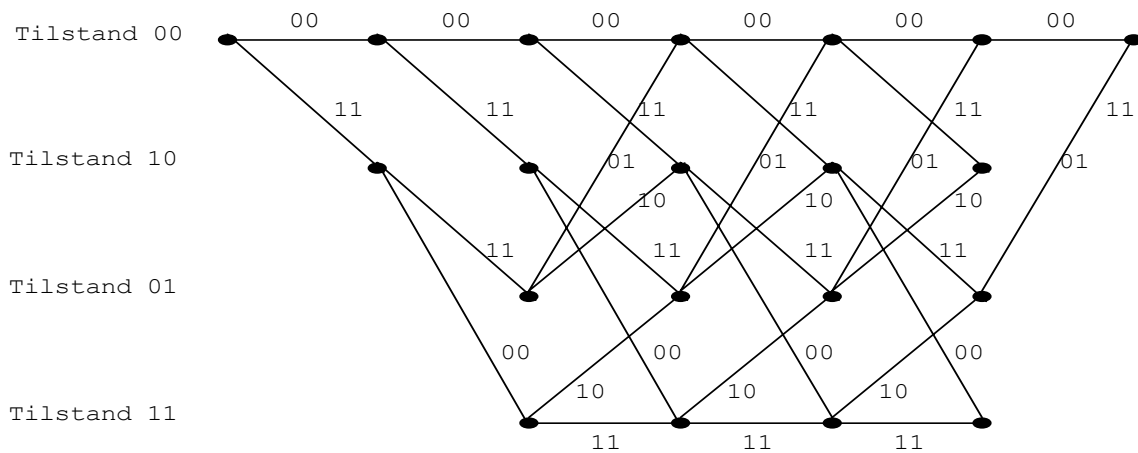
Figur 3.3: Enkoder til en (2,1,2)-kode.



Figur 3.4: Kodetreet til (2,1,2)-koden.

Figur 3.5 viser trellisdiagrammet til den samme enkoderen. Trellis diagrammet inneholder den samme informasjonen som kodetreet. Nodene i trelliset representerer tilstanden til enkoderen. Hver node har to stier ut som tilsvarende inndata til enkoderen. I dette tilfellet er det kun en bit inndata, 0 og 1. Den nedre stien tilsvarende 0, den øvre

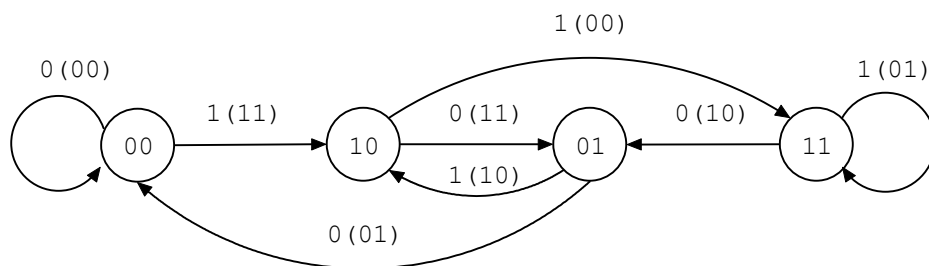
tilsvarer 1. Merkelappene på hver sti forteller hva utdata til enkoderen blir i hvert tilfelle. Av figur 3.5 ser vi at trelliset ikke vokser eksponentielt slik som kodetreet.



Figur 3.5: Trellisdiagram til en (2,1,2)-koden.

Trellisdiagrammet er altså en mer hensiktsmessig representasjon for lange informasjonssekvenser. I det generelle tilfellet vil 2^k stier forlate hver node og tilsvarende vil 2^k stier ende opp i hver node.

Man kan også laget et tilstandsdiagram for en kode. Figur 3.6 viser et slikt for (2,1,2)-koden i figur 3.3. Tilstandsdiagrammet inneholder noder som representerer tilstandene til enkoderen og grener ut fra disse. Tallene over hver gren beskriver inndata og tallene i parentes utdata til enkoderen. Vi ser at dette er en rettet graf.



Figur 3.6: Tilstandsdiagram for (2,1,2)-koden

En rettet graf representeres ofte som en matrise. I matrisen kan man slå opp den nye tilstanden ved gitt inndata og tilstand, se figur 3.7. På samme måte kan man også lage en matrise som viser utdata. Vi ser at denne tabellen kan bli stor for koder med stor registerlengde. For en vilkårlig stor kode vil størrelsen bli $2^m \cdot 2^n$. For en rate 2/3 kode med registerlengde 19 vil tabellen inneholde over 2 millioner elementer.

Inndata	0	1	...	2^k-2	2^k-1
Tilstand					
	Ny tilstand				
	0				
	1				
	.				
	.				
	2^m-2				
	2^m-1				

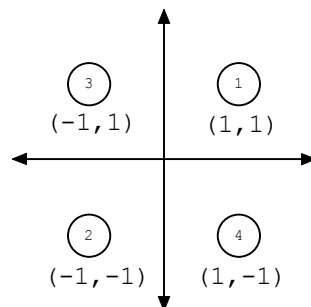
Figur 3.7: Tillstandsmatrise for en generell kode.

3.2 Den frie avstanden

For blokk-koder [7, s. 2] bruker vi minimum Hamming avstanden til en kode for å finne kodens feildeteksjons og feilkorreksjonsegenskaper. På tilsvarende måte kan vi bruke trellisdiagrammet til å finne den frie Hamming avstanden til en konvolusjonskode. Siden vi ikke har kodeord som i blokk-koder, må vi finne minimumsavstanden mellom kodesekvenser. Minimumsavstanden er da definert som den minste avstanden mellom alle kodesekvensene som starter og ender opp i samme tilstand. P.g.a. at koden er lineær må vi bare sammenligne par av sekvenser som starter og ender i 0 tilstanden [6, s. 184]. Siden den ene kodesekvensen da er "null sekvensen", sekvensen som bare inneholder 0'ere, vil vi kunne finne minimumsavstanden ved å finne Hamming vekten til de andre kodeordene. Hamming vekten er antall 1'ere i kodesekvensen. Minimum Hamming vekt blir derfor den frie avstanden til koden.

I trådløse systemer byttes gjerne biter ut med signalpunkter fra en signalkonstellasjon før dataene sendes over en kanal. Når vi gjør dette, kan vi ikke regne Hamming avstanden slik som forklart over. For å finne minimumsavstanden når signalpunkter er lagt på må vi beregne den Euklidske avstanden mellom to kodesekvenser.

Eksempel



Figur 3.8: 4-QAM.

Figur 3.8 viser en 4-QAM konstellasjon. Avstanden mellom punkt 1 og punkt 3 blir da:

$$d(1,3) = \sqrt{((1 - (-1))^2 + (1 - 1))^2} = 2$$

Den euklidske avstanden mellom punktene er definert som kvadratet av avstanden mellom punktene.

$$d^2(1,3) = 2^2 = 4$$

På samme måte kan vi finne den euklidske avstanden mellom de andre punktene i konstellasjonen.

$$d^2(1,2) = \sqrt{((1 - (-1))^2 + (1 - (-1)))^2} = 8$$

Den frie avstanden til en kode er definert i [1] og er

$$d_{fri} = \min_{\{a_n\} \neq \{a'_n\}} \left[\sum_n d^2(a_n, a'_n) \right]^{1/2}$$

der a og a' er symboler i en kodesekvens og $d^2(a_n, a'_n)$ er den euklidske avstanden mellom n -te komponenten til a og a' ved tidspunkt n .

Den frie avstanden er den minste avstanden mellom alle par av signalsekvenser som kan dannes av enoderen. Beregningen gjøres ved å finne den Euklidske avstanden mellom hvert symbol i hver sekvens og summere disse.

3.3 Partisjonering av signalkonstellasjoner

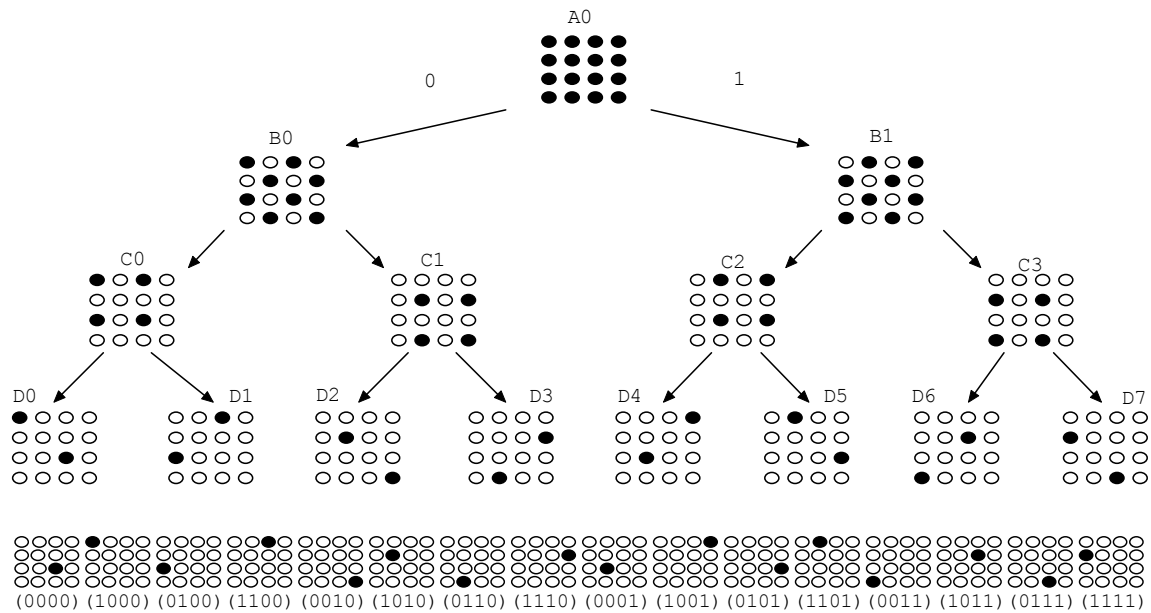
Ved bruk av feilkorrigerende koder er det nødvendig å legge til redundant informasjon for å kunne finne og korrigere eventuelle feil. Underboeck [2] introduserte en måte å legge til redundante biter uten å øke signal båndbredden. Dette kan gjøres gjennom 3 steg:

1. Legge til en redundant bit etter hver m -te inndata bit.
2. Utvide signalkonstellasjonen fra 2^m til 2^{m+1} signalpunkter.
3. Bruke de $m + 1$ kodete bitene til å velge signaler i den utvidede konstellasjonen.

Underboeck viser også hvordan han partitionerer konstellasjonene og hvordan han velger symboler ut fra de binære utdata fra enkoderen. Underboeck partitionerer konstellasjonene som vist i figur 3.9. Hvert partisjoneringsnivå gis en bokstav med A som utgangspunkt. Partisjonen på nivå A deles i to subkonstellasjoner som da blir nivå B. Disse to subkonstellasjonene splittes videre opp i 4 nye subkonstellasjoner som da er på nivå C. Slik fortsetter oppsplittingen helt til hver subkonstellasjon inneholder kun et signalpunkt. Subkonstellasjonene nummereres også som vist i figuren. Hver subkonstellasjon har høyere minimumsavstand en den forrige. Hvis vi setter

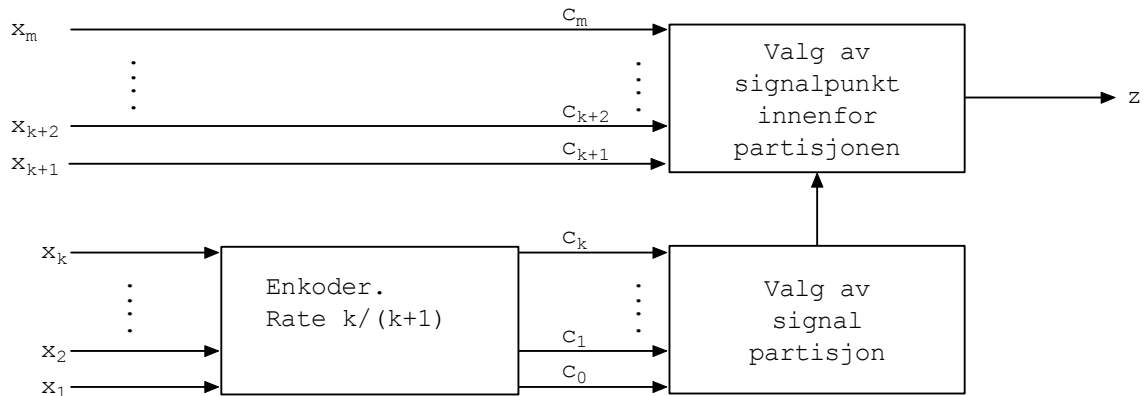
minimumsavstanden $d_{\min} = \Delta$ for A0, får vi $d_{\min} = \sqrt{2}\Delta$ for B0 og B1. For hvert nivå vi går nedover i partisjoneningeren ganger vi med $\sqrt{2}$ for å få minimumsavstanden. Generelt får vi at $d_{\min} = (\sqrt{2})^v \Delta$, hvis v angir nivået vi er på.

Nederst i figur 3.9 ser vi at hvert symbol har fått en binær merkelapp. Disse merkelappene får man ved å følge stien fra A0 ned til punktet. Går man til venstre får man 0, går man til høyre får man 1. Disse merkelappene brukes til å velge symboler ut i fra utdata til konvolusjonsenkoderen. Figur 3.10 viser hvordan dette gjøres i en rate $k/(k+1)$ enkoder. Vi kan bruke 16-QAM konstallasjonen i figur 3.9 i et eksempel.



Figur 3.9: Partisjonering av en 16 QAM konstallasjon [1].

I en 16-QAM konstallasjon trengs det 4 biter for å representere et symbol. Merkelappene til disse symbolene vises som nevnt nederst i figur 3.9. I dette tilfellet sendes det 3 biter inn i enkoderen for å få ut de 4 bitene som trengs for å velge et symbol i konstallasjonen. Altså en rate $\frac{3}{4}$ enkoder. I dette tilfellet velges en partisjon som kun inneholder et punkt og kodingen er ferdig. La oss si at vi fortsatt ønsker å bruke den samme koden med rate $\frac{3}{4}$, men vi ønsker å øke størrelsen på konstallasjonen til 64-QAM. I dette tilfellet trengs ytterligere 2 biter for å velge et symbol i konstallasjonen. Disse to bitene velges ukodet fra informasjonsbitene. Dvs. at de fire bitene som kommer ut av konvolusjonsenkoderen brukes til å velge en av partisjonene på nivå E. I en 64-QAM vil partisjoner på nivå E inneholde 4 punkter (tilsvarende nivå C i figur 3.9). De to siste bitene brukes dermed til å velge signalpunktet innenfor denne subkonstallasjonen.



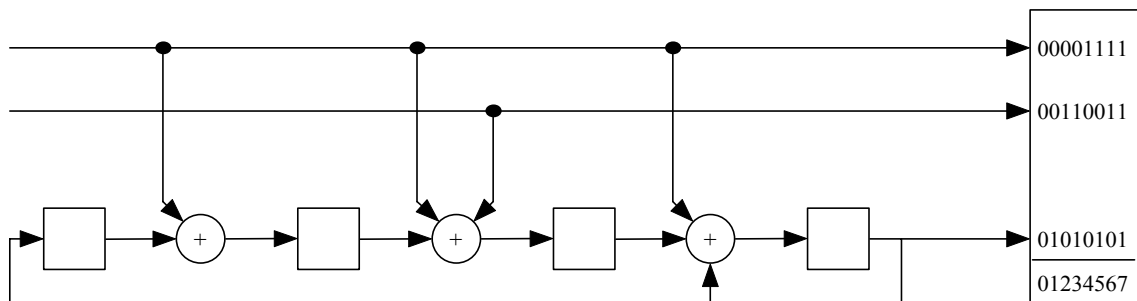
Figur 3.10: Signalvalg i en enkoder.

Figur 3.10 viser hvordan m informasjonsbiter x_1, x_2, \dots, x_m fra en informasjonssekvens velger et signal z fra en 2^{m+1} stor konstellasjon. Hvis $k=m$ som i eksempelet med 16-QAM tennes ingen ekstra biter for å finne et punkt innenfor en subkonstellasjon da subkonstellasjonen kun inneholder et punkt.

Ved å øke størrelsen på konstellasjonen så minker andelen av redundans biter og feilkorleksjonen ved dekoding vil bli svekket. Fordelen er at man kan sende flere informasjonsbiter pr. symbol og dermed øke den spektrale effektiviteten.

3.4 Feedback enkodere

Enkoderen i figur 3.2 og figur 3.3 er en såkalte "feedforward" enkoder. I en slik enkoder vil alle inndata biter og biter som ligger i minnet gå til neste minneblokk eller direkte til utdata. Det finnes også en annen type enkoder, en "feedback" enkoder (figur 3.11). En slik enkoder vil ha en forbindelse fra en minneblokk til en tidligere minneblokk. En slik enkoder vil man ikke kunne føre tilbake til nulltilstanden ved å putte inn m antall 0'er, da en 1'er vil kunne føres tilbake i tidligere minneblokker. Dette vil jeg komme tilbake til senere. "Feedforward" enkodere med rate $k/(k+1)$ kan ofte gjøres om til feedback enkodere ved litt arbeid [2 og 14]. Hvordan man gjør dette vil jeg ikke gå inn på i denne oppgaven.

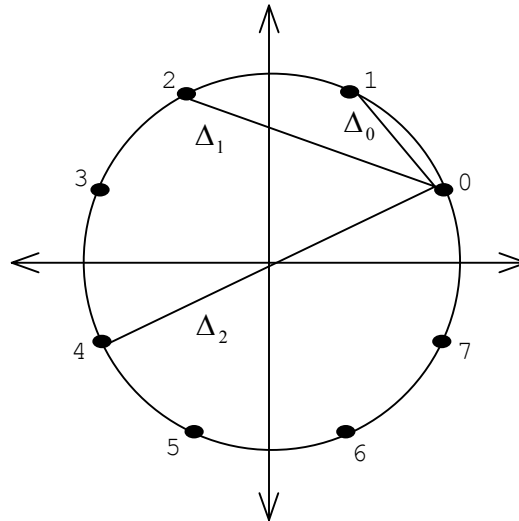


Figur 3.11: Feedback enkoder rate 2/3 med $G(D)=[23,04,16]$.

Figur 3.11 viser en feedback enkoder med $G(D)=[23,04,16]$. Vi ser her at $g^{(0)}=23$ definerer polynomet med tilbakeføringskanalen mens de to andre polynomene $g^{(1)}$ og $g^{(2)}$ definerer hvordan inndata bitene skal innvirke på enkoderen. Tallene i boksen til høyre i figuren viser alle mulige utdata ved ulike inndata og innhold i minneblokkene.

3.5 Koding i halen

Som nevnt tidligere er det ikke mulig å føre en feedback enkoder tilbake til null tilstanden ved å sende inn m nullere. Dette kan føre til at minimums avstanden mellom de kodete sekvensene kan bli mindre en den frie avstanden til koden. I [3] beskrives det hvordan dette problemet kan løses for en 8-PSK konstellasjon. I stede for å buke vanlig ”mapping” i halen, kan man endre på ”mappingen ” for å unngå reduksjonen i minimumsavstanden.



