Rotasjonsinvariante turbokoder

Christian Skauge Knudtsen
Institutt for informatikk, Universitetet i Bergen
skauge@ii.uib.no

13. juni 2003

Forord

Jeg vil først og fremst takke min veileder Øyvind Ytrehus for all hjelp han har gitt meg i løpet av hovedfaget. Vi har i samarbeid kommet frem til resultatene som presenteres i denne oppgaven. Uten hans veiledning hadde det heller ikke vært mulig å gjennomføre den. Han har alltid vært tilgjengelig når jeg har hatt behov for assistanse, noe jeg setter stor pris på.

Jeg vil også takke mine medstudenter for hjelp til LaTeX og debugging. Og ikke minst for å være med på å gjøre tiden som hovedfagsstudent til en positiv opplevelse.

Innhold

Kapittel 1

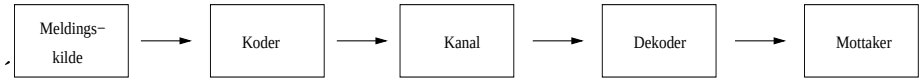
Innledning

1.1 Introduksjon

Kodeteori er et fagområde som omfatter både matematikk, informatikk og ingeniørteknikk. Det oppstod i 1940 årene fra studier av metoder for å overføre data over kommunikasjonskanaler med støy. En pioner innenfor kodeteori var Claude Shannon, som i 1948 publiserte "*A mathematical theory of communication*". I denne artikkelen viste Shannon at det for en støyutsatt kommunikasjonskanal eksisterer et tall, kalt kanalens kapasitet. Dersom man overfører data med en rate lavere enn kanalkapasiteten vil man kunne få en pålitelig dataoverføring forutsatt at gode kodings- og dekodingsteknikker brukes. Kodeteori omhandler sending av data på støyutsatte kanaler, og å gjenskape meldinger som er endret av støy. I årene som har gått har kodeteori vokst kraftig, og bruksområdene er blitt mange. Spesielt innen trådløs kommunikasjon og datalagring er kodeteori en nødvendighet. I jakten på koder som kommer nærmere Shannons kapasitet har stadig nye metoder blitt utviklet. "Klassiske" systemer som blokkoder og konvolusjonskoder har vært i bruk lenge, og de siste årene har nye paradigmer som turbokoder kommet til. Turbokoder ble introdusert i artikkelen "Near Shannon limit error-correcting coding and decoding: Turbo codes" av Berrou, Glavieux, og Thitimajshima i 1993. Denne typen koder har vist seg å komme veldig nært kanalkapasiteten på båndbreddebegrensede kanaler. De har blant annet blitt implementert i UMTS standarden for mobilkommunikasjon og DVB standarden for digital videoverføring. Det er gjort mye forskning på turbokoder siden de ble presentert. Noen aspekter ved turbokoder er det imidlertid jobbet mindre med. Et eksempel er rotasjonsinvarians.

1.2 Innledning

I alle former for kommunikasjon er støy en forstyrrende faktor. Ved sending av binære signaler over satellittkanaler, radiolinker, telefonlinjer, eller lignende vil støy kunne endre meldingen. Lagring av digital informasjon vil også være utsatt. Støyen kan ha mange årsaker: fra stråling, atmosfærisk støy og lynnedslag til forstyrrende kommunikasjonssignaler. For å motvirke dette kan man inkludere redundant informasjon i signalene i form av en feilkorrigerende kode. Slik kan man forhåpentligvis gjenopprette den opprinnelige informasjonen selv om den er blitt endret. Et enkelt kommunikasjonssystem kan beskrives skjematisk slik:



Dataoverføringsystem

Anta at systemet er binært, og i stand til å overføre to meldinger (f.eks. "ja" og "nei") over en kanal med støy. Disse to meldingene kan representeres ved "0" og "1". Dersom en melding (for eksempel "1") sendes vil det alltid være en sjanse for at støyen påvirker signalet så mye at en annen melding ("0") blir registrert hos mottakeren. Mottakeren vil imidlertid ikke ha noen indikasjon på om den mottatte meldingen er korrekt eller korrumpert av støy. Hvis man anvender en feilkorrigerende kode vil imidlertid situasjonen forandre seg. I stedet for å sende "1" kan man sende "111" og for "0" tilsvarende "000". Denne metoden har innført redundans i form av en enkel repetisjonskode. I stedet for å sende en bit over kanalen per informasjonsbit sendes nå 3 biter, noe som reduserer dataoverføringsraten til $\frac{1}{3}$ av den ukodete. Men dersom støy endrer en bit av signalet nå, for eksempel fra "000" til "010" blir situasjonen en annen. For det første kan vi nå være sikker på at signalet er endret av støy, siden "010" ikke er et gyldig kodeord. Samtidig kan vi også se at det som ble mottatt ligner mer på "000" enn "111". Vi kan nå enten be om å få meldingen sendt på nytt, eller gjette på den mest sannsynlige informasjonen. Dermed har man både fått innført feildeteksjon og korreksjon. Det er lett å se at minst to av bitene må endre verdi før meldingen kan bli dekodet feil.

Et generelt prinsipp er at dersom man i en gitt kode øker kodelengden så reduseres feilsannsynligheten. Men samtidig som kodelengden øker vokser arbeidsmengden for å dekode meldingen. I tillegg overføres færre informasjonssymboler per kodesymbol. Dersom kodene blir lange nok går feilsannsynligheten mot null, og dekodingskompleksiteten går mot uendelig. En av de store utfordringene innen kodeteori ligger i å finne gode koder av overkommelig lengde.

Kapittel 2

Teoretisk bakgrunn

2.1 Noen definisjoner

Siden de fleste kommunikasjonssystemer er kodet binært (med symbolene "0" eller "1"), vil jeg bare se nærmere på lineære koder over GF(2). I hele oppgaven antar jeg at kanalen det sendes over er en binær symmetrisk kanal eller BSC. Det innebærer at for hver enkelt bit som sendes er sannsynligheten for at den ikke mottas korrekt p. Sannsynligheten for at den mottas korrekt er gitt ved $(1\text{-}p)$. Alle bitene regnes som uavhengige, og kanalen er minneløs. Dette gir

$$P(1\ mottatt\,|\,0\,sendt) = P(\,0\ mottatt\,|\,1\,sendt) = p$$

$$P(\,0\ mottatt|\,0\,sendt) = P(\,1\,mottatt\,|\,1\,sendt) = 1-p$$

Anta at kodeord fra en kode C er sendt over kommunikasjonskanalen. Hvis ordet $\mathbf{x}$ blir mottatt kan vi beregne sannsynligheten

$$P(\ \mathbf{x}\ mottatt|\ \mathbf{c}\ sendt)$$

for alle kodeord $\mathbf{c} \in C$. Maksimal sannsynlighets dekodings (MLD) regelen konkluderer med at $\mathbf{c}_x$ er det mest sannsynlige kodeordet dersom $\mathbf{c}_x$ maksimerer sannsynligheten

$$P(\,\mathbf{x}\,mottatt|\,\mathbf{c}_x\,sendt) = \max_{\mathbf{c}\in C} P(\,\mathbf{x}\,mottatt|\,\mathbf{c}\,sendt)$$

Dersom to kodeord er like sannsynlig etter beregningen kan et velges vilkårlig.

Shannons teorem sier at feilfri dataoverføring er mulig dersom senderen ikke overskrider kanalens *kapasitet* C. Hvis vi ser på sannsynligheten for dekodingsfeil så vil denne gå mot null når kodelengden vokser, forutsatt at kodens rate R er mindre enn kanalens kapasitet C. Hvis man bruker teoremet motsatt vei kan man konkludere med at dersom R > C så er feilfri overføring umulig. For å finne hvor denne grensen går kan man se på skranken utledet av Shannon i [1]. Uttrykt i dB er den definert slik for kanaler med additiv hvit Gaussisk støy (AWGN):

$$SNR_c = 10\log_{10}\frac{2^{2R}-1}{2R}$$

der

$$SNR = 10 \log_{10} \frac{E_b}{N_0}$$

R er kodens rate (se definisjon 2.3), E_b er gjennomsnittlig mottatt signalenergi per informasjonsbit, og N_0 er enkeltsidig effekt-spektraltetthet for den Gaussiske støyen.

2.2 Lineære blokkoder

Noen definisjoner:
u = informasjonsvektoren som skal kodes.
k = lengden av **u**.
v = kodevektoren, generert ved å sende **u** gjennom koderen.
n = lengden av **v**
Det er en en-til-en korrespondanse mellom **u** og **v**. Det finnes altså inntil 2^k informasjonsvektorer, og like mange kodevektorer.

Definisjon 2.1 En blokkode av lengde n med 2^k kodeord kalles en lineær (n,k) kode hvis og bare hvis dens 2^k kodeord utspenner et k-dimensjonalt underrom av vektorrommet av alle n-tupler over kroppen GF(2).

Definisjon 2.2 En binær blokkode er lineær hvis og bare hvis mod(2) summen av to kodeord også er et gyldig kodeord.

Lineære blokkoder har fordelen at de har en lite kompleks definisjon og implementasjon. De kan beskrives ved hjelp av generatormatriser (**G**) og paritetsjekkmatriser (**H**). Radene i matrisen **G** utspenner koden. **G** kan bygges opp av k lineært uavhengige kodeord. Hvis vi tar utgangspunkt i generatormatrisen vil kodingsprosessen foregå slik: $\mathbf{u} * \mathbf{G} = \mathbf{v}$

Eksempel 1.1: En (7,4) lineær blokkode

$$[1\,0\,1\,0] \;*\; \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} = [0\,0\,1\,1\,0\,1\,0]$$

Forholdet mellom **G** og **H** kan beskrives slik: $\mathbf{G} * \mathbf{H}^\top = 0$. **G** kan endres ved elementære matriseoperasjoner uten at kodens feilkorrigerende og feildetekterende egenskaper påvirkes. Hvis **G** settes på følgende form: $\mathbf{G} = [\mathbf{P} \,|\, \mathbf{I}_k]$ der I_k er en k*k elementærmatrise, vil kodeordene få formen $\mathbf{v} = [\mathbf{p} \,|\, \mathbf{u}]$. Det vil med andre ord være mulig å lese informasjonsvektoren direkte fra kodeordene. Dette kalles *systematisk form.* Koden i eksempel 1.1 er på systematisk form, det er lett å kjenne igjen de fire bitene i **u** på slutten av **v**. Når kodeordet sendes over kanalen er det utsatt for støy som kan påvirke innholdet i **v**. Prosessen kan uttrykkes slik: $\mathbf{r} = \mathbf{v} + \mathbf{e}$. Der **r** er den mottatte vektoren og **e** er feilmønsteret, en vektor av samme lengde som **v** og som har enere i de posisjonene hvor **v** bytter verdi på grunn av støy. Dersom **e** bare inneholder nuller, oppstår det ingen feil.

Blokkoder har flere nyttige parametere. En av disse er kodens rate:

Definisjon 2.3 En kodes koderate (R) er antall informasjonssymboler pr. kodesymbol eller $\frac{k}{n}$.

Det er også nyttig å kunne vurdere hvor effektiv en kode er til å detektere eller korrigere feil. Dette bestemmes av kodens vektfordeling og avstandsegenskaper.

Definisjon 2.4 Et kodeords Hammingvekt er lik det antallet posisjoner hvor kodeordet ikke har 0'er. I eksempel 1.1 har kodeordet vekt lik 3.

Definisjon 2.5 Hamming-avstanden mellom to kodeord er lik antall posisjoner hvor de to kodeordene er ulike. I binære koder kan denne størrelsen finnes ved å addere kodeordene sammen (mod 2).

Teorem 2.1 Minimumsavstanden (d_{min}) i en kode er lik vekten av kodeordet med lavest vekt, utenom null-kodeordet. (Null-kodeordet vil alltid være med i en lineær blokkode.)

Minimumsavstanden i koden, eller d_{min} beskriver hvor effektiv koden er til å oppdage og rette feil. En blokkode med minimumsavstand d_{min} kan garantert oppdage alle feilmønstre med vekt $\leq d_{min}$ - 1, hvis den bare brukes til å oppdage feil. I tillegg vil den oppdage noen feilmønstre med større lengde. Disse feilmønstrene er alle binære n-tupler som ikke er kodeord. Det finnes $2^n - 2^k$ slike tupler. Grunnen til dette er at dersom feilmønsteret er lik et kodeord blir (mod 2) summen $\mathbf{r} = \mathbf{v} + \mathbf{e}$ et annet, men gyldig kodeord (jfr. definisjon 2.2). En slik feil vil ikke kunne oppdages.

Teorem 2.2 Hvis en lineær blokkode brukes kun til feilkorreksjon kan den rette t feil der $d_{min} = 2t + 1$

Bevis for teorem 2.1 og 2.2 finnes i [5]. I tillegg til minimumsavstanden har vektfordelingen i koden en del å si for dens egenskaper. Det er en fordel om koden har få kodeord av lav vekt, uavhengig av minimumsavstanden. Dette fordi alle kodeord tilsvarer feilmønstre som ikke kan oppdages, og lavvekts feilmønstre vanligvis er mer sannsynlige enn høyvekts feilmønstre.

For å regulere en kodes rate kan det brukes en teknikk som kalles punktering. Det består i å fjerne paritetsbiter etter et gitt mønster. Koden i eksempel 1.1 har 3 paritetsbiter og rate $\frac{4}{7}$. Dersom den første paritetsbiten punkteres vekk får koden rate $\frac{2}{3}$, og kodeordet i eksempelet blir $[0\,1\,1\,0\,1\,0]$. Å justere raten er praktisk når man bruker kraftige feilkorrigerende koder i situasjoner der feilsannsynligheten er liten. Man vil da overføre mindre data per informasjonssymbol, noe som øker effektiviteten.

2.3 Konvolusjonskoder

Konvolusjonskoder kan uttrykkes som et spesialtilfelle av blokkoder, og beskrives med generatormatrise og paritetssjekkmatrise. Karakteristisk for disse kodene er at koderen har minne. Kodingen av en enkelt bit er avhengig av de m foregående bitene, der m er størrelsen på koderens minne. Paritetsjekkbitene er altså en sum av noen utvalgte foregående informasjonsbiter. En slik kode har parametere (n,k,m) der n og k har samme betydning som for blokkoder og m er den maksimale størrelsen på koderens minne. Et viktig spesialtilfelle oppstår for $k = 1$. Da er informasjonen ikke delt opp i blokker, men kan ses på som en kontinuerlig datastrøm. For å gi en

effektiv implementasjon av konvolusjonskoder kan man bruke skiftregistre.

Når konvolusjonskoder skal beskrives kan man ta utgangspunkt i kodens generatorsekvenser. Hvis en kode har y slike kan de se slik ut:

$$g^{(1)} = g_0^{(1)}, g_1^{(1)}...., g_m^{(1)} \quad g^{(2)} = g_0^{(2)}, g_1^{(2)}, ..., g_m^{(2)} \quad .... \quad g^{(y)} = g_0^{(y)}, g_1^{(y)}, ..., g_m^{(y)}$$

Hvis koderen for eksempel har to generatorsekvenser:

$$g(1) = 1011 \ , \ g(2) = 1111$$

Hvis **u** er informasjonsvektoren, vil kodingen foregå slik når i'te bit av **u** kommer inn i koderen: (All addisjon er mod 2)

$$v(i)^{(1)} = u_i + u_{i-2} + u_{i-3} \qquad v(i)^{(2)} = u_i + u_{i-1} + u_{i-2} + u_{i-3}$$

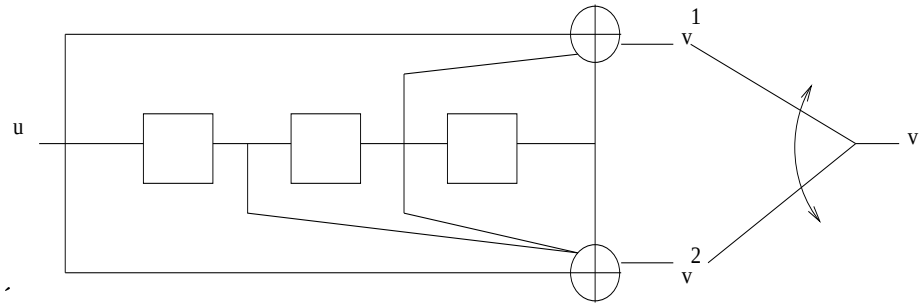
Hvis informasjonsvektoren u ser slik ut (symbolet lengst til venstre går først inn i koderen ved tid $i = 0$) :

$$u = (0110.....)$$

vil den kodede sekvensen ved $i = 3$ være:

$$v^{(1)}(3) = 0 + 1 + 0 = 1 \quad , \quad v^{(2)}(3) = 0 + 1 + 1 + 0 = 0 \quad \Rightarrow \quad v(3) = 1, 0$$

Skiftregisteret som svarer til de overnevnte generatorsekvensene ser slik ut:



En (2,1,3) Konvolusjonskoder

Sekvensene som kommer ut av koderen blir multiplekset ut på kanalen. Konvolusjonskodere kan også ha flere input eller output sekvenser. Eksempelvis vil en (5,3,4) koder ta inn 3 informasjonsekvenser og generere 5 kodesekvenser.

Generatormatrisen **G** kan bygges opp med utgangspunkt i generatorsekvensene. Hvis koderen har to slike sekvenser gjøres det slik:

$$G = \begin{bmatrix} g_0^{(1)} & g_0^{(2)} & g_1^{(1)} & g_1^{(2)} & g_2^{(1)} & \cdots \\ & & g_0^{(1)} & g_0^{(2)} & g_1^{(1)} & \cdots \\ & & & & g_0^{(1)} & \cdots \\ & & & & & \ddots \\ & & & & & \end{bmatrix}$$

På samme måte som for blokkoder kan kodingsprosessen uttrykkes som $\mathbf{u} * \mathbf{G} = \mathbf{v}$. Men når sekvensene som skal kodes blir lange, vokser $\mathbf{G}$ fort, noe som gjør denne metoden upraktisk i forhold til for eksempel skiftregisterversjonen. I tillegg blir matrisen enda mer komplisert dersom man har flere input sekvenser.
Konvolusjonskoder kan også beskrives ved hjelp av generatorpolynomer. I en (2,1,m) kode blir ligningene slik: (alle operasjoner over GF(2))

$$\mathbf{v}^{(1)}(D) \;=\; \mathbf{u}(D)g^{(1)}(D) \quad , \quad \mathbf{v}^{(2)}(D) \;=\; \mathbf{u}(D)g^{(2)}(D)$$

hvor

$$\mathbf{u}(D) \;=\; u_0 \;+\; u_1 D \;+\; u_2 D^2 \;+ \quad .....$$

er informasjonssekvensen. Kodesekvensene ser slik ut:

$$\begin{aligned}\mathbf{v}^{(1)}(D) \;&=\; v_0^{(1)} \;+\; v_1^{(1)} D \;+ v_2^{(1)} D^2 \;+ \quad ....\\ \mathbf{v}^{(2)}(D) \;&=\; v_0^{(2)} \;+\; v_1^{(2)} D \;+ v_2^{(2)} D^2 \;+ \quad ....\end{aligned}$$

Og generatorpolynomene til koden blir

$$\begin{aligned}\mathbf{g}^{(1)}(D) \;&=\; g_0^{(1)} \;+\; g_1^{(1)} D \;+\; ... \;+\; g_m^{(1)} D^m\\ \mathbf{g}^{(2)}(D) \;&=\; g_0^{(2)} \;+\; g_1^{(2)} D \;+\; ... \;+\; g_m^{(2)} D^m\end{aligned}$$

Etter multipleksing blir kodeordet:

$$\mathbf{v} \;=\; \mathbf{v}^{(1)}(D^2) \;+\; D\mathbf{v}^{(2)}(D^2)$$

I (2,1,3) koden i eksempelet over er:

$$g_1(D) \;=\; 1 + D^2 + D^3 \qquad g_2(D) \;=\; 1 + D + D^2 + D^3$$

Kodingsprosessen foregår slik hvis man bruker generatorpolynomene: (Alle operasjoner er over GF(2))

$$u = 0110 = D + D^2$$

$$\mathbf{v}^{(1)} = (D + D^2) * (1 + D^2 + D^3) = D + D^2 + D^3 + D^5 = 0111010$$

Tilsvarende for $\mathbf{v}^{(2)}$. Igjen implementeres kodingen effektivt via skiftregistre.

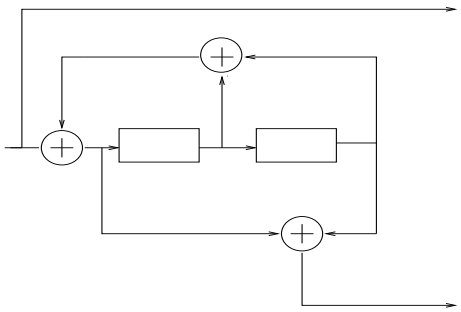
På samme måte som for blokkoder kan konvolusjonskoder settes på systematisk form. Hvis man ser på skiftregisteret vil det innebære et det går en kobling direkte fra input ($\mathbf{u}$) til output ($\mathbf{v}$). Et av generatorpolynomene til en systematisk koder vil ha grad null. For eksempel: $[g_1(D)\,,\, g_2(D)] = [1\,,\, D + D^2 + ...]$. En systematisk koder har flere fordeler. To eksempler er enklere elektronikk i koderen, og at man

ikke trenger en inverteringskrets for å finne informasjonssekvensen $\mathbf{u}$.

En svakhet med systematiske konvolusjonskoder er at siden de må ha et generatorpolynom av grad null, så begrenses valgene av disse. Dette reduserer effektiviteten til koderen. For å unngå problemstillingen kan man bruke en rekursiv systematisk konvolusjonskoder, eller RSC. Med utgangspunkt i generatorpolynomene bygges de opp slik:

$$[g_1(D)\,,\, g_2(D)] \quad \rightarrow \quad \left[\frac{g_1(D)}{g_1(D)}\,,\, \frac{g_2(D)}{g_1(D)}\right] \quad \rightarrow \quad \left[1\,,\, \frac{g_2(D)}{g_1(D)}\right]$$

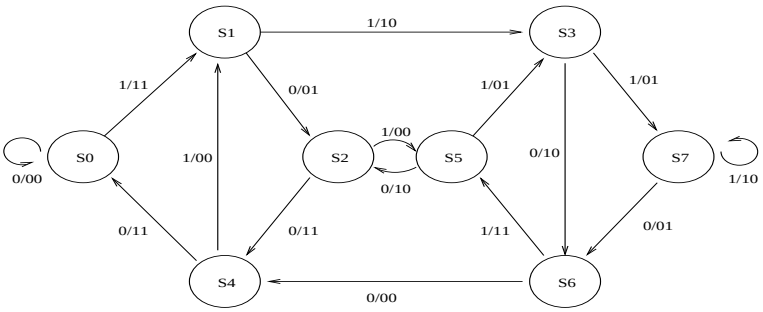
Et eksempel på en RSC



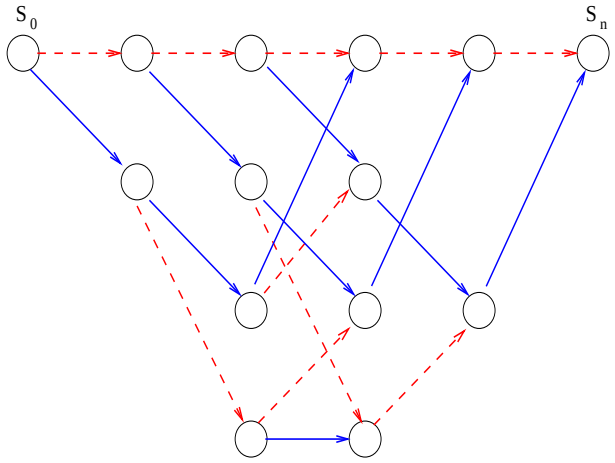
En rekursiv systematisk koder er kjennetegnet ved at de har en *feedback* kobling fra kodet output til input. Tilbakekoblingen medfører at enere som kommer inn i koderen ikke nødvendigvis forlater den etter m klokkeslag. Dette betyr at dersom en ener fulgt av en lang sekvens med nuller kommer inn i koderen vil eneren sirkulere og gi utputtsekvensen høy vekt. Derfor produserer RSC kodere vanligvis høyere vekt utputtsekvenser enn tilsvarende ikke-rekursive kodere for lavvekts inputsekvenser. Det finnes riktig nok noen bestemte bitsekvenser som fører koderen tilbake til null-tilstanden, og disse varierer med koderen. Spesielt innputt vektorer av vekt to har vist seg å være problematiske.