Figur 3.12: 8-PSK signal konstellasjon.

Utdata i figur 3.11 tilsvarer hvert sitt punkt i 8-PSK konstellasjonen i figur 3.12. Hvis vi setter radius i sirkelen i figur 3.12 til 1 får vi at kvadratet av avstandene mellom de ulike punktene blir:

$$\Delta_0^2 = 0,586$$

$$\Delta_1^2 = 2,0$$

$$\Delta_2^2 = 4,0$$

Figur 3.13 (a) viser vanlig ”mapping” i halen. Tallene inne i punktene viser tilstanden til enkoderen, mens tallene langs stiene viser utdata (konstellasjonssymboler) fra enkoderen. Fra første punkt starter vi fra null tilstanden og tar alle mulige utdata fra denne. De neste stegene består av m antall steg som tilsvarer halen i kodetreet. Her er inndata 0 og utdata blir da kun avhengig av de bitene som sirkulerer i minneblokkene på grunn av ”feedback”. Derfor får vi kun symbolene 0 eller 1 som utdata langs de resterende stiene.

Minimumsavstanden mellom stiene i treet finner vi ved å regne ut avstanden mellom hver av de seks parene med stier. Den Euclidske avstanden mellom to signaler a_n og a'_n er $d(a_n, a'_n)$. Minimumsavstanden mellom stiene i halen blir da:

$$d(0,1) = \Delta_1^2 + \Delta_0^2 = 2,0 + 0,586 = 2,586$$

$$d(0,2) = \Delta_1^2 + 2 * \Delta_0^2 = 2,0 + 2 * 0,586 = 3,172$$

$$d(0,3) = \Delta_2^2 + 3 * \Delta_0^2 = 4,0 + 3 * 0,586 = 5,758$$

$$d(1,2) = \Delta_2^2 + 3 * \Delta_0^2 = 4,0 + 3 * 0,586 = 5,758$$

$$d(1,3) = \Delta_1^2 + 2 * \Delta_0^2 = 2,0 + 2 * 0,586 = 3,172$$

$$d(2,3) = \Delta_1^2 + \Delta_0^2 = 2,0 + 0,586 = 2,586$$

Merk! Kvadratet av avstanden mellom punkt 0 og 2 er den samme som mellom 0 og 6 osv. Figur 3.13 viser hvilke avstander som er de samme.

Ut fra dette ser vi at minimumsavstanden mellom stiene i figur 3.11 (a) er 2,586. I [3] kan vi finne at miniumsavstanden til koden er 4.586. Vi ser at disse tallene ligger langt fra hverandre. Dette medfører en større feilsannsynlighet i halen på treet. I figur 3.13 (b) er det presentert en bedre løsning. Siden halen kun består av 0 eller 1 signalpunkter kan vi endre mappingen i halen slik at vi får størst mulig avstand mellom de to signalpunktene. I figur 3.13 (b) har vi byttet ut alle signalpunkt 1 med signalpunkt 4. Vi ser i figur 3.12 at signalpunkt 0 og 4 er de punktene som ligger lengst fra hverandre i konstellasjonen. Ved å endre mappingen på denne måten vil vi kunne øke minimumsavstanden i halen.

Minimumsavstanden mellom stiene blir nå :

$$d(0,1) = \Delta_1^2 + \Delta_2^2 = 2,0 + 4,0 = 6,0$$

$$d(0,2) = \Delta_1^2 + 2 * \Delta_2^2 = 2,0 + 2 * 4,0 = 10,0$$

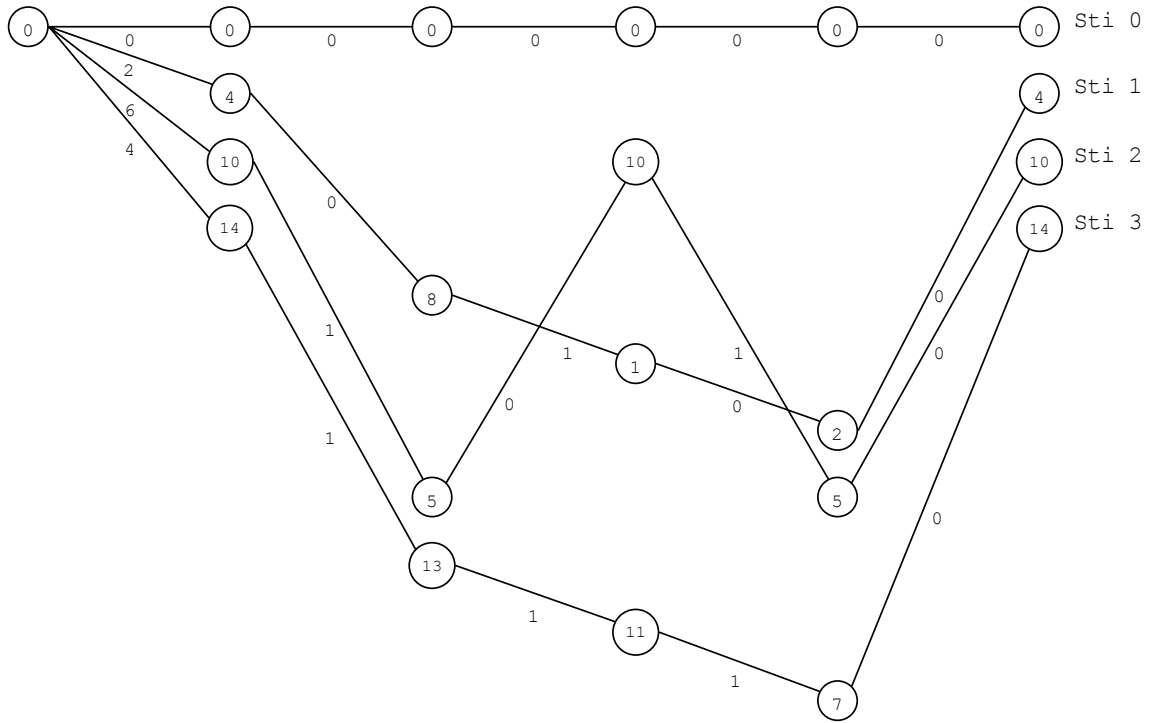
$$d(0,3) = \Delta_2^2 + 3 * \Delta_2^2 = 4,0 + 3 * 4,0 = 16,0$$

$$d(1,2) = \Delta_2^2 + 3 * \Delta_2^2 = 4,0 + 3 * 4,0 = 16,0$$

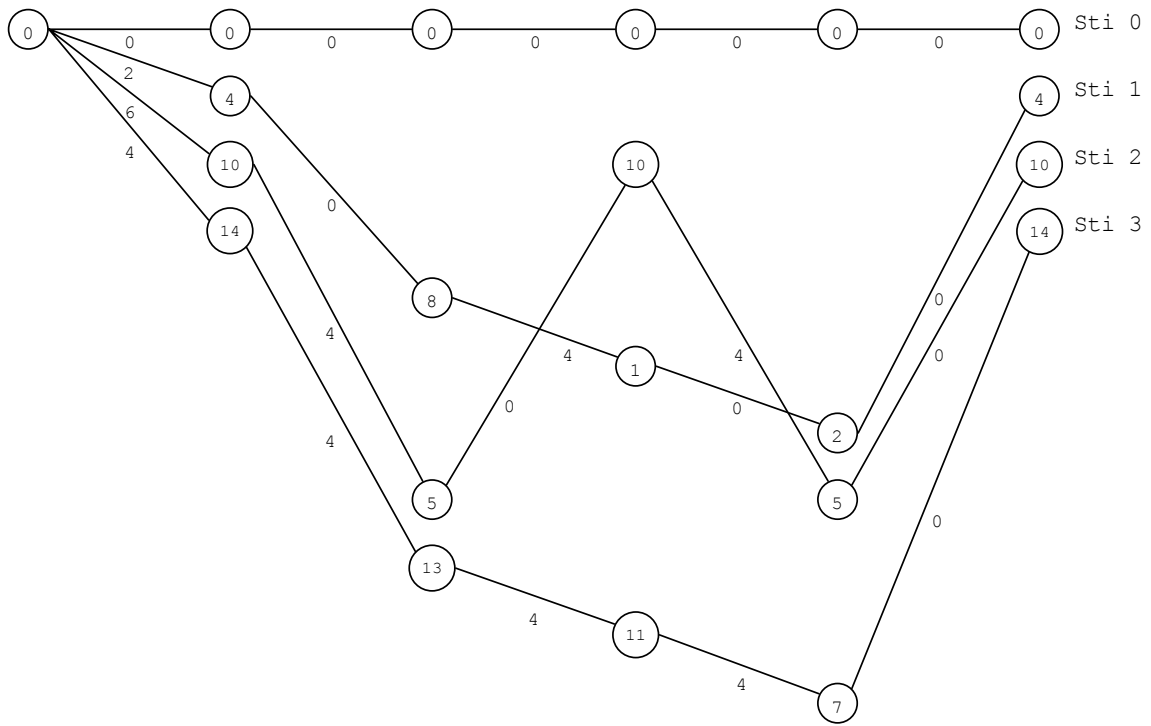
$$d(1,3) = \Delta_1^2 + 2 * \Delta_2^2 = 2,0 + 2 * 4,0 = 10,0$$

$$d(2,3) = \Delta_1^2 + \Delta_2^2 = 2,0 + 4,0 = 6,0$$

Her ser vi at minimumsavstanden i halen har økt til 6,0 og at halen ikke svekker feilsannsynligheten til informasjonsbiter i enden av hver blokk.



(a)



(b)

Figur 3.13: Trellis diagram som viser ”mapping” i halen ved 8-PSK.

En tilsvarende teknikk benyttes også for ”mapping” i halen for større signalkonstellasjoner. Vi finner det punktet som har størst avstand til 0 og bytter ut punkt

1 med dette punktet. På denne måten oppnår vi størst mulig avstand mellom de to punktene.

I [3] er det gjort forsøk der man sammenligner BER i de siste informasjonsbitene og lengden på halen. Jo lenger hale vi har jo mindre blir BER. Kurven vil flate ut etter hvert og det vil være mindre og mindre å tjene på å forlenge halen jo lenger halen er. For å minimere BER trengs det en hale med en lengde som minimum tilsvarer registerlengden til koden.

Halens innvirkning på feilsannsynligheten avhenger også av størrelsen på blokkene. Jo lengre blokker vi har desto mindre innflytelse vil halen ha. Om vi bruker kontinuerlig dekodning uten oppdeling i blokker kan vi nesten se bort fra påvirkning fra halens lengde.

3.6 Dekoding

Shanons teorem om kanalkapasitet [1, s. 10] forteller at alle kanaler har en kanalkapasitet C og at for enhver rate $R < C$ finnes det en kode med rate R som har en vilkårlig liten feilsannsynlighet $P(E)$ ved MLD.

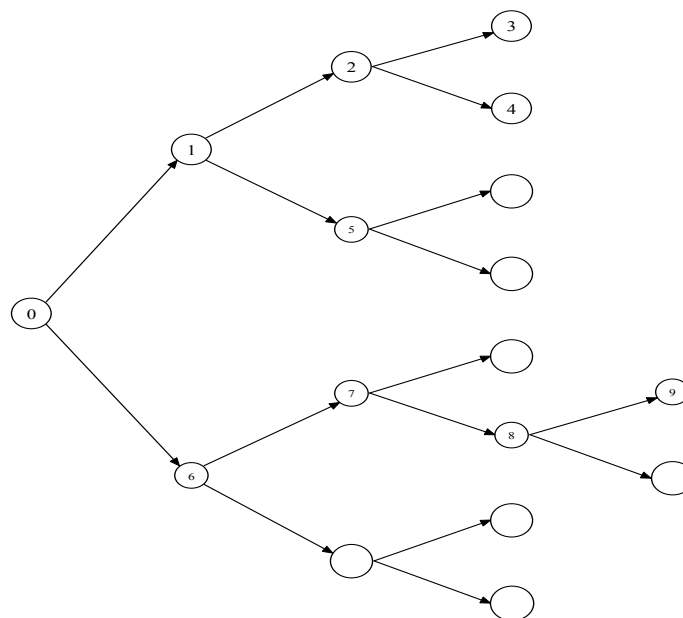
Ved bruk av viterbidekodning av konvolusjonskoder vil ikke denne grensen kunne nås i praksis. Dette skyldes at det kun kan benyttes små registerlengder på grunn av begrensninger i minnet på dekoderen. Antall beregninger ved bruk av viterbidekodning er 2^m uansett hvor mange feil som oppstår, der m er antall minneblokker i enkoderen. Om det oppstår få eller ingen feil i overføringen vil dette være sløsing av resurser. Sekvensiell dekodning tar hensyn til dette. Dekodingen tilpasser seg støynivået slik at det ved liten støy vil være få beregninger, mens det ved mange feil vil være mange beregninger. Ved sekvensiell dekodning er antall beregninger uavhengig av m . Dette vil si at man kan bruke koder med store registerlengder og Shanons teorem om vilkårlig liten feilsannsynlighet kan nås. Problemet med sekvensiell dekodning er at det blir utført veldig mange beregninger når det kommer mye støy på kanalen. Her vil det finnes en øvre grense for hvor mange feil som kan rettes før informasjon går tapt.

To av de mest vanlige typene sekvensiell dekodning er Fano algoritmen som ble introdusert av Fano i 1963. Den andre er Stack-algoritmen eller ZJ-algoritmen som ble oppdaget av Zigangirov og Jelinek noen år senere [1].

Sammenlignet med blokk-koder blir dekodningen av konvolusjonskoder mer komplekst siden vi ikke har kodeord med fastsatt lengde. Dette medfører at dekoderen må vente i ubestemt tid før den kan skille mellom to forskjellige kodesekvenser. Det finnes to grupper teknikker å utføre dekodningen. MLSD (Maximum Likelihood Sequence Detection) og sekvensiell dekodning. MLSD kan også beskrives som et bredde først søk, mens sekvensiell dekodning kan beskrives som dybde først søk. Den mest kjente MLSD algoritmen er Viterbi algoritmen [1, s. 315].

Viterbi algoritmen søker gjennom alle stier i trellisdiagrammet og er derfor garantert å finne den nærmeste stien til den mottatte stien. Kompleksiteten i algoritmen øker eksponentielt med registerlengden. Registerlengder over 16 vil i praksis kreve for mange beregninger. Sekvensiell dekodning tar hensyn til dette og vil være mer effektivt for større koder.

Figur 3.14 viser hvordan et dybde først søk kan bli utført i et kodetre. Et dybde først søk vil aldri søke gjennom hele trelliset. Derfor er man heller ikke garantert å dekode til den stien som ligger nærmest. Denne typen algoritmer er derfor ikke MLD. Stabel-algoritmen og Fano-algoritmen er to typer dybde først søk algoritmer. Her vil jeg konsentrere meg om Stabel-algoritmen da denne er enklest å forstå og enklest å implementere i en simulator. Jeg har også lagt vekt på at Stabel-algoritmen er raskere da den besøker færre noder enn Fano-algoritmen.



Figur 3.14: sekvensiell dekodning.

3.7 Metrikk

En metrikk beregnes for å kunne fortelle oss i hvor stor grad en sti er lik en annen sti. Teknikken brukes i både Viterbidekodning og i sekvensiell dekodning. Når man sjekker om en sti i treet ligger nær en mottatt sti i dekoderen beregnes metrikken mellom disse stiene. Metrikken i Viterbialgoritmen er gitt ved $d(\bar{r}, \bar{v})$ der \bar{r} er det mottatte kodeordet og \bar{v} er en sti i kodetreet. Metrikken er altså gitt ved Hamming avstanden mellom de to kodeordene.

Eksempel:

$$\bar{r} = 000\ 101\ 110\ 001\ 111\ 101$$

$$\bar{v}_1 = 000\ 001\ 010\ 001\ 111\ 101$$

$$\bar{v}_2 = 001\ 101\ 010$$

$$\bar{v}_3 = 001$$

$$d(\bar{r}, \bar{v}_1) = 2$$

$$d(\bar{r}, \bar{v}_2) = 2$$

$$d(\bar{r}, \bar{v}_3) = 1$$

Vi ser her at beregningen ikke tar hensyn til lengden av kodeordene. Selv om $d(\bar{r}, \bar{v}_3)$ har den beste metrikken trenger ikke det å bety at dette er den beste stien. For sekvensiell dekoding er det derfor utviklet en annen måte å beregne metrikken på. Denne beregningen tar også hensyn til stiens lengde. Bitmetrikken ble innført av Fano og har derfor fått navnet Fanometrikken. Fanometrikken er definert til:

$$M(r_i | v_i) = \log_2 \frac{P(r_i | v_i)}{P(r_i)} - R$$

Her er $P(r_i | v_i)$ er feilsannsynligheten ved overføring av en bit, og $P(r_i)$ er sannsynligheten for at et symbol blir valgt. For en BSC (Binary Symmetric Channel) med feilsannsynlighet p og rate R er det vist ([1], s. 353) at metrikken blir

$$M(r_i | v_i) = \begin{cases} \log_2 2p - R & \text{for } r_i \neq v_i \\ \log_2 2(1-p) - R & \text{for } r_i = v_i \end{cases}$$

Hvis vi setter $p=0.10$ og $R=1/3$ får vi følgende metrikker.

$$M(r_i | v_i) = \begin{cases} -2.65 & \text{for } r_i \neq v_i \\ 0.52 & \text{for } r_i = v_i \end{cases}$$

Fra kodesekvensene i eksempelet kan vi da beregne metrikkene som nå tar hensyn til lengden på kodesekvensene. Legg merke til at man her beregner metrikken for alle komponentene i kodesekvensen og legger disse sammen.

$$M(\bar{r}, \bar{v}_1) = 16*0,52+2*(-2.65)=3.02$$

$$M(\bar{r}, \bar{v}_2) = 7*0,52+2*(-2.65)=-1.66$$

$$M(\bar{r}, \bar{v}_3) = 2*0,52+1*(-2.65)=-1.61$$

Her ser vi at \bar{v}_1 er den beste metrikken og at denne sekvensen da vil bli foretrukket.

Hvis vi sender QAM signalpunkter over en kanal og ikke kun biter, trenger vi en annen måte å beregne metrikken på. Når vi overfører en 1'er bit over en kanal vil den med tilstrekkelig støy kunne endres til 0. Ved bruk av QAM symboler vil symbolet ved påvirkning av støy bevege seg i et koordinatsystem. Jo mer støy jo lenger vekk fra sin opprinnelige posisjon vil punktet vandre. Dette er nærmere beskrevet i kapittelet om støy. Fanometrikken som brukes for dekoding med signalkonstellasjoner presenteres i [3, s.1805] og er:

$$M_B(a_l, z_l) = \log_2 \frac{\exp\left(-|z_l - a_l|^2 / 2\sigma^2\right)}{\sum_{i=0}^{N-1} \exp\left(-|z_l - a^i|^2 / 2\sigma^2\right)} + n(1 - R)$$

der $|z_l - a_l|^2 = (z_l^x - a_l^x)^2 + (z_l^y - a_l^y)^2$

- z_l er mottatt signal på tidspunkt l . z_l^x og z_l^y er mottatt signal delt inn i x og y komponent.
- a_l er merkelapp på gren i treet på tidspunkt l .
- $R = k/n$ er raten til koden.
- σ^2 er variansen som er nærmere definert i støy kapittelet.
- K er antall punkter i signalkonstellasjonen.
- a^i er punkt nr i i signalkonstellasjonen.

For trelliskoder der n er lik $k+1$ og raten $R = k/(k+1)$, kan vi forenkle det siste leddet.

Det siste leddet i formel (3) blir da:

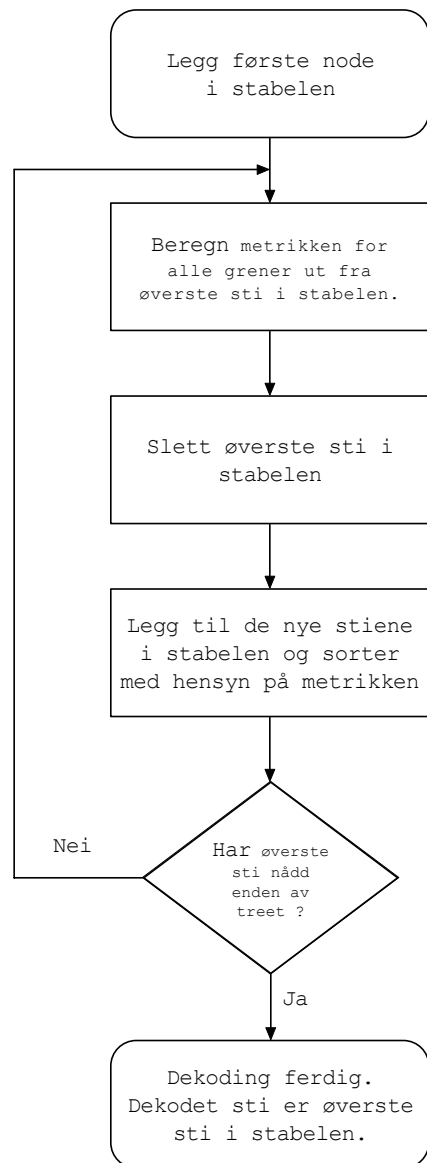
$$n(1 - R) = n\left(1 - \frac{k}{n}\right) = (k+1)\left(1 - \frac{k}{k+1}\right) = k+1 - k = 1$$

Det siste leddet vil for alle trelliskoder med rate $R = k/(k+1)$ derfor alltid bli 1.

3.8 Stabel-algoritmen

Stabel-algoritmen er en av de mest brukte algoritmene til sekvensiell dekoding [1, s. 351]. Stabel-algoritmen bruker en stabel til å organisere ulike grener i et kodetre. Hvert element i stabelen inneholder stien i treet og metrikken til denne stien. Stabelen blir sortert etter metrikk med største metrikk øverst. Algoritmen utvider hele tiden den stien med best metrikk, altså den stien som ligger øverst i stabelen. Det vil alltid være 2^k stier

som går ut fra hver node i et tre. Stabel algoritmen utvider den øverste stien i stabelen med 2^k nye delstier og legger metrikken til disse stiene til den opprinnelige stien. Det blir nå dannet 2^k nye stier som er en gren lenger enn den forrige. Disse grenene blir så lagt inn i stabelen. Den opprinnelige stien blir slettet. Stabelen blir nå sortert på ny og de samme operasjonene vil bli utført på stien som nå ligger øverst i stabelen. Slik fortsetter algoritmen helt til en av stiene når enden av treet. Dette blir da den dekodete stien. Figur 3.15 viser et flytdiagram over stabelalgoritmen.



Figur 3.15: Flytdiagram over stabel algoritmen ([1], s. 356).

For koder med $k=1$ vil stabelen legge til to nye stier og slette den gamle stien i hvert steg med unntak av halen der inndata kun er 0 og algoritmen kun legger til 1 sti. Dette vil si at lengden på stabelen utvides med 1 for hvert steg i algoritmen så lenge den ikke dekode i halen. Med store blokker vil halen være så kort i forhold til resten av treet at størrelsen på

stabelen vil være tilnærmet lik antall steg i algoritmen. For vilkårlig stor k vil stabelen utvides med $2^k - 1$ elementer i hvert steg.

Eksempel

Vi ønsker å kode en bitsekvens. Legge på støy på denne sekvensen og dekode sekvensen igjen. Til koding brukes enkoderen i figur 3.11. Dette er en rate $\frac{2}{3}$ enkoder med generatorpolynom $G(D)=[23,04,16]$. Registerlengden til denne koden er 4. Vi ønsker å bruke 8-PSK konstallasjonen i figur 3.12 med gjennomsnittlig energi lik 1. For å sikre feilkorreksjon i de siste bitene legger vi også til en hale med lengde 4. I dekodingen av halen endrer vi mappingen som vist i figur 3.13 (b).

Vi velger følgende inndata-sekvens:

00 00 00 10 11

Vi legger så til 8 nullere på grunn av halen og får:

00 00 00 10 11 00 00 00 00

Disse bitene vil nå bli kjørt gjennom enkoderen som koder signalet og returnerer en sekvens av PSK-symboler. Ut fra enkoderen i figur 3.11 vil vi kunne lage tabellene i figur 3.16. Tabellene viser hvilke tilstand enkoderen kommer til (a) og hva slags utdata den vil gi når enkoderen er i en gitt tilstand og man gir en viss inndata (b). Inndata i dette tilfellet er to biter og med inndata 0 menes inndata 00, inndata 1 er 10, 2 er 01 og 3 er 11. Den første biten er altså den minst signifikante.

Tilstand \ Inndata	Inndata			
	0	1	2	3
0	0	4	14	10
1	2	6	12	8
2	4	0	10	14
3	6	2	8	12
4	8	12	6	2
5	10	14	4	0
6	12	8	2	6
7	14	10	0	4
8	9	13	7	3
9	11	15	5	1
10	13	9	3	7
11	15	11	1	5
12	1	5	15	11
13	3	7	13	9
14	5	1	11	15
15	7	3	9	13

(a)

Tilstand \ Inndata	Inndata			
	0	1	2	3
0	0	2	4	6
1	0	2	4	6
2	0	2	4	6
3	0	2	4	6
4	0	2	4	6
5	0	2	4	6
6	0	2	4	6
7	0	2	4	6
8	1	3	5	7
9	1	3	5	7
10	1	3	5	7
11	1	3	5	7
12	1	3	5	7
13	1	3	5	7
14	1	3	5	7
15	1	3	5	7

(b)

Figur 3.16: (a) viser neste tilstand ved gitt inndata og tilstand. (b) viser utdata ved gitt inndata og tilstand.

Enkoderen starter alltid i 0 tilstanden. Når inndata er 0, ser vi ut fra tabellene at neste tilstand og utdata også vil bli 0. Fra bitsekvensen som starter med 00 00 00 får vi derfor

0 0 0 som utdata i de tre første stegene. Tilstanden vil fortsatt være 0, men i steg 4 vil inndata være 10 (1). Dette fører til at neste tilstand blir 4 og utdata blir 2. I steg 5 vil inndata være 11(3). Enkoderen er nå i tilstand 4 og fra tabell (a) får vi nå neste tilstand lik 2 og utdata lik 6 fra tabell (b). På denne måten kan vi fortsette til hele bitsekvensen er kodet. Vi får:

00→0
00→0
00→0
10→2
11→6
00→0
00→0
00→1
00→1

Utdata i tabell b er nummererte PSK symboler. Vi får dermed følgende sekvens av PSK symboler:

0,0,0,2,6,0,0,1,1

Som nevnt tidligere ønsker vi å endre mappingen i halen. Halen i dette tilfellet er de siste 4 PSK symbolene. Alle PSK symboler i halen som er lik 1 blir endret til 4 for å oppnå størst mulig avstand mellom punktene. Se figur 3.12. Når vi innfører dette får vi følgende kodeblokk:

0,0,0,2,6,0,0,4,4

Over kanalen sender vi nå signalpunktene som x- og y-koordinater.

Kodeblokken med koordinater blir da:

0 - (0.92388,0.382683)
0 - (0.92388,0.382683)
0 - (0.92388,0.382683)
2 - (-0.382683,0.92388)
6 - (0.382683,-0.92388)
0 - (0.92388,0.382683)
0 - (0.92388,0.382683)
4 - (-0.92388,-0.382683)
4 - (-0.92388,-0.382683)

Når PSK-symbolene overføres over kanalen blir de påvirket av støy. I dette eksemplet er det lagt til tilfeldig generert støy på signalpunktene. For mer informasjon om støy se kapittel 2. Signalpunktene med støy blir nå:

0 ~ (1.16973,-0.29095)
0 ~ (0.0337866,0.0825665)
0 ~ (1.04701,0.843747)
2 ~ (-1.15868,1.26253)
6 ~ (-0.00341626,-0.88884)
0 ~ (0.854852,0.648295)
0 ~ (1.26871,-0.313643)
4 ~ (-1.04148,-0.308528)
4 ~ (-1.08928,-0.316096)

Vi bruker nå stabelalgoritmen til dekoding. I metrikkberegningen setter vi $\sigma = 0.45$. Figuren viser stabelen for hvert steg i stabelalgoritmen. Den opprinnelige stien, altså den korrekte dekodete stien er uthevet. I sekvensene i figuren er PSK-symbolene dekodet til biter.

<p>Steg 1</p> <p>00 (-1.12) 11 (-2.95) 10 (-13.34) 01 (-15.17)</p>	<p>Steg 2</p> <p>0010 (-2.73) 0000 (-2.74) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 10 (-13.34) 01 (-15.17)</p>	<p>Steg 3</p> <p>001000 (-2.43) 0000 (-2.74) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 001010 (-9.04) 10 (-13.34) 001011 (-14.53) 01 (-15.17) 001001 (-21.14)</p>	<p>Steg 4</p> <p>00100010 (-2.67) 0000 (-2.74) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 00100000 (-8.69) 001010 (-9.04) 10 (-13.34) 001011 (-14.53) 01 (-15.17) 00100001 (-19.17) 001001 (-21.14) 00100011 (-25.19)</p>
<p>Steg 5</p> <p>0000 (-2.74) 0010001001 (-2.78) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 0010001010 (-11.19) 10 (-13.34) 001011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00100001 (-19.17) 001001 (-21.14) 00100011 (-25.19)</p>	<p>Steg 6</p> <p>000000 (-2.44) 0010001001 (-2.78) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 0010001010 (-11.19) 10 (-13.34) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19)</p>	<p>Steg 7</p> <p>00000010 (-2.26) 0010001001 (-2.78) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0010001010 (-11.19) 10 (-13.34) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>	<p>Steg 8</p> <p>0000001011 (-2.40) 0010001001 (-2.78) 11 (-2.95) 0001 (-3.65) 0011 (-3.65) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>

<p>Steg 9</p> <p>00000101100 (-2.14) 0010001001 (-2.78) 11 (-2.958) 0001 (-3.65) 0011 (-3.65) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>	<p>Steg 10</p> <p>0010001001 (-2.78) 11 (-2.95) 0000010110000 (-3.35) 0001 (-3.65) 0011 (-3.65) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>	<p>Steg 11</p> <p>11 (-2.95) 00000010110000 (-3.35) 0001 (-3.65) 0011 (-3.65) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 001000100100 (-16.76) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>	<p>Steg 12</p> <p>00000010110000 (-3.35) 0001 (-3.65) 0011 (-3.65) 1100 (-4.38) 1110 (-5.02) 1111 (-5.03) 1101 (-5.67) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 001000100100 (-16.76) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>
<p>Steg 13</p> <p>0000001011000000 (-2.89) 0001 (-3.65) 0011 (-3.65) 1100 (-4.38) 1110 (-5.02) 1111 (-5.03) 1101 (-5.67) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 001000100100 (-16.76) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>	<p>Steg 14</p> <p>000000101100000000 (-2.40) 0001 (-3.65) 0011 (-3.652) 1100 (-4.38) 1110 (-5.02) 1111 (-5.03) 1101 (-5.67) 0000001001 (-5.85) 0010001011 (-6.30) 00100000 (-8.69) 001010 (-9.04) 000010 (-9.04) 00000001 (-9.67) 0000001000 (-10.83) 0010001010 (-11.19) 10 (-13.34) 0000001010 (-14.28) 001011 (-14.53) 000011 (-14.53) 0010001000 (-14.71) 01 (-15.17) 001000100100 (-16.76) 00000000 (-18.19) 00100001 (-19.17) 001001 (-21.14) 000001 (-21.14) 00100011 (-25.19) 00000011 (-25.60)</p>		

Vi ser i steg 14 at vi ender opp med samme bitsekvens som vi startet med: 00 00 00 10 11 00 00 00 00. Stabel algoritmen dekodet derfor sekvensen uten feil.

I eksemplet ser vi at stabelen utvides med 3 elementer for hvert steg så lenge vi ikke dekker i halen. Dette kan vi også se ved å sette inn $k=2$ i formelen $2^k - 1$. For koder

med større k og lengre kodesekvenser ser vi at stabelen kan bli veldig lang etter hver. Spesielt hvis vi har mye støy. For vilkårlig stor k og blokk lengde L vil vi trenge et uendelig stort minne til å oppbevare stabelen. Om stabelen skulle bli større enn det tilgjengelige minnet vil vi kunne få en overflyt og miste elementer i stabelen. Dette er årsaken til at vi ofte deler opp data i blokker med lengde L og ikke benytter kontinuerlig dekoding.

Hvis vi skulle få en overflyt i stabelen løses dette ved å slette elementene som ligger nederst i stabelen. Metrikken er et estimat for sannsynligheten for at en sti er den korrekte. Det er derfor svært liten sannsynlighet for at stiene som ligger nederst i stabelen noen gang vil vandre helt til toppen og bli det endelige dekodete resultatet.

Et annet problem kan være at sortering av alle elementene i stabelen kan bli tidkrevende. Jelinek har introdusert *stack-bucket* algoritmen der han sorterer stiene i "bøtter" eller intervaller der man ikke sorterer metrikker innenfor samme intervall ([1], s. 360). Jeg kommer ikke til å gå nærmere inn på denne algoritmen.

En annen og mye brukt algoritme til sekvensiell dekoding er Fano-algoritmen. Algoritmen er ikke like rask som stabel-algoritmen, men krever til gjengjeld svært lite med minne. I motsetning til stabel-algoritmen hopper ikke Fano-algoritmen fra et sted i kodetreet til et annet, men jobber seg hele tiden frem og tilbake på samme gren. I et steg så beveger den seg enten fremover til en av de 2^k neste nodene eller tilbake til den forrige noden. Metrikken til stien blir lagt til eller trukket fra og man trenger ikke lagre metrikker til andre stier. En ulempe er at det ofte blir nødvendig å besøke samme node flere ganger og at man derfor må gjøre samme metrikkberegning flere ganger. Algoritmen fortsetter fremover så lenge metrikken ikke blir mindre enn en satt grense. Den søker da andre stier ut fra samme node. Hvis ingen stier holder seg over den gitte grensen, senkes grensen og søket begynner på nytt. Når algoritmen når enden av treet avsluttes den og gjeldende sti er nå den dekodete stien. I [1, s. 360-364] står en nærmere beskrivelse av Fano-algoritmen.

Kapittel 4

Nye Resultater

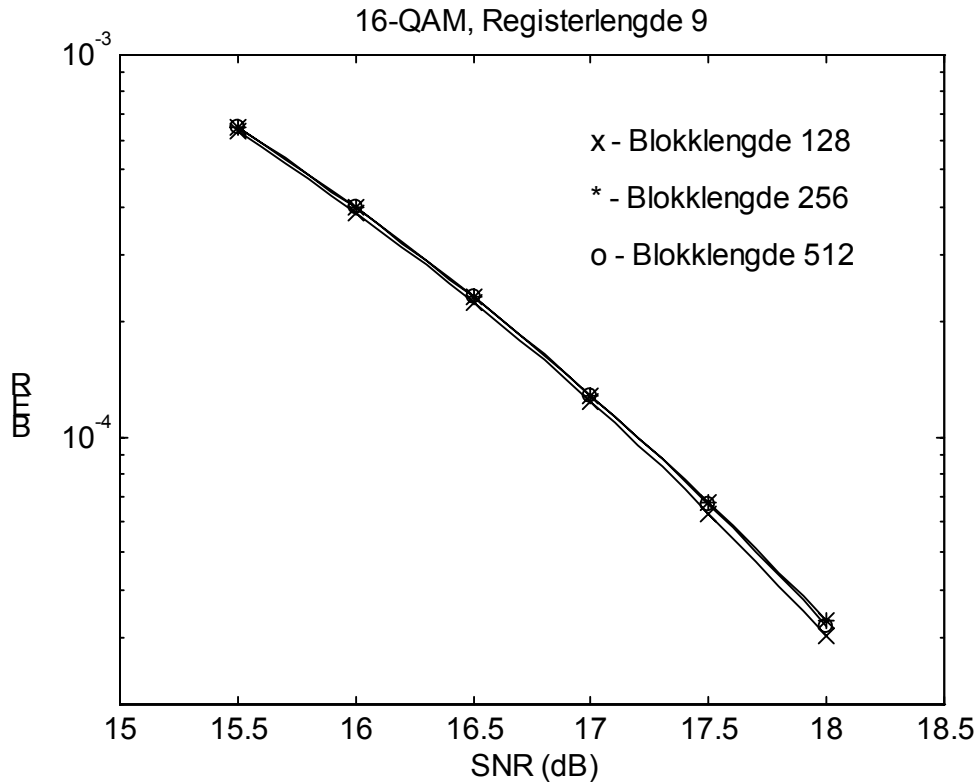
I dette kapittelet presenteres simuleringsresultater fra simuleringer med kjente og tilfeldig genererte trelliskoder. Simuleringene er utført av et eget utviklet simuleringsprogram. Se vedlegg 2. Programmet tar inn tilfeldig genererte data som kodes og sendes over en simulert AWGN kanal der støyen er normalfordelt med forventning 0 og varians σ^2 som beskrevet i kapittel 2. Til dekoding er stabel-algoritmen benyttet. Inndataene er splittet opp i blokker siden kontinuerlig dekoding raskt fører til overflyt i stabelen om mengden med inndata er stor.