2.4 Dekoding av konvolusjonskoder

Siden en konvolusjonskoder er en sekvensiell krets, kan den bli beskrevet av et tilstandsdiagram. Hver innputt bit fører koderen til en av 2^k mulige tilstander. Med denne informasjonen kan alle kombinasjoner av koderens input og registerinnhold beskrives grafisk. Dette kalles et tilstandsdiagram, og det gir en fullgod beskrivelse av koderen. For (2,1,3) koderen i eksempelet over ser diagrammet slik ut:



Den første biten i figuren representerer innputt, de to påfølgende bitene er korresponderende utputt. Hvis man utvider en kodes tilstandsdiagram til å inkludere tid så får man en *trellis*. Hver tidsenhet vil da bli representert ved sitt eget tilstandsdiagram. Resultatet er en rettet graf. Den har generelt 2^m tilstander per tidsenhet, utenom de første og siste m tilstandene. En enkel trellis med 4 tilstander er vist i figuren:



For konvolusjonskoder har den initiale og den siste dybden i trellisen bare en tilstand. Dette medfører at $S_0 = \{s_0\}$ og $S_n = \{s_n\}$. En kant $\mathbf{e}$ fra tilstand S_{i-1} terminerer i tilstand S_i, $1 \leq i \leq n$. La $s^S(\mathbf{e})$ og $s^E(\mathbf{e})$ være tilstander hvor $\mathbf{e}$ henholdsvis begynner og slutter. En terminert trellis for en binær rate $\frac{1}{2}$ konvolusjonskode har lengde n $= k + v$. Trellisen i figuren over har $k = 3$ og $v = 2$. I løpet av de første k dybdene er det to kanter ut av hver node, men fra og med k 'te dybde går det bare en kant fra hver tilstand, og disse kantene er en del av en unik sti som terminerer trellisen ved å lede til nulltilstanden, S_n , i dybde n. Hver kant $\mathbf{e}$ har følgende merkelapper:

- $u(\mathbf{e})$, informasjonssymbolet assosiert med $\mathbf{e}$. Dette er vist med farger i figuren; (blå er null, rød er en).
- $c(\mathbf{e})$, paritetsjekksymbolet assosiert med $\mathbf{e}$. Dette er ikke vist i figuren.

Vurder en sti som starter i S_0 og terminerer i S_n i trellisen. Dersom man samler opp alle par av merkelapper fra hver kant man passerer får man en merkelappsekvens, noe som tilsvarer et kodeord i koden. Ved å traversere alle mulige stier fra S_0 til S_n vil de tilhørende merkelappsekvensene inneholde alle de 2^k kodeordene i konvolusjonskoden. For trellisterminering legges det til $m = k - 1$ biter til slutt. Disse er valgt slik at de tvinger koderen tilbake til nulltilstanden. For å kunne bruke trellisen i dekodingen må man ha et mål for sannsynligheten til de forskjellige stiene:

Anta at en informasjonssekvens $\mathbf{u}$ kodes til et kodeordet $\mathbf{v}$, og at den Q-ære sekvensen $\mathbf{r}$ er mottatt over en binær innputt, Q-ær utputt diskret minneløs kanal (DMC). Dekoderen må nå produsere et estimat $\mathbf{v}$' for kodeordet $\mathbf{v}$ basert på $\mathbf{r}$. En ML dekoder for en DMC velger det kodeordet som maksimerer den logaritmiske sannsynlighetsfunksjonen $\log P(\mathbf{r} \,|\, \mathbf{v})$. I [5] er det vist at

$$\log P(\mathbf{r} \,|\, \mathbf{v}) = \sum_{i=0}^{n-1} \log P(r_i \,|\, v_i)$$

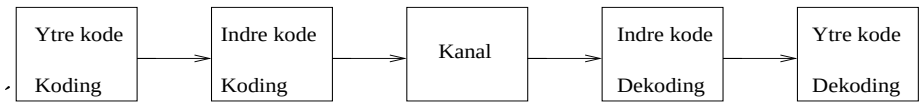
hvor $P(r_i \,|\, v_i)$ er sannsynligheten for at det oppstår en feil på kanalen. Funksjonen $P(\mathbf{r} \,|\, \mathbf{v})$ er metrikken forbundet med stien $\mathbf{v}$. Metrikk for hver enkelt kant kan også

utledes. Når dette inkluderes i trellisen blir grafen vektet, og det blir mulig å sammenligne sannsynlighetene for ulike stier. Dette legger grunnlaget for dekodingen, som kan utføres ved å finne korteste sti gjennom trellisen.

Konvolusjonskodenes spesielle struktur kan altså utnyttes for å effektivisere dekodingen. Dette er utnyttet i Viterbialgoritmen, som jobber på trellisrepresentasjonen av koden. Det er vist at Viterbi tilsvarer en dynamisk programmert algoritme som finner korteste sti gjennom en vektet graf. Det er også vist at Viterbi gir en ML dekoding av konvolusjonskoder, se [5]. Viterbialgoritmen eliminerer kandidatsekvenser i hvert steg, noe som gjør den svært effektiv. Når algoritmen implementeres i elektronisk utstyr må man ofte nøye seg med en redusert versjon for å spare minne. Det er da ikke lengre snakk om ML dekoding, men ytelsen kan likevel komme tett opptil det optimale. Det er utvilket flere slike forenklede varianter. For en detaljert beskrivelse av algoritmen og variasjoner av denne se [5].

2.5 Serielt konkatenerte koder

Det er naturlig å tenke at dersom det å kode en melding medfører redusert tap av data, hva vil skje dersom vi koder to ganger etter hverandre? I noen tilfeller gir dette en kraftig forbedring. Et eksempel er serielt konkatenerte koder:

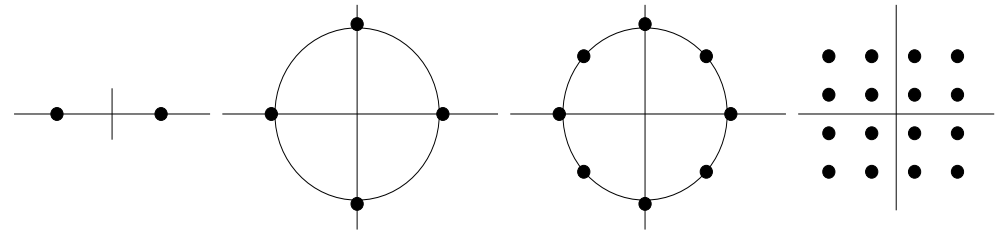


Serielt konkatenert kode

Dersom den indre koden dekoder et kodeord feil, vil den ytre koden få mange påfølgende uriktige biter, eller et feilskred. For å håndtere denne oppførselen kan man bruke to strategier: Enten kan man bruke en kode som er optimalisert for feilskreddeteksjon og -korreksjon. Eller så kan man bruke en interleaver mellom de to kodene slik at feilskredene blir brutt opp. Da bruker en isteden en ytre kode optimalisert for tilfeldige feil. Resultatet blir en $(n_i n_y\ ,\ k_i k_y)$ - kode med minimumsavstand $(d_i d_y)$. Serielt konkatenerte koder bruker vanligvis relativt enkle komponentkoder. Koderne og dekoderne til disse enkle kodene er billige og enkle å implementere. Den resulterende koden vil som oftest ha en ytelse som er på nivå med mye mer komplekse koder. Dette gjør serielt konkatenerte koder enkle og effektive.

2.6 Modulasjon

Modulasjon - demodulasjon er den signaloverføringsmetoden som tradisjonelt er brukt i blant annet modemer. (Modem = modulator - demodulator). Modulatoren mottar kodede bitsekvenser fra koderen. En kontinuerlig tone, kalt sinus bærebølge legger grunnlaget for signalet. Signaler sendes enten ved å variere frekvens, amplitude, eller ved faseskift av bærebølgen. Demodulatoren på den andre siden av kanalen gjør bølgene om til bitsekvenser igjen før resultatet sendes til dekoderen. Hvordan modulasjonen bygges opp av faseskift og amplitudeendringer kan uttrykkes ved en konstellasjon.



Noen konstellasjoner

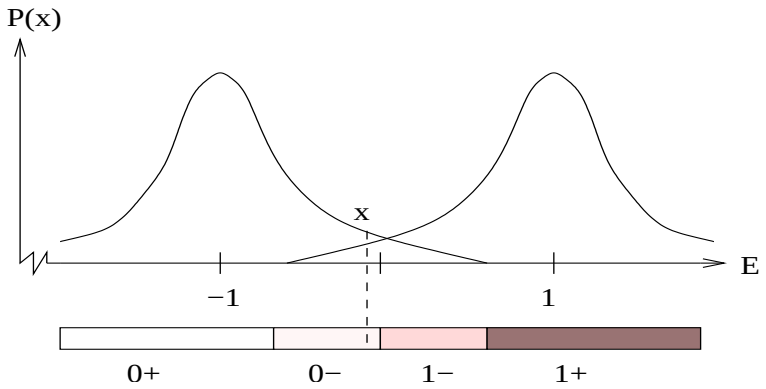
I det første eksempelet har hvert symbol en av to mulige verdier. Nyquist båndbredden til et kommunikasjonssignal er lik raten symboler overføres med. Hvis symbolene er binære krever en R_s bit per sekund rate en R_s Hz Nyquist båndbredde. Den spektrale effektiviteten til et system er definert som antall biter per sekund overført per 1 Hz båndbredde. BPSK har en spektral effektivitet på 1 bit/sek/Hz. Symbolfeilraten er gitt ved sannsynligheten for at støy påvirker signalet så mye at det ligner mer på et annet. ML dekoding velger det euklidsk nærmeste hvis alle symboler er like sannsynlige.
Et vanlig problem i kommunikasjonssituasjoner er at man ønsker en høyere datarate enn tilgjengelig båndbredde tillater. For å bøte på dette må man øke den spektrale effektiviteten. Men det har sin pris: Energien som kreves for å overføre et symbol er gitt ved $d = \sqrt{E}$, der d er euklidsk avstand mellom konstellasjonspunktet og origo. Hvis man utvider BPSK til 2^m signalpunkter uniformt fordelt på x-aksen så kan man sende m biter per symbol. For hver ekstra bit man kan sende firedobles gjennomsnittlig energimengde, dersom d_{min} skal holdes fast. Dette gjelder også de sirkulære konstellasjonene i eksempel 2 og 3. Den mest effektive av disse konstellasjonen er den kvadratiske. For hver ekstra bit dobles gjennomsnittlig energimengde per symbol. Konstellasjonene i eksempel 2, 3 og 4 er todimensjonale. De bygges opp av to ortogonale bærebølger. Slike bølger vil ikke påvirke hverandre underveis og kan demoduleres separat. Multidimensjonale konstellasjoner har gjort det mulig å produsere modemer med raskere overføringsrate.

2.7 Soft decision - Hard decision

Den enkleste metoden for å tolke demodulerte signaler kalles hard decision. Når man vurderer signalene, er det bare å undersøke hvilket symbol det som ble mottatt lignet mest på. Deteksjonen er enkel: velg det symbolet som ligner mest. Med denne strategien mister man mye informasjon. I mer avanserte systemer kan man sende mottatt kanalinformasjonen til dekoderen. Dette gir et mer nyansert bilde av de mottatte symbolene. Dette kalles soft decision dekoding. Med slik informasjon kan dekoderen bygge opp sannsynlighetsberegninger for alle mottatte symboler. Med utgangspunkt i figuren vil HD gi "0", men SD for eksempel kan gi

$$P(x|0) = 0.10 \quad P(x|1) = 0.08$$

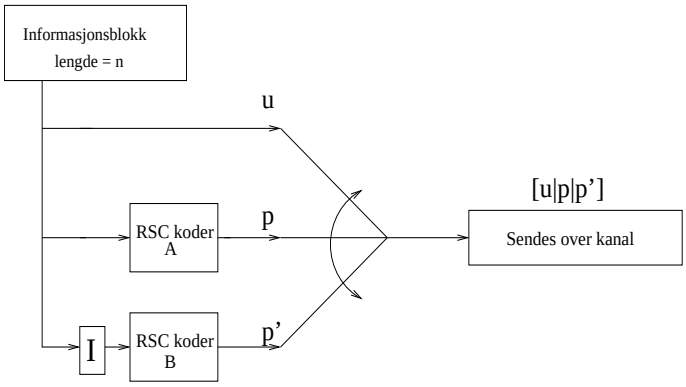
Soft decision er mye mer effektivt i praksis, men krever avanserte kretser og algoritmer i dekoderen.



I stedet for å bruke HD med to verdier (0 eller 1) kan man tillate flere verdier. F.eks. 4 slike: 0_+ , 0_- , 1_- , 1_+. Der '1_-' er en "dårlig" ener. Ved å legge til en tilstrekkelig nyansert kvantisering vil man få resultater som nærmer seg SD uten at det krever så avansert elektronikk. Man må imidlertid bruke kretser og algoritmer i dekoderen som kan gjøre bruk av den økte informasjonsmengden.

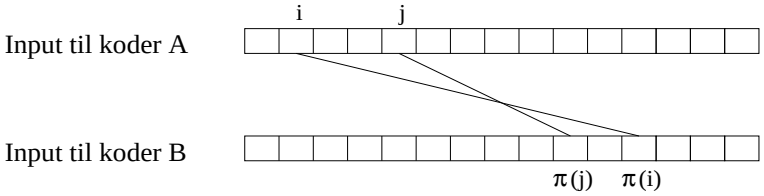
2.8 Turbokoder - koding

Turbokoder ble først beskrevet i [2]. Turbokoder er parallellkonkatenerte konvolusjonskoder, for eksempel generert av koderen i figuren under. Koderkretsen består av to komponentkoder knyttet sammen av en interleaver. De to komponentkodene kan godt være identiske, og er i eksempelet rate $\frac{1}{2}$ systematisk rekursive konvolusjonskoder. Koderne kan betegnes A og B. Selve kodingen foregår slik: Informasjonsvektoren **u** kommer inn i koder A, og behandles på normal måte. Samtidig går også **u** inn i interleaveren, og permuteres. Den permuterte informasjonsvektoren, **u'** behandles så av dekoder B. Dekoder A produserer [**p**] og dekoder B [**p'**]. Til slutt multiplekses **u, p,** og **p'** ut på kanalen. Kodesekvensen blir sammensatt slik: $[u_0, p_0, p'_0, u_1, p_1, p'_1, u_2, p_2, p'_2 .....]$ Dersom man ikke bruker punktering får den totale koderkretsen en rate på $\frac{1}{3}$. Det legges også til noen biter til slutt for trellisterminering.



Det brukes rekursive komponentkoder for å forbedre kodens vektegenskaper. Anta at output fra koder A er en lavvekts kodesekvens. For å forhindre at koder B også genererer en lavvekts sekvens brukes interleaveren til å permutere input. Håpet er at dekoder B produserer en høy(ere) vekts sekvens, noe som gir det totale kodeordet middels vekt. Det er lett å se at interleaveren er en viktig komponent av koderen. Den vil ha direkte innflytelse på kodens avstandsegenskaper. En god interleaver unngår flere lavvekts kodeord, som igjen forbedrer kodens BER. Det er

gjort mye forskning på interleaverdesign, og det har blant annet vist seg at tilfeldige interleavere ofte har bra ytelse. En av de viktigste tingene en interleaver må unngå er *sykler.* En førsteordens sykel bygges opp av to tall, i og j der $0 \leq i < j < k$. Lengden av sykelen blir $l = l(i,j) = j - i + |\pi(i) - \pi(j)|$. Se figur:

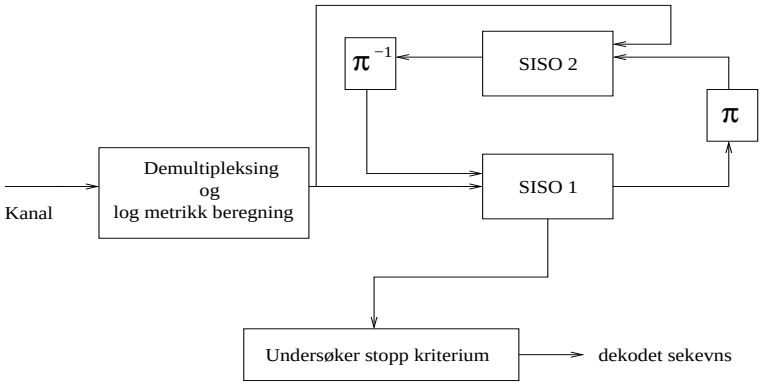


En kort sykel medfører at beregningene i dekoderen ikke er uavhengige, noe som kan skape feilforplantning. Også minimumslengden til andre- og tredjeordens sykler kan det være interessant å regulere. En strategi for å bygge gode interleavere er å generere mange tilfeldige interleavere, og forkaste de som ikke oppfyller visse krav til for eksempel minimale sykellengder. Interleaveren kan også brukes til å spre bitmønstre som fører komponentkoderne inn i nulltilstanden. Gitt en komponentkode kan den skreddersys for å oppnå dette best mulig.

2.9 Turbokoder - dekoding

Koderen til en turbokode har en relativt enkel beskrivelse. Dekoderen er ikke like enkel. Hvis man ser på en ML dekoder for en rate $\frac{1}{2}$ konvolusjonskode, og antar en ordlengde på N, der N$\geq$1000 (ikke uvanlig for en turbokode). Dersom kodens struktur ignoreres, så vil en "naiv" ML dekoding måtte sammenligne mottatt sekvens med 2^Nkodeord, og velge det kodeordet som ligner mest. En slik fremgangsmåte er beregningsmessig svært tung. Shannons teori beskriver det samme: Koders effektivitet øker med voksende N, men det samme gjør dekodingskompleksiteten. I stedet for en slik "brute-force" fremgangsmåte, kan man bruke Viterbidekoding, som utnytter konvolusjonskodens struktur, og gir en kraftig kompleksitetsforenkling i forhold til en "naiv" dekoding. Viterbialgoritmen tillater eliminering av kandidatsekvenser, men kan imidlertid ikke brukes til å dekode turbokoder. Interleaveren kompliserer kodens struktur i en slik grad, at turbokoder ligner mer på blokkoder enn konvolusjonskoder. Men det finnes alternativer til ML-dekoding, nærmere bestemt suboptimale dekodingsmetoder. Disse metodene forsøker å bryte ML dekodingen opp i flere enkle steg.

En analyse av feilsannsynlighet forutsetter bruk av MLD, noe som ser ut til å være for komplekst for turbokoder. I forsøket på å finne en alternativ fremgangsmåte gjenoppfant Berrou & co. iterativ dekoding. En slik dekoding ser ut til å yte svært tett opp til MLD ved moderat til høy SNR, men det finnes imidlertid ingen formelle bevis for dette. Iterert dekoding baserer seg på samarbeid mellom modulene som dekoder hver av komponentkodene. Hver dekoder har to typer informasjon tilgjengelig: Kanalverdier for hvert mottatt symbol, og "a priori" informasjon generert av den andre dekoderen. Å få feil a priori informasjon kan bli katastrofalt for dekodingsprosessen. Dessverre er det umulig å vite på forhånd hvilken a priori informasjon som er korrekt. Iterativ dekoding løser dette ved å utveksle "ekstrinsik" informasjon (defineres senere) i stedet for komplett informasjon mellom komponent dekoderne. I løpet av flere iterasjoner vil forhåpentligvis det gjennomsnittlige bidraget fra riktig ekstrinsik informasjon overdøve effekten av feil ekstrinsik informasjon.



En turbodekoder

Anta en stokastisk variabel X som tar verdien x med sannsynlighet p(x), kanskje betinget av et eller annet. Det kan være praktisk å representere denne sannsynligheten som en "log likelihood ratio" eller LLR. Det er en funksjon av typen

$$\lambda(x) \;=\; \log\left(\frac{p(x)}{1 \;-\; P(x)}\right)$$

Selv om det ikke formelt er helt korrekt, kalles ofte LLR'ene for "informasjon" i litteraturen. Bruk av LLR gjør at operasjoner på sannsynlighetene blir additive i stedet for multiplikative, noe som er beregningsmessig mer effektivt.

La $\underline{\lambda}(x)$ være en blokk av LLR'er for eksempel $(\lambda(x_1)\,,\,\lambda(x_2)\,,\,....\,,\,\lambda(x_m)$ for en eller annen vektor $(x_1\,,\,.....\,,\,x_m)$. Mottaker demultiplekser mottatt sekvens til tre blokker med LLR'er:

- $\underline{\lambda}^{(R:U)} \;=$ Kanalverdier av informasjonssymboler
- $\underline{\lambda}^{(R:C)_A} \;=$ Kanalverdier med paritetsymboler fra dekoder A
- $\underline{\lambda}^{(R:C)_B} \;=$ Kanalverdier med paritetsymboler fra dekoder B

Det å bestemme sannsynlighetsfordelingene krever en nøyaktig verdi for SNR, noe som er lett å beregne hvis man har en stabil kanal. Symboler som fjernes ved punktering, betegnes med en 0'er i LLR blokken. For hver komponentkode brukes en *Soft-in-soft-out* eller SISO dekoder for å produsere a posteriori informasjon $\underline{\lambda}^{(komplett)}$. Denne størrelsen er basert på a priori informasjon $\underline{\lambda}^{(A)}$, kanalinformasjonen $\underline{\lambda}^{(K)}$og på det strukturelle forholdet mellom bitene generert av koderen. På blokknivå er:

$$\underline{\lambda}^{(Komplett)} = \underline{\lambda}^{(A)} + \underline{\lambda}^{(K)} + \underline{\lambda}^{(E)}$$

Hvis vi trekker $\underline{\lambda}^{(A)}$ og $\underline{\lambda}^{(K)}$ fra $\underline{\lambda}^{(Komplett)}$ får vi ekstrinsik informasjonen $\underline{\lambda}^{(E)}$. Denne størrelsen representerer hvor mye informasjon som kom ut av denne runden i dekoderen. Ekstrinsik informasjon $\underline{\lambda}^{(E)}(u_j)$ på en spesifikk informasjonsbit u_j er uavhengig av a priori informasjonen $\underline{\lambda}^{(A)}(u_j)$ på den samme biten. Ved starten av dekodingsprosessen blir blokkene $\underline{\lambda}^{(R:U)}$og $\underline{\lambda}^{(R:C)_A}$ sendt til SISO$_A$ mens den initiale a priori informasjonen $\underline{\lambda}^{(A)}$ er null for alle symbolene. Deretter blir blokkene $\underline{\lambda}^{(R:U)}$ (etter interleaving), og $\underline{\lambda}^{(R:C)_B}$ sendt til SISO$_B$, sammen med ekstrinsik informasjonen $\underline{\lambda}^{(E)}$ fra forrige steg som nå brukes som a priori informasjon etter interleaving i B. Etter dette blir ekstrinsik informasjonen fra SISO$_B$ deinterleavet

og sendt til SISO_A som a priori informasjon i 2. runde av dekodingen.

Etter hvert som dekodingen går sin gang vil ekstrinsik informasjonen gradvis forbedres og informasjonen for hele blokken brukes til å gi et endelig estimat for hver enkelt informasjonsbit $u_i\,,\ 0 \leq j \leq K$.

Dersom resultatet konvergerer må man ha et kriterium for å vurdere når man er fornøyd med resultatet, slik at dekodingen ikke utfører unødig mye arbeid. I tillegg trengs et absolutt maksimum antall iterasjoner for å fange opp de tilfellene dekodingsprosessen ikke konvergerer. I en kommunikasjonssituasjon kan forsinkelsen skapt av dekodingen være problematisk. I tillegg til at enkelte rammer kan behandles i maksimalt antall iterasjoner, jobber turbokodene på store blokker av data. Den totale forsinkelsen kan gjøre turbokoder uegnet for applikasjoner som er avhengig av en jevn datastrøm og kort forsinkelse (for eksempel telefoni).

2.10 SISO algoritmen

SISO blokkene er den viktigste komponenten i iterert dekoding. Det finnes flere algoritmer for slike blokker, blant annet BCJR og Soft Output Viterbi. SISO algoritmen jeg bruker er en additiv variant basert på BCJR, hentet fra [4]. Denne algoritmen jobber på en trellis-representasjon av koden (se kapittel 2.4). Rekursjonen beveger seg først fremover gjennom trellisen, og så bakover, og til slutt slås resultatene sammen. Når algoritmen jobber seg fremover, begynner den med $\alpha_0(s_0) = 0$, og beregner:

$$\alpha_j(s) = \overset{*}{\underset{e:s^E(e)=s}{\max}} \{\alpha_{j-1}(s^s(e)) \ + \ u(e)\lambda_j^{(A)} \ + \ u(e)\lambda_j^{(R:U)} \ + \ c(e)\lambda_j^{(R:C)}\}$$

der første ledd representerer foregående tilstand, andre ledd a priori verdier, og de to siste kanalverdier.

$\max^* \alpha_j$ som opererer på T tall $\alpha_1 .... \alpha_T$ er sum operasjonen $\log\left[\sum_{t=1}^{T} e^{at}\right]$.

Størrelsen $\alpha_j(s)$ er en LLR som representerer sannsynligheten for at komponentkoderen er i trellis tilstand s ved tid j, gitt at tilgjengelig foregående kanalinformasjon og a priori informasjon gjennom alle symboler fram til tid j. For å unngå numeriske problemer normaliseres α (og β)-verdiene på hver dybde. Med normalisering menes her at man finner den største α-verdien og trekker den fra alle verdiene. Den største verdien blir lik null etterpå, og alle andre verdier negative. Tilsvarende gjøres for β-verdiene. Den andre rekursjonen begynner med $\beta_n(s_n) = 0$, og beregner:

$$\beta_j(s) = \overset{*}{\underset{e:s^E(e)=s}{\max}} \{\beta_{j+1}(s^E(e)) + u(e)\lambda_{j+1}^{(A)} + u(e)\lambda_{j+1}^{(R:U)} + c(e)\lambda_{j+1}^{(R:C)}\}$$

Siste steg av BJCR algoritmen er å beregne for alle $j\ ,\ 0 \leq j \leq K$ ekstrinsik-log metrikk for j'te informasjonsbit $\lambda_j^{(E)}$. For u = {0,1} beregn

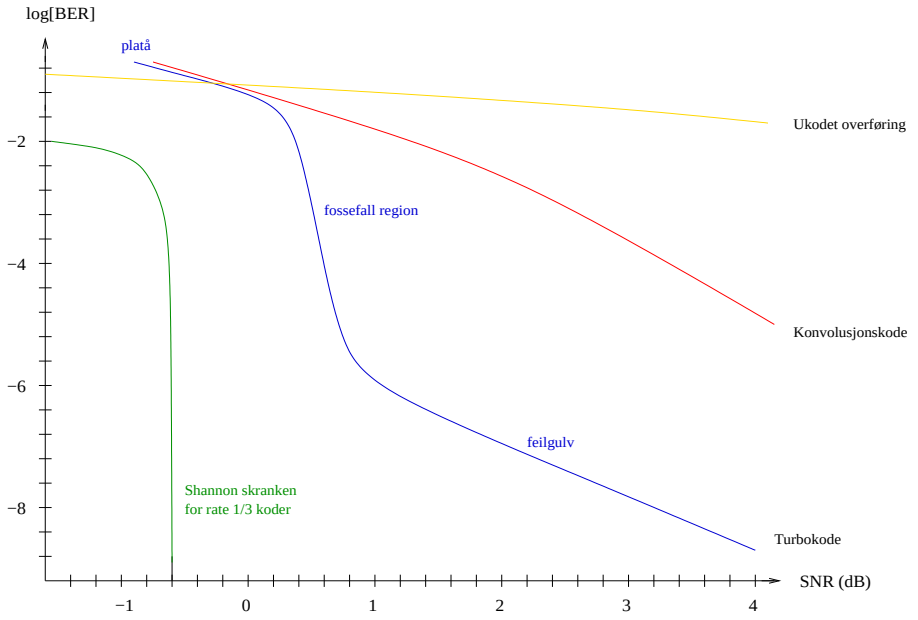
$$\mu(u, j) = \overset{*}{\underset{e:s^E(e)=s}{\max}} \{\alpha_{j-1}(s^s(e)) + \beta_j(S^E(e)) + u(e)\lambda_j^{(R:U)} + c(e)\lambda_j^{(R:C)}\}$$

Og så

$$\lambda_j^{(E)} = \mu(1, j) - \mu(0, j)$$

En ting som er verd å nevne er at ekstrinsik informasjonen $\lambda_j^{(E)}$ til j'te informasjonsbit er uavhengig av a priori verdien $\lambda_j^{(A)}$på j'te bit.

2.11 Turbokoders egenskaper



Figuren illustrerer forbindelsen mellom BER og SNR. Shannonskranken er inntegnet ved ca. -0.55dB for rate 1/3. Det er også tegnet inn BER kurver for ukodet sending, en rate 1/3 konvolusjonskode med 64 tilstander, og en rate 1/3 turbokode med en enkel tilfeldig interleaver. Informasjonslengden $N = 1000$.

Som man ser i figuren over er turbokoder knapt bedre enn ukodet dataoverføring ved lave signal/støyforhold. Så følger *fossefallregionen* hvor kodens feilrate faller fort med forbedret signal/støyforhold. Dette flater til slutt ut i *feilgulvet* der kodens effektivitet styres av noen få, mest sannsynlige feilvektorer. Å undersøke en turbokodes ytelse ved dårlige signal-støy forhold er noe man kan gjøre effektivt ved datasimulering. Men etter hvert som feilsannsynligheten synker kreves det stadig mer arbeid for å gjøre en grundig analyse. For å få et riktig bilde må man observere hundrevis av dekodingsfeil, og ved høye signal-støyforhold kan det ta veldig lang tid før en feil i det hele tatt inntreffer. En annen fremgangsmåte er å utlede beskrankninger for kodens ytelse. Disse baseres på kodens vektfordeling, og er først og fremst avhengig av lavvektskodeordene.

En ytelsesskranke for turbokoder kan beregnes slik: Hvis man antar bruk av MLD kan rammefeilsannsynligheten til enhver feilkorrigerende kode utrykkes ved *union bound* for høye SNR:

$$FER \leq \sum_{w=d}^{N} A_w Q\left(\sqrt{2wR \cdot SNR}\right)$$

hvor A_w er lik antallet kodeord med Hammingvekt w, og d er kodens minimumsavstand. $Q()$ er den komplementære feilfunksjonen

$$Q(x) = \frac{1}{2\pi} \int_x^\infty e^{-\frac{t^2}{2}}\, dt$$

Tallsettet $\{A_w \,|\, w = 0\,, \ldots.\,, N\}$ er vektfordelingen til koden. Hvis man vi se på

bitfeilsannsynligheten, så kan den utrykkes ved

$$BER \leq \sum_{w=d}^{N} \frac{B_w}{K} Q\left(\sqrt{2wR \cdot SNR}\right)$$

hvor $B_w = \sum_{i=1}^{w} A_{i,w-i}$ er den totale informasjonsvekten av alle kodeord av vekt w. *Union bound* kan også forbedres. Dette gir en tettere skranke, og tillater å begrense summasjonene til færre ledd:

$$FER \leq \sum_{w=d}^{d+m} A_w Q\left(\sqrt{2wR \cdot SNR}\right) + \Omega_1$$

Og bitfeilraten blir

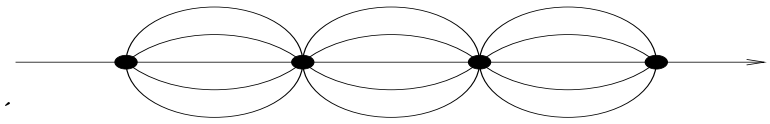
$$BER \leq \sum_{w=d}^{d+m} \frac{B_w}{K} Q\left(\sqrt{2wR \cdot SNR}\right) + \Omega_2$$

der Ω er noe som går mot null ved høyt SNR, og m er et lite heltall. Ved feilgulvet er det tilstrekkelig å beregne de første par leddene av summasjonen for å få et nøyaktig bilde av bitfeilsannsynligheten.

Inngående analyser av turbokoder viser at de generelt har dårligere minimumsavstand enn klassiske koder med tilsvarende rate og lengde. De har imidlertid mye færre kodeord av lav vekt enn en klassisk kode. Hvis tilfeldige interleavere brukes vokser d_{min} langsomt, samtidig som antall kodeord med lav vekt sakte reduseres med økende interleaverlengde K. Hvor nært en turbokode er Shannonskranken er av hengig av informasjonslengde, valg av komponentkoder og interleaver, samt implementasjonsdetaljer i dekoderne.

2.12 Kodet modulasjon

Hvis man betrakter trellisen til vanlig modulasjon, får man en enkel trellis med en tilstand per tidsenhet. Dette fordi hvert symbol er uavhengig av det forrige, og fordi det ikke er noen form for redundans.



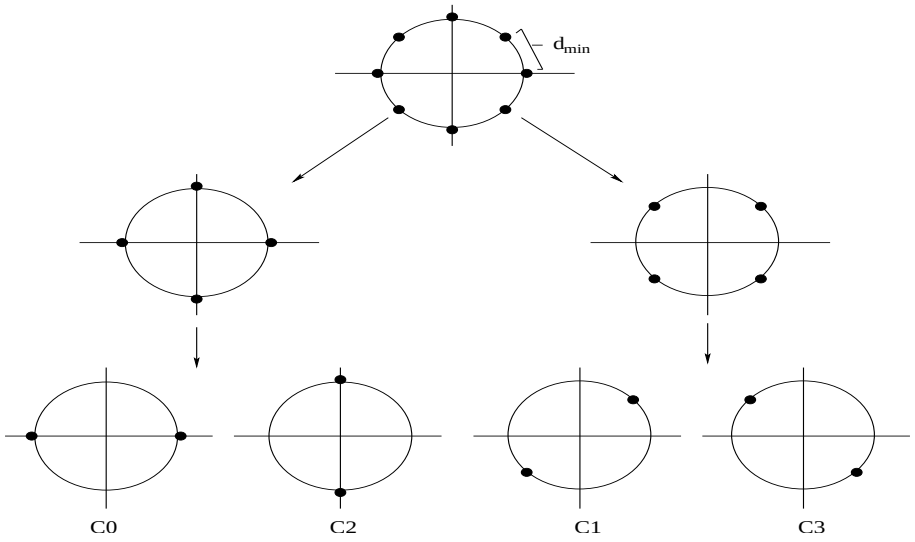
Trellis for ukodet modulasjon

De feilkorrigerende kodene får sine egenskaper fra å inkludere redundant informasjon som gjør noen mønstre ugyldige. For å forbedre ytelsen til modulasjon lanserte Ungerboeck [8] kodet modulasjon basert på de samme ideene. Ungerboecks metode introduserer redundans uten å øke signalbåndbredden via tre steg:

1. Legg en redundant bit til alle m kildebiter
2. Ekspander signalkonstellasjonen fra 2^m til 2^{m+1} symboler

3. Bruk den m+1'te bitkodete kildeblokken til å velge signalsett i den utvidete konstellasjonen

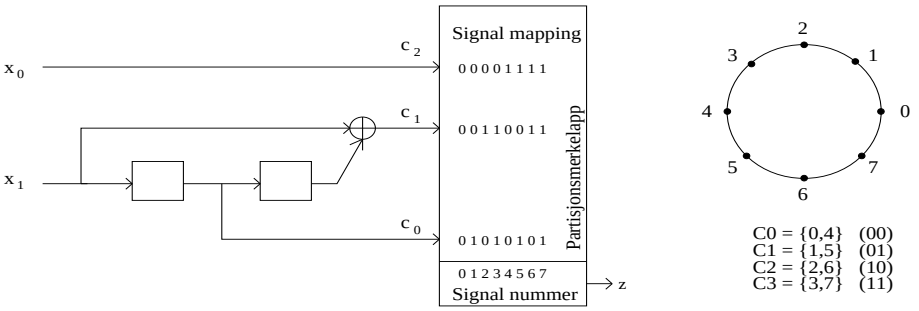
Symboloverføringsraten blir fremdeles den samme som for ukodet. Redundans oppstår fordi en 2^{m+1} konstellasjon brukes til å sende med en rate på m biter/sek/Hz. Det spesielle i metoden er hvordan de m informasjonsbitene knyttes til de 2^{m+1} signalpunktene i den utvidete konstellasjonen. Metoden kalles sett-partisjonering. Konstellasjonen partisjoneres til en serie mindre subkonstellasjoner. Figuren viser partisjonering av 8-PSK:



Minimumsavstanden innad i konstellasjonen øker etter partisjonering:

- 8 punkter : $d_{min} = 2\sin\left(\frac{\pi}{8}\right) = \Delta$
- 4 punkter : $d_{min} = 2\sin\left(\frac{\pi}{4}\right) \approx 1.85\Delta$
- 2 punkter : $d_{min} = 2\sin\left(\frac{\pi}{2}\right) \approx 2.61\Delta$

k av de m informasjonsbitene kodes med en rate $\frac{k}{k+1}$ konvolusjonskode. Resten (k+1) av bitene brukes til å velge en av de 2^{m+1} konstellasjonene i (k+1)'te nivå av partisjonstreet. De siste (m-k) bitene brukes til å velge punkt i den valgte konstellasjonen. En slik koder skaper en mer avansert trellisstruktur på de kodete sekvensene. Dette tillater igjen feilkorreksjon. Dekoding foregår ved hjelp av for eksempel Viterbialgoritmen. Et av Ungerboecks enklere eksempler er vist i figuren under:



Man ekspanderer en 4-PSK til et 8-PSK system. Begge gir en spektral effektivitet på 2 bits/sek/Hz. 8-PSK konstellasjonen partisjoneres til 4 subkonstellasjoner

{C0, C1, C2, C3} (Se figur over). Hver subkonstellasjon har en tilhørende 2-bit merkelapp, som korresponderer med en unik 2-bit output fra rate $\frac{1}{2}$ koderen. Den siste informasjonsbiten velger et av de to punktene i subkonstellasjonen. Forbedringen et slikt system gir over ukodet modulasjon kan uttrykkes ved *asymptotic coding gain* eller γ:

$$\gamma = \frac{(S_{ukodet})}{(S_{kodet})} \cdot \frac{\left(d^2_{fri/kodet}\right)}{\left(d^2_{fri/ukodet}\right)}$$

der S er gjennomsnittlig energi per punkt. Ved 8-PSK varianten beskrevet ovenfor er $\gamma = 2$ eller 3.01 dB. Se [7] for beregningen og mer om emnet.

2.13 Rotasjonsinvarians

Hvis man bruker symmetriske signalsett som PSK eller QAM vil systemet være sårbart for uønsket rotasjon. Dersom mottakeren kommer ut av fase vil den mottatte sekvensen oppfattes som rotert i det euklidske planet. Hvis denne roterte sekvensen ikke tilhører koden kan dette medføre en lang rekke dekodingsfeil. For å håndtere dette fenomenet kan man bruke rotasjonsinvarians. En rotasjonsinvariant kode sørger for at en rotert kodesekvens alltid er en annen kodesekvens, og at alle rotasjoner av kodesekvensen dekodes til den samme informasjonssekvensen. Målet er å gjøre dekodingsprosessen helt upåvirket av eventuell rotasjon. *Rotasjonskanalen* er utsatt for støy, men i tillegg også for rotasjon, som oppstår dersom mottakeren velger feil fase: $\mathbf{r} = \mathbf{v} + \mathbf{e} + \rho$.

I [3] presenteres tre strategier for å oppdage og korrigere uønsket rotasjon av signalene:

1. Opptrening: Åpner kommunikasjonen med et testsignal som "måler" signalrotasjonen.
   - Hvis andre kanalkarakteristikker også må måles, kan dette gjøres samtidig
   - Hvis forholdene endres underveis må man teste på nytt
   - Krever avansert elektronikk
2. Bruke en kode med kodeord som ikke er rotasjoner av hverandre
   - Krever mange symboler
3. Bruke en rotasjonsinvariant kode.
   - Mottakeren sender problemet videre til dekoderen.

I praksis vil man kunne lage koder som er rotasjonsinvariante ved faseskift på et gitt antall grader, typisk 180, 90 eller 45. Dette kan forbedre ytelsen til en kode, men man ofrer antallet kodeord. Hvis man eksempelvis ser på 180 graders rotasjonsinvarians vil det være to symboler som skal dekodes til samme informasjon. For alle slike par (x,y) gjelder det at å rotere konstellasjonspunktet x 180 grader vil gi posisjonen til punktet y. Videre i dekodingsprosessen vil det ikke være mulig å se forskjell på slike par. I mange situasjoner vil det å legge til rotasjonsinvarians gjøre en kode mer effektiv, inntil et visst punkt. Det er i praksis sjeldent lønnsomt å gjøre en kode mer enn 45 grader rotasjonsinvariant. I dag brukes ofte numeriske søk etter gode koder. Ved å legge til visse kriterier til disse søkene, kan man få rotasjonsinvariante koder. Noen algebraiske definisjoner av rotasjonsinvarians:

Definisjon 2.6 Et signalsett $S \subset R^{2N}$ er Φ - invariant hvis $\rho(S) = S$ for alle $\rho \in \Phi$.

Definisjon 2.7 En partisjon $Y = \{ y_1 , y_2 , \ldots , y_n \}$ av et signalsett S er Φ -

invariant hvis $\rho(Y) = Y$ for alle $\rho \in \Phi$, hvor $\rho(Y) = \{ \rho(y_1) , \rho(y_2) , ... , \rho(y_n) \}$. Hvis $\Phi = R(S)$ så er Y rotasjonsinvariant.

Definisjon 2.8 En kode C over en Φ-invariant partisjon Y er Φ-invariant hvis $\rho(\mathbf{c}) \in C$ for alle $\rho \in \Phi$ og for alle $\mathbf{c} \in C$. Hvis $\Phi = R(S)$ så er C en rotasjonsinvariant kode.

Man kan vurdere om en kode er rotasjonsinvariant ved å undersøke dens graf. Alle koder kan beskrives ved hjelp av en rettet graf (se kapittel 2.4). En kodes *Fischer cover* er en unik graf G som genererer C og oppfyller to krav: alle nodene er distinkte og alle kantene som forlater en node har forskjellige merkelapper. For de fleste trellis koder laget for AWGN kanaler, er Fiscer coveret identisk med kodens minimale trellisdiagram. Symmetri i denne grafen betyr at koderen er rotasjonsinvariant:

Teorem 2.3 En kode C over en Φ-invariant partisjon Y er Φ-invariant hvis og bare hvis dens *Fischer cover* er Φ-invariant.

Bevis finnes i [3].

Kapittel 3

Oppgaven

3.1 Om oppgaven

Målsetningen med oppgaven var å bygge en rotasjonsinvariant turbokode. Grunnen til at jeg valgte å se nærmere på denne problemstillingen er at det er publisert svært lite forskning på dette feltet tidligere. Det var derfor et åpent spørsmål i hvilken grad dette var mulig å få til. Siden turbokoder brukes i stadig flere sammenhenger var resultatene av dette arbeidet interessant å få klarhet i. I løpet av oppgaven prøvde jeg ut to forskjellige metoder for å oppnå rotasjonsinvarians. Jeg så først på om de fungerte, deretter vurderte jeg ytelsen til metodene.

Jeg vil først gi en beskrivelse av metodene jeg vil prøve ut. Deretter vil jeg gjennomgå implementasjonen av metodene, og til slutt vil jeg bruke datasimulering for å fastslå hvordan de fungerer.

3.2 Beskrivelse

Teorem 2.3 forteller at man kan vurdere om en gitt kode er rotasjonsinvariant ved å undersøke dens graf. Problemet med å bruke denne fremgangsmåten på turbokoder er at koderen har en svært komplisert graf. Den er bygget opp av to komponentkodere knyttet sammen av en interleaver, der interleaveren anbefales å være stor, og gjerne tilfeldig. Det går greit å sette opp grafer til komponentkoderne, men når interleaveren kommer med blir det hele mye vanskeligere. Dette er kanskje en av de viktigste årsakene til at så lite forskning er gjort på dette feltet. Selv korte og svært strukturerte interleavere gir svært kompliserte grafer, og disse vil i tillegg ha dårlig ytelse. For å unngå hele problemstillingen valgte jeg å prøve metoder som var uavhengig av kodens graf. Fremgangsmåtene baserte seg heller på "smarte triks", som kunne brukes med en vilkårlig turbokode. I løpet av oppgaven testet jeg to forskjellige metoder. Jeg brukte til enhver tid modulasjon med BPSK, og begrenset rotasjonsinvariansen til 180 grader. Med et slikt oppsett vil eventuell 180 graders rotasjon medføre at kanalverdiene bytter fortegn, noe som også betyr at bitverdiene inverteres. De modifiserte dekoderne må da kunne avgjøre om de mottatte kanalverdiene er korrekte (ikke rotert) eller har feil fortegn (rotert). Da begge metodene innebærer en del ekstra prosessering, vil en del av oppgaven gå ut på å vurdere om økningen i kjøretid er akseptabel. Et annet viktig spørsmål som skal undersøkes er hvordan det å legge til rotasjonsinvarians påvirker kodens ytelse.

3.3 Metode 1 - Dobbel dekoding

Det mulig å lage en 180-grader rotasjonsinvariant dekoding med et lite triks. Man kan dekode på vanlig måte, og så vurdere resultatet. Dersom dekodingen gir et dårlig resultat (antydet av f.eks. maksimalt antall iterasjoner gang på gang) kan man invertere kanalverdiene og dekode på nytt. Kombinert med den overnevnte teorien vil dette oppdage og rette eventuell 180 graders rotasjon, men det vil medføre en kraftig forsinkelse fordi dekodingsprosessen må kjøres to ganger. Det er sannsynlig at dekodingen med uriktige bitverdier går maksimalt antall iterasjoner i tilnærmet alle tilfeller, og det gjør det forhåpentligvis enkelt å oppdage hvilken dekoder som er ute av fase. Et unntak kan være ved dårlige signal-støyforhold, da er det kanskje vanskelig å avgjøre hvilken dekoding som jobber med riktig datasett. Siden denne fremgangsmåten vil medføre en kraftig økning i kjøretid, kan et kompromiss for å øke ytelsen være å sette det maksimale antallet iterasjoner så lavt som mulig. Men settes dette antallet for lavt vil det gjøre det enda vanskeligere å finne ut hvilken dekoder som er i fase.