4.1 Resultater fra kjente koder

4.1.1 Tidsbruk

Etter å ha kjørt en del simuleringer ble det raskt klart at simuleringer er en tidkrevende prosess. Mest innvirkning på tiden hadde størrelsen på konstallasjonen. Dette er årsaken til at jeg ikke har laget simuleringer for større konstallasjoner enn 64-QAM. Blokk lengden hadde også betydning på kjøretiden og jeg ønsket derfor å finne ut om blokk lengden hadde stor innvirkning på BER. Jeg valgte derfor en registerlengde 9 kode fra [8] og gjorde 3 simuleringer med disse kodene hvor jeg satte blokk lengden til henholdsvis 512, 256 og 128. Koden som er brukt har paritetssjekkpolynom [1401,0166,0300]. I simuleringene er det benyttet en hale med lengde lik registerlengden. Fra kapittel 4 husker vi at en hale på minimum denne lengden var nødvendig for å minimere BER. Ved å benytte en hale på lengde 9 skulle man derfor i dette tilfellet oppnå gode resultater uansett blokk lengde. Simuleringsresultatene er vis i figur 4.1.

Som ventet ser vi at BER er nesten identisk for de tre simuleringene. Dette stemmer godt over ens med resultater i [3]. For å spare tide har jeg derfor valgt å bruke mindre blokk lengder for store konstallasjoner da dette vil ha liten innvirkning på resultatet. Jeg har allikevel valgt å benytte en blokk lengde på 512 i simuleringene mine for 8-PSK og 16-QAM. Årsaken er at det i [8] er benyttet blokk lengde på 512 og at jeg ønsker å foreta de samme simuleringene under de samme forholdene.



Figur 4.1: Simuleringsresultater for blokk lengde 128, 256 og 512.

Som nevnt øker simuleringstiden ved å øke størrelsen på konstellasjonen eller blokk lengden. Registerlengden til koden og SNR har også innvirkning på denne tiden. Registerlengden er den som spiller minst inn. Forskjellen mellom koder med små registerlengder og koder med store registerlender er så små at jeg ikke har måttet ta hensyn til dette i mitt valg av hvilke simuleringer jeg skal gjøre.

Siden et lavt SNR vil føre til flere feil når signalet overføres over en kanal, er det også logisk at dekodingen vil ta lenger tid om man senker SNR. Når man bruker simuleringssprogrammet vil man hele tiden kunne se lengden på stabelen i stabelalgoritmen under dekodingen. Ved lavt SNR vil man se at stabelen blir betraktelig større under dekoding av hver blokk. Dette har satt en naturlig grense for hvor lavt SNR jeg har kunnet benytte. Grensen er mest avhengig av størrelsen på minnet på datamaskinen som benyttes. Når stabelen blir tilstrekkelig stor vil den fylle hele minnet og datamaskinen begynner da å bruke harddisken som virtuelt minne. Når dette skjer økes kjøretiden drastisk. Simulering under slike forhold er ikke gjennomførbart i praksis. Stort minne er derfor viktig om man ønsker å simulere under et dårlig SNR.

Simuleringene har vist at disse minneprobemene oppstår ved høyere BER for små signalkonstellasjoner enn for store.

4.1.2 Simulering for 8-PSK

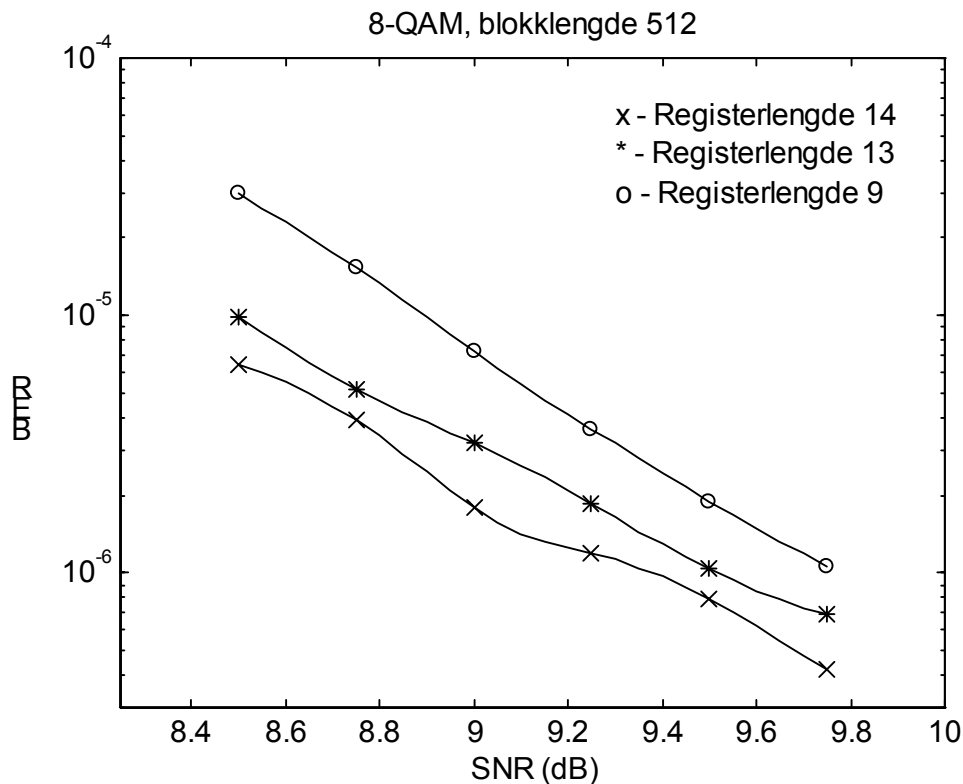
Jeg vil her presentere simuleringresultater for koder som er beskrevet av Wang og Costello i [8]. Disse kodene er funnet gjennom søkealgoritmer som skal generere gode trelliskoder til bruk ved sekvensiell dekoding. Disse kodene er utviklet for signalkonstellasjonene 8-PSK og 16-QAM. Kodene er designet for å tilfredsstille krav om høy feilkorreksjon og god beregningsytelse.

Simuleringer for de samme kodene er utført av Wang og Costello i [3]. Den eneste kjente forskjellen i disse simuleringene er at det i [3] er benyttet Fanoalgoritmen til dekoding, mens det i mine resultater er benyttet stabelalgoritmen. Figur 4.2 viser simuleringresultater for tre ulike koder med forskjellig registerlengde. Paritetssjekkpolynomene til kodene som er benyttet i simuleringen er:

Registerlengde 9: [1401,0166,0300]

Registerlengde 13: [33001,16226,01400]

Registerlengde 14: [57001,22266,35400]



Figur 4.2: Simuleringresultater for koder med registerlengde 9, 13 og 14.

Ved å sammenligne disse simuleringene med simuleringene gjort i [3], ser vi at resultatene ikke er nøyaktig de samme. Vi ser at resultater i mine simuleringer ligger

nesten 1 dB dårligere enn simuleringene i [3]. Til tross for noe avvikende resultater gir en registerlengde 9 kode med sekvensiell dekoding et noe bedre resultat enn optimal Viterbidekoding av en registerlengde 6 kode. Med andre ord kan man ved å øke registerlengden oppnå bedre resultater med sekvensielldekoding enn med Viterbidekoding. Vi kan også ut fra figur 4.2 se at feilsannsynligheten minsker ved å øke registerlengden.

For en 8-PSK konstellasjon trengs det et SNR=7,6 dB for å nå cutoffraten $R_0^* = 2$ biter/T [3].

4.1.3 Simulering for 16-QAM

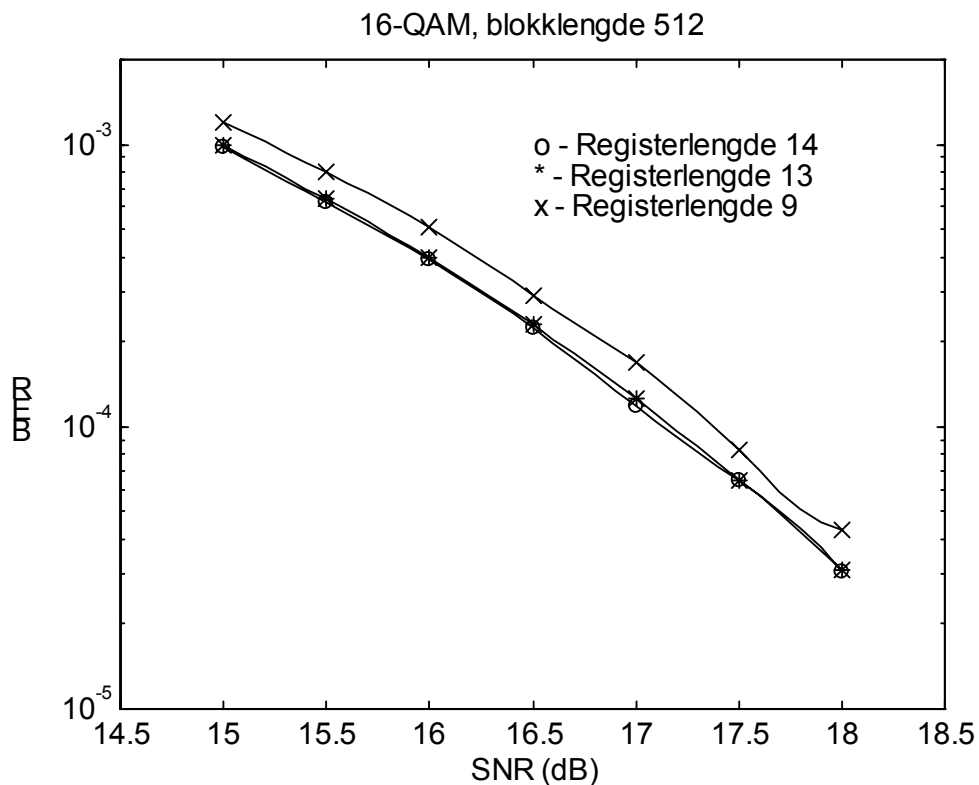
Jeg presenterer her simuleringresultater der 16-QAM konstellasjonen er benyttet. Som for 8-PSK konstellasjonen er også disse kodene hentet fra [8]. Paritetssjekkpolynomene i disse simuleringene er:

Registerlengde 9: [1401,0166,0300]

Registerlengde 13: [35153,06452,13500,16000]

Registerlengde 14: [57001,22266,35400]

Resultatene fra disse simuleringene er vist i figur 4.3.



Figur 4.3: Simuleringsresultater for koder med registerlengde 9, 13 og 14.

Figur 4.3 viser simuleringer med 16-QAM konstellasjon. Wang og Costello viser ingen simuleringresultater for denne konstellasjonen i [3]. Jeg har derfor ingen tidligere resultater å sammenligne med. Ut fra mine resultater ser det ut til at det blir mindre forskjell på BER ved å variere registerlengden enn ved 8-PSK. Til tross for en meget grundig gjennomgang av simuleringsprogrammet har jeg ikke funnet noen forklaring på dette.

For en 16-QAM konstellasjon trengs det et SNR=11,1 dB for å nå cutoffraten $R_0^* = 3$ biter/T [3].

4.2 Resultater fra tilfeldig genererte koder

I [15] vises det at en tilfeldig valgt kode vil gi gode resultater med høy sannsynlighet. Wang og Costello har i [9] funnet gode trelliskoder med høy registerlengde for 256-QAM ved hjelp av en algoritme som baserer seg på tilfeldig genererte koder. Da jeg ikke har funnet tidligere testede trelliskoder for 32-QAM og 64-QAM, har jeg valgt ut en mengde tilfeldig genererte koder og gjort simuleringer på disse for å finne ut hvilke av disse kodene som gir de beste resultatene. Algoritmen for å lage disse tilfeldig genererte kodene finner du i vedlegg 1.

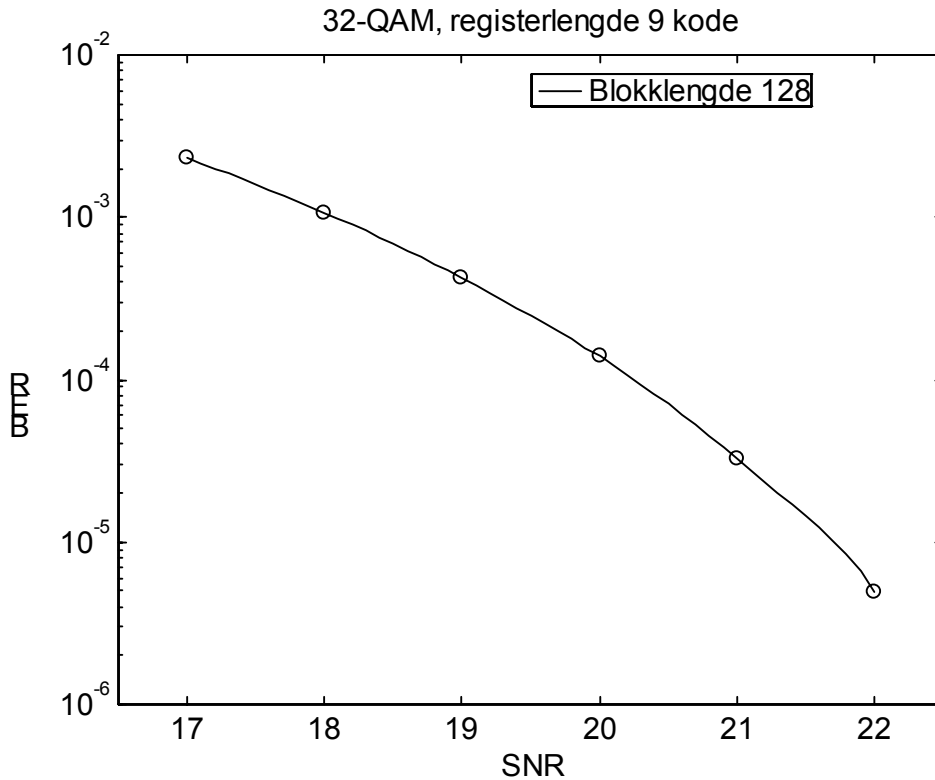
4.2.1 Simulering for 32-QAM

Her presenteres simuleringresultater der en 32-QAM signalkonstellasjon er benyttet. Tabell 4.1 viser hvilke koder som er testet ut og BER ved simulering av disse kodene. Jeg har brukt SNR=17 dB for å finne BER i tabell 4.1.

Kode	$g^{(0)}$	$g^{(1)}$	$g^{(2)}$	BER
1	1333	0542	0542	0.00225391
2	1213	0206	0400	0.00237793
3	1365	0236	0702	0.00225977
4	1377	0512	0512	0.00227246
5	1363	0170	0674	0.00233008
6	1415	0072	0376	0.00232129
7	1421	0002	0300	0.00224316
8	1677	0230	0044	0.00228027
9	1037	0346	0166	0.00230762
10	1333	0542	0502	0.00225391

Tabell 4.1: Simuleringresultater for 32-QAM konstellasjon med en registerlengde 9 kode.

Tabellen viser at BER ikke varierer mye mellom de forskjellige kodene. Vi ser at kode nr. 1 og kode nr. 10 gir det beste resultatet. Jeg har brukt kode nr. 1 med paritetssjekkpolynom [1333,0542,0542] og registerlengde 9 for å danne plottet i figur 4.5.



Figur 4.5: Simuleringsresultater for 32-QAM konstellasjon.

I tabell 4.1 ser vi at det er forholdsvis liten variasjon i BER mellom de ulike tilfeldig genererte kodene. Dette støtter opp om teorien om at man med høy sannsynlighet finner en god trelliskode ved å velge en tilfeldig. I følge denne teorien er det også svært sannsynlig at kodene som er benyttet i til simulering i figur 4.5 er gode koder for en 32-QAM konstellasjon.

For en 32-QAM konstellasjon trengs det et SNR=14,5 dB for å nå cutoffraten $R_0^* = 4$ biter/T [3].

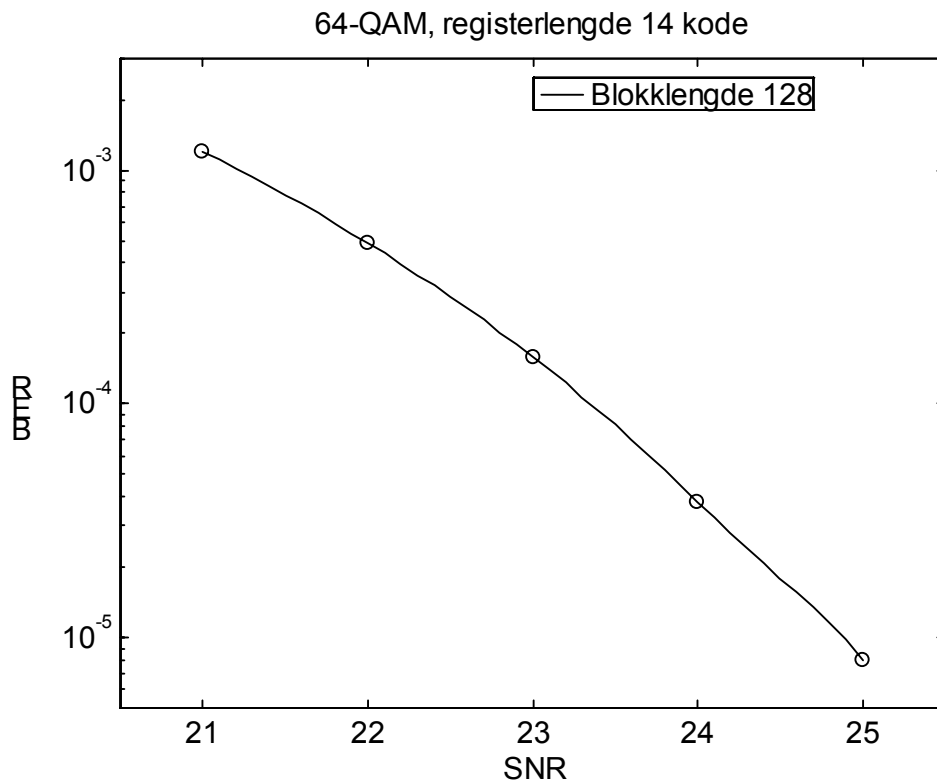
4.2.2 Simulering for 64-QAM

På samme måte som for 32-QAM konstellasjonen har jeg gjort simuleringer på tilfeldig valget koder for 64-QAM konstellasjon. Jeg har her valgt koder med registerlengde 14. Et SNR=20 dB er brukt for å generere BER. Resultatene kan du se i tabell 4.2.

Kode	$g^{(0)}$	$g^{(1)}$	$g^{(2)}$	$g^{(3)}$	BER
1	55561	24102	21172	65170	0.002575
2	35207	00044	06422	21722	0.00258484
3	53611	60416	22744	46526	0.00255688
4	16107	65066	04404	60236	0.00257016
5	33053	20732	25404	27752	0.00257953
6	13501	66610	60444	23770	0.00256938
7	14613	01564	26402	46130	0.00256125
8	31375	66404	42202	04460	0.00256234
9	14723	64616	25712	60416	0.00256625
10	14247	27372	05462	40372	0.00257484

Tabell 4.2: Simuleringsresultater for 64-QAM konstellasjon med en registerlengde 14 kode.

Også i dette tilfellet er det små forskjeller mellom BER for de ulike kodene. Best er kode nr. 3 med paritetssjekkpolynom [53611,60416,22744,46526]. Denne koden er derfor brukt til å lage plottet i figur 4.6.



Figur 4.6: Simuleringsresultater for 64-QAM konstellasjon.