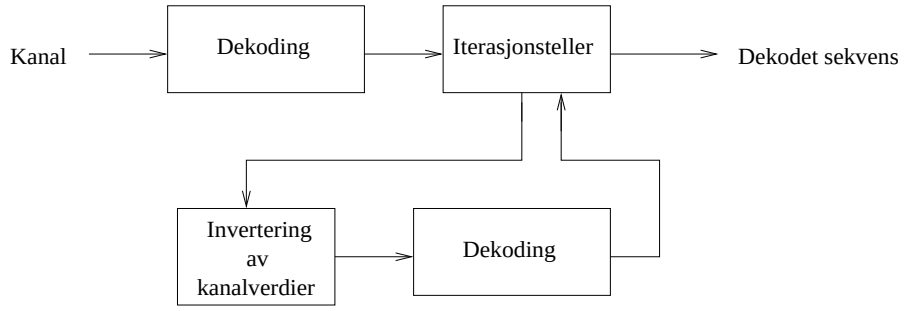
3.4 Implementasjon av dobbel dekoding

Pseudokoden til algoritmen ser slik ut:

Algorithm 1 Dobbel dekoding

```
Dekod mottatt sekvens
lagre resultat
undersøk om antall iterasjoner per ramme ≥ terskel (= max. antall iterasjoner?)
hvis ja
  inverter mottatt sekvens og dekod på nytt.
  lagre resultat og tell antall iterasjoner.
  returner resultat med lavest gjennomsnittlig antall iterasjoner
hvis nei
  returner resultat.
```

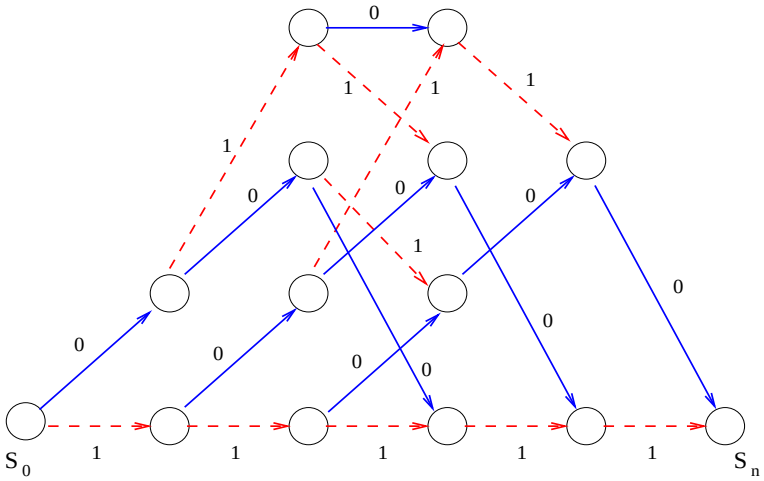
I praksis burde denne metoden ikke gi økt bitfeilrate ved moderat til høyt SNR siden det er snakk om en fullverdig dekoding. Men ved dårlige signal-støyforhold er det imidlertid mulig at det blir vanskelig å avgjøre hvilken dekoding som er i fase, noe som vil medføre en økning i feilsannsynligheten. Dekodingsprosessen blir slik:



Denne metoden er enkel å implementere, og vil ikke medføre endringer i selve koden. Hele implementasjonen kan legges i kontroll-programmet som post-prosessering.

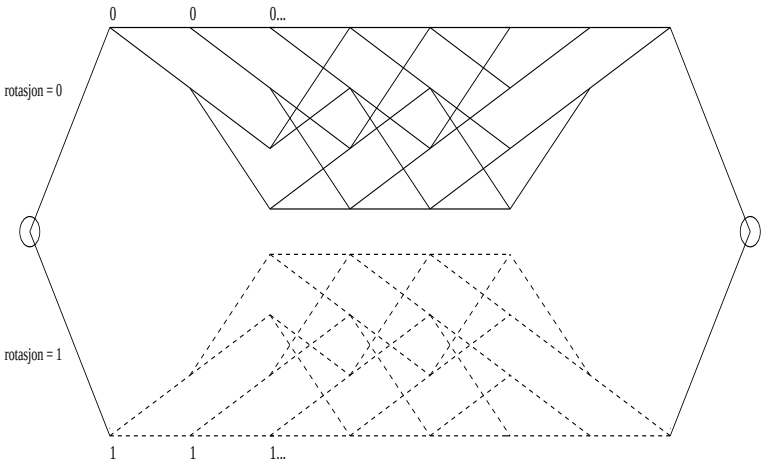
3.5 Metode 2 - Skyggetrellis

Den andre metoden er mer avansert enn den første. Fremgangsmåten er som følger: Beskrivelsen tar utgangspunkt i et vanlig turbokodeoppsett. Lengden av informasjonsvektoren økes med en. Denne siste biten går aldri inn i koderen. Dersom den er en ener, så vil alle bitene i den kodete sekvensen inverteres, og hvis den er en nuller skjer ingen ting. Kodet sekvens sendes så over kanalen. Dekoderen vet nå ikke om mottatt vektor er invertert eller ikke. For å finne ut av det kan den bruke følgende modell: Dersom man lager en speiling av trellisen til en kode, og inverterer bitene på alle kantene, vil man få en trellis som kan brukes til å dekode en invertert kodesekvens.



Kodeord [10010] –> Invertering –> [01101]

En slik "skyggetrellis" kan slås sammen med en vanlig trellis, ved å knytte dem sammen i S_0 og S_n. Det vil da finnes stier gjennom den sammensatte trellisen som tilhører både normale og inverterte kodeord:



Resultatet blir en utvidet kode der alle inverterte kodeord også er gyldige kodeord. For eksempel medfører **0**-kodeordet i den opprinelige koden at et **1**-kodeord finnes i den utvidete. Dersom kanalverdiene er invertert vil den korrekte stien gjennom trellisen finnes i skyggetrellisen, ellers vil den være i den normale. Siden koden fremdeles representeres ved hjelp av en trellis vil normale kodings- og dekodings-

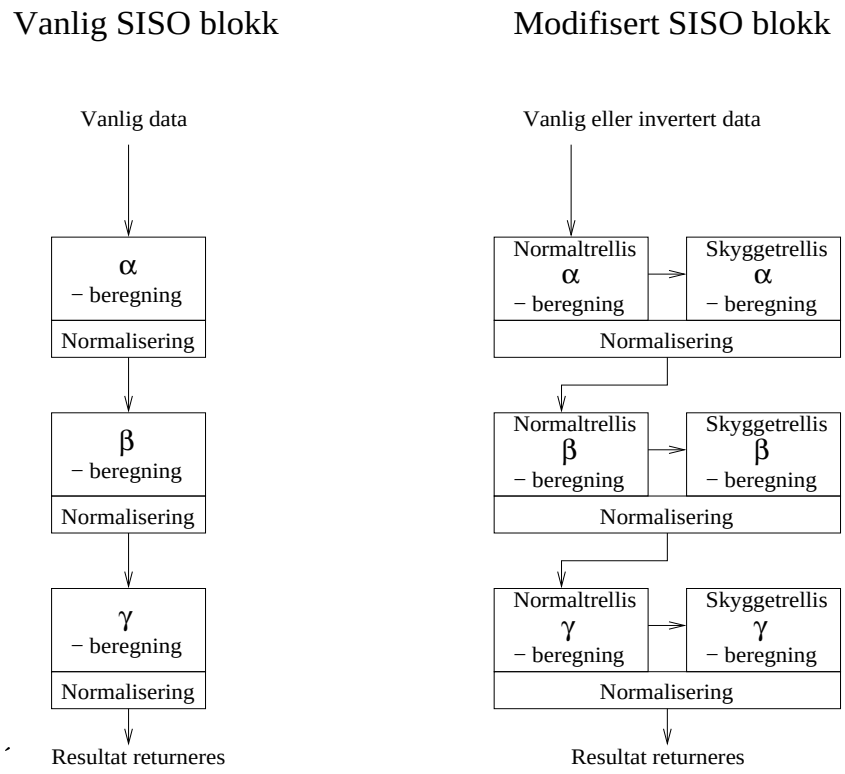
teknikker kunne brukes med minimal modifikasjon.

Brukt sammen med den overnevnte teorien vil man oppnå 180 graders rotasjonsinvarians. Dekoderen må kunne dekode riktig, selv om kanalverdiene har feil fortegn, noe den nye trellisen vil kunne håndtere. Det vil imidlertid introduseres større feilsannsynlighet når antallet gyldige kodeord økes (se kapittel 2.2). I tillegg vil kodens vektfordeling endres. Nå må både maksimum- og minimumvekt i den opprinnelige koden undersøkes, siden invertere høyvektskodeord blir lavvektskodeord. Det er en fare for at en interleaver som øker maksimumsvekten gjør mer skade enn nytte, og dermed medfører at den utvidete koden blir mindre effektiv. Dette vil bli undersøkt senere i oppgaven. Selve implementasjonen av skyggetrellisen kan legges i enten trellisrepresentasjonen eller i SISO blokkene. Jeg har valgt å legge endringene i SISO blokkene i simuleringsprogrammet.

Det er kanskje mulig å skape høyere grad av rotasjonsinvarians ved å utvide denne fremgangsmåten, nærmere bestemt ved å øke antallet tvillingtrelliser. Dersom man bruker to tvillingtrelliser sammen med QPSK vil man kanskje kunne oppnå 90 graders rotasjonsinvarians. Jeg vil ikke gå inn på dette i denne oppgaven, men bare konsentrere meg om 180 graders rotasjonsinvarians.

3.6 Implementasjon av skyggetrellis

For å gjøre dekodingen rotasjonsinvariant, vil jeg modifisere SISO blokkene slik at de jobber med både den vanlige trellisen og skyggetrellisen. SISO blokkene gjør da alle de rekursive operasjonene på dobbelt så mye data. I implementasjonen av SISO blokkene brukes det tre tabeller som alle representerer trellisen. Disse tabellene har dimensjoner lik (trellislengde * maksimal trellisdybde), og brukes til å lagre resultater i α, β, og γ-beregningene. I den modifiserte versjonen blir laget dobbelt så lange α, β, og γ-tabeller. De får nå dimensjoner ((2*trellislengde) * maksimal trellisdybde). I den ene delen av α-tabellen beregnes metrikken til stiene i normaltrellisen, og i den andre delen beregnes metrikken for skyggetrellisen. Tilsvarende for β og γ tabellene. Deltabellene normaliseres sammen for å finne den beste stien i den totale trellisen. Til slutt blir de to γ-deltabellene vurdert mot hverandre. Den stien i den totale trellisen som har best metrikk presenteres til den andre SISO blokken.



Skjematisk beskrivelse av SISO blokker

Jeg vil også la SISO blokkene utveksle informasjon om rotasjon på samme måte som de utveksler informasjon om bitverdier. Dette gjøres ved hjelp av et rotasjonsparameter som er beskrevet i kapittel 3.7 Programmet vil testes både med og uten denne informasjonsutvekslingen, og jeg vil vurdere simuleringsresultatene for å se om den er hensiktsmessig. Utover dette skal det skal ikke være nødvendig å gjøre noen endringer i turbokoden.

3.7 Rotasjonsparameteret

For at de to SISO blokkene skal kunne utveksle informasjon om signalet er i fase eller ikke valgte jeg å lage et parameter for dette formålet. Det er basert på metrikken innad i SISO blokkene. Hvilken trellis som har stien med den beste metrikken (og hvor mye bedre den er) skal formidles til den andre SISO blokken. Den deltrellisen som har best metrikk ville få et tillegg i α, β, og γ-tabellene i den andre SISO blokken. Den deltrellisen som kommer dårligst ut vil få metrikken redusert tilsvarende.

Pseudokoden for informasjonsutvekslingen blir som følger:

Algorithm 2 Bruk av rotasjonsparameteret

Rotasjonsparameteret initialiseres til null i starten på hver ramme
For hver iterasjon:
  Motta rotasjonsparameter (r1) fra den andre SISO blokken
  Beregn nytt rotasjonsparameter i denne SISO blokken (r2)
  Beregn SISO rekursjonene med parameter (r1)
  forkast (r1)
  send (r2) til den andre SISO blokken

Parameteret som kalkuleres i en SISO blokk brukes ikke til beregningene innad i denne. Selve verdien av parameteret beregnes fra α-tabellen etter rekursjonen. Algoritmen er utformet som en vanlig α beregning, uten bruk av rotasjonsparameter. Etter rekursjonen beregnes parameteret slik fra den ferdig utfylte tabellen:

$$rot = \left(\sum_{i=0}^{2^m} \alpha_n(s_i)\right)_{normaltrellis} - \left(\sum_{i=0}^{2^m} \alpha_n(s_i)\right)_{skyggetrellis}$$

Der n er lik trellislengde og m antall tilstander i trellisen. Programkoden for beregning av parameteret blir slik:

for $0 \leq i <$ antall tilstander
rot = ($\sum$ ($\alpha_{normaltrellis}$[tabellengde][i])) - ($\sum$ ($\alpha_{skyggetrellis}$[tabellengde][i]))

Parameteret blir altså positivt hvis normaltrellisen har best metrikk og negativt hvis skyggetrellisen har best metrikk. Deltrellisen som inneholder den antatt beste stien premieres i neste SISO blokk. Pseudokoden for hele operasjonen blir slik:

Algorithm 3 SISO blokk med rotasjonsparameter

For hver SISO blokk:
Motta rotasjonsparameter (r1) fra andre SISO blokk
α beregning uten rot parameter.
  beregn rot parameter (r2) fra α tabellen etter rekursjonen.
  forkast α beregning.
  Lagre (r2) parameteret til senere.
ny normaltrellis α bergning, legg til (r1) parameter fra andre SISO blokk
  skyggetrellis α bergning, trekk fra (r1) parameter fra andre SISO blokk
tilsvarende β beregninger med (r1) parameter fra andre SISO blokk
tilsvarende γ beregninger med (r1) parameter fra andre SISO blokk
forkast rot parameter fra andre SISO blokk (r1)
send rotasjonsparameter (r2) beregnet i steg tre til den andre SISO blokken.

Hvis rot $= +1$ vil alle verdiene i $\alpha_{normaltrellis}$ få et tillegg på $+1$ pr. kolonne, mens alle verdiene i $\alpha_{skyggetrellis}$ vil reduseres med 1 pr. kolonne.

I kildekoden blir dette utført inne i rekursjonen:

```
double gam = (Alpha[i-1][oldstate] +rot);
```

og for den roterte versjonen:

```
double gam = (Alpha[i-1][oldstate + numberofstates] -rot);
```

Absoluttverdien av tillegget til en sti gjennom trellisen er altså (rot*trellislengde).

En slik beregning øker kjøretiden på programmet en del. For å vurdere om rotasjonsparameteret ga noe utbytte testet jeg beregninger med en versjon av programmet der dette ikke var implementert. Jeg ville også vurdere om økningen i kjøretid var akseptabel.

3.8 Problemer med SISO modifikasjonen

Da jeg holdt på å implementere endingene i SISO blokkene hadde jeg innledningsvis en del problemer. Det viste seg at dersom jeg overførte en informasjonssekvens med bare 0'ere (som kodes til **0**) ville dekodingen alltid konkludere med at kodeordet var rotert. Dette medførte at dersom jeg slo av rotasjon og bare sendte 0-informasjon så fikk jeg 100% feil. Da jeg undersøkte nærmere oppdaget jeg at **0** vektoren alltid hadde en metrikk som var mye lavere enn en tilsvarende kodet sekvens med bare 1'ere, som den ble sammenlignet med. Metrikken i **0** sekvensen økte svært langsomt i forhold til **1** sekvensen. De store verdiene i metrikken til **1** sekvensen førte til at den alltid gikk seirende ut av SISO blokken, uansett rotasjon. For alle stier gjennom trellisen er det slik at metrikkens numeriske størrelse er avhengig av hvor stien går. De numeriske metrikktilleggene vokser med økende dybde i trellisen. Slik som SISO blokkene var modifisert var det gitt på forhånd hvilken sti som kom til å få best metrikk, nemlig den som gikk dypest i trellisen. Dette medførte en feilrate på ca. 50% ved simulering for gode signal-støyforhold, siden halvparten av kodeordene alltid vil bli oppfattet som rotert og andre halvparten som ikke-rotert, uansett faktisk rotasjon.

Problemene var forårsaket av at den initiale SISO modifiseringen var feil. Den var programmert slik at to stier gjennom en og samme trellis ble sammenlignet. Og innad i denne trellisen var det metrikkstørrelsen som avgjorde hvilken sti som gikk seirende ut. Når jeg fikk ordnet problemet oppstod skyggetrellisen, og SISO blokkene vurderte endelig en dobbelt så stor trellis. Feilen i kildekoden lå her:

```
gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j]*(-1)));
```

mens korrekt var:

```
gam+=( Mult((1-codebits[j]),ApriCode[n*(i-1)+j]));
```

Etter denne korreksjonen fungerte SISO blokkene som forventet.

3.9 Oversikt over simuleringsprogrammet

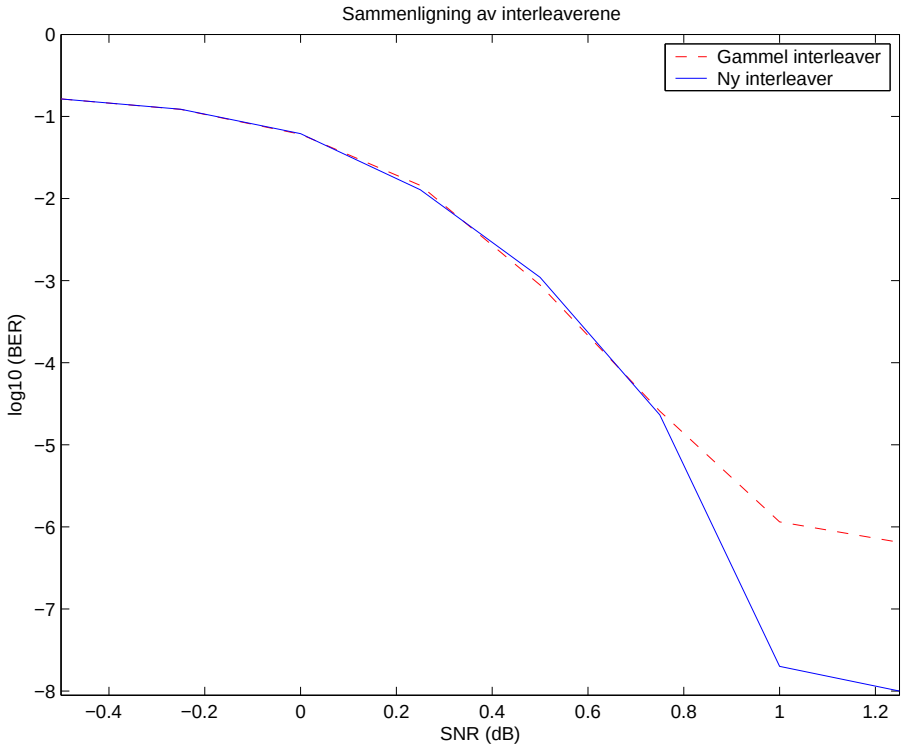
Simuleringsprogrammet "SimulatorQAM" simulerer trådløs kommunikasjon over en AWGN-kanal. Programmet utfører følgende operasjoner: Først genereres tilfeldig data, som skal brukes som informasjon. Deretter kodes informasjonen med valgt feilkorrigerende kode. Så moduleres den kodete sekvensen og sendes ut på kanalen. Signalet utsettes for støy. Sekvensen demoduleres og dekodes. Resultatet av dekodingen vurderes så mot den opprinnelige informasjonssekvensen. Det føres regnskap på antallet feil, målt signal-støyforhold og til slutt kodes den mottatte informasjonssekvensen på nytt, for å sammenligne den med mottatt kodesekvens. Simuleringen foregår enten til et oppgitt antall rammer er overført, eller til maksimalt antall rammefeil er overskredet.

Turbokoden jeg bruker i simuleringene er "g42" som er bygget opp av to identiske (2,1,4) systematisk rekursive kodere. De har generatorsekvenser $g(1) = 1101$, $g(2) = 1111$. Jeg har valgt informasjonslengde n = 1250, og punkterer bare vekk informasjonsbiten fra dekoder B med mindre annet er spesifisert. Hver ramme behandles i inntil 50 iterasjoner. For å gjøre rotasjonsinvariansen så enkel som mulig har jeg valgt BPSK modulasjon. 180 graders rotasjon medfører som nevnt at bitverdiene inverteres.

Programmet eksisterer i fire versjoner. En versjon er en standard turbokode. Den andre versjonen er en rotasjonsinvariant turbokode som inkluderer beregning av et rotasjonsparameter i SISO blokken. Denne vurderingen av rotasjonen presenteres til den andre komponentdekoderen, som beskrevet i kapittel 3.7. Denne versjonen utfører en del ekstraarbeid og har følgelig en del dårligere kjøretid. Den tredje versjonen er også rotasjonsinvariant men har ingen eksplisitt beregning av rotasjon. Dette programmet vil bare finne korteste sti gjennom totaltrellisen, og basere alle vurderinger på det. Den siste versjonen er en standard turbokode som dekoder to ganger dersom antallet iterasjoner er høyt. Den dekodingen som har færrest antall iterasjoner antas som korrekt.

3.10 Interleavere

For å undersøke om det oppstod problemer dersom maksimalvekten økte, konstruerte jeg to interleavere. En som ville gi moderat ytelse, og en interleaver med strenge parametere som skulle gi bedre ytelse. Før jeg begynte simuleringene, sammenlignet jeg resultatene fra en test av de to interleaverene. Resultatet er plottet i grafen under. Informasjonslengden n =1250, og koderaten er $\frac{1}{3}$.



Den nye interleaveren var åpenbart bedre, og forbedret kodens ytelse betraktelig. For detaljer om interleaverene se appendiks.

Kapittel 4

Resultatanalyse

4.1 Simulering

Den viktigste målsetningen med simuleringene var å sammenligne de modifiserte metodene med tilsvarende ikke-rotasjonsinvariante turbokoder. Først og fremst for å finne ut i hvilken grad rotasjonsinvariansen påvirket kodens ytelse. Simuleringen foregikk med SNR mellom -2 og +3 dB. Alle simuleringene ble kjørt inntil 350.000 rammer og informasjonslengde $n = 1250$. Rammene ble behandlet i inntil 50 iterasjoner, og simuleringen på et signal-støyforhold ble avbrutt når det hadde oppstått 500 rammefeil. Simuleringene ble utført med to forskjellige interleavere som har forskjellig ytelse (se 3.10). Jeg målte også kjøretiden til de forskjellige variasjonene av programmet underveis. En annen ting som ble undersøkt var ytelse med og uten rotasjonsparameter. I hvilken grad rotasjonsparameteret påvirket kjøretiden ble også vurdert. Jeg satt $P(rotasjon) = 0.5$ for hver ramme i rotasjonsinvariante metoder. En begrensning i programmet er at rotasjon bare påvirker hele rammer. Dette er kanskje ikke så realistisk, men det påvirker ikke resultatene utover at noen problemstillinger unngås. For å kunne utlede ytelsesskranker for kodene, undersøkte jeg også kodenes vektfordeling. Dette ble gjort med et spesiallaget program, se kapittel 4.3.

Jeg brukte koderate på $\frac{1}{3}$ i simuleringene med mindre noe annet er spesifisert.

4.2 Måling av kjøretid

Jeg undersøkte hvordan rotasjonsinvariansen påvirket kjøretiden ved å måle tiden på simuleringene. Det ga disse resultatene:

Variant	Relativ kjøretid
Normal	1
Rotasjonsinvariant uten rot parameter	1.3 - 1.5
Rotasjonsinvariant med rot parameter	2.8 - 3.0
Dobbel dekoding ved -2 dB	3
Dobbel dekoding ved +5 dB	30

Resultatene er basert på simuleringer både i området med 100% feil og maksimalt antall iterasjoner (mindre enn -2 dB) og i området med 100% riktig dekoding, og lavt antall iterasjoner (5dB+). Dette gjør de mest mulig sammenlignbare. Testene er kjørt på stor kodelengde, og med høyt antall rammer for at initialiseringsoperasjoner skal spille inn minst mulig. Resultatene i begge regionene var identiske for alle metoder unntatt dobbel dekoding. Denne metoden har bedre ytelse i situasjoner med

100% feil fordi den relative mengden ekstraarbeid da er mye lavere (50 iterasjoner + 50 iterasjoner). Ved gode signal-støyforhold viser denne metoden tydelig sin svakhet (2 iterasjoner + 50 iterasjoner). Dersom dobbel dekoding skulle implementeres i praksis ville det nok vært mulig å lage optimaliseringer som forhindret at maksimalt antall iterasjoner ble kjørt gang på gang ved gode signal-støyforhold. Men hvis rotasjonen veksler hyppig ville det vært vanskelig å unngå en høy kjøretid selv med optimaliseringer. Tabellen viser også at skyggetrellismetodens kjøretid nesten dobles dersom rotasjonsparameteret skal brukes. Dette er på grunn av flere initialiseringer og rekursive kall i SISO blokkene.

4.3 Beregning av kodens vektfordeling

Jeg kjørte også et program for å finne vektfordelingene i koden med de forskjellige interleavere. For å beregne ytelseskranker til kodene må man kjenne kodenes lavvektskodeord (se kapittel 2.11). Vektfordelingsprogrammet genererte all mulig innputt av vekt inntil en gitt verdi. Så ble denne lavvekts innputten kjørt gjennom koderen, og vekten til de korresponderende kodeordene ble registrert. Jeg kjørte programmet for innputtvekt ≤ 35, noe som burde være nok til å finne de mest kritiske lavvekts kodeordene. Informasjonsvekten til et kodeord er lik vekten av dets korresponderende informasjonssekvens. Total informasjonsvekt er summen av informasjonsvektene til alle kodeord med en gitt vekt. Denne størrelsen brukes i beregningen av "Union bound". Med interleaver 1 fikk jeg disse resultatene:

Vekt	Antall kodeord	Total informasjonsvekt
0	1	0
12	1	1
28	1	4
30	4	13
32	24	46
33	1	4
34	43	89
35	2	8

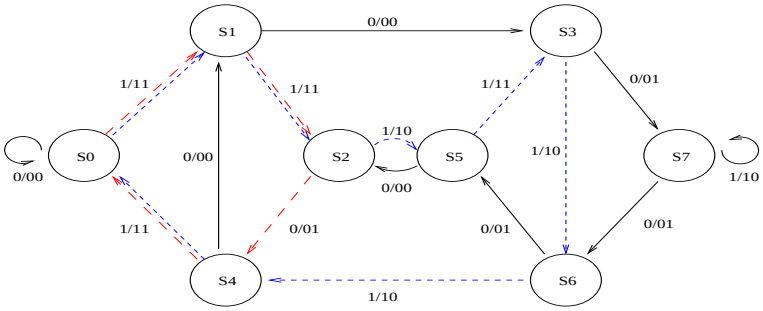
Med interleaver 2 fikk jeg disse resultatene:

Vekt	Antall kodeord	Total informasjonsvekt
0	1	0
28	4	16
30	1	3
32	16	64
33	2	8
34	5	15
35	2	8

Dette viser at koden har en minimumsvekt på henholdsvis 12 og 28 med de to interleaverene. Det forklarer den store forskjellen i ytelse mellom de to. Med utgangspunkt i disse resultatene beregnet jeg union bound for de to interleaverene ved rate $\frac{1}{3}$. Dette er en nyttig skranke å bruke fordi det tar veldig lang tid å få nok feil til å vurdere ytelsen ved høyt SNR med simulering.

4.4 Vurdering av kodens maksimalvekt

Når jeg hadde funnet minimumsavstanden i kodene kunne jeg begynne å se litt på den opprinnelige kodens maksimalvekt. Dersom maksimalvekten skulle redusere kodens minimumsavstand må den være ekstremt høy: Med kodelengde $n = 1250$ og rate $\frac{1}{3}$ blir kodelengden 3750. For å redusere minimumsavstanden må det finnes kodeord av denne lengden med mindre enn henholdsvis 12 og 28 nuller. Dette er svært høye vekter. Tilstandsdiagrammet til komponentkoderne er slik:



Tilstandsdiagram for "g42" komponentkoden

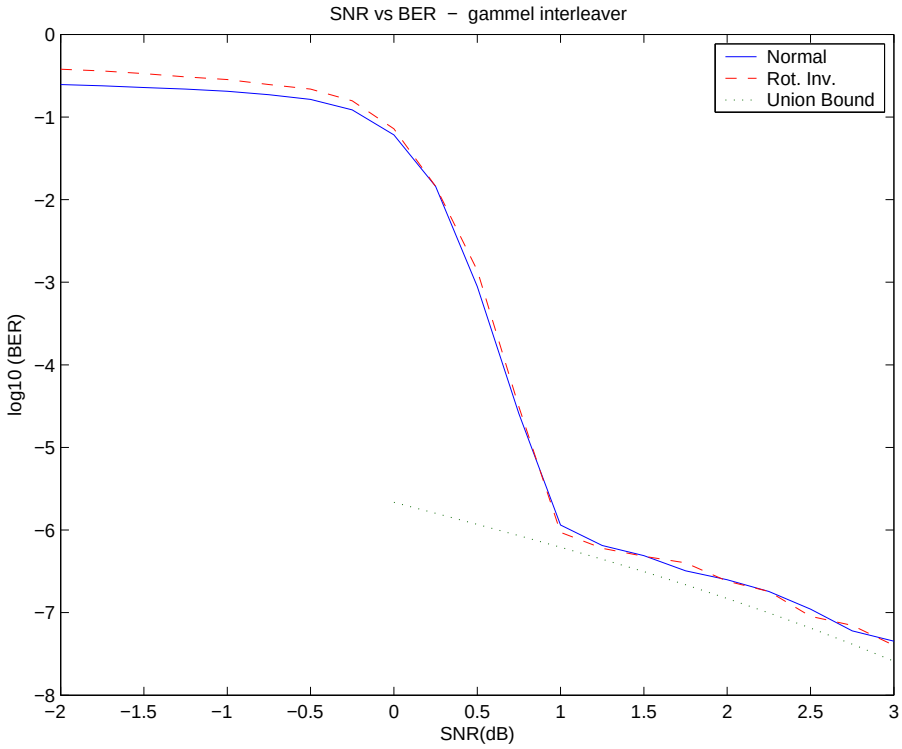
De to syklene som gir høyest vekt er merket med farge i figuren. Den røde sykelen har lengde 4. Innputt er 1101 og utputt blir 11 11 01 11 (2 nuller per 12 biter). Den blå sykelen har lengde 7. Innputt er 1111111 og utputt blir 11 11 10 11 10 10 11 (3 nuller per 21 biter). Det er alt for mange nuller med i disse syklene for å komme i nærheten av tilstrekkelig høye maksimalvekter ved rate $\frac{1}{3}$. I tillegg gjelder dette bare en enkelt komponentkoder. Interleaveren og den andre koderen må også tas med i betraktningen. Det er imidlertid mulig å punktere på en slik måte at man lager problemer. Dersom informasjonssekvensen er en repetisjon av 1101, og punkteringskartet 0101 brukes vil komponentkoderen gi utputt $\underline{\mathbf{1}}$. Et annet problematisk punkteringskart er dersom informasjonsvektoren $\underline{\mathbf{1}}$ brukes sammen med punkteringskartet 11 11 10 11 10 10 11. Dette vil også gi $\underline{\mathbf{1}}$ som utputt fra koderen.

Dersom begge komponentkodene punkteres slik at de kan gi $\underline{\mathbf{1}}$ som utputt vil koden ikke fungere. For at skyggetrellismetoden skal kunne fungere er den avhengig av at $\underline{\mathbf{1}}$-kodeordet ikke er en del av den opprinnelige koden. Dersom $\underline{\mathbf{1}}$-kodeordet er tilstede vil det finnes et identisk kodeord i den vanlige og den utvidete delen av koden, noe som blir katastrofalt for dekodingen. Skulle $\underline{\mathbf{1}}$-kodeordet finnes bare i en av komponentkodene vil det skape problemer, men den andre komponentkoden vil fremdeles fungere, og den vil kanskje kunne redde dekodingen. Hva som skjer dersom punkteringskartet 0101 brukes på en av komponentkoderne (slik at $\underline{\mathbf{1}}$-kodeordet finnes i **en** av komponentkodene) vil bli undersøkt i simuleringen. Generelt ved implementasjon av skyggetrellismetoden er det viktig å velge punkteringen slik at denne typen problemer ikke oppstår.

4.5 Simuleringsresultater

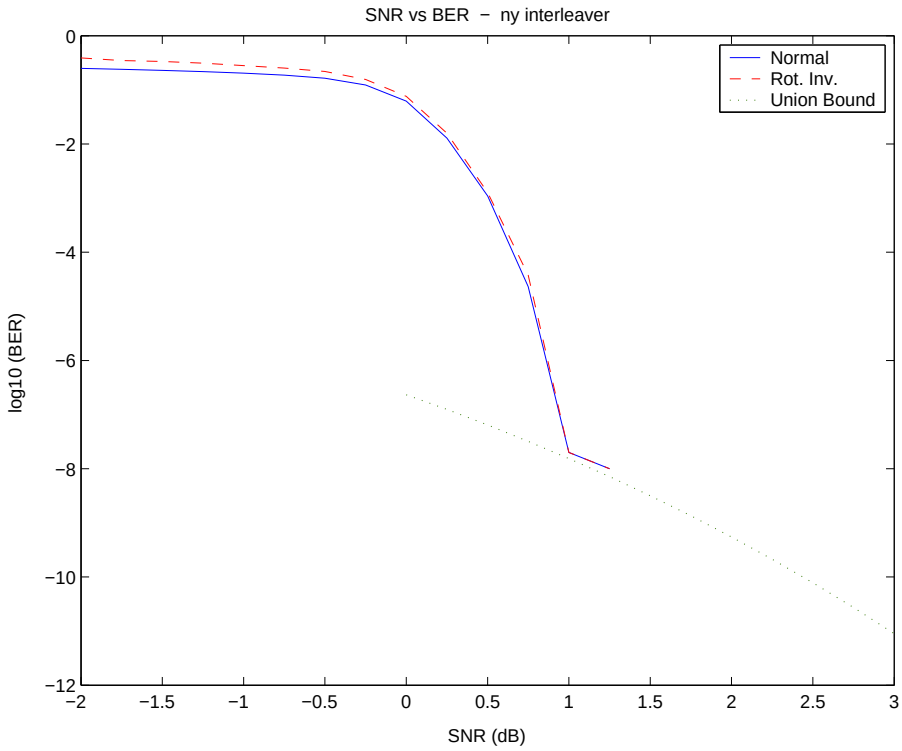
Jeg har valgt å presentere resultatene fra skyggetrellismetoden først. Utputt fra programmet er presentert i appendiks. Her er det plottet i matlab for -2 til 3dB.

Resultater med den gamle interleaveren:



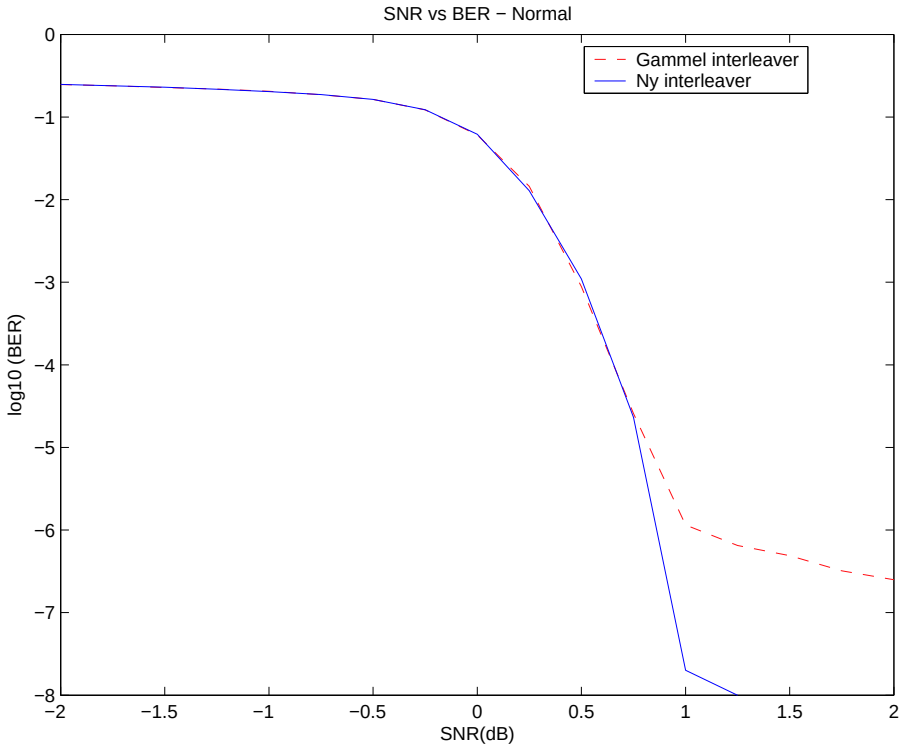
Det er liten forskjell i resultatene for de to variantene. For signal-støyforhold under 0 dB har den rotasjonsinvariante koden litt dårligere ytelse. For bedre signal-støyforhold det ingen merkbar forskjell mellom de to metodene. Resultatene i området mellom 2dB og 3dB er usikre på grunn av få observerte feilhendelser, men de viser tendensen. Sammen med skranken gir det et godt bilde av utviklingen. Det er interessant å se at både den rotasjonsinvariante og den umodifiserte metoden har samme feilgulv. Det betyr at ytelsen til den rotasjonsinvariante metoden er meget god.

Det neste jeg ville undersøke var om en bedre interleaver medførte dårligere ytelse i den rotasjonsinvariante koden. Grafen under viser resultatene med den forbedrete interleaveren:

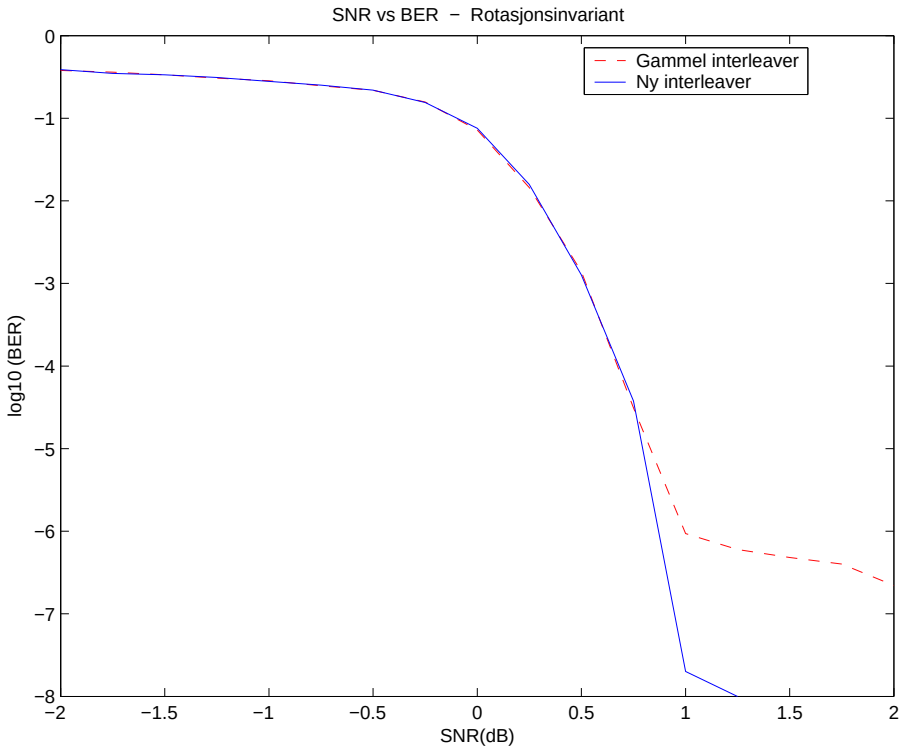


Den nye interleaveren var såpass mye bedre at simulering begynte å bli svært tidkrevende for SNR over 1 dB. 48 timers prosessering kunne gi 0 feil, så jeg valgte å bruke union bound som skisse for ytelsen. Igjen er det tydelig at den rotasjonsinvariante versjonen bare er merkbart dårligere for SNR under 0 dB. De to versjonene ser også her ut til å ha det samme feilgulvet, så ytelsen er ikke dårligere enn med den originale interleaveren.

Sammenligning av de to forskjellige interleavere ga:



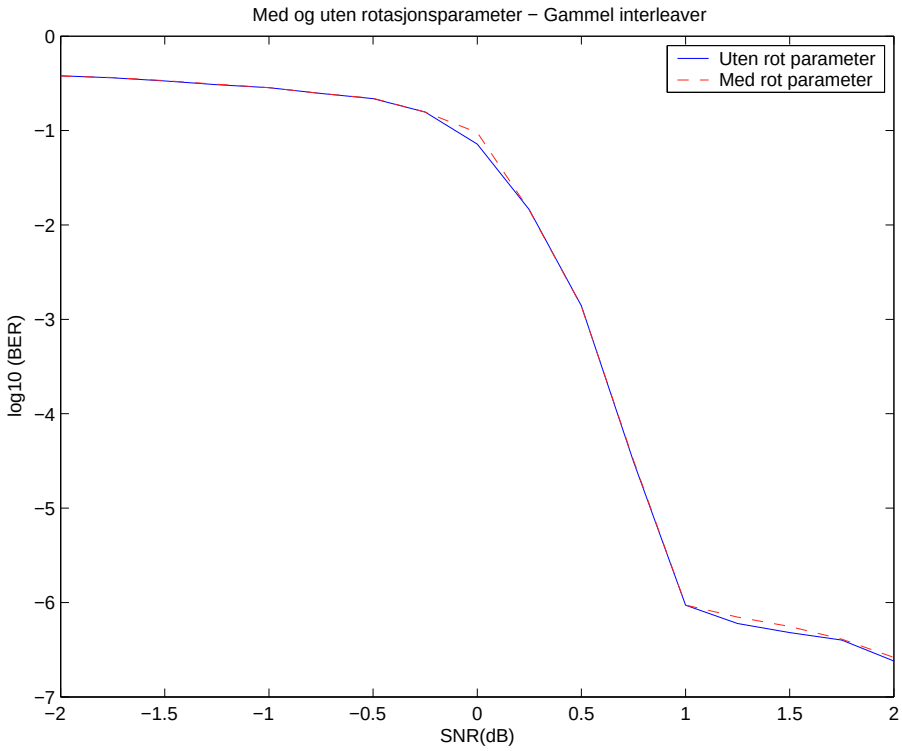
og



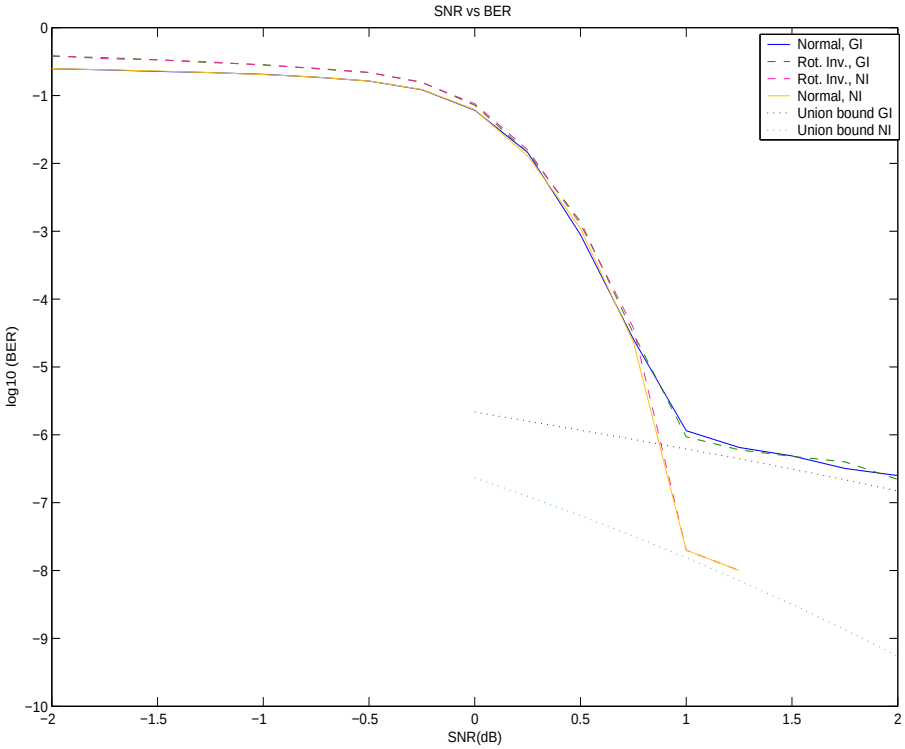
Ser at kodenes ytelse ved SNR > 0 dB helt og holdent er avhengig av interleaveren, og ikke er påvirket av eventuell rotasjonsinvarians. Resultatene stemmer også godt overens med interleavertesten fra tidligere. Det viser seg igjen at det bare er for

SNR < 0 dB at den rotasjonsinvariante koden yter dårligere. Det er også tydelig at den forbedrete interleaveren ikke reduserer ytelsen på noen måte for den rotasjonsinvariante koden, noe man kanskje kunne frykte. Den økte minimumsavstanden i koden ser ut til å være udelt positivt.