Vi ser at man også for 64-QAM får en liten variasjon i BER mellom de ulike tilfeldig genererte kodene. Dette bygger ytterligere opp om teorien om at man med stor sannsynlighet finner en god trelliskode ved å velge en tilfeldig. Mye tyder derfor på at også koden som er brukt til å generere plottet i figur 4.6 er en god kode for 64-QAM. For en 64-QAM konstellasjon trengs et SNR=17,8 for å nå cutoffraten $R_0^* = 5$ biter/T [3].

Kapittel 5

Oppsummering

Her vil jeg gi en kort oppsummering og beskrive eventuelle videre arbeider som kan gjøres rundt innholdet i denne oppgaven.

5.1 Sammendrag

I denne hovedfagsoppgaven har jeg presentert simuleringresultater for skevensiell dekodning av forskjellige trelliskoder. For å få en grunnleggende forståelse av hva oppgaven skulle handle om, ble det hele satt inn i en historisk sammenheng. Det var så nødvendig å forklare hvordan digital kommunikasjon foregår i dag og gi eksempler på noen av dagens systemer. Det ble forklart hvordan støy påvirker informasjon som overføres over en AWGN-kanal og hvordan feilkorrigerende koder kunne sikre pålitelig kommunikasjon.

Videre ble det gitt en innføring i signalkonstellasjoner med hovedvekt på QAM konstellasjoner. Punkter i en signalkonstellasjon påvirkes av støy når de overføres over en kanal. Det ble i kapittel 2 beskrevet hvordan støyen påvirker et signalpunkt som overføres over en kanal og hvor stor energi som er nødvendig for å sende dette signalpunktet. Det ble også gitt uttrykk for cutoffraten som forteller hvor stor overføringsrate som er mulig for å opprettholde pålitelig kommunikasjon og fornuftig kompleksitet.

Koding og dekodning ble forklart i kapittel 3. Konvolusjonskoder og sekvensiell dekodning ble satt i fokus og den sekvensielle dekodingsalgoritmen stabelalgoritmen ble gått gjennom i detalj. Det blir også vist hvordan man kan nummerere punktene i en signalkonstellasjon for å oppnå størst mulig minimumsavstand mellom signalpunktene. Halen i en kodesekvens kan føre til svekket feilkorreksjon i de siste bitene i en kodesekvens. En forklaring på hvordan man kan unngå dette problemet blir gitt i kapittel 3.

I kapittel 4 presenteres egne simuleringresultater for trelliskoder med forskjellig registerlengde. Her vises det hvordan man ved å øke registerlengden til koden kan oppnå en lavere BER. Det er gjort simuleringer med 8-PSK, 16-QAM, 32-QAM og 64-QAM. For 8-PSK og 16-QAM ble det benyttet kjente koder fra [8]. Simuleringene av 8-PSK sammenlignes med tilsvarende simuleringer i [3]. Resultatene viste at jeg fikk en noe høyere BER enn i [3]. For 32-QAM og 64-QAM fant jeg noen tilfeldige koder og sammenlignet resultatene fra simuleringer med disse. Resultatet viste at jeg med stor sannsynlighet fant gode koder for simulering med disse signalkonstellasjonene.

5.2 Videre arbeid

Simuleringer er en tidkrevende prosess og med mer tilgjengelig tid kan man produsere flere resultater. Om man har tilgang på stor datakraft og mye minne kunne det vært av

interesse og kjøre simuleringer for lavere SNR enn det som er gjort i denne oppgaven. På grunn av tidsbegrensningen har jeg ikke fått testet ut så mange tilfeldige koder som jeg hadde ønsket. Derfor hadde det også vært interessant å finne bedre koder enn de som jeg har funnet. Man kan også gjøre tilsvarende søk for større signalkonstellasjoner. I første rekke 128-QAM. Ved å utvide simulatoren så den takler større signalkonstellasjoner kan man også gjøre tilfeldige søk etter koder for disse. I [9] er det gjort simuleringer med gode trelliskoder for 256-QAM. Om 256-QAM implementeres i simulatoren kan man verifisere disse resultatene.

For å gi simulatoren større omfang kan man implementere Fano-algoritmen [1] til dekodning. Man kan da gjøre nye simuleringer og sammenligne resultatene med resultatene i denne oppgaven. Det er også av interesse å sammenligne kjøretider på de to algoritmene.

Adaptiv koding og modulasjon kan også innføres i simulatoren [17]. Man kunne da sammenligne resultater fra dette adaptive systemet med resultatene i denne oppgaven og finne ut hvor mye man kan tjene på å gjøre et system adaptivt. Resultatene kunne også sammenlignes med teorier om ytelse for slike systemer.

For å øke simuleringshastigheten og minske minnebehovet kan man sette en grense for hvor stor stabelen i stabelalgoritmen kan være. For eksempel sette grensen slik at simuleringene aldri vil starte å bruke virtuelt minne. Ved å gjøre dette vil det i enkelte tilfeller bli nødvendig å slette elementer fra stabelen. Hvis man hele tiden ved overflyt sletter de elementene som ligger nederst i stabelen er det lite sannsynlig at man sletter stier som vil ende opp som den øverste stien da de nederste elementene i stabelen alltid inneholder den minst sannsynlige stien. Her vil det også være enkelt å legge inn en sjekk som forteller deg om det har oppstått en situasjon der et element som skulle nådd toppen i stabelen er blitt slettet.

Brukervennlighet og visuelle presentasjoner er ikke vektlagt i min utgave av simulatoren. Her er det store forbedringsmuligheter. Et grafisk brukergrensesnitt kan implementeres og resultater fra simuleringene kan plottes direkte inn i et koordinatsystem. Ulempen med dette kan være at kjøretiden på simuleringene kan øke.

Referanseliste

- [1] S. Lin og D. J. Costello, Jr., *Error control coding: Fundamentals and applications*, Prentice Hall, 1983.
- [2] G. Underboeck, "Channel coding with multilevel/phase signals" IEEE Trans. Inform. Theory, vol. IT-28, pp. 55-67, Jan. 1982.
- [3] F. Q. Wang og D. J. Costello, Jr. "Erasurefree sequential decoding of trellis codes", IEEE Trans. Inform. Theory, vol. 40, pp. 1803-1817, nov. 1994.
- [4] S. S. Pietrobon og D. J. Costello, Jr. "Trellis Coding with Multidimensional QAM Signal Sets" IEEE Trans. Inform. Theory, vol. 39, NO.2, March 1993.
- [5] G. D. Forney, Jr. "Efficient Modulation for Band-Limited Channels", IEEE Journals On Selected Areas In Communication, vol. SAC-2, NO. 5 September 1984.
- [6] A. Burr, *Modulation and Coding for Wireless Communication* Prentice Hall, 2001.
- [7] R. Hill, *A First Course In Coding Theory* Oxford University Press, 1986.
- [8] F. Q. Wang og D. J. Costello, Jr. "Robusty Good Trellis Codes", IEEE Trans. Inform. Theory, vol. 44, pp. 791-798, july 1996.
- [9] F. Q. Wang og D. J. Costello, Jr. "Sequential Decoding Of Trellis Codes at High Spectral Efficiencies", IEEE Trans. Inform. Theory, vol. 43, pp. 2013-2019, nov. 1997.
- [10] R. Johannesson, "The error probability of general trellis codes with applications to sequential decoding, "IEEE Trans. Inform. Theory, vol. IT-23 pp. 609-611, Sept. 1977.
- [11] Connected Earth, URL:<http://www.connected-earth.com/>
- [12] C.E. Shannon "A Mathematical Theory of Communication," Bell Syst. Tech. J., 27, pp. 379-423 (Part I), 623-656 (Part II), July 1948.
- [13] M. C. Jeruchim, P. Balban, K. S. Shanmugan, *Simulation Of Communication Systems* Plenum Press, New York 1992.
- [14] R. Johannesson, Kamil Sh. Zigangirov, *Fundamentals Of Convolutional Coding*, IEEE Press 1999.
- [15] R. G. Gallager, *Information Theory and Reliable Communication*. New York: Wiley, 1968.
- [16] M. S. Gast, *802.11 Wireless Networks The Definitive Guide*, O'Reilly & Associates; 1st edition, April 2002.
- [17] K. J. Hole, H. Holm og G. E. Øien, "Adaptive Multidimensional Coded Modulation Over Flat Fading Channels", IEEE Journals On Selected Areas In Communication, vol. 18, No. 7 July 2000.

Appendiks

Oversikt over programmet

Her gis en innføring i hvordan simuleringsprogrammet fungerer og hvordan man kan endre parametere etter eget ønske.

Oppstart av programmet

Ved oppstart av programmet kan man velge om man ønsker å taste inn parametrene selv eller om man ønsker å kjøre en ferdig definert simulering.

Valg av parametere

Hvis man velger å taste inn parametrene selv, vil man kunne velge mellom en rekke ferdig definerte valg. Parametrene som skal velges er: kode, blokk lengde, type konstellasjon, SNR og antall kodeblokker som skal dekodes. Når parametrene er valgt, utfører programmet en simulering ut fra valgene. Når simuleringen er ferdig vil du få resultatet skrevet til skjerm. Resultatene er: antall dekodingsfeil, BER og gjennomsnittlig antall biter dekodet pr. sekund. Under simuleringen vil man hele tiden se hvor mange biter som er dekodet og hvor mange biter som skal dekodes totalt. Etter at simuleringen har startet blir det også beregnet et tidsestimat for simuleringen. En klokke vil hele tiden vise hvor lang tid som er brukt. Simuleringene viser også hele tiden hvor lang stabelen i stabelalgoritmen er.

Valg 1:

Når man velger å taste inn parameterene selv, er det første valget valg av kode. Valgene er da:

1. Constraintlength = 16, H=[270463,037772,136264,015274]
2. Constraintlength = 17, H=[747541,370212,012210,261332]
3. Constraintlength = 18, H=[1227115,0516070,0732646,0742142]
4. Constraintlength = 19, H=[2236213,0742212,1005224,1455130]
5. Constraintlength = 13, H=[35153,06452,13500,16000]
6. Constraintlength = 13, H=[33001,16226,01400]
7. Constraintlength = 9, H=[1401,0166,0300]
8. Constraintlength = 9, H=[1055,0502,0400]
9. Constraintlength = 11, H=[4047,2302,0400]
10. Constraintlength = 11, H=[4001,3352,1500]

Kodene er hentet fra Wang og Costellos arbeider i [8,9]. Kodene som er nevnt over er de eneste kodene som er definert i programmet.

Valg 2:

Her skal man velge blokk lengden. Man har her 4 valg:

1. 64
2. 128
3. 256
4. 512

Dette er de eneste valgene som er definert i programmet.

Valg 3:

Her skal man velge type konstellasjon. Alle konstellasjonene er definert i QAM.h fila. QAM konstellasjonene har fått en mapping som sikrer en god Euclidsk avstand. Hvordan dette er gjort står nærmere forklart i kapittelet om signalkonstellasjoner. I simuleringene kan man velge mellom følgende 5 konstellasjoner.

1. 8-PSK
2. 16-QAM
3. 32-QAM
4. 64-QAM
5. 128-QAM

Valg 4:

Valg av SNR under simuleringen. Her kan man taste inn signal til støy forholdet i dB som et desimaltall. Tallet må ligge mellom 1 og 30.

Valg 5:

Her velger man antall kodeblokker som skal kodes og dekodes i simuleringen. Jo flere blokker man velger desto mer nøyaktig vil resultatet bli.

Når disse valgene er gjort, vil simuleringen starte. Etter at simuleringen har avsluttet vil resultatet skrives til fila CodeRes.txt. Resultatet blir lagt til det innholdet som eventuelt allerede er i fila. På denne måten vil man kunne se resultatet av alle simuleringene som er kjørt. Man kan slette fila om man ikke ønsker å ta vare på resultatene. En ny fil vil da bli laget ved neste simulering.

Fila inneholder alle parametrene som er definert for simuleringen. Formatet på fila er:

Kode|Blokk lengde|Konstellasjon|SNR|Antall biter dekodet pr. sekund|Antall kodeblokker som skal kodes og dekodes|BER

Koden er representert som et tall fra 1-10 som tilsvarer valget av kode i valg 1. Blokk lengden skrives direkte til fil. Konstellasjonen forteller hvor mange punkter den inneholder. SNR gir et desimaltall for signal-til-støy forholdet man har valgt. Antall biter dekodet pr. sekund skrives som et desimaltall. Antall kodeblokker som skal kodes og dekodes skrives også til filen. Til slutt legges bitfeilraten til som et desimaltall. Eksemplet under viser CodeRes.txt etter flere simuleringer.

Eksempel på CodeRes.txt

7	512	8	8.25	5746.13	10000	5.79102e-05
7	512	8	8.75	6520.28	10000	1.05469e-05
7	512	8	9.25	7066.39	10000	2.53906e-06
7	512	8	9.5	6751.51	10000	1.66016e-06
8	512	8	8	3054.12	10000	9.02344e-05
8	512	8	8.5	474.648	10000	3.38867e-05
8	512	8	9	4506.96	10000	4.98047e-06
9	512	8	8	2554.56	10000	2.87109e-05
7	512	8	9	6565.35	50000	7.5e-06
7	512	8	8.5	7101	10000	3.44727e-05
7	512	8	8.25	6094.97	10000	5.54688e-05
7	512	8	9	5290.54	250000	7.23047e-06
7	512	8	8.5	7037.75	250000	2.95898e-05

Under simuleringen opprettes også filen Errors.txt. Denne filen inneholder samtlige dekodingsfeil som har oppstått under simuleringen. Filen viser hvilke bit i hvilke kodeblokk som er dekodet feil.

Eksempel på Errors.txt

Error at bit nr 1023 in block nr 3911
Error at bit nr 292 in block nr 6968
Error at bit nr 1018 in block nr 9289
Error at bit nr 1019 in block nr 9289
Error at bit nr 586 in block nr 12600
Error at bit nr 646 in block nr 12600
Error at bit nr 1018 in block nr 24672

Under simuleringen opprettes også Input.txt og Result.txt. Disse filene brukes av simulatoren til å lagre inndata og resultatet i simuleringen. Input.txt inneholder den slumpfallsgenererte bitsekvensen som skal kodes og Result.txt inneholder det dekodete resultatet. Filene blir etter dekodingen sammenlignet og eventuelle uoverensstemmelser blir registrert som feil i Errors.txt og lagt til i beregningen av BER.

Benytte ferdigdefinerte simuleringer

Hvis man velger å ikke taste inn parametrene selv vil en ferdig definert simulering kjøres. For å endre på denne ferdigdefinerte simuleringen må man inn i C++ koden.

Simuleringsprogrammet inneholder en funksjon som heter *simulate()*. Funksjonen tar inn alle parametrene og setter i gang simuleringen. Funksjonen tar inn 6 parametre på formen: *simulate(sjekkbit, kode, blokk lengde, konstellasjon, støy, #blokker)*.

- Sjekkbit forteller om man skal taste inn parametrene selv eller om man skal benytte parametrene som er ferdig definert. Sjekkbit er 1 for egen inntasting, 0 hvis ferdigdefinert.
- Resten av parametrene tilsvarer de samme valgene som man gjør ved manuell inntasting.

Eksempel

simulate(0,7,4,1,8,5000) :

Kjører en simulering med ferdigdefinerte parametre, kode nr 7 (H=[1401,0166,0300]), blokk lengde nr. 4 (512 bit), konstellasjon nr. 1(8-PSK), SNR lik 8 og man dekker 5000 blokker.

simulate(0,1,3,4,9.34,2350) :

Kjører en simulering med ferdigdefinerte parametre, kode nr 1 (H=[270463,037772,136264,015274]), blokk lengde nr. 3 (256 bit), konstellasjon nr. 4(64-QAM), SNR lik 9.34 og man dekker 2350 blokker.

Simulate() funksjonen startes fra *main()* funksjonen i programmet.

Endring av parametre i koden

Med litt arbeid kan man ganske enkelt legge til andre parametre til simuleringene. Med dette menes for eksempel å legge til flere typer koder, andre signalkonstellasjoner, andre blokk lengder eller endre intervallet for SNR. Alle valg som tas i programmet er satt opp i funksjonen *get_parameters()*. Denne funksjonen har akkurat de samme inndataene som *simulate()*. Altså får vi

get_parameters(sjekkbit, kode, blokk lengde, konstellasjon, støy, #blokker). Valgene kode, blokk lengde og konstellasjon er listet opp i hver sin *switch()* sekvens der hver *case* definerer hvert enkelt valg. Her kan man enkelt legge til flere *case* enheter.

Eksempel

Oprinnelig:

```
while (1>valg<10)
{
switch(valg)
{
case 1 :
    {blokk lengde=64;break;}
case 2 :
    {blokk lengde=128;break;}
```

```

        .
        .
        .
    case 10 :
        {blokk lengde=1024;break;}
    }
}

```

Etter endring:

```

while (1>valg<11)
{
switch(valg)
{
case 1 :
    {blokk lengde=64;break;}
case 2 :
    {blokk lengde=128;break;}
    .
    .
    .
case 11 :
    {blokk lengde=2048;break;}
}
}
}

```

Eksempelet viser hvordan man enkelt kan legge til et valg nr. 11 med blokk lengde 2048. Det eneste som skal endres er at *while* løkken som må inneholde riktig antall valg og *switch* enheten der dette valget må legges til.

På tilsvarende måte kan man endre alle de andre valgene i programmet også. Hver *case* inneholder i for valg av kode noe mer programkode enn i dette eksemplet, men ved å studere disse valgene vil man forstå hvordan man skal lage nye valg.

Hvis man ønsker å legge til en ny signalkonstellasjon må man i tillegg til de nevnte endringene også legge den nye konstellasjonen inn i fila QAM.h som definerer de ulike signalkonstellasjonene. En konstellasjon er definert som i eksemplet nedenfor.

Eksempel

```

double QAM8[8][3]= {
    {0,0.923879532511,-0.382683432365},
    {1,-0.923879532511,0.382683432365},
    {2,0.382683432365,0.923879532511},
    {3,-0.382683432365,-0.923879532511},
    {4,0.923879532511,0.382683432365},
    {5,-0.923879532511,-0.382683432365},
    {6,-0.382683432365,0.923879532511},
    {7,0.382683432365,-0.923879532511},
};

```

Konstellasjonen defineres altså som en 2-dimensjonal array Der hver rad inneholder et signalpunkt. Hvert signalpunkt har 3 parametere: punktnummer, x-koordinat og y-koordinat. PS! Her er det viktig at konstellasjonene mappes til de mest hensiktsmessige punktnummerene som vist i kapittel 3 for å oppnå best resultat. Det er kun 8-PSK konstellasjonen som representeres som desimaltall. De andre konstellasjonene representeres ved heltall.

SNR og antall runder er definert i en *if* setning som gir et intervall for dette valget. Her kan man forandre dette intervallet etter eget ønske.