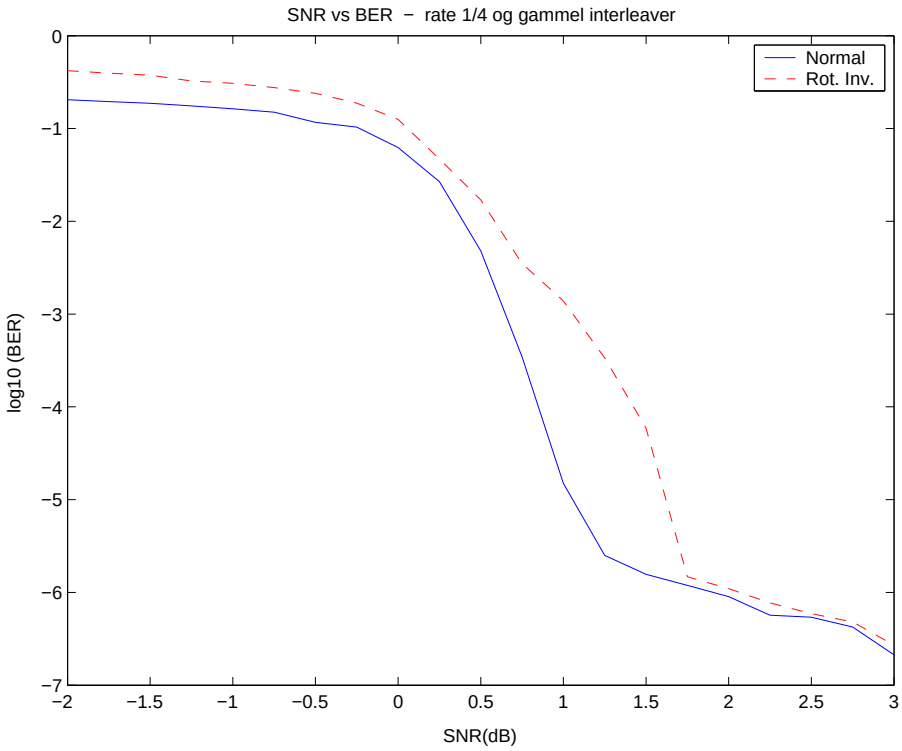
Alle rotasjonsinvariante metoder ble testet med og uten rotasjonsparameter. Det medførte ingen endring av ytelsen å inkludere det, som den neste grafen viser:



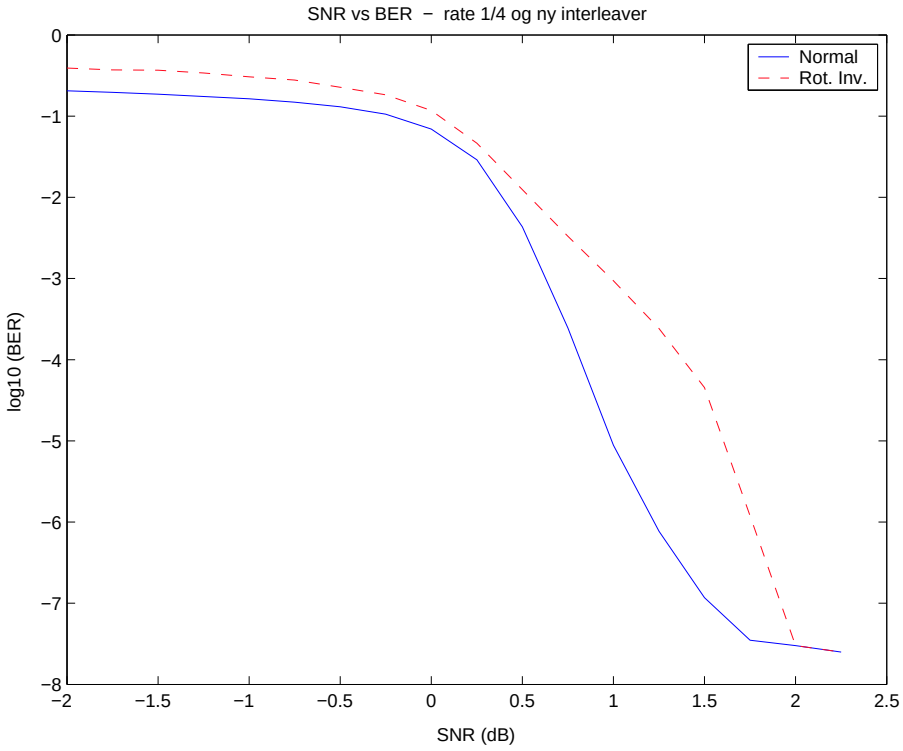
Simuleringer med den andre interleaveren ga tilsvarende resultater. Det ser ikke ut til å være noen grunn til å bruke rotasjonsparameteret ved de forholdene jeg tester. Her medførte det bare en økning i kjøretiden. For å gi et mer oversiktlig bilde har jeg plottet alle grafene i samme diagram:



Så langt var resultatene for skyggetrellismetoden svært gode. Resultatene ble imidlertid annerledes når jeg testet koder uten noen punktering (det vil si at all utputt fra begge komponentkoderne er tatt med). Dette blir strengt tatt en rate $\frac{1}{4}$ kode, men hver informasjonsbit vil også opptre to ganger, noe som gjør det til en slags repetisjonskode. Raten alene gir altså ikke en fullgod beskrivelse av koden. Resultatene for denne varanten ble slik:

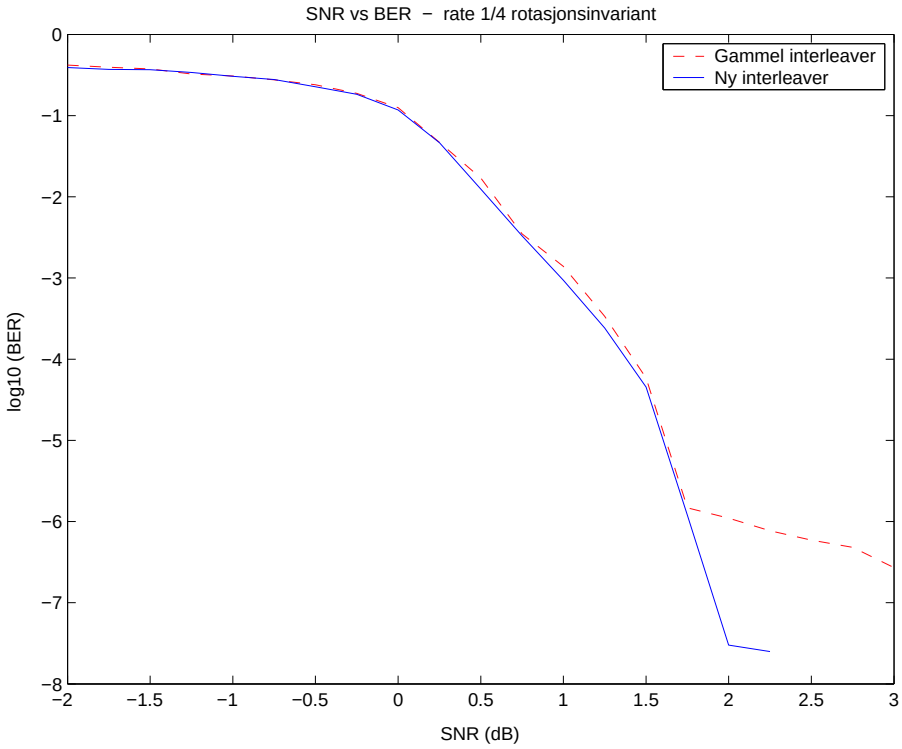


Med den andre interleaveren skjer det samme:



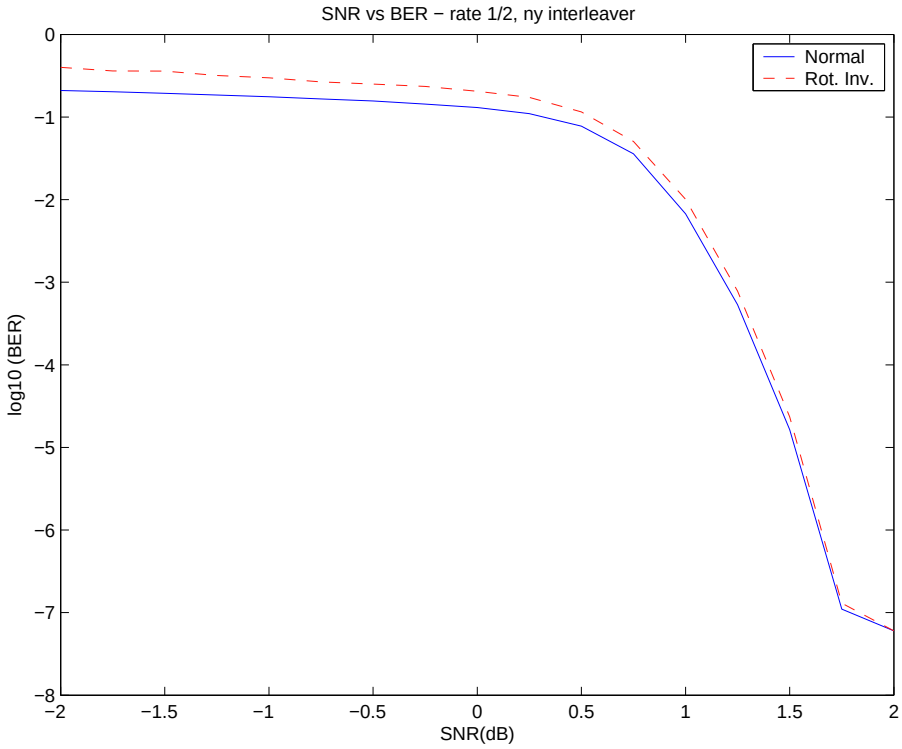
Jeg prøvde både med og uten rotasjonsparameteret for å se om det endret noe, men det gjorde det ikke. Hvorfor dette skjer med denne versjonen har jeg ingen god forklaring på. I fossefallregionen er den rotasjonsinvariante koden markert dårligere

enn den umodifiserte. Det er ikke mulig å se denne tendensen for rate $\frac{1}{3}$ kodene. Feilgulvet ser fremdeles ut til å være identisk, men den rotasjonsinvariante koden treffer det 0.5dB senere. Det kan se ut som om dekodinger av visse kodeord alltid går galt. Jeg sammenlignet også de to rotasjonsinvariante upunkterte grafene for å se om de oppførte seg forskjellig:



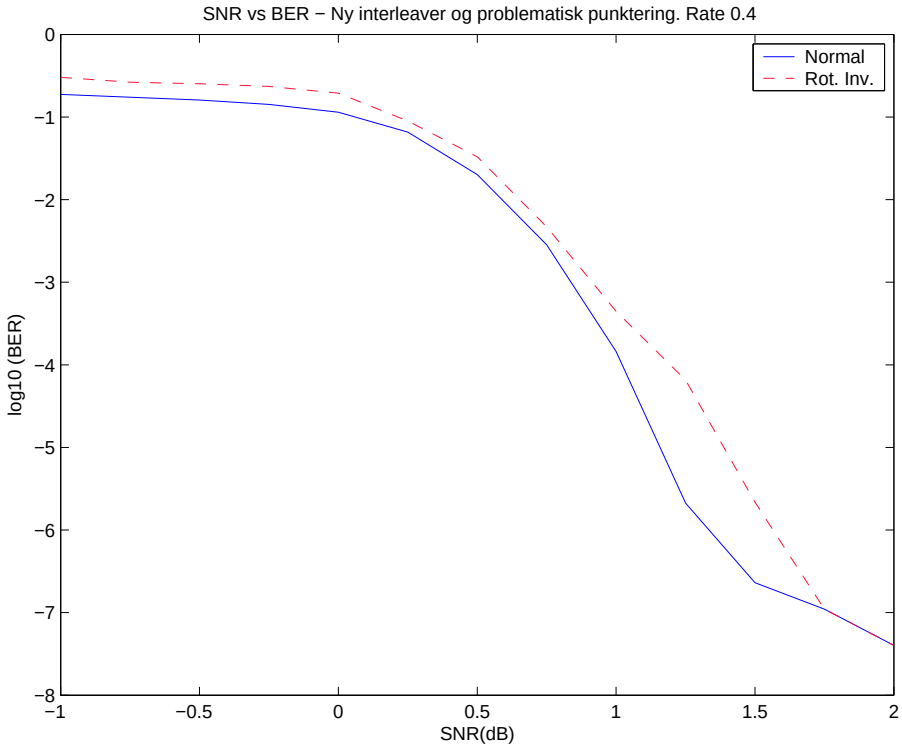
Det er lett å se at begge grafene utvikler seg på samme måte til interleaverene skiller dem. Men grafen viser også at den nye interleaveren treffer feilgulvet senere enn den gamle. Den karakteristiske grafformen er riktignok felles for begge to. Uansett er ytelsen til denne versjonen ikke så dårlig at den er ubrukelig, men årsaken til de spesielle resultatene er ukjent.

Jeg ville også undersøke om skyggetrellismetoden får problemer med kode som har høyere rate enn $\frac{1}{3}$, og om det kunne ha noen sammenheng med oppførselen til de upunkterte kodene. Jeg valgte derfor å teste rate $\frac{1}{2}$ koder for å se om det oppstod problemer dersom jeg økte kodens rate. Det ga disse resultatene:



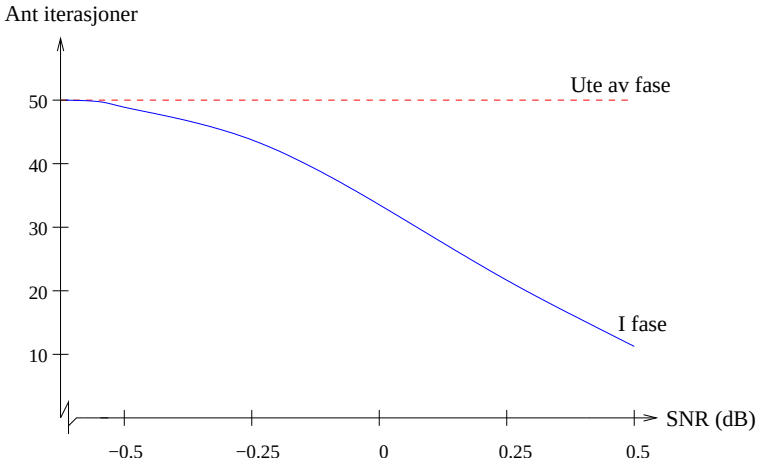
Det viser seg at rate $\frac{1}{2}$ koder fungerer utmerket. Feilgulvet er også her identisk, og resultatene viser at høyere rate ikke nødvendigvis medfører problemer for skyggetrellismetoden. Dette kan være annerledes for koder med veldig høy rate, men det har jeg ikke undersøkt i denne oppgaven.

Til slutt undersøkte jeg hva som skjer dersom en av komponentkodene har det problematiske punkteringskartet 0101 (se kapittel 4.4). Punkteringen til den andre komponentkoden var 1110, noe som ga en rate på $\frac{2}{5}$. Resultatet ble slik:



Det er tydelig at dette punkteringskartet medfører svakere ytelse i fossefallregionen. Ved noen typer innputt er det sannsynligvis lite hjelp å hente fra den ene komponentdekoderen. Feilgulvet ser ut til å være identisk. Men selv i denne situasjonen er ikke ytelsestapet så stort som for den upunkterte versjonen.

Dobbel dekoding gir fullgode dekodingsresultater når dekoderen som er i fase synker under maksimalt antall iterasjoner. Den som er ute av fase vil aldri gå under maksimalt antall. Figuren illustrerer når det inntreffer, og resultatene var identiske for begge interleaverene ved rate $\frac{1}{3}$:



Dekoderen i fase hadde ved -0.75dB 100% feil og i gjennomsnitt 50 iterasjoner per ramme. Ved -0.5 dB hadde den 90% feil og ca. 48 iterasjoner per ramme. Dekoderne kan altså skilles fra hverandre på grunnlag av antall iterasjoner ved SNR

bedre enn ca. -0.5 dB. Dekoderen som er ute av fase får konstant målt sin BER til ca. 0.91, 50 iterasjoner og 100% feil. Etter dette punktet yter dobbel dekoding akkurat like bra som den umodifiserte metoden, men kjøretiden er veldig høy. Det kan riktignok gjøres noe arbeid for å forbedre dette. Man kan for eksempel tjene litt på å huske om forrige ramme invertert, og så antar at den neste er på samme måte. Maksimum antall iterasjoner bør også velges med omhu.

Kapittel 5

Oppsummering og konklusjon

5.1 Konklusjon

Når det gjelder dobbel dekoding, fikk jeg bekreftet at metoden fungerer inntil et visst punkt. Under -0,5dB klarer denne implementasjonen ikke å skille mellom fase og ute av fase, men da er også feilsannsynligheten så stor at koden er ubrukelig i praksis. Over -0,5dB er feilsannsynligheten lik umodifiserte koden. Metoden får svært høy kjøretid når det oppstår rotasjon, men det kan legges til optimaliseringer for å gjøre metoden mer anvendelig dersom det skulle være interessant. Dobbel dekoding har fordelen at den kan brukes med en vilkårlig turbokode, og den krever bare små endringer i kontrollprogrammet for å implementere. Dette medfører at det kreves betraktelig mindre for å få denne metoden til å fungere med eksisterende utstyr sammenlignet med skyggetrellismetoden. En svakhet med dobbel dekoding er at den ikke er særlig robust dersom rotasjon oppstår midt i en ramme. Dette vil medføre feilskred, men del vil bare gå utover en enkelt ramme. I tillegg gjør den ekstremt store kjøretidsøkningen at metoden er uegnet til applikasjoner som er følsom ovenfor forsinkelse.

Skyggetrellismetoden fungerte meget bra: Simuleringsresultatene for rate $\frac{1}{3}$ viser ingen vesentlig effektivitetstap i koden, og en kjøretidsøkning med en faktor på kun 1,4. Skyggetrellisen kan i tillegg brukes sammen med en vilkårlig turbokode, fordi den er uavhengig av koderens graf og interleaver. Det viste seg også at rotasjonsparameteret var unødvendig fordi det ikke forbedret kodens ytelse, noe som tillot at kjøretidsøkningen forblie liten. En svakhet er at den ikke kan håndtere rotasjon som oppstår midt i rammer, noe som vil medføre feilskred. Implementasjon av denne metoden vil medføre endringer i SISO blokkene, noe som kan være problematisk i enkelte tilfeller. Økningen i kjøretid kan også være nok til at noen typer applikasjoner ikke vil kunne benytte denne metoden. Det er også usikkert om fremgangsmåten fungerer tilfredsstillende for koder med veldig høy rate. Dette bør belyses ved videre forskning. Hva som skaper problemer i den upunkterte versjonen er også usikkert, og bør undersøkes videre. Men jeg kan i alle fall konkludere med at skyggetrellismetoden hadde veldig bra ytelse brukt med både rate $\frac{1}{3}$ og rate $\frac{1}{2}$ kodene jeg testet. Den moderate økningen i kjøretid er en liten pris å betale for den utvidete funksjonaliteten. Det er imidlertid veldig viktig å kontrollere at punkteringen ikke skaper problemer. Punkteringskart som medfører at enerkodeordet finnes i en av komponentkodene vil redusere kodens ytelse. Skulle dette kodeordet finnes i begge komponentkodene vil metoden ikke fungere.

5.2 Videre arbeid

Det er flere ting som kan være interessant å forske videre på etter denne oppgaven. Hvordan skyggetrellismetoden fungerer med høyrate koder er et viktig spørsmål. Det kunne også vært interessant å undersøke den utvidete kodens vektfordeling, noe som for eksempel kan gjøres ved å se på den opprinelige kodens høyvektskodeord. I denne sammenhengen er også punkteringsmønsteret som brukes viktig å kontrollere. En annen ting som kan gjøres er å finne ut hvordan metoden håndterer at rotasjon opptrer midt i en ramme, og om det eventuelt går an å legge til funksjonalitet som håndterer dette. Det hadde også vært interessant å få klarhet i hvordan det fungerer å bygge 90 eller 45 graders rotasjonsinvarians med flere tvillingtrelliser. Det bør også forskes på hva som skjer med den upunkterte koden siden den har en såpass spesiell oppførsel. Et siste tema som bør belyses er om det er mulig å konstruere rotasjonsinvariante turbokoder basert på koderens graf. Dersom det er mulig vil det ikke medføre inngrep i SISO blokkene, noe som kan være en fordel. Det kan også tenkes at det kan presenteres bevis for at det ikke er mulig å konstruere slike rotasjonsinvariante turbokoder, gitt kriteriene i [3]. Bare videre forskning vil kunne gi svaret.

Bibliografi

[1] C. E. Shannon, "A Mathematical Theory of Communication, " *Bell Syst. Tech. J., 27,* July 1948.

[2] C. Berrou, A. Glavieux, P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," *Proceedings of ICC'93*, May 1993.

[3] M. D. Trott, S. Benedetto, R. Garello, M. Mondin, "Rotational invariance of trellis codes," *IEEE Transactions of infomation theory vol 42 no 3,* May 1996.

[4] S. Benedetto, D. Divsalar, G. Montorsi, F. Pollara, "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatinated codes" *TDA Progress report 42-127*, November 1996.

[5] S. Lin and D. Costello "Error control codes," *Prentice Hall* 1983

[6] Ø. Ytrehus "An introduction to turbo codes and iterative decoding," *Telektronikk vol 98 no 1,* 2002

[7] S. Wicker "Error control systems for digital communications and storage," *Prentice Hall* 1995

[8] G. Ungerboeck "Channel Coding with Multilevel/Phase Signals", IEEE Trans. on Information Theory, vol. IT-28, No. 1, Jan. 1982.

Tillegg A

Resultattabeller

A.1 Diverse

Den nye interleaveren er generert ved hjelp av et eget program kalt RealInt. Parametrene som er brukt er:
S= 40, S_2=10, S_3=60, S_{32} = 15, og looplength = 7.
Interleaverene er lagt ved oppgaven på CD-ROM. De som er brukt er Interleaver1250.txt (gammel) og Int.txt (ny)

Matlab kommando for beregning av union bound:
U_{SNR} = log10 (((((*antall_kodeord*)/1250)*Q((sqrt((*tot_info_vekt*)*(1/3)*2*SN(*SNR*)))))) + ...), *SNR* = 0, 0.25,.....,3

A.2 Resultattabeller

Normal turbokode, gammel interleaver, rate $\frac{1}{3}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.6060
-1.75	500	-0.6220
-1.50	500	-0.6434
-1.25	500	-0.6623
-1.00	500	-0.6890
-0.75	500	-0.7303
-0.50	500	-0.7862
-0.25	500	-0.9140
0.00	500	-1.2160
0.25	400	-1.8394
0.50	400	-3.0516
0.75	400	-4.5904
1.00	400	-5.9393
1.25	100	-6.1871
1.50	100	-6.3098
1.75	100	-6.4949
2.00	100	-6.6021
2.25	<100	-6.7447
2.50	<100	-6.9586
2.75	<100	-7.2218
3.00	<100	-7.3468

Rotasjonsinvariant turbokode, gammel interleaver, uten rotasjonsparameter, rate $\frac{1}{3}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.4205
-1.75	500	-0.4412
-1.50	500	-0.4740
-1.25	500	-0.5142
-1.00	500	-0.5463
-0.75	500	-0.6064
-0.50	500	-0.6615
-0.25	500	-0.8040
0.00	500	-1.1429
0.25	400	-1.8383
0.50	400	-2.8574
0.75	400	-4.5064
1.00	400	-5.0283
1.25	100	-6.2218
1.50	100	-6.3188
1.75	100	-6.3979
2.00	100	-6.6198
2.25	<100	-6.7447
2.50	<100	-7.0458
2.75	<100	-7.1549
3.00	<100	-7.3979

Rotasjonsinvariant turbokode, gammel interleaver, med rotasjonsparameter, rate $\frac{1}{3}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.4225
-1.75	500	-0.4377
-1.50	500	-0.4731
-1.25	500	-0.5086
-1.00	500	-0.5451
-0.75	500	-0.6073
-0.50	500	-0.6592
-0.25	500	-0.8038
0.00	500	-1.1021
0.25	400	-1.8422
0.50	400	-2.8618
0.75	400	-4.4927
1.00	400	-6.0237
1.25	100	-6.1549
1.50	100	-6.3143
1.75	100	-6.3882
2.00	<100	-6.5821

Normal turbokode, ny interleaver, rate $\frac{1}{3}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.6045
-1.75	500	-0.6212
-1.50	500	-0.6389
-1.25	500	-0.6630
-1.00	500	-0.6920
-0.75	500	-0.7275
-0.50	500	-0.7861
-0.25	500	-0.9101
0.00	500	-1.2095
0.25	400	-1.8920
0.50	400	-2.9598
0.75	400	-4.6356
1.00	400	-7.6986
1.25	<100	-8.0042

Rotasjonsinvariant turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{1}{3}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.4115
-1.75	500	-0.4578
-1.50	500	-0.4747
-1.25	500	-0.5063
-1.00	500	-0.5531
-0.75	500	-0.5988
-0.50	500	-0.6602
-0.25	500	-0.8088
0.00	500	-1.1211
0.25	400	-1.8006
0.50	400	-2.8980
0.75	400	-4.4235
1.00	400	-7.6991
1.25	<100	-7.9978

Normal turbokode, gammel interleaver, rate $\frac{1}{4}$ upunktert:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.6893
-1.75	500	-0.7103
-1.50	500	-0.7284
-1.25	500	-0.7569
-1.00	500	-0.7873
-0.75	500	-0.8246
-0.50	500	-0.9341
-0.25	500	-0.9855
0.00	500	-1.2048
0.25	400	-1.5721
0.50	400	-2.3206
0.75	400	-3.4590
1.00	400	-4.8247
1.25	100	-5.6015
1.50	100	-5.8055
1.75	100	-5.9245
2.00	100	-6.0448
2.25	<100	-6.2457
2.50	<100	-6.2676
2.75	<100	-6.3726
3.00	<100	-6.6737

Rotasjonsinvariant turbokode, gammel interleaver, uten rotasjonsparameter, rate $\frac{1}{4}$ upunktert:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.3768
-1.75	500	-0.4034
-1.50	500	-0.4248
-1.25	500	-0.4881
-1.00	500	-0.5129
-0.75	500	-0.5591
-0.50	500	-0.6198
-0.25	500	-0.7258
0.00	500	-0.9014
0.25	400	-1.3298
0.50	400	-1.7696
0.75	400	-2.4559
1.00	400	-2.8580
1.25	100	-3.4733
1.50	100	-4.2311
1.75	100	-5.8318
2.00	100	-5.9586
2.25	<100	-6.1135
2.50	<100	-6.2291
2.75	<100	-6.3188
3.00	<100	-6.5686