Vedlegg 1

Generering av tilfeldige koder

Dette er et program som genererer tilfeldige trelliskoder. For å endre registerlengden på kodene må man inn i programkoden. Antbits variabelen definerer størrelsen på koden. Legg merke til at `antbits=registerlengde+1`.

```
#include <fstream.h>
#include <iomanip.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h> //For random funksjon
#include <time.h> //For randomize funksjon

int main()
{
    //Her må man manuelt inn i koden for å endre parameterene til kodene som skal genereres
    int antbits=15;//(constraintlength+1)
    int antpoly=4;
    int firstpol=antbits%3;
    if (firstpol==0){firstpol=3;}
    int antcodes=20;
    int out=0;
    double MAX=RAND_MAX;
    int randnr;
    int* bitarray;
    bitarray=new int[antbits];
    cout<<"Constraintlength "<<antbits-1<<" kode med "<<antpoly<<" polynomer.\n\n";

    for (int y=0;y<antcodes;y++)
    {
        cout<<"\nKode "<<(y+1);
        cout<<" [";
        for (int x=0;x<antpoly;x++)
        {
            for (int i=0;i<antbits;i++)
            {
                randnr=rand();
                if (randnr<(RAND_MAX/2))
                    {bitarray[i]=0;}
                else {bitarray[i]=1;}
            }
            out=0;
            for (int i=0;i<firstpol;i++)
            {
                if((x==0)&&(i==0)){bitarray[i]=1;}
                if((x!=0)&&(i==0)){bitarray[i]=0;}
                out=bitarray[i]*pow(2,i)+out;
            }
            cout<<out;
            for (int i=0;i<(((antbits-1)/3));i++)
```

```

    {
    out=0;
    int k=0;
    for (int j=2;j>=0;j--)
    {
        if((x==0)&&((i*3)+j+firstpol)==antbits-1){bitarray[(i*3)+j+firstpol]=1;}
        if((x!=0)&&((i*3)+j+firstpol)==antbits-1){bitarray[(i*3)+j+firstpol]=0;}
        out=out+bitarray[(i*3)+j+firstpol]*pow(2,k);
        k++;
    }
    cout<<out;
    }
    if (x<(antpoly-1)) {cout<<" ";}
    //cout<<"\n";
    for (int i=0;i<antbits;i++)
    {
        //cout<<bitarray[i];
    }
    //cout<<"\n\n";
    //getch();
    }

    cout<<"]";}
    getch();
    }
    }

```

Vedlegg 2

Simuleringsprogrammet

Her ligger kildekoden til selve simulatorprogrammet. Se appendiks for en kort innføring i bruken av dette programmet.

```
#include <conio.h>
#include <fstream.h>
#include <iomanip.h>
#include <math.h>
#include <stdlib.h> //For random funksjon
#include <time.h> //For randomize funksjon
#include <dos.h> //Brukes til clock
#include <QAM.h> //Her ligger QAM konstellasjonene, 16-, 32-, 64- og 128-QAM

class trelliskode
{
private:
    int constraintlength; //constraint length til koden
    int anth; //antall H'er
    int QAMstr; //Størrelsen på konstellasjonen
    float SNR; //Signal til støy forhold
    long loops; //Amtall runder programmet skal kjøre
    double avr_energi; //gjennomsnittlig energi til en konstellasjon
    int code;
    double ro;
    double QAM[128][3]; //Brukes til å lagre de ulike konstellasjonene. 128-QAM er max
    int ant_dummy_bits; //Antall bits som ikke involveres i kodingen, men som innvirker på
    //output
    //D.v.s. går rett gjennom encoderen
    int block_length; //Setter lengden på en blokk med symboler som skal bli
    //kodet.Når en blokk er kodet,
    //starter man med neste blokk fra state 0 igjen.

public:
    void simulate(int,int,int,int,float,long);
    int* get_parameters(int,int,int,int,float,long);//Lar brukeren taste inn data, returnerer koden
    //input: manuell/ikke manuell inntasting, kode, blokklengthe,konstellasjon, støy, antall runder
    double get_avrage_energy(int);
    int* get_parity_matrix();
    int get_ant_bits();
    int get_QAM();
    void show_parity_matrix(int*);
    int get_qamstr(); //Tar inn størrelsen på konstellasjonen
    void make_encoder(int*);//Lager encoderen
    long get_next_state(int*,long,int,int);//int* matrise med H'ene i.
    //long state,int input. Returnerer ny state, output og input
    long** make_state_list(int*); //input paritetssjekk matrisene, returnerer state_list
    long** make_out_list(int*); //input paritetssjekk matrisene, returnerer out_list
    long get_outval(long,long);
    void show_state_list(long**,long,long); //viser state listen
    double find_metric(double,double,double,double);//finne metrikken mellom to punkt
    int* generate_bits(int);//lager bits til input i funksjonen
    double* make_code_block(long**,int*);//Tar inn peker til (state_list og pathlist)(int**),
    //lengden i x retning og lengden i y retning regnes ut i funksjonen
    void stack_algorithm(long**,double*); //listen med fra-til state, symboler som skal dekodes;
```

```

        long real_input(long,int,long);//Gjør om input til input uten dummybits
        long dummmmy_add(long,int,long);//Regner ut tilleggs output p.g.a. dummybits
        double* add_noise(double*);
        double* add_manual_noise(double*);
        int result(int,long); //input lengden av kodeblokken og hvilke kode blokk som testes,output antall
                                //errors på kodeblokken
        int run_sim(long**,int); //returnerer antall errors. input statelist, outlist
    };

//Strukten er et element i stacken som brukes i stackalgoritmen
//path bør gjøres om til en array av bits slik at den fungerer for større blokker.
struct link
    {
        int* path; //Inneholder stien i dekodingsreet på titallsform
        int path_length;
        double metric;
        int state;
        link* next;
        link* previous;
    };

//stacken som brukes i stackalgoritmen. består av link elemener
class stack
    {
private:
        link* first;
public:
        long stacklength;
        stack()
            {first=NULL;stacklength=0;}
        ~stack()//Sletter hele stacken for å spare minne
            {
                link* first;
                link* midl;
                first=getfirst();
                while (first!=NULL)
                    {
                        midl=first->next;
                        delete[] first->path;
                        delete first;
                        first=midl;
                    }
                stacklength=0;
                delete midl;
            }
        void additem(int*,int,double,int);//legger til element i stacken. Input er input til koderen, lengden av input +
                                //metrikken + state
        void removeitem();
        void showstack(); //Viser stacken.
        link* getfirst();
    };

void main()
    {
        trelliskode k2;
        char answer=0;
        cout<<"Vil du taste inn parameterene selv ? (y/n) ";
        while ((answer!='y')&&(answer!='n'))
            {
                answer=getche();
                if ((answer!='y')&&(answer!='n')){cout<<"\nTast y eller n: ";}
            }
    }

```

```

    }
    clrscr();
    if (answer=='y')
        {k2.simulate(1,0,0,0,0);}
    else
        {
            k2.simulate(0,7,4,1,8,100);
            k2.simulate(0,7,4,1,8.25,100);
        }
    getch();
}

//Regner ut toerlogaritmen til et tall
double log2(double x)
{
    double answer=(log10(x)/log10(2));
    return answer;
}

//Returnerer antall bits som skal kodes i en blokk
int trelliskode::get_ant_bits()
{
    int antbits=(block_length*((anth-1)+ant_dummy_bits));
    cout<<"\n\nBlokk lengden : "<<block_length;
    cout<<"\nKonstellasjonstype : "<<QAMstr<<"-QAM";
    cout<<"\nAntall Dummybits : "<<ant_dummy_bits;
    cout<<"\nAntall bits i hver kodeblokk : "<<antbits;
    cout<<"\nSNR : "<<SNR<<" dB"<<" ==> ro="<<ro<<" , gj.energi="<<avr_energi;//getch();
    return antbits;
}

double trelliskode::get_avrage_energy(int str)
{
    double midl=0;
    for (int i=0;i<str;i++)
        {
            midl=midl+(pow(QAM[i][1],2))+pow(QAM[i][2],2));
        }
    avr_energi=(midl/str);
    ro=(sqrt(avr_energi/(2*pow(10,(SNR/10)))));
    return avr_energi;
}

void trelliskode::simulate(int f,int c1,int c2,int c3,float n,long r)
{
    ofstream outfile1("Errors.txt"); //Kun for å slette innholdet hvis filen eksisterer fra før
    int* m=get_parameters(f,c1,c2,c3,n,r);
    get_QAM();
    get_avrage_energy(QAMstr);
    show_parity_matrix(m);
    int numberofbits=get_ant_bits();
    long** slist;
    slist=make_state_list(m);
    run_sim(slist,numberofbits);
}

//Hvis "fill"=1 taster man inn parameterene, hvis ikke må de sendes med i funksjonen
int* trelliskode::get_parameters(int fill,int choice1,int choice2,int choice3,float noice,long rounds)
{

```

```

SNR=noise;
loops=rounds;
code=choice1;
int* matrix;
if (fill==1)
{
choice1=0;
cout<<"Velg Kode:\n1. Constraintlength = 16, H=[270463,037772,136264,015274]\n"
<<"2. Constraintlength = 17, H=[747541,370212,012210,261332]\n"
<<"3. Constraintlength = 18, H=[1227115,0516070,0732646,0742142]\n"
<<"4. Constraintlength = 19, H=[2236213,0742212,1005224,1455130]\n"
<<"5. Constraintlength = 13, H=[35153,06452,13500,16000]\n"
<<"6. Constraintlength = 13, H=[33001,16226,01400]\n"
<<"7. Constraintlength = 9, H=[1401,0166,0300]\n"
<<"8. Constraintlength = 9, H=[1055,0502,0400]\n"
<<"9. Constraintlength = 11, H=[4047,2302,0400]\n"
<<"10. Constraintlength = 11, H=[4001,3352,1500]\n"
<<"11. Constraintlength = 10, H=[2201,0666,1400]\n"
<<"12. Constraintlength = 12, H=[10517,06462,04400]\n"
<<"13. Constraintlength = 14, H=[57001,22266,35400]\n"
<<"14. Constraintlength = 15, H=[104001,045666,035400]\n"
<<"Tast nr : ";
while ((choice1<1)||((choice1>14)))
{
cin>>choice1;
code=choice1;
if ((choice1<1)||((choice1>14))) {cout<<"\nFeil valg av kode\nTast nr : ";}
}
}
switch(choice1)
{
case 1 :
{
anth=4;
constraintlength=16;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på
//oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[24]={2,7,0,4,6,3,0,3,7,7,2,1,3,6,2,6,4,0,1,5,2,7,4};
for(int i=0;i<(anto*anth);i++)
{matrix[i]=midl[i];}
break;
}
case 2 :
{
anth=4;
constraintlength=17;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på
//oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[24]={7,4,7,5,4,1,3,7,0,2,1,2,0,1,2,2,1,0,2,6,1,3,3,2};
for(int i=0;i<(anto*anth);i++)
{matrix[i]=midl[i];}
break;
}
case 3 :
{
anth=4;

```

```

constraintlength=18;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[28]={1,2,2,7,1,1,5,0,5,1,6,0,7,0,0,7,3,2,6,4,6,0,7,4,2,1,4,2};
for(int i=0;i<(anto*anth);i++)
    {matrix[i]=midl[i];}
break;
}
case 4 :
{
anth=4;
constraintlength=19;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[28]={2,2,3,6,2,1,3,0,7,4,2,2,1,2,1,0,0,5,2,2,4,1,4,5,5,1,3,0};
for(int i=0;i<(anto*anth);i++)
    {matrix[i]=midl[i];}
break;
}
case 5 :
{
anth=4;
constraintlength=13;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[20]={3,3,0,0,1,1,6,2,2,6,0,1,4,0,0};
for(int i=0;i<(anto*anth);i++)
    {matrix[i]=midl[i];}
break;
}
case 6 :
{
anth=3;
constraintlength=13;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[20]={3,5,1,5,3,0,6,4,5,2,1,3,5,0,0,1,6,0,0,0};
for(int i=0;i<(anto*anth);i++)
    {matrix[i]=midl[i];}
break;
}
case 7 :
{
anth=3;
constraintlength=9;
int anto;
if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
else anto=(constraintlength+1)/3;
matrix = new int[anto*anth];
int midl[20]={1,4,0,1,0,1,6,6,0,3,0,0};
for(int i=0;i<(anto*anth);i++)
    {matrix[i]=midl[i];}
break;
}
}

```

```

case 8 :
{
    anth=3;
    constraintlength=9;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={1,0,5,5,0,5,0,2,0,4,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
case 9 :
{
    anth=3;
    constraintlength=11;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={4,0,4,7,2,3,0,2,0,4,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
case 10 :
{
    anth=3;
    constraintlength=11;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={4,0,0,1,3,3,5,2,1,5,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
case 11 :
{
    anth=3;
    constraintlength=1;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={2,2,0,1,0,6,6,6,1,4,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
case 12 :
{
    anth=3;
    constraintlength=12;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={1,0,5,1,7,0,6,4,6,2,0,4,4,0,0};
    for(int i=0;i<(anto*anth);i++)

```

```

        {matrix[i]=midl[i];}
    }
    break;
}
case 13 :
{
    anth=3;
    constraintlength=14;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={5,7,0,0,1,2,2,2,6,6,3,5,4,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
case 14 :
{
    anth=3;
    constraintlength=15;
    int anto;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[20]={1,0,4,0,0,1,0,4,5,6,6,6,0,3,5,4,0,0};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    break;
}
default :;
}
}

if (fill==1)
{
    choice2=0;
    cout<<"\nVelg blokklengthe:\n1. 64\n"
        <<"2. 128\n"
        <<"3. 256\n"
        <<"4. 512\n"
        <<"Tast nr : ";
    while ((choice2!=1)&&(choice2!=2)&&(choice2!=3)&&(choice2!=4))
    {
        cin>>choice2;
        if ((choice2!=1)&&(choice2!=2)&&(choice2!=3)&&(choice2!=4))
            {cout<<"\nFeil valg av blokklengthe\nTast nr : ";}
    }
}

switch(choice2)
{
    case 1 :
    {
        block_length=64;
        break;
    }
    case 2 :
    {
        block_length=128;
        break;
    }
    case 3 :
    {
        block_length=256;

```

```

        break;
    }
    case 4 :
    {
        block_length=512;
        break;
    }
    default ;;
}

if (fill==1)
{
    choice3=0;
    cout<<"\nVelg QAM konstellasjon:\n1. 8-PSK\n"
        <<"2. 16-QAM\n"
        <<"3. 32-QAM\n"
        <<"4. 64-QAM\n"
        <<"5. 128-QAM\n"
        <<"Tast nr : ";
    while ((choice3!=1)&&(choice3!=2)&&(choice3!=3)&&(choice3!=4)&&(choice3!=5))
    {
        cin>>choice3;
        if ((choice3!=1)&&(choice3!=2)&&(choice3!=3)&&(choice3!=4)&&(choice3!=5))
            {cout<<"\nFeil valg av QAM\nTast nr : ";}
    }
}

switch(choice3)
{
    case 1 :
    {
        QAMstr=8;
        break;
    }
    case 2 :
    {
        QAMstr=16;
        break;
    }
    case 3 :
    {
        QAMstr=32;
        break;
    }
    case 4 :
    {
        QAMstr=64;
        break;
    }
    case 5 :
    {
        QAMstr=128;
        break;
    }
    default ;;
}

if (fill==1)
{
    SNR=0;
    cout<<"\nVelg SNR f.eks mellom 1-30:\nTast tallet : ";
    while ((SNR<=0)||((SNR>50))
        {

```

```

        cin>>SNR;
        if ((SNR<=0)||((SNR>50))) {cout<<"\nFeil valg av SNR\nTast tallet : ";}
    }
}

if (fill==1)
{
    loops=0;
    cout<<"\nVelg antall kodeblokker som skal dekodes (antall runder):\nTast tallet : ";
    while ((loops<=0)||((loops>1000000000)))
    {
        cin>>loops;
        if ((loops<=0)||((loops>1000000000)))
            {cout<<"\nFeil valg av antall kodeblokker\nTast tallet : ";}
    }
}

ro=(sqrt(1/(2*pow(10,(SNR/10)))));
ant_dummy_bits=-1;
int i=0;
if(QAMstr==pow(2,anth)){ant_dummy_bits=0;}
else if (QAMstr<pow(2,anth)){cout<<"Konstellasjonen er for liten for denne koden";getch();}
else
{
    while ((QAMstr!=pow(2,anth))&&(i<10))
    {
        i++;
        if (QAMstr==pow(2,(anth+i)))
            {ant_dummy_bits=i;}
    }
}
if (ant_dummy_bits==-1){cout<<"Error, kan ikke definere ant_dummy_bits";getch();}
clrscr();
return matrix;
delete[] matrix;
}

int* trelliskode::get_parity_matrix()
{
    block_length=128;
    QAMstr=32; //Settes til 16,32,64 eller 128
    SNR=9; //2 vil gi feil
    ro=(sqrt(1/(2*pow(10,(SNR/10)))));
    //Brukes ved ferdigdefinerte H'er
    //Test for constrainthlength=16, #h=4, H=[270463,037772,136264,015274], Antall dummy bits =0
    anth=4;
    constraintlength=16;
    int anto;int* matrix;
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    matrix = new int[anto*anth];
    int midl[24]={2,7,0,4,6,3,0,3,7,7,7,2,1,3,6,2,6,4,0,1,5,2,7,4};
    for(int i=0;i<(anto*anth);i++)
        {matrix[i]=midl[i];}
    ant_dummy_bits=-1;
    int i=0;
    if(QAMstr==pow(2,anth)){ant_dummy_bits=0;}
    else if (QAMstr<pow(2,anth)){cout<<"Konstellasjonen er for liten for denne koden";getch();}
    else
    {
        while ((QAMstr!=pow(2,anth))&&(i<10))
        {

```

```

        i++;
        if (QAMstr==pow(2,(anth+i)))
            {ant_dummy_bits=i;}
    }
}
if (ant_dummy_bits==-1){cout<<"Error, kan ikke definere ant_dummy_bits";getch();}
return matrix;
delete[] matrix;
}

int trelliskode::get_QAM()
{
    switch(QAMstr)
    {
        case 8 :
        {
            for (int i=0;i<8;i++)
            {
                for (int j=0;j<3;j++)
                    {QAM[i][j]=QAM8[i][j];}
            }
            break;
        }
        case 16 :
        {
            for (int i=0;i<16;i++)
            {
                for (int j=0;j<3;j++)
                    {QAM[i][j]=QAM16[i][j];}
            }
            break;
        }
        case 32 :
        {
            for (int i=0;i<32;i++)
            {
                for (int j=0;j<3;j++)
                    {QAM[i][j]=QAM32[i][j];}
            }
            break;
        }
        case 64 :
        {
            for (int i=0;i<64;i++)
            {
                for (int j=0;j<3;j++)
                    {QAM[i][j]=QAM64[i][j];}
            }
            break;
        }
        case 128 :
        {
            for (int i=0;i<128;i++)
            {
                for (int j=0;j<3;j++)
                    {QAM[i][j]=QAM128[i][j];}
            }
            break;
        }
        default : cout<<"\nFeil input til get_QAM()";getch();
    }
}
}