Rotasjonsinvariant turbokode, gammel interleaver, med rotasjonsparameter, rate $\frac{1}{4}$ upunktert:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.4088
-1.75	500	-0.4099
-1.50	500	-0.4332
-1.25	500	-0.4881
-1.00	500	-0.5154
-0.75	500	-0.5441
-0.50	500	-0.5895
-0.25	500	-0.7741
0.00	500	-0.9288
0.25	400	-1.2676
0.50	400	-1.8125
0.75	400	-2.4469
1.00	400	-2.9576
1.25	100	-3.5845
1.50	100	-4.3118
1.75	100	-5.8281
2.00	100	-5.9914
2.25	<100	-6.1024
2.50	<100	-6.2366
2.75	<100	-6.3279
3.00	<100	-6.4815

Normal turbokode, ny interleaver, rate $\frac{1}{4}$ upunktert:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.6882
-1.75	500	-0.7079
-1.50	500	-0.7293
-1.25	500	-0.7575
-1.00	500	-0.7867
-0.75	500	-0.8278
-0.50	500	-0.8850
-0.25	500	-0.9750
0.00	500	-1.1586
0.25	400	-1.5381
0.50	400	-2.3637
0.75	400	-3.6109
1.00	400	-5.0526
1.25	100	-6.1135
1.50	100	-6.9318
1.75	100	-7.4559
2.00	100	-7.5229
2.25	<100	-7.6021

Rotasjonsinvariant turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{1}{4}$ upunktert:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.4077
-1.75	500	-0.4303
-1.50	500	-0.4353
-1.25	500	-0.4682
-1.00	500	-0.5164
-0.75	500	-0.5553
-0.50	500	-0.6444
-0.25	500	-0.7376
0.00	500	-0.9303
0.25	400	-1.3336
0.50	400	-1.9054
0.75	400	-2.4800
1.00	400	-3.0291
1.25	100	-3.6163
1.50	100	-4.3430
1.75	100	-5.9245
2.00	100	-7.5229
2.25	<100	-7.6101

Normal turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{1}{2}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.6769
-1.75	500	-0.6936
-1.50	500	-0.7120
-1.25	500	-0.7330
-1.00	500	-0.7535
-0.75	500	-0.7802
-0.50	500	-0.8063
-0.25	500	-0.8432
0.00	500	-0.8855
0.25	400	-0.9588
0.50	400	-1.096
0.75	400	-1.4459
1.00	400	-2.1725
1.25	100	-3.2715
1.50	100	-4.7841
1.75	100	-6.9586
2.00	<100	-7.2218

Rotasjonsinvariant turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{1}{2}$:

SNR (dB)	Antall obs. feil	log10 (BER)
-2.00	500	-0.3985
-1.75	500	-0.4415
-1.50	500	-0.4438
-1.25	500	-0.4972
-1.00	500	-0.5262
-0.75	500	-0.5745
-0.50	500	-0.6009
-0.25	500	-0.6301
0.00	500	-0.6884
0.25	400	-0.7616
0.50	400	-0.9380
0.75	400	-1.2953
1.00	400	-1.9964
1.25	100	-3.1082
1.50	100	-4.6295
1.75	100	-6.8861
2.00	<100	-7.2209

Normal turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{2}{5}$ problematisk punktering:

SNR (dB)	Antall obs. feil	log10 (BER)
-1.00	500	-0.7258
-0.75	500	-0.7595
-0.50	500	-0.7935
-0.25	500	-0.8478
0.00	500	-0.9412
0.25	500	-1.1819
0.50	500	-1.6975
0.75	500	-2.5466
1.00	500	-3.8386
1.25	400	-5.6737
1.50	400	-6.6383
1.75	100	-6.9578
2.00	100	-7.3982

Rotasjonsinvariant turbokode, ny interleaver, uten rotasjonsparameter, rate $\frac{2}{5}$ problematisk punktering:

SNR (dB)	Antall obs. feil	log10 (BER)
-1.00	500	-0.5193
-0.75	500	-0.5785
-0.50	500	-0.5974
-0.25	500	-0.6293
0.00	500	-0.7099
0.25	500	-1.0487
0.50	500	-1.4814
0.75	500	-2.3288
1.00	500	-3.3513
1.25	400	-4.1843
1.50	400	-5.6635
1.75	100	-6.9686
2.00	100	-7.3971

Tillegg B

Programkode

Til simuleringene brukte jeg et ferdig laget simuleringsprogram, og gjorde endringer der det var nødvendig for å implementere rotasjonsinvariansen. Simuleringene med normale turbokoder er gjort i det umodifiserte programmet. For å gi en oversikt over hvilken kildekode jeg har endret har jeg satt det opp i en tabell. "U" betyr at simuleringene brukte en umodifisert variant, mens "M" betyr at programmet brukte en modifisert variant. SimulatorQAM er hovedkontrollprogrammet, ParSchem er et kontrollprogram for SISO dekodingen og LogDec_code er implementasjonen av SISO blokkene.

	Normal	RI med rot parameter	RI uten rot parameter
SimulatorQAM	U	M	M
ParSchem	U	M	M
LogDec_code	U	M	*M
Trellis, kanal, kode, etc.	U	U	U

*M betyr at de to metodene brukte forskjellige modifiserte versjoner av programmet. I tillegget vil jeg først ta med de mest sentrale metodene i sin opprinnelige form, og så vil jeg ta med de endrete versjonene. Dette fordi endringene jeg har gjort da kommer klart frem. Komplett kildekode vil legges ved oppgaven på en CD-ROM.

Nr	Fil
1	Umodifisert SimulatorQAM
2	Umodifisert LogDec_code
3	Modifisert SimulatorQAM
4	Modifisert LogDec_code
5	*Modifisert Logdec_code

B.1 Umodifisert SimulatorQAM

```
// Simulator of a pragmatic turbo coded modulation scheme
//
// define __QAM_MOD__ for higher order modulation
// if __QAM_MOD__ is not defined, binary modulation is used

#include <stdio.h>
#include <math.h>
```

```
#include <time.h>

/*************************************************************/

FILE *fresultat;
FILE *parameters;
FILE *fp_report;

#define PERFECT_CHANNEL_ESTIM

//#define __QAM_MOD__

/*************************************************************/

#if !defined(__CONSTELLATION__H)
#include "../Auxiliary/Constellation.h"
#endif

#if !defined(__READGEN_H)
#include "../Auxiliary/ReadGen.h"
#endif

#if !defined(__PARALLEL__H)
#include "../ParSchem/ParSchem.h"
#endif

#if !defined(__CHANNEL_AWGN__H)
#include "../Channels/Channel_AWGN.h"
#endif

#if !defined(__DATAGEN__H)
#include "../Auxiliary/DataGen.h"
#endif

//*************************************************************************

int d_Hamming(int *c1, int *c2, int length)
{
int distance = 0;
for (int i=0;i<length;i++)
if(c1[i] != c2[i])
distance++;
return distance;
}

//*************************************************************************

int main(int argc,char *argv[])
{

   //************* INITIALISERING AV MODELL *************


   Code_itr *My_code;
```

```
My_code = new Parallel(argv[1]);     int code_type = 0;


//************************************************

FILE *fp_codeword;

int MinErr=500,MaxRound=500,MinRound=20;
printf("\nPlease enter Min. number of errors: ");
scanf("%d",&MinErr);
printf("\nPlease enter Minimum number of frames and Maximum number of frames: ");
scanf("%d %d",&MinRound,&MaxRound);

int *Info,*codeword,*sentcodeword,*codeword_int,*reencoded;
int *reencoded_int;
double mean_num_of_itr;

double SNRdB;

sentcodeword = new int[My_code->code_length];
codeword_int = new int[My_code->code_length];
reencoded_int = new int[My_code->code_length];

printf("\nPlease enter SNRdB in simulation ");
scanf(" %lf",&SNRdB);

double *airword = new double[My_code->code_length];
double *airword_int;
double eff_rate;

int *received;
int *received2;
int i;
int millionbits;
int Teller2;

printf("Antall iterasjoner %d\n",My_code->AntIter);

int eff_code_length = My_code->code_length;
eff_rate = (double) (1.0*My_code->info_length)/(1.0*eff_code_length);


//****************** CHOOSE THE CHANNEL ***************************

int num_channel;
Channel **My_channel;

//-------------------------------------------------------------
// Additive white Gaussian noise channel
//-------------------------------------------------------------

num_channel=1;
My_channel = new Channel*[num_channel];
```

```
   for (i=0;i<num_channel;i++)
      My_channel[i] = new Channel_AWGN(SNRdB,eff_code_length,eff_rate);


   //-----------------------------------------------------------------------

   int ll;

   int num_snr = 5;
   double *snr_vector;
   snr_vector = new double[num_snr];
   for (ll=0;ll<num_snr;ll++)
     snr_vector[ll] = (double) (ll*0.25+SNRdB);

   double *QS=new double[eff_code_length];
   double *QS_deint = new double[eff_code_length];

   printf("The Frame length is %d \n",eff_code_length);
   printf("The Rate is %1.2f\n",eff_rate);
   printf("The simulation will be done for SNR = %1.3f dB\n",SNRdB);

   RandData cg(My_code->info_length);

   //-----------------------------------------------------------------------
   // Signal constellation mapping
   //-----------------------------------------------------------------------

#if defined(__QAM_MOD__)
   //int m = 3;                // 64-QAM
   //int m = 2;                  // 16-QAM
   int m = 1;               // BPSK

   int num_symbols = eff_code_length/m;
   Constellation *signal;
   signal = new Constellation(m,eff_code_length);

   if (eff_code_length%m != 0)
   {
     printf("Error: Incorrect info length!\n");
     return 0;
   }
#endif

   //-----------------------------------------------------------------------
   // Printing of parameters to the file 'parameters.txt'
   //-----------------------------------------------------------------------

   parameters = fopen("parameters.txt","w");
   fprintf(parameters,"\n");

#if !defined(PERFECT_CHANNEL_ESTIM)
   fprintf(parameters,"Estimation of channel parameters is performed.\n");
   fprintf(parameters,"\n");
```

```
#else
   fprintf(parameters,"Perfect channel estimates are assumed.\n");
   fprintf(parameters,"\n");
#endif

   My_code->print_parameter_file(parameters);
   My_channel[0]->print_parameter_file(parameters);

   fprintf(parameters,"Minimum number of errors: %d\n",MinErr);
   fprintf(parameters,"Minimum number of frames: %d\n",MinRound);
   fprintf(parameters,"Maximum number of frames: %d\n",MaxRound);
   fprintf(parameters,"Minimum of two iterations\n");
   fprintf(parameters,"\n");

#if defined(__QAM_MOD__)
   signal->print_parameter_file(parameters);
#endif

   fprintf(parameters,"Initial SNR in dB: %.2f\n",SNRdB);
   fprintf(parameters,"\n");
   fclose(parameters);


   //*******************************************************
   //
   // Random interleaver....
   //
   //*******************************************************

   int *leaver = new int[My_code->code_length];
   int *leaver_inv = new int[My_code->code_length];
   int num_cand = My_code->code_length;

   int map_int;
   printf("Read mapping interleaver from file? (yes(1) or no(0)) ");
   scanf("%d",&map_int);

   if (map_int == 0)
   {
     for (i=0;i<My_code->code_length;i++)
       leaver_inv[i] = i;

     // Building interleaver...

     while (num_cand > 0)
     {
       int index1 = rand() % num_cand;
       leaver[num_cand-1] = leaver_inv[index1];
       leaver_inv[index1] = leaver_inv[num_cand-1];
       num_cand--;
     }
   }
   else
   {
```

```
      FILE *finter;

      if ((finter=fopen("Mapinterleaver.txt","r"))== NULL)
        printf("\nCould not open the file. Terminates....\n");
      else
        for (i=0;i<My_code->code_length;i++)
          fscanf(finter,"%d",&leaver[i]);
      fclose(finter);
    }

    for (i=0;i<My_code->code_length;i++)
      leaver_inv[leaver[i]] = i;

    parameters = fopen("Mapinterleaver.txt","w");
    for (i=0;i<My_code->code_length;i++)
      fprintf(parameters,"%d ",leaver[i]);

    fprintf(parameters,"\n");
    fprintf(parameters,"\n");
    fprintf(parameters,"Interleaver used before mapping to constellation points\n");
    fprintf(parameters,"The interleaver is randomly generated\n");
    fprintf(parameters,"\n");
    fclose(parameters);

    //*******************************************************
    //
    // Simulation of different signal-to-noise ratios...
    //
    //*******************************************************

    time_t t1,t2,t_start,t_end;

    (void) time(&t1);
    (void) time(&t_start);

    for (ll=0;ll<num_snr;ll++)
    {
      int i;
      SNRdB = snr_vector[ll];
      int frames=0;
      int Ant_blocks_inerror=0;
      int Ant_wrong_blocks=0;
      int Ant=0;
      double kjisum=0.0;

      millionbits = 1000000 / My_code->info_length;

#if defined(__QAM_MOD__)
      double noise_var = (double) (1<<(2*m)-1)/
                         (double) (12.*m*eff_rate*pow(10.,SNRdB/10.));
#endif

      // Reinitialize the channels...
      for (i=0;i<num_channel;i++)
```

```
     {
       My_channel[i]->ReInit(SNRdB);

#if defined(__QAM_MOD__)
       My_channel[i]->ReInitnoise(noise_var);
#endif

     }

#if defined(__QAM_MOD__)
     printf("Noise variance: %f\n",noise_var);
#endif

     while (((Ant_blocks_inerror<MinErr) && (frames<MaxRound)) ||
            (frames<MinRound))
     {
       Info=cg.Data();

       codeword=My_code->Encode(Info);

       for (i=0; i<My_code->code_length; i++)
         codeword_int[leaver[i]] = codeword[i];

#if defined(__QAM_MOD__)
       airword=signal->generate_QAM_seq(codeword_int);
#else
       for (i=0;i<My_code->code_length;i++)
         airword[i] = 2.*codeword_int[i]-1.;
#endif

       for (i=0;i<num_channel;i++)
         My_channel[i]->Channel_mod(airword);


       //-------------------------------------------------
       // Demodulation and decoding
       //-------------------------------------------------

#if defined(__QAM_MOD__)
       QS = signal->bit_channel(My_channel[0]->data_seq_out,
                                My_channel[0]->noise_var);
#else
       double *recseq = My_channel[0]->data_seq_out;
       for (i=0;i<My_code->code_length;i++)
         QS[i] = 2./My_channel[0]->noise_var*recseq[i];
#endif

       for (i=0; i<My_code->code_length; i++) {
         QS_deint[i] = QS[leaver[i]];

       received = My_code->Decode(QS_deint);

       //-------------------------------------------------
       //Count the number of errors:
```

```
      //-------------------------------------------------

      {
        int old_Ant;
    double d2sent,d2received;

 old_Ant = Ant;
        d2sent= 0.; d2received = 0.;


    for (i=0; i<My_code->info_length; i++) {
          if (Info[i]!=received[i])
      {
        Ant++;
      }
 }


    if (Ant>old_Ant)
    {
          Ant_blocks_inerror++;
          reencoded = My_code->Encode(received);



          fp_codeword = fopen("Codewords.txt","a");

          fprintf(fp_codeword,"Codeword, SNR = %f dB\n",SNRdB);
          for (i=0; i<My_code->code_length; i++)
            if (sentcodeword[i]!=reencoded[i])
              fprintf(fp_codeword,"[%d]",i);

          fprintf(fp_codeword,"\nPositions in info seq., SNR = %f dB\n",SNRdB);
          for (i=0; i<My_code->info_length; i++)
            if (Info[i]!=received[i])
              fprintf(fp_codeword,"[%d]",i);

          //-------------------------------------------------------------

          for (i=0;i<num_channel;i++)
            d2sent += My_channel[i]->distance2(airword);

          for (i=0;i<My_code->code_length;i++)
            reencoded_int[leaver[i]] = reencoded[i];

          for (i=0;i<num_channel;i++)
          {
#if defined(__QAM_MOD__)
            airword=signal->generate_QAM_seq(reencoded_int);
#else
            for (int j=0;j<My_code->code_length;j++)
              airword[j] = 2.*reencoded_int[j]-1.;
#endif
            d2received += My_channel[i]->distance2(airword);
```

```
  }

      kjisum += d2sent;

      printf("Code distance: %d, d2(sent,y)=%1.3f, d2(decoded,y)=%1.3f",
              d_Hamming(sentcodeword,reencoded,
              My_code->code_length),d2sent,d2received);

          fprintf(fp_codeword,"\n");
          fprintf(fp_codeword,"Code distance: %d, d2(sent,y)=%1.3f, d2(decoded,y)=%1.3f",
                  d_Hamming(sentcodeword,reencoded,
              My_code->code_length),d2sent,d2received);

      if (d2sent < d2received)
      {
        printf("**** wrong decoding");
            fprintf(fp_codeword,"**** wrong decoding");
        Ant_wrong_blocks++;
      }
          fprintf(fp_codeword,"\nThe effective number of iterations is: %d\n",
                  My_code->eff_AntIter);

          fprintf(fp_codeword,"\n");
          fprintf(fp_codeword,"\n");
          fclose(fp_codeword);

      printf("\n");
          printf("The effective number of iterations is: %d\n",
                  My_code->eff_AntIter);
    }
      }

      frames++;
      if (frames % millionbits ==0)
      {
        double mean_itr = My_code->get_mean_num_of_itr();
FILE *fp_0 = fopen("Output.txt","a");
        (void) time(&t_end);

     printf(" %d frames completed. %d errors so far. %d block errors, %d incorrectly decoded
                frames, Ant,Ant_blocks_inerror, Ant_wrong_blocks);

        fprintf(fp_0," %d frames completed. %d errors so far. %d block errors, %d incorrectl
                   frames, Ant,Ant_blocks_inerror, Ant_wrong_blocks);

        printf("Mean number of iterations so far: %1.3f\n",mean_itr);
        printf("Used time for the last %d frames: %d\n",millionbits,
(int) t_end-t_start);
        fprintf(fp_0,"Used time for the last %d frames: %d\n",millionbits,
                (int) t_end-t_start);
        fprintf(fp_0,"Mean number of iterations so far: %1.3f\n",mean_itr);
        (void) time(&t_start);
        fclose(fp_0);
      }
```

```
    }//SLUTT PÅ SIMULERINGSLOOP

    //--------------------------------------------------------------------
    // A specific value of the signal-to-noise ratio is completed, and
    // results are written to file...
    //
    //--------------------------------------------------------------------

    fresultat = fopen("feilrate.txt","a");
    fprintf(fresultat,"%1.2f\t%1.8f\t%1.8f\n",SNRdB,
           (float)Ant/((float)(frames*My_code->info_length)),
           (float)Ant_blocks_inerror/((float)(frames)));
    fclose(fresultat);

    fp_report = fopen("report_error.txt","a");
    mean_num_of_itr = My_code->get_mean_num_of_itr();
    My_code->report_errors_file(fp_report);

    fprintf(fp_report,"[%1.2f,log10(%1.8f)], %d bit errors, %d frames (out of %d) in error.
          (float)Ant/((float)(frames*My_code->info_length)),
          Ant,Ant_blocks_inerror,frames,Ant_wrong_blocks,mean_num_of_itr);

    fprintf(fp_report,"\n");
    fclose(fp_report);

    printf("[%1.2f,log10(%1.8f)], %d bit errors, %d frames (out of %d) in error. %d incorrec
         (float)Ant/((float)(frames*My_code->info_length)),
         Ant,Ant_blocks_inerror,frames,Ant_wrong_blocks,mean_num_of_itr);

    printf("Corrected 180 deg rotation on %d out of %d frames\n", Teller2, frames);

    Teller2 = 0;
    My_code->reset_errors();
  }

  delete []QS;

  (void) time(&t2);
  printf("Tidsforbruk: %d\n",(int) t2-t1);
  return 1;
}
```