```

```

void trelliskode::show_parity_matrix(int* matr)
{
    int anto; //Antall elementer på oktal form
    cout<<"Paritetssjekk matrisen er :\n\n";
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1;
    else anto=(constraintlength+1)/3;
    cout<<"\nconstraintlength = "<<constraintlength<<" H = "<<anth;
    for (int i=0;i<(anto*anth);i++)
    {
        if (i%anto==0)cout<<"\nH"<<i/anto<<" : ";
        cout<<matr[i]<<" ";
    }
}

void trelliskode::make_encoder(int* matrix)
{
    ant_dummy_bits=0;
    int anto,p,a; //p brukes til å lagre pow(2,k), vet ikke hvorfor dette må gjøres
    int skipbits;
    int* memory;
    int* oldmem;
    int* input;
    skipbits=2-((constraintlength)%3); //bits i hver H som ikke teller
    input= new int[anth-2]; //input til encoder
    int* dummy_input = new int[ant_dummy_bits];
    memory= new int[constraintlength-1];
    oldmem= new int[constraintlength-1];
    for (int i=0;i<=constraintlength;i++){memory[i]=0;oldmem[i]=0;} //nullstille memory
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;
    char test=0;cout<<"\nTast inn input : ";test=getche();
    while (test!='q')
    {
        //*****Tast input*****
        input[0]=test-48;
        for (int i=1;i<(anth-1);i++){input[i]=getche()-48;}
        for (int i=0;i<(ant_dummy_bits);i++){dummy_input[i]=getche()-48;}
        for (int j=0;j<anto;j++) //Går igjennom alle de oktale pos. i h'ene
        {
            for (int k=0;k<3;k++) //Går igjennom 1,2 og 3 bit i hver oktal h
            {
                for (int i=0;i<(anth);i++)//Går gjennom alle H'ene
                {
                    if (j==0&&k==0){k=skipbits;}
                    if (i==0) //Spesiell behandling for H0 (rotasjonspolynomet)
                    {
                        p=pow(2,k);
                        a=((5-(k%2))-p); //a=((5-(k%2))-p) AND med 4 så 2 så 1
                        if ( ((matrix[j]&a)!=0&&(j==0&&k==skipbits)) //Ikke første bit i H0 og matrix er 1
                            &&((matrix[j]&a)!=0&&(!((j*3)+k)==constraintlength+skipbits))) )
                            //Ikke siste bit i H0 og matrix er 1
                            {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]
                                +oldmem[constraintlength-1]+oldmem[((j*3)+k)-skipbits-1])%2);
                                cout<<"(#1)"<<memory[(j*3)+k-1];}
                        else if(((j*3)+k)==constraintlength+skipbits) //Siste bit i H
                            {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]
                                +oldmem[((j*3)+k)-skipbits-1])%2;cout<<"(#2)";}
                        else if(j==0&&k==skipbits)//Første bit i H
                            {
                                if(skipbits==2){memory[0]=(memory[0]+oldmem[constraintlength-skipbits+1])%2;}

```

```

        else if (skipbits==0) {memory[0]=(memory[0]
+oldmem[constraintlength-skipbits-1])%2;}
        else {memory[0]=(memory[0]+oldmem[constraintlength-skipbits])%2;}
        cout<<"(#3)";
        }
        else {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]
+oldmem[(j*3)+k-skipbits-1])%2;cout<<"(#4)";} //Ikke første, siste og matrix er 0
        }
    else if ((matrix[(i*(anto))+j]&a)!=0)
        {memory[(j*3)+k-skipbits]=((memory[(j*3)+k-skipbits])+input[i-1])%2);

    else {cout<<"(-)";}
    }
}
cout<<"\nMemory er : ";
for (int i=0;i<constraintlength;i++){cout<<memory[i]<<" ";}
cout<<"\nOutput er : ";
int outsum=oldmem[constraintlength-1]; //Brukes til å lagre output som tall i titallsystemet
cout<<oldmem[constraintlength-1]<<" ";getch();
int old_i;
for (int i=0;i<(anth-1);i++)
    {cout<<input[i]<<" ";outsum=outsum+(pow(2,(i+1))*input[i]);old_i=i;}
for (int i=0;i<(ant_dummy_bits);i++)
    {cout<<dummy_input[i]<<" ";outsum=outsum+(pow(2,(i+old_i+2))*dummy_input[i]);}
cout<<" -> "<<outsum<<" (Signal punkt nummer)";
for (int i=0;i<=constraintlength;i++){oldmem[i]=memory[i];memory[i]=0;}
cout<<"\n Tast neste input (q for avslutt) : ";test=getche();
}
}

```

//Funksjonen tar inn matrisen med H'ene, hvilke state encoderen skal være i, og input til encoderen.
//Den returnerer den nye staten til encoderen og output. Både input og output returneres som heltall.
//f.eks. 1010=5, 11001=19 osv...
//Funksjonen er klargjort for dummy input, men har ingen betydning før input er stor nok. D.v.s.
//større enn mulig vanlig input.
//type return inneholder hvilke retval verdi som skal returneres.
//memory, oldmem og input arrayene er satt med lengde 50. Hvis constraintlengden skulle overstige 50 vil det oppstå problemer

```

long trelliskode::get_next_state(int* matrix, long state, int inp, int typereturn)
{
    int anto,p,a; //p brukes til å lagre pow(2,k), vet ikke hvorfor dette må gjøres
    int skipbits;
    skipbits=2-((constraintlength)%3); //bits i hver H som ikke teller

    long memory[50];
    long oldmem[50];
    int input[50];
    for (int i=0;i<constraintlength;i++) //Legger state inn i memory
    {
        long midl=(pow(2,i));
        if((state&midl)!=0){memory[i]=0;oldmem[i]=1;}
        else {memory[i]=0;oldmem[i]=0;}
    }
    if ((constraintlength+1)%3!=0) anto=(constraintlength+1)/3+1; //Antall elementer på oktal form
    else anto=(constraintlength+1)/3;

    for (int i=0;i<(anth-1);i++) //Legger input over i array
    {
        long midl=(pow(2,i));
        if((inp&midl)!=0){input[i]=1;}
        else {input[i]=0;}
    }
}

```

```

}
for (int j=0;j<anto;j++) //Går igjennom alle de oktale pos. i h'ene
{
    for (int k=0;k<3;k++) //Går igjennom 1,2 og 3 bit i hver oktal h
    {
        for (int i=0;i<(anth);i++)//Går gjennom alle H'ene
        {
            if (j==0&&k==0){k=skipbits;}//for å starte i riktig posisjon
            if (i==0) //Spesiell behandling for H0 (rotasjonspolynomet)
            {
                p=pow(2,k);
                a=((5-(k%2))-p);
            }
            if ( ((matrix[j]&a)!=0&&!(j==0&&k==skipbits)) //Ikke første bit i H0 og matrix er 1
                &&((matrix[j]&a)!=0&&(((j*3)+k)==constraintlength+skipbits))) )
                //Ikke siste bit i H0 og matrix er 1
            {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]
            +oldmem[constraintlength-1]+oldmem[(j*3)+k-skipbits
            -1])%2);}//cout<<"#1"<<memory[(j*3)+k-skipbits];}
            else if(((j*3)+k)==constraintlength+skipbits) //Siste bit i H
            {
                memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]+oldmem[(j*3)+k-skipbits-1])%2;
            }
            else if(j==0&&k==skipbits)
            {
                if(skipbits==2){memory[0]=(memory[0]+oldmem[constraintlength-skipbits+1])%2;}
                else if (skipbits==0){memory[0]=(memory[0]+oldmem[constraintlength-skipbits-1])%2;}
                else {memory[0]=(memory[0]+oldmem[constraintlength-skipbits])%2;}
            }
            else {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]+oldmem[(j*3)
            +k-skipbits-1])%2;}//cout<<"#4"; //Ikke første, siste og matrix er 0
            }
            else if ((matrix[(i*(anto))+j]&a)!=0) //a=((5-(k%2))-p) AND med 4 så 2 så 1
            {memory[(j*3)+k-skipbits]=(memory[(j*3)+k-skipbits]
            +input[i-1])%2);
            }
            else {}
        }
    }
}

long retval[2];
retval[0]=0;retval[1]=0;retval[2]=0;
for (int i=0;i<constraintlength;i++) //Legger state fra array til heltall
{retval[0]=(retval[0]+(memory[i]*(pow(2,i))));}
for (int i=0;i<(anth-1);i++) //Legger output fra array til heltall
{retval[1]=(retval[1]+(input[i]*(pow(2,(i+1)))));}
retval[1]=retval[1]+oldmem[constraintlength-1];
for (int i=0;i<(anth-1);i++)
{retval[2]=(retval[2]+(input[i]*(pow(2,i))));}
long back;
if (typereturn==0)
{back=retval[0];}
else if (typereturn==1)
{back=retval[1];}
else {back=retval[2];}
return back;
}

```

//parity_matrix er paritets matrisen,xlength er $2^{(anth-1)}$, ylength er $2^{constraintlength}$
//PS! xlength kan økes til å inkludere dummybits (vet ikke om det trengs, kan inkluderes i en annen funksjon)
long** trelliskode::make_state_list(int* parity_matrix)

```

    {
    long xlength=pow(2,(anth-1));
    long ylength=pow(2,constraintlength);
    long** state_list;
    state_list=new long*[xlength-1];//Inneholder tabell over hvilke state man kan gå til fra en gitt state
    for (long i=0;i<xlength;i++)
        {state_list[i]=new long[ylength];}
    long retval;//retval=new int[2];
    long counter=(((xlength-1)*(ylength-1))/100);
    long count=0;
    cout<<"\n\nLager State List\n0%";
    for (long i=0;i<xlength;i++)//Går gjennom alle input
        {
        for (long j=0;j<ylength;j++)//Går gjennom alle states
            {
            if(((i*j)>(counter*count))&&(count<100)){count++;cout<<"\b\b\b" <<count<<"%";}
            retval=get_next_state(parity_matrix,j,i,0);
            state_list[i][j]=retval;
            }
        }
    return state_list; //list er en array med to pekere. En til state_list og en til path_list.
    }

//parity_matrix er paritets matrisen,xlength er 2^(anth-1), ylength er 2^constraintlength
//PS! xlength kan økes til å inkludere dummybits (vet ikke om det trengs, kan inkluderes i en annen funksjon)
long** trelliskode::make_out_list(int* parity_matrix)
    {
    long xlength=pow(2,(anth-1));
    long ylength=pow(2,constraintlength);
    long** path_list;
    path_list=new long*[xlength-1]; //Inneholder tabell over output når man går fra en state til en annen ved gitt input
    for (long i=0;i<xlength;i++)
        {path_list[i]=new long[ylength];}
    long retval;//retval=new int[2];
    long counter=(((xlength-1)*(ylength-1))/100);
    long count=0;
    cout<<"\n\nLager Out List\n0%";
    for (long i=0;i<xlength;i++)
        {
        for (long j=0;j<ylength;j++)
            {
            if(((i*j)>(counter*count))&&(count<100)){count++;cout<<"\b\b\b" <<count<<"%";}
            retval=get_next_state(parity_matrix,j,i,1);
            path_list[i][j]=retval;
            }
        }
    return path_list; //list er en array med to pekere. En til state_list og en til path_list.
    }

void trelliskode::show_state_list(long** state_list,long x,long y)
    {
    cout<<"\n\n          STATE LIST :\n\n ";
    for (int j=0;j<x;j++)
        {
        if(j<10){cout<<" ";}
        cout<<j<<" ";
        }
    cout<<"\n";
    for (int i=0;i<y;i++)
        {
        cout<<"\n";if(i<10){cout<<" ";}cout<<i<<"->";

```

```

for (int j=0;j<x;j++)
{
    if(state_list[j][i]<10)
        {cout<<" ";}
    cout<<state_list[j][i]<<" ";
}
}
getch();
}

```

//Finner metrikken mellom to punkter. z verdiene er mottatt signal. a verdiene er merkelapp på gren
double trelliskode::find_metric(double zx,double zy,double ax,double ay)

```

{
    double n=anth;
    double R;R=(n-1)/n;
    int K=QAMstr; //Antall punkt i konstellasjonen
    double metric,temp,temp2=0;
    temp=exp( ((pow((zx-ax),2)+pow((zy-ay),2))*-1)/(2*pow(ro,2)) );//Over brøkstreken
    for (int i=0;i<K;i++)
    {
        temp2=temp2+(exp( ((pow((zx-QAM[i][1]),2)+pow((zy-QAM[i][2]),2))*-1)/(2*pow(ro,2)) ));
    }
}

```

//Her må det settes en grense for hvor lite tallet over brøkstreken skal være. Hvis puktene er veldig
//langt fra hverandre, blir verdien over brøkstreken tilnærmet 0. Man kan da ikke ta logaritmen.

```

if (temp<(pow(10,-300)))
    {temp=pow(10,-300);}
if (temp2==0)
    {cout<<"\nError, metric calculation, cant divide by 0.";metric=0;}
else
    {
        metric=(log2(temp/temp2))+(n*(1-R));
    }
return metric;
}

```

int* trelliskode::generate_bits(int antbits)

```

{
    //srand(2);//Ved å kjøre denne, vil man få samme kodeblokk hver gang
    double MAX=RAND_MAX;
    int randnr;
    int* bitarray;
    bitarray=new int[antbits];
    for (int i=0;i<antbits;i++)
    {
        randnr=rand();
        if (randnr<(RAND_MAX/2))
            {bitarray[i]=0;}
        else {bitarray[i]=1;}
    }
    return bitarray;
    delete[] bitarray;
}

```

//Koder en array med input bits til signalpunkter.

double* trelliskode::make_code_block(long** state_list, int* input_arr)

```

{
    int xlength=pow(2,(anth-1));
    int ylength=pow(2,constraintlength);
    int ant_bits_in_block; //#bits i en blokk før koding. D.v.s. #bits inn i encoderen pr. blokk * #blokker
}

```

```

ant_bits_in_block=((anth-1)+ant_dummy_bits)*block_length;
int ant_bits_pr_symb;    //Antall bits pr. symbol
ant_bits_pr_symb=((anth-1)+ant_dummy_bits);
ofstream outfile("Input.txt");
for(int i=0;i<(((anth-1)+ant_dummy_bits)*block_length);i++)
    {outfile<<input_arr[i]<<' ';}

//Finner det punktet som ligger lengst fra 0. Brukes i koding av halen
int talepoint;    //Punktet som ligger lengst fra punkt 0 i konstallasjonen
double midl=0;
for (int i=0;i<QAMstr;i++)
    {
        double newmid=sqrt(pow((QAM[i][1]-QAM[0][1]),2)+pow((QAM[i][2]-QAM[0][2]),2));
        if (newmid>midl){talepoint=QAM[i][0];midl=newmid;}
    }
int* code_block;code_block=new int[block_length+constraintlength];
double* QAM_block;QAM_block=new double[((block_length+constraintlength)*2)];
int input=0;
int dummy_bit_extra=0; //Bruke til å lagre tillegg til output p.g.a. dummy bits
int current_state=0; //Starter alltid i 0 state;
for(int i=0;i<(block_length+constraintlength);i++)
    {
        for(int j=0;j<ant_bits_pr_symb;j++)
            {
                if (i<block_length)
                    {
                        if (j<(ant_bits_pr_symb)-ant_dummy_bits)
                            {
                                input=input+(input_arr[j+(i*ant_bits_pr_symb)]*pow(2,j));
                                //Gjør om bits til heltall, finner de riktige bitsene
                            }
                        else
                            {
                                dummy_bit_extra=dummy_bit_extra+(input_arr[j+(i*ant_bits_pr_symb)]*pow(2,(j+1)));
                                //j+1 fordi output gir en ekstra bit
                            }
                    }
                else
                    {
                        input=0;dummy_bit_extra=0;
                    }
            }
        if (i<block_length)//Hvis det fortsatt er igjen blokker som skal kodes
            {
                code_block[i]=get_outval(input,current_state)+dummy_bit_extra;
            }
        else
            {
                if(get_outval(input,current_state)==1)
                    {code_block[i]=talepoint;}
                else {code_block[i]=0;}
            }
        current_state=state_list[input][current_state];
        input=0;dummy_bit_extra=0;
    }
for (int i=0;i<block_length+constraintlength;i++)
    //legger QAM symboler inn i QAM_block. Hvert symbol har to koordinater
    {
        QAM_block[(i*2)]=QAM[code_block[i]][1];
        QAM_block[(i*2)+1]=QAM[code_block[i]][2];
    }
delete[] code_block;

```

```

delete[] state_list;//delete[] out_list;
return QAM_block;
delete[] QAM_block;
}

void trelliskode::stack_algorithm(long** state_list,double* codeblock)
{
link* current;current=NULL;
stack metricstack;
long xlength=pow(2,(anth-1));
long ylength=pow(2,constraintlength);
long exlength=pow(2,ant_dummy_bits);//Tilleggs lengde p.g.a. dummy bits i x retningen (skal ganges inn)
if (ant_dummy_bits==0)exlength=0;

//Finner det punktet som ligger lengst fra 0. Brukes i dekodning av halen
int tailpoint; //Punktet som ligger lengst fra punkt 0 i konstallasjonen
double midl=0;
for (int i=0;i<QAMstr;i++)
{
double newmid=sqrt(pow((QAM[i][1]-QAM[0][1]),2)+pow((QAM[i][2]-QAM[0][2]),2));
if (newmid>midl){tailpoint=QAM[i][0];midl=newmid;}
}
long current_state=0;
double treemetric;
double oldtreemetric=0;
int* treepath; //Inneholder stien gjennom treet
int oldtreepath[1700];//Fikk minneproblemer ved å lage denne dynamisk. Hvor skal den i såfall slettes
int treepathlength;
int oldtreepathlength=0;
long current_next_state;
int j=0;
int counter=0;
while (j<(block_length+constraintlength))//Går gjennom hver enkelt blokk
{
counter++;
if (current==NULL)//Når stacken er tom
{
current_next_state=0;
treemetric=0;
treepathlength=0;
}
if (j<block_length)//Hvis det dekodes punkter i kodeblokken
{
int runval=xlength*exlength;
if (runval==0){runval=xlength;}
for(long i=0;i<runval;i++)//Går gjennom alle mulig valg for hver blokk
{
//oppretter stien gjennom treet (regner ut lengden av denne)
treepath=new int[((anth-1)+ant_dummy_bits)*(block_length+constraintlength)];
for(int l=0;l<oldtreepathlength;l++)
{treepath[l]=oldtreepath[l];} //Legger gammel path til starten av den nye
for(int k=(j*((anth-1)+ant_dummy_bits));k<(((anth-1)+ant_dummy_bits)
+(j*((anth-1)+ant_dummy_bits)));k++) //Legger til de nye bitene
{
int test=(pow(2,(k-(j*((anth-1)+ant_dummy_bits)))));
if((i&test)!=0){treepath[k]=1;}
else {treepath[k]=0;}
}
}
long val;
val=real_input(i,ant_dummy_bits,xlength); //Henter input uten dummybits
treepathlength=anth-1+ant_dummy_bits;
current_next_state=state_list[val][current_state]; //Henter output etter gitt input
}
}

```

```

//Henter de to tilhørende verdiene ut av qam blokken
double qamval1=QAM[get_outval(val,current_state)+dummmmy_add(i,ant_dummy_bits,xlength)][1];
double qamval2=QAM[get_outval(val,current_state)+dummmmy_add(i,ant_dummy_bits,xlength)][2];
    treemetric=find_metric(codeblock[j*2],codeblock[(j*2)+1],qamval1,qamval2)+oldtreemetric;
    metricstack.additem(treepath,treepathlength+oldtreepathlength,treemetric,current_next_state);
    }
else //Hvis det skal dekodes bit i halen
    {
    treepath=new int[((anth-1)+ant_dummy_bits)*(block_length+constraintlength)];
    for(int l=0;l<oldtreepathlength;l++)
        {treepath[l]=oldtreepath[l];} //Legger gammel path til starten av den nye
    for(int k=j*((anth-1)+ant_dummy_bits);k<(((anth-1)+ant_dummy_bits)
        +(j*((anth-1)+ant_dummy_bits)));k++) //Legger til de nye bitene
        {
        treepath[k]=0;
        }
    treepathlength=anth-1+ant_dummy_bits;
    current_next_state=state_list[0][current_state];
    if(get_outval(0,current_state)==1)
        {
        treemetric=find_metric(codeblock[j*2],codeblock[(j*2)+1],QAM[tailpoint][1],QAM[tailpoint][2])
            +oldtreemetric;
        metricstack.additem(treepath,treepathlength+oldtreepathlength,treemetric,current_next_state);
        }
    else
        {
        treemetric=find_metric(codeblock[j*2],codeblock[(j*2)+1],QAM[0][1],QAM[0][2])+oldtreemetric;
        metricstack.additem(treepath,treepathlength+oldtreepathlength,treemetric,current_next_state);
        }
    }
//Tar elementet med best metrikk av stacken for å fortsette på dette
current=metricstack.getfirst();
current_state=current->state;
oldtreemetric=current->metric;
oldtreepathlength=current->path_length;
//kopierer path over i oldpath for å ta vare på den
for(int l=0;l<oldtreepathlength;l++)
    {
    oldtreepath[l]=current->path[l];
    }
//Hvis dekodingen ikke er ferdig, fjern øverste element
if (j<((block_length+constraintlength)-1))
    {
    metricstack.removeitem();
    }
j=(oldtreepathlength/(anth-1+ant_dummy_bits));
}
//Skriver ut resultatet av dekodingen til fil
ofstream outfile("Result.txt");
current=metricstack.getfirst();
for (int i=0;i<((current->path_length));i++)
    {outfile<<(current->path[i]<<' ');} //cout<<input[i];
delete state_list;//delete out_list;
delete[] codeblock;
}

//Funksjonen gjør om input med dummy bits til input uten dummybits. D.v.s. fjerner dummybits fra input
long trelliskode::real_input(long icount,int dummy_bits,long lengthx)
    {
    int midl=0;int j=0;

```

```

while(midl<lengthx)//Finner antall bits som skal til for å representere lengthx
{
    midl=midl+pow(2,j);
    j++;
}
int nrbits=j-1;
int retval=0;
for (long i=0;i<nrbits;i++)//Regner ut verdien av output uten dummybits
{
    long midl2=pow(2,i);
    if((icount&midl2)!=0)
        {retval=retval+pow(2,i);}
}
return retval;
}

long trelliskode::dummy_add(long icount,int dummybits, long lengthx)
{
    int midl=0;int j=0;
    while(midl<lengthx)//Finner antall bits som skal til for å representere lengthx
    {
        midl=midl+pow(2,j);
        j++;
    }
    int nrbits=j-1;
    int retval=0;
    for (int i=0;i<dummybits;i++)//Regner ut tilleggs output p.g.a. dummybits
    {
        long midl2=pow(2,(i+nrbits));
        if((icount&midl2)!=0)
            {retval=retval+pow(2,(i+nrbits+1));}
    }
    return retval;
}

//Henter størrelsen på konstellasjonen og returnerer m (der 2~m er #punkter)

int trelliskode::get_qamstr()
{
    int nr;
    cout<<"Velg størrelse på QAM konstellasjon : \n1) QPSK(m=2)\n2) 8-CROSS(m=3)
        (Ikke klargjort for denne enda)";
    cout<<"\n3) 16-QASK(m=4)\n4) 32-CROSS(m=5)\n5) 64-QASK(m=6)\n6) 128-CROSS(m=7)\n7) 256-
        QASK(m=8)\n";
    cout<<"8) 512-CROSS(m=7)\n9) 1024-QASK(m=8)\n";
    cout<<"Skriv nr : ";
    nr = getche()-48;
    cout<<"\nDu valgte : "<<pow(2,(nr+1))<<"-QAM";getch();
    return (nr+1);
}

//Setter elementer inn i stacken slik at stacken holdes sortet. Høyeste metrikk øverst
//Input er stien i treet, lengden av stien og metrikk
void stack::additem(int* this_path,int p_length ,double metr, int this_state)
{
    stacklength++;
    gotoxy(50,19);cout<<"Stacklength : "<<stacklength<<" ";
    link* newlink=new link;
    newlink->metric=metr;
    newlink->path=this_path;
    newlink->path_length=p_length;
}

```

```

newlink->state=this_state;
int bitset=0;
if (first==NULL)
{
    newlink->next=first;
    newlink->previous=first;
    first=newlink;
}
else
{
    link* current=first;
    while (current!=NULL && bitset==0)
    {
        if (newlink->metric>=current->metric)
        {
            //cout<<"#1 ";
            newlink->previous=current->previous;
            newlink->next=current;
            current->previous=newlink;
            if(newlink->previous!=NULL)
                {newlink->previous->next=newlink;}
            if(newlink->previous==NULL)
                {first=newlink;}
            bitset=1;
        }
        else if(current->next==NULL)
        {
            //cout<<"#2 ";
            newlink->previous=current;
            newlink->next=current->next;
            current->next=newlink;
            bitset=1;
        }
        current=current->next;
    }
}
}

```

```

void stack::showstack()
{
    link* current=first;
    cout<<"\n\nStacken :\n";
    while (current!=NULL)
    {
        cout<<"\n";
        for (int i=0;i<((current->path_length));i++)
        {
            cout<<current->path[i];
        }
        //cout<<"("<<current->path<<") ";
        cout<<" ("<<current->metric<<")";
        cout<<" Current State = "<<current->state;
        current=current->next;
    }
    getch();
    //delete[] input;
}

```

```

//Sletterførste element i stacken
void stack::removeitem()

```

```

{
    stacklength--;
}

```

```

gotoxy(50,19);cout<<"Stacklength : "<<stacklength<<" ";
link* current;
current=first;
first=current->next;
first->previous=NULL;
delete current->path;
delete current;
}

link* stack::getfirst()
{
return first;
}

//Funksjonen genererer normalfordelt random støy
//RAND_MAX er definert i stdlib.h og er satt til 32767 som er max størrelse for en int
double* trelliskode::add_noise(double* QAMblock)
{
double Pi=3.14159265;
double MAX=RAND_MAX;
int randnr1,randnr2;
double un,um,yn,ym; //un=u_n, um=u_{n+1} osv... n og n+1 er index
for (int i=0;i<block_length+constraintlength;i++)
{
randnr1=rand();randnr2=rand();
un=(randnr1+1)/MAX;
um=(randnr2+1)/MAX;
yn=(sqrt((-2)*log(un))*ro*(sin(2*Pi*um)))/+0.7;
ym=(sqrt((-2)*log(un))*ro*(cos(2*Pi*um)))/-0.9;
QAMblock[2*i]=QAMblock[2*i]+yn;
QAMblock[(2*i)+1]=QAMblock[(2*i)+1]+ym;
}
cout<<"\nCode block with noise :\n";
return QAMblock;
}

//Legger til støy manuelt (kun for testing)
//Må gjøres om hvis block_length forandres. 20 bit for blocklength 10. 8 bit hale for constraintlength 4
double* trelliskode::add_manual_noise(double* QAMblock)
{
double noise[28]={1,1.1,-0.5,1.5,-0.8,2,1.9,0.5,-1.1,1,0.9,-0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0,0,0,0,0,0};
cout<<"\nCode block with noise :\n";
for (int i=0;i<28;i++)
{
QAMblock[i]=QAMblock[i]+noise[i];
}
for (int i=0;i<(block_length+constraintlength);i++)
{
cout<<" ("<<QAMblock[2*i]<<","<<QAMblock[(2*i)+1]<<")\n";
}
}
getch();
return QAMblock;
}

int trelliskode::result(int length, long blocknumber)
{
int* a;
int* b;
a=new int[length];
b=new int[length];
int errors=0;int i=0;
ifstream infile1("Input.txt");

```

```

ifstream infile2("Result.txt");
ofstream outfile1("Errors.txt",ios::app);//Legger til informasjon i fila
for (int i=0;i<length;i++)
    {
    infile1>>a[i];
    }
for (int i=0;i<length;i++)
    {
    infile2>>b[i];
    if (a[i]!=b[i]){errors++;outfile1<<"\nError at bit nr "<<i<<" in block nr "<<blocknumber;}
    }
delete[] a;delete[] b;
return errors;
}

```

```

long trelliskode::get_outval(long inp,long state)

```

```

{
long retval=0;
long midl=pow(2,constraintlength-1);
if((state&midl)!=0)
    {retval=(inp*2)+1;}
else
    {retval=(inp*2);}
return retval;
}

```

```

int trelliskode::run_sim(long** slist,int size)

```

```

{
long testnumber=loops;
srand(1);//initsierer pseudo random generatoren med et seed som parameter
clock_t start, end, midl;
float errorcount=0;
int timer, min, sek;
float midl2;
int* bits;
double* block;
cout<<"\n\nAntall dekodded bits :\nBeregnet tid:\nTid:\n";
start=clock();
for(long j=0;j<testnumber;j++)
    {
    bits=generate_bits(size);
    block=make_code_block(slist,bits);

//legger til støy. Gjøres her p.g.a. problemer rundt srand() da jeg skulle fortsette
//å plukke random verdier uten å starte srand på nytt hver gang funksjonen startet
//double ro=sqrt(1/(2*pow(10,(SNR/10))));cout<<"ro="<<ro;getch();

double Pi=3.14159265;
double MAX=RAND_MAX;
int randnr1,randnr2;
double un,um,yn,ym; //un=u_n, um=u_n+1 osv... n og n+1 er index
for (int i=0;i<(block_length+constraintlength);i++)
    {
    randnr1=rand();randnr2=rand();
    un=(randnr1+1)/(MAX+1);
    um=(randnr2+1)/(MAX+1);
    yn=(sqrt((-2)*log(un)))*ro*(sin(2*Pi*um));//+0.7;
    ym=(sqrt((-2)*log(un)))*ro*(cos(2*Pi*um));//-0.9;
    block[2*i]=block[2*i]+yn;
    block[(2*i)+1]=block[(2*i)+1]+ym;
    }
}
}

```

```

        }

stack_algorithm(slist,block);
    errorcount=result(size,j)+errorcount;
gotoxy(1,22);
cout<<(j+1)*size<<"/"<<(testnumber*size);

midl=clock();midl=(midl-start)/CLK_TCK;//cout<<"midl="<<midl;getch();
gotoxy(15,21);
timer=(midl/3600);min=(midl-(timer*3600))/60;sek=(midl-(timer*3600)-(min*60));
cout<<timer<<"t"<<min<<"m"<<sek<<"s  ";
if (j==10)
    {
        midl=clock();midl=(midl-start);//cout<<"midl="<<midl;getch();
        midl=((midl*testnumber)/CLK_TCK)/10;
        timer=(midl/3600);min=(midl-(timer*3600))/60;sek=(midl-(timer*3600)-(min*60));
        gotoxy(15,20);cout<<timer<<"t"<<min<<"m"<<sek<<"s";
    }
delete[] bits;
}
end=clock();
cout<<"\n\n\n  RESULTAT\n";
cout<<"Antall dekodingsfeil : "<<errorcount;
cout<<"\nBER : "<<(errorcount/(testnumber*size));
cout<<"\nBits/sekund dekodet : "<<(((testnumber-1)*size)/((end - start) / CLK_TCK));
ofstream outfile("CodeRes.txt",ios::app);//Legger til informasjon i fila
outfile<<code<<"\t"<<block_length<<"\t"<<QAMstr<<"\t"<<SNR<<"\t"
    <<(((testnumber-1)*size)/((end - start) / CLK_TCK))<<"\t"<<testnumber<<"\t"
    <<(errorcount/(testnumber*size))<<"\n";
}

```


Vedlegg 3

QAM.h

I QAM.h filen ligger alle signalkonstellasjonene definert. Konstellasjonene er splittet opp som beskrevet i kapittel 3. Denne filen må inkluderes hvis man skal kunne kompilere simulatoren i vedlegg 2.

```
double QAM8[8][3]= {
    {0,0.923879532511,0.382683432365},
    {1,0.382683432365,0.923879532511},
    {2,-0.382683432365,0.923879532511},
    {3,-0.923879532511,0.382683432365},
    {4,-0.923879532511,-0.382683432365},
    {5,-0.382683432365,-0.923879532511},
    {6,0.382683432365,-0.923879532511},
    {7,0.923879532511,-0.382683432365},
};

int QAM16[16][3]= {
    {0,1,-1},
    {1,-3,3},
    {2,-3,-1},
    {3,1,3},
    {4,3,-3},
    {5,-1,1},
    {6,-1,-3},
    {7,3,1},
    {8,-1,-1},
    {9,3,3},
    {10,3,-1},
    {11,-1,3},
    {12,-3,-3},
    {13,1,1},
    {14,1,-3},
    {15,-3,1}
};

int QAM32[32][3]= {
    {0,-1,3},
    {1,3,3},
    {2,3,-1},
    {3,3,-5},
    {4,-1,-5},
    {5,-1,-1},
    {6,-5,-1},
    {7,-5,3},
    {8,-3,1},
    {9,-3,-3},
    {10,1,-1},
    {11,5,-3},
    {12,5,1},
    {13,1,1},
```

```

{14,1,5},
{15,-3,5},
{16,-3,-1},
{17,-3,3},
{18,1,3},
{19,5,3},
{20,5,-1},
{21,1,-1},
{22,1,-5},
{23,-3,-5},
{24,3,1},
{25,3,-3},
{26,-1,-3},
{27,-5,-3},
{28,-5,1},
{29,-1,1},
{30,-1,5},
{31,3,5},
};

```

```

int QAM64[64][3]={
    {0,-7,7},
    {1,1,-1},
    {2,1,7},
    {3,-7,-1},
    {4,-3,3},
    {5,5,-5},
    {6,5,3},
    {7,-3,-5},
    {8,-3,7},
    {9,5,-1},
    {10,5,7},
    {11,-3,-1},
    {12,-7,3},
    {13,1,-5},
    {14,1,3},
    {15,-7,-5},
    {16,-5,5},
    {17,3,-3},
    {18,3,5},
    {19,-5,-3},
    {20,-1,1},
    {21,7,-7},
    {22,7,1},
    {23,-1,-7},
    {24,-1,3},
    {25,7,-5},
    {26,-1,5},
    {27,7,-3},
    {28,-5,1},
    {29,3,-7},
    {30,-5,1},
    {31,3,-7},
    {32,3,1},
    {33,-5,-7},
    {34,-5,7},
};

```

```
{35,3,-1},
{36,3,7},
{37,-5,-1},
{38,-1,5},
{39,7,-3},
{40,7,5},
{41,-1,3},
{42,-1,7},
{43,7,-1},
{44,7,7},
{45,-1,-1},
{46,-5,3},
{47,3,-5},
{48,3,3},
{49,-5,-5},
{50,-7,5},
{51,1,-3},
{52,1,5},
{53,-7,-3},
{54,-3,1},
{55,5,-7},
{56,5,1},
{57,-3,-7},
{58,-3,5},
{59,5,-3},
{60,5,5},
{61,-3,-3},
{62,1,1},
{63,-7,-7},
```

```
};
```

```
int QAM128[128][3]={
    {0,-1,7},
    {1,7,3},
    {2,3,-5},
    {3,-5,-1},
    {4,7,7},
    {5,7,-5},
    {6,-5,-5},
    {7,-5,7},
    {8,7,-1},
    {9,-1,-5},
    {10,-5,3},
    {11,3,7},
    {12,7,-9},
    {13,-9,-5},
    {14,-5,11},
    {15,11,7},
    {16,-1,-9},
    {17,-9,3},
    {18,3,11},
    {19,11,-1},
    {20,-1,-1},
    {21,-1,-3},
    {22,3,3},
```

{23,3,-1},
{24,-9,-1},
{25,-1,11},
{26,11,3},
{27,3,-9},
{28,-9,7},
{29,7,11},
{30,11,-5},
{31,-5,-9},
{32,-7,-1},
{33,-3,7},
{34,5,3},
{35,1,-5},
{36,-7,7},
{37,5,7},
{38,5,-5},
{39,-7,-5},
{40,1,7},
{41,5,-1},
{42,-3,-5},
{43,-7,3},
{44,9,7},
{45,5,-9},
{46,-11,-5},
{47,-7,11},
{48,9,-1},
{49,-3,-9},
{50,-11,3},
{51,1,11},
{52,1,-1},
{53,-3,-1},
{54,-3,3},
{55,1,3},
{56,1,-9},
{57,-11,-1},
{58,-3,11},
{59,9,3},
{60,-7,-9},
{61,-11,7},
{62,5,11},
{63,9,-5},
{64,1,-7},
{65,-7,-3},
{66,-3,5},
{67,5,1},
{68,-7,-7},
{69,-7,5},
{70,5,5},
{71,5,-7},
{72,-7,1},
{73,1,5},
{74,5,-3},
{75,-3,-7},
{76,-7,9},
{77,9,5},
{78,5,-11},

{79,-11,-7},
{80,1,9},
{81,9,-3},
{82,-3,-11},
{83,-11,1},
{84,1,1},
{85,1,-3},
{86,-3,-3},
{87,-3,1},
{88,9,1},
{89,1,-11},
{90,-11,-3},
{91,-3,9},
{92,9,-7},
{93,-7,-11},
{94,-11,5},
{95,5,9},
{96,7,1},
{97,3,-7},
{98,-5,-3},
{99,-1,5},
{100,7,-7},
{101,-5,-7},
{102,-5,5},
{103,7,5},
{104,-1,-7},
{105,-5,1},
{106,3,5},
{107,7,-3},
{108,-9,-7},
{109,-5,9},
{110,11,5},
{111,7,-11},
{112,-9,1},
{113,3,9},
{114,11,-3},
{115,-1,-11},
{116,-1,1},
{117,3,1},
{118,3,-3},
{119,-1,-3},
{120,-1,9},
{121,11,1},
{122,3,-11},
{123,-9,-3},
{124,7,9},
{125,11,-7},
{126,-5,-11},
{127,-9,-5},

};