B.2 Umodifisert LogDec_code

```
#define _LOGDEC_CODE__H

#if !defined(_LOGDEC__H)
#include "LogDec.h"
#endif

#if !defined(_TRELLIS_CODE__H)
#include "Trellis_code_ext.h"
#endif
```

```
class LogDec_code: public LogDec
{
   public:

   LogDec_code(int _k,int _n,int trellis_lengde,Trellis_code *_trellis);
   ~LogDec_code();

   void SISO(double *ApriCode,double *ApriInfo,
     double *OutInfo,double *OutCode,
             int computeOutCode);

   double Max2(double tall1,double tall2);

   private:
Trellis_code *trellis;
       double **gamma,**gammac;
};


LogDec_code :: LogDec_code(int _k,int _n,int trellis_lengde,
                           Trellis_code *_trellis):LogDec()
{
int i;
n=_n;
    InfoLengde=_k;
    TrellisLengde = trellis_lengde;
    trellis = _trellis;

gamma = new double*[InfoLengde];
   gammac = new double*[n];
   for (i=0; i<n; i++)
      gammac[i] = new double[2];

        for (i=0; i<InfoLengde; i++)
      gamma[i] = new double[2];

Gamma = new double[trellis->numberofstates];
   Alpha=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Alpha[i]=new double[trellis->numberofstates];
    Beta=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Beta[i]=new double[trellis->numberofstates];
}

LogDec_code :: ~LogDec_code()
{
   int i;

   for (i=0; i<TrellisLengde+1; i++)
    {
    delete []Alpha[i];
      delete []Beta[i];
```

```
    }
   delete []Alpha;
   delete []Beta;
   delete []Gamma;

   for (i=0; i<n; i++)
     delete []gammac[i];

   for (i=0; i<InfoLengde; i++)
     delete []gamma[i];

   delete []gammac;
   delete []gamma;
}

inline double LogDec_code :: Max2(double tall1,double tall2)
{
   double maximum, difference,dummy;
   int intdiff;

   if (tall1 > tall2)
    {
    maximum=tall1;
    difference = tall1-tall2;//+maximum;
    }
   else
      {
    maximum=tall2;
    difference = tall2-tall1;//+maximum;
    }

   //intdiff = (int) floor(difference*GRANULARITY);
   if (difference < DIFFERENCE_THRESHOLD)
   {
     intdiff = (int) (difference*GRANULARITY);
     if (intdiff < TABLESIZE)
       return maximum + log_exp_tab[intdiff];
     else
       return maximum;
   }
   else
     return maximum;

   //return (maximum+log(1+exp(-difference)));
}

inline void LogDec_code ::SISO(double *ApriCode,double *ApriInfo,
                               double *OutInfo,double *OutCode,
                               int computeOutCode)
{
   int i,state,j;
   int numberofstates = trellis->numberofstates;
   int numberofedges = trellis->numberofedges;
   double sum;
```

```
   double Infty=1000000000.0;

   // Initialize the probability before loop
Alpha[0][0]=0.0;
   for (i=1; i<numberofstates; i++) Alpha[0][i]=-Infty;
   Beta[TrellisLengde][0]=0.0;
   for (i=1; i<numberofstates; i++) Beta[TrellisLengde][i]=-Infty;
    // Always terminated trellis..ending in zero state

   // Computation of the probability Alpha

   for (i=1; i<=TrellisLengde; i++)
    {
        // for all edges:

        for (state=0; state<numberofstates; state++)
                Gamma[state]=-Infty;
        for (int edgeno=0; edgeno<numberofedges; edgeno++)
                {
                int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam = Alpha[i-1][oldstate];

                for (int j = 0; j<InfoLengde; j++)
                  gam+=Mult(infobits[j],ApriInfo[InfoLengde*(i-1)+j]);

                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],ApriCode[n*(i-1)+j]);

                Gamma[newstate]=Max2(gam,Gamma[newstate]);

                }

        for (state=0; state<numberofstates; state++)
                {
                Alpha[i][state]=Gamma[state];

                }

       sum=Alpha[i][0];
       for (j=1; j<numberofstates; j++) if (Alpha[i][j]>sum) sum=Alpha[i][j];
       for (j=0; j<numberofstates; j++) Alpha[i][j]=Alpha[i][j]-sum;
   }

   // Computation of probability Beta

   for (i=TrellisLengde-1; i>=0; i--)
    {
    for (state=0; state<numberofstates; state++)
    Gamma[state]=-Infty;
```

```
 // for all edges:

      for (int edgeno=0; edgeno<numberofedges; edgeno++)
       {
       int *infobits = trellis->edge[edgeno].u;
       int oldstate = trellis->edge[edgeno].start;
       int newstate = trellis->edge[edgeno].end;
       int *codebits = trellis->edge[edgeno].c;

      double gam = Beta[i+1][newstate];

              for (int j = 0; j<InfoLengde; j++)
gam+=Mult(infobits[j],ApriInfo[InfoLengde*i+j]);

              for (int j = 0; j<n; j++)
                gam+=Mult(codebits[j],ApriCode[n*i+j]);

              Gamma[oldstate]=Max2(gam,Gamma[oldstate]);
              }

      for (state=0; state<numberofstates; state++)
     {
        Beta[i][state]=Gamma[state];
     }
     sum=Beta[i][0];
     for (j=1; j<numberofstates; j++) if (Beta[i][j]>sum) sum=Beta[i][j];
     for (j=0; j<numberofstates; j++) Beta[i][j]=Beta[i][j]-sum;
  }


 // Computation of the soft output, likelihood ratio of symbols in the frame
 for (i=1; i<=TrellisLengde; i++)
 {
      for (int infobitno=0;infobitno<InfoLengde;infobitno++)
  {
              gamma[infobitno][0]=-Infty;
 gamma[infobitno][1]=-Infty;
 }

 //gamma[0]=-Infty;
 //gamma[1]=-Infty;

 for (int edgeno=0; edgeno<numberofedges; edgeno++)
 {
 int *infobits = trellis->edge[edgeno].u;
              int oldstate = trellis->edge[edgeno].start;
              int newstate = trellis->edge[edgeno].end;
              int *codebits = trellis->edge[edgeno].c;

              double gam=Alpha[i-1][oldstate]+Beta[i][newstate];

              //for (int j = InfoLengde; j<n; j++)
              //  gam+=Mult(codebits[j],ApriCode[n*(i-1)+j]);
```

```
                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],ApriCode[n*(i-1)+j]);

                for (int infobitno=0;infobitno<InfoLengde;infobitno++)
                  {
  double gam1 = gam;
                  int infobit = infobits[infobitno];
                  for (int j = 0; j<InfoLengde; j++)
                    if (j != infobitno)
    {
                      gam1+=Mult(infobits[j],ApriInfo[InfoLengde*(i-1)+j]);
                      //gam1+=Mult(codebits[j],ApriCode[n*(i-1)+j]);
    }

                     gamma[infobitno][infobit]=
                   Max2(gam1,gamma[infobitno][infobit]);
  }
                }

        for (int infobitno=0;infobitno<InfoLengde;infobitno++)
    OutInfo[InfoLengde*(i-1)+infobitno]=
                  gamma[infobitno][1]-gamma[infobitno][0];
    }


   if (computeOutCode) // computeOutCode ==1: Brukes i SISOouter
    for (i=1; i<=TrellisLengde; i++)
    {
    for (int codebitno=0;codebitno<n;codebitno++)
    {
                gammac[codebitno][0]=-Infty;
   gammac[codebitno][1]=-Infty;
   }

   for (int edgeno=0; edgeno<numberofedges; edgeno++)
   {
   int *infobits = trellis->edge[edgeno].u;
                  int oldstate = trellis->edge[edgeno].start;
                  int newstate = trellis->edge[edgeno].end;
                  int *codebits = trellis->edge[edgeno].c;

                double gam=Alpha[i-1][oldstate]+Beta[i][newstate];

                       for (int j=0;j<InfoLengde;j++)
                  gam+=Mult(infobits[j],ApriInfo[InfoLengde*(i-1)+j]);

                for (int codebitno=0;codebitno<n;codebitno++)
                {
double gam1 = gam;
                       int codebit = codebits[codebitno];
                 for (int j = 0; j<n; j++)
                   if (j != codebitno)
                       gam1+=Mult(codebits[j],ApriCode[n*(i-1)+j]);
```

```
                gammac[codebitno][codebit]=
                     Max2(gam1,gammac[codebitno][codebit]);

                }
    }
    for (int codebitno=0;codebitno<n;codebitno++)
    OutCode[n*(i-1)+codebitno]=
                gammac[codebitno][1]-gammac[codebitno][0];
    }

  // End Computation of OutCode


#if defined(___TEST__SKRIVUT)
   FILE *fp = fopen("Alpha_org.txt","w");
   fprintf(fp,"ny SISO\n");
   for (i=1; i<=TrellisLengde; i++)
    {
   for (state=0; state<numberofstates; state++)
    {
    fprintf(fp,"i=%d,state=%d: Alpha=%1.5f Beta=%1.5f \n",
    i,state,Alpha[i][state],
    Beta[i][state]);
    }
    //printf("OutInfo[%d]=%1.5f,OutCode=%1.5f,%1.5f\n",
    // i-1,OutInfo[i-1],OutCode[n*(i-1)],OutCode[n*(i-1)+1]);
//fprintf(fp,"ApriInfo[%d]=%1.5f,ApriCode=%1.5f\n",
    //       i-1,ApriInfo[i-1],ApriCode[n*(i-1)]);
    }
   fclose(fp);
#endif

}
```

B.3 Modifisert SimulatorQAM

```
// Simulator of a pragmatic turbo coded modulation scheme
//
// define __QAM_MOD__ for higher order modulation
// if __QAM_MOD__ is not defined, binary modulation is used

#include <stdio.h>
#include <math.h>
#include <time.h>

/*************************************************************/

FILE *fresultat;
FILE *parameters;
FILE *fp_report;
```

```
#define PERFECT_CHANNEL_ESTIM

//#define __QAM_MOD__

/*************************************************************/

#if !defined(__CONSTELLATION__H)
#include "../Auxiliary/Constellation.h"
#endif

#if !defined(__READGEN_H)
#include "../Auxiliary/ReadGen.h"
#endif

#if !defined(__PARALLEL__H)
#include "../ParSchem/ParSchem.h"
#endif

#if !defined(__CHANNEL_AWGN__H)
#include "../Channels/Channel_AWGN.h"
#endif

#if !defined(__DATAGEN__H)
#include "../Auxiliary/DataGen.h"
#endif

//*************************************************************************

int d_Hamming(int *c1, int *c2, int length)
{
int distance = 0;
for (int i=0;i<length;i++)
if(c1[i] != c2[i])
distance++;
return distance;
}

//*************************************************************************

int main(int argc,char *argv[])
{

   //************* INITIALISERING AV MODELL *************


   Code_itr *My_code;

   My_code = new Parallel(argv[1]);     int code_type = 0;


   //**************************************************

   FILE *fp_codeword;
```

```
int MinErr=500,MaxRound=500,MinRound=20;
printf("\nPlease enter Min. number of errors: ");
scanf("%d",&MinErr);
printf("\nPlease enter Minimum number of frames and Maximum number of frames: ");
scanf("%d %d",&MinRound,&MaxRound);

int *Info,*codeword,*sentcodeword,*codeword_int,*reencoded;
int *reencoded_int;
double mean_num_of_itr;

double SNRdB;

sentcodeword = new int[My_code->code_length];
codeword_int = new int[My_code->code_length];
reencoded_int = new int[My_code->code_length];

printf("\nPlease enter SNRdB in simulation ");
scanf(" %lf",&SNRdB);

double *airword = new double[My_code->code_length];
double *airword_int;
double eff_rate;

int *received;
int *received2;
int i;
int millionbits;
int Teller2;

printf("Antall iterasjoner %d\n",My_code->AntIter);

int eff_code_length = My_code->code_length;
eff_rate = (double) (1.0*My_code->info_length)/(1.0*eff_code_length);


//***************** CHOOSE THE CHANNEL *************************

int num_channel;
Channel **My_channel;

//-------------------------------------------------------------
// Additive white Gaussian noise channel
//-------------------------------------------------------------

num_channel=1;
My_channel = new Channel*[num_channel];

for (i=0;i<num_channel;i++)
   My_channel[i] = new Channel_AWGN(SNRdB,eff_code_length,eff_rate);


//-----------------------------------------------------------------------

int ll;
```

```
   int num_snr = 5;
   double *snr_vector;
   snr_vector = new double[num_snr];
   for (ll=0;ll<num_snr;ll++)
     snr_vector[ll] = (double) (ll*0.25+SNRdB);

   double *QS=new double[eff_code_length];
   double *QS_deint = new double[eff_code_length];

   printf("The Frame length is %d \n",eff_code_length);
   printf("The Rate is %1.2f\n",eff_rate);
   printf("The simulation will be done for SNR = %1.3f dB\n",SNRdB);

   RandData cg(My_code->info_length);

   //---------------------------------------------------------------------
   // Signal constellation mapping
   //---------------------------------------------------------------------

#if defined(__QAM_MOD__)
   //int m = 3;                // 64-QAM
   //int m = 2;                  // 16-QAM
   int m = 1;               // BPSK ?

   int num_symbols = eff_code_length/m;
   Constellation *signal;
   signal = new Constellation(m,eff_code_length);

   if (eff_code_length%m != 0)
   {
     printf("Error: Incorrect info length!\n");
     return 0;
   }
#endif

   //---------------------------------------------------------------------
   // Printing of parameters to the file 'parameters.txt'
   //---------------------------------------------------------------------

   parameters = fopen("parameters.txt","w");
   fprintf(parameters,"\n");

#if !defined(PERFECT_CHANNEL_ESTIM)
   fprintf(parameters,"Estimation of channel parameters is performed.\n");
   fprintf(parameters,"\n");
#else
   fprintf(parameters,"Perfect channel estimates are assumed.\n");
   fprintf(parameters,"\n");
#endif

   My_code->print_parameter_file(parameters);
   My_channel[0]->print_parameter_file(parameters);
```

```
   fprintf(parameters,"Minimum number of errors: %d\n",MinErr);
   fprintf(parameters,"Minimum number of frames: %d\n",MinRound);
   fprintf(parameters,"Maximum number of frames: %d\n",MaxRound);
   fprintf(parameters,"Minimum of two iterations\n");
   fprintf(parameters,"\n");

#if defined(__QAM_MOD__)
   signal->print_parameter_file(parameters);
#endif

   fprintf(parameters,"Initial SNR in dB: %.2f\n",SNRdB);
   fprintf(parameters,"\n");
   fclose(parameters);


   //*******************************************************
   //
   // Random interleaver....
   //
   //*******************************************************

   int *leaver = new int[My_code->code_length];
   int *leaver_inv = new int[My_code->code_length];
   int num_cand = My_code->code_length;

   int map_int;
   printf("Read mapping interleaver from file? (yes(1) or no(0)) ");
   scanf("%d",&map_int);

   if (map_int == 0)
   {
     for (i=0;i<My_code->code_length;i++)
       leaver_inv[i] = i;

     // Building interleaver...

     while (num_cand > 0)
     {
       int index1 = rand() % num_cand;
       leaver[num_cand-1] = leaver_inv[index1];
       leaver_inv[index1] = leaver_inv[num_cand-1];
       num_cand--;
     }
   }
   else
   {
     FILE *finter;

     if ((finter=fopen("Mapinterleaver.txt","r"))== NULL)
       printf("\nCould not open the file. Terminates....\n");
     else
       for (i=0;i<My_code->code_length;i++)
         fscanf(finter,"%d",&leaver[i]);
     fclose(finter);
```

```
   }

   for (i=0;i<My_code->code_length;i++)
     leaver_inv[leaver[i]] = i;

   parameters = fopen("Mapinterleaver.txt","w");
   for (i=0;i<My_code->code_length;i++)
     fprintf(parameters,"%d ",leaver[i]);

   fprintf(parameters,"\n");
   fprintf(parameters,"\n");
   fprintf(parameters,"Interleaver used before mapping to constellation points\n");
   fprintf(parameters,"The interleaver is randomly generated\n");
   fprintf(parameters,"\n");
   fclose(parameters);

   //*******************************************************
   //
   // Simulation of different signal-to-noise ratios...
   //
   //*******************************************************

   time_t t1,t2,t_start,t_end;

   (void) time(&t1);
   (void) time(&t_start);

   for (ll=0;ll<num_snr;ll++)
   {
     int i;
     SNRdB = snr_vector[ll];
     int frames=0;
     int Ant_blocks_inerror=0;
     int Ant_wrong_blocks=0;
     int Ant=0;
     double kjisum=0.0;

     millionbits = 1000000 / My_code->info_length;

#if defined(__QAM_MOD__)
     double noise_var = (double) (1<<(2*m)-1)/
                        (double) (12.*m*eff_rate*pow(10.,SNRdB/10.));
#endif

     // Reinitialize the channels...
     for (i=0;i<num_channel;i++)
     {
       My_channel[i]->ReInit(SNRdB);

#if defined(__QAM_MOD__)
       My_channel[i]->ReInitnoise(noise_var);
#endif

     }
```

```
#if defined(__QAM_MOD__)
     printf("Noise variance: %f\n",noise_var);
#endif

     while (((Ant_blocks_inerror<MinErr) && (frames<MaxRound)) ||
            (frames<MinRound))
     {
       Info=cg.Data();

       codeword=My_code->Encode(Info);


       //---------------------------------------
       //HER SKJER INVERTERING
       //Kodeord inverteres dersom dette ønskes.
       //---------------------------------------

       for (i=0; i<My_code->code_length; i++){
 sentcodeword[i]=codeword[i];

//   if (true) {         //Invert? y=true n=false
 if(Info[0] == 0) {
   if (codeword[i] == 1)
     codeword[i] = 0;
   else
     codeword[i] = 1;
 }
       }

       for (i=0; i<My_code->code_length; i++)
         codeword_int[leaver[i]] = codeword[i];

#if defined(__QAM_MOD__)
       airword=signal->generate_QAM_seq(codeword_int);
#else
       for (i=0;i<My_code->code_length;i++)
         airword[i] = 2.*codeword_int[i]-1.;
#endif

       for (i=0;i<num_channel;i++)
         My_channel[i]->Channel_mod(airword);


       //-------------------------------------------------
       // Demodulation and decoding
       //-------------------------------------------------

#if defined(__QAM_MOD__)
       QS = signal->bit_channel(My_channel[0]->data_seq_out,
                                My_channel[0]->noise_var);
#else
       double *recseq = My_channel[0]->data_seq_out;
       for (i=0;i<My_code->code_length;i++)
```

```
         QS[i] = 2./My_channel[0]->noise_var*recseq[i];
#endif

       for (i=0; i<My_code->code_length; i++) {
         QS_deint[i] = QS[leaver[i]];
        //printf(" %f ", QS_deint[i]);  //Utskrift av kanal verdier
 //if(QS_deint[i]>0) printf("1");
// else printf("0");
}
       //printf("\n\n");

       received = My_code->Decode(QS_deint);

       //_-_-_-_-_-_-Anti inverter-_-_-_-_-_-_-_
       //
       // Husk at dette er snakk om KANALVERDIER ikke bineære verdier
       // Ved BPSK er det bare å bytte fortegn
       // SKAL IKKE BRUKES ved normale forhold
       if(false) {
 for (i=0; i<My_code->code_length; i++){
   QS_deint[i] = QS_deint[i] * -1;
 }
 Teller2++;
       }


       //--------------------------------------------------
       //Count the number of errors:
       //--------------------------------------------------

       {
         int old_Ant;
     double d2sent,d2received;

 old_Ant = Ant;
         d2sent= 0.; d2received = 0.;


//*****  Utskrift av sendt og mottatt kodeord.
/*  for (i=0; i<My_code->info_length; i++) {
   printf("%d", Info[i]);
 }

 printf("\n\n");

 for (i=0; i<My_code->info_length; i++) {
   printf("%d", received[i]);
 }
*/
//***
     for (i=0; i<My_code->info_length; i++) {
           if (Info[i]!=received[i])
       {
         //printf("[%d]",i);
```

```
        Ant++;
      }
}


    if (Ant>old_Ant)
    {
          Ant_blocks_inerror++;
          reencoded = My_code->Encode(received);



          fp_codeword = fopen("Codewords.txt","a");

          fprintf(fp_codeword,"Codeword, SNR = %f dB\n",SNRdB);
          for (i=0; i<My_code->code_length; i++)
            if (sentcodeword[i]!=reencoded[i])
              fprintf(fp_codeword,"[%d]",i);

          fprintf(fp_codeword,"\nPositions in info seq., SNR = %f dB\n",SNRdB);
          for (i=0; i<My_code->info_length; i++)
            if (Info[i]!=received[i])
              fprintf(fp_codeword,"[%d]",i);

          //-------------------------------------------------------

          for (i=0;i<num_channel;i++)
            d2sent += My_channel[i]->distance2(airword);

          for (i=0;i<My_code->code_length;i++)
            reencoded_int[leaver[i]] = reencoded[i];

          for (i=0;i<num_channel;i++)
          {
#if defined(__QAM_MOD__)
            airword=signal->generate_QAM_seq(reencoded_int);
#else
            for (int j=0;j<My_code->code_length;j++)
              airword[j] = 2.*reencoded_int[j]-1.;
#endif
            d2received += My_channel[i]->distance2(airword);
   }

       kjisum += d2sent;

       printf("Code distance: %d, d2(sent,y)=%1.3f, d2(decoded,y)=%1.3f",
              d_Hamming(sentcodeword,reencoded,
              My_code->code_length),d2sent,d2received);

          fprintf(fp_codeword,"\n");
          fprintf(fp_codeword,"Code distance: %d, d2(sent,y)=%1.3f, d2(decoded,y)=%1.3f",
                  d_Hamming(sentcodeword,reencoded,
              My_code->code_length),d2sent,d2received);
```

```
      if (d2sent < d2received)
      {
        printf("**** wrong decoding");
            fprintf(fp_codeword,"**** wrong decoding");
        Ant_wrong_blocks++;
      }
          fprintf(fp_codeword,"\nThe effective number of iterations is: %d\n",
                  My_code->eff_AntIter);

          fprintf(fp_codeword,"\n");
          fprintf(fp_codeword,"\n");
          fclose(fp_codeword);

      printf("\n");
          printf("The effective number of iterations is: %d\n",
                  My_code->eff_AntIter);
    }
      }

      frames++;
      if (frames % millionbits ==0)
      {
        double mean_itr = My_code->get_mean_num_of_itr();
FILE *fp_0 = fopen("Output.txt","a");
        (void) time(&t_end);

     printf(" %d frames completed. %d errors so far. %d block errors, %d incorrectly decoded
                frames, Ant,Ant_blocks_inerror, Ant_wrong_blocks);

        fprintf(fp_0," %d frames completed. %d errors so far. %d block errors, %d incorrectl
                    frames, Ant,Ant_blocks_inerror, Ant_wrong_blocks);

        printf("Mean number of iterations so far: %1.3f\n",mean_itr);
        printf("Used time for the last %d frames: %d\n",millionbits,
(int) t_end-t_start);
        fprintf(fp_0,"Used time for the last %d frames: %d\n",millionbits,
                (int) t_end-t_start);
        fprintf(fp_0,"Mean number of iterations so far: %1.3f\n",mean_itr);
        (void) time(&t_start);
        fclose(fp_0);
      }
    }//SLUTT PÅ SIMULERINGSLOOP

    //-------------------------------------------------------------------
    // A specific value of the signal-to-noise ratio is completed, and
    // results are written to file...
    //
    //-------------------------------------------------------------------

    fresultat = fopen("feilrate.txt","a");
    fprintf(fresultat,"%1.2f\t%1.8f\t%1.8f\n",SNRdB,
           (float)Ant/((float)(frames*My_code->info_length)),
           (float)Ant_blocks_inerror/((float)(frames)));
    fclose(fresultat);
```

```
      fp_report = fopen("report_error.txt","a");
      mean_num_of_itr = My_code->get_mean_num_of_itr();
      My_code->report_errors_file(fp_report);

      fprintf(fp_report,"[%1.2f,log10(%1.8f)], %d bit errors, %d frames (out of %d) in error.
            (float)Ant/((float)(frames*My_code->info_length)),
            Ant,Ant_blocks_inerror,frames,Ant_wrong_blocks,mean_num_of_itr);

      fprintf(fp_report,"\n");
      fclose(fp_report);

      printf("[%1.2f,log10(%1.8f)], %d bit errors, %d frames (out of %d) in error. %d incorrec
           (float)Ant/((float)(frames*My_code->info_length)),
           Ant,Ant_blocks_inerror,frames,Ant_wrong_blocks,mean_num_of_itr);

      printf("Corrected 180 deg rotation on %d out of %d frames\n", Teller2, frames);

      Teller2 = 0;
      My_code->reset_errors();
   }

   delete []QS;

   (void) time(&t2);
   printf("Tidsforbruk: %d\n",(int) t2-t1);
   return 1;
}
```

B.4 Modifisert LogDec_code, uten rotasjonsparameter

```
#define _LOGDEC_CODE__H

#if !defined(_LOGDEC__H)
#include "LogDec.h"
#endif

#if !defined(_TRELLIS_CODE__H)
#include "Trellis_code_ext.h"
#endif

class LogDec_code: public LogDec
{
   public:

   LogDec_code(int _k,int _n,int trellis_lengde,Trellis_code *_trellis);
   ~LogDec_code();

   double SISO(double *ApriCode,double *ApriInfo,
     double *OutInfo,double *OutCode,
              double rot, int computeOutCode);
```

```
   double Max2(double tall1,double tall2);

   private:
Trellis_code *trellis;
       double **gamma1,**gamma2, **gammac, *Delta_G, **Delta_A;
};


LogDec_code :: LogDec_code(int _k,int _n,int trellis_lengde,
                           Trellis_code *_trellis):LogDec()
{
int i;
n=_n;
    InfoLengde=_k;
    TrellisLengde = trellis_lengde;
    trellis = _trellis;

gamma1 = new double*[InfoLengde];
gamma2 = new double*[InfoLengde];

   gammac = new double*[n];
   for (i=0; i<n; i++)
      gammac[i] = new double[2];

        for (i=0; i<InfoLengde; i++)
      gamma1[i] = new double[2];

for (i=0; i<InfoLengde; i++)
  gamma2[i] = new double[2];


Gamma = new double[(trellis->numberofstates)*2];
Delta_G =  new double[(trellis->numberofstates)*2];

   Alpha=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Alpha[i]=new double[(trellis->numberofstates)*2];

Delta_A = new double*[TrellisLengde+1];
for (i=0; i<TrellisLengde+1; i++)
  Delta_A[i]=new double[(trellis->numberofstates)*2];

    Beta=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Beta[i]=new double[(trellis->numberofstates)*2];
}

LogDec_code :: ~LogDec_code()
{
   int i;

   for (i=0; i<TrellisLengde+1; i++)
    {
```

```
    delete []Alpha[i];
       delete []Beta[i];
delete []Delta_A[i];
    }
   delete []Alpha;
   delete []Beta;
   delete []Gamma;
   delete []Delta_A;
   delete []Delta_G;


   for (i=0; i<n; i++)
     delete []gammac[i];

   for (i=0; i<InfoLengde; i++)
     delete []gamma1[i];

   for (i=0; i<InfoLengde; i++)
     delete []gamma2[i];


   delete []gammac;
   delete []gamma1;
   delete []gamma2;
}

inline double LogDec_code :: Max2(double tall1,double tall2)
{
   double maximum, difference,dummy;
   int intdiff;

   if (tall1 > tall2)
    {
    maximum=tall1;
    difference = tall1-tall2;//+maximum;
    }
   else
      {
    maximum=tall2;
    difference = tall2-tall1;//+maximum;
    }

   //intdiff = (int) floor(difference*GRANULARITY);
   if (difference < DIFFERENCE_THRESHOLD)
   {
     intdiff = (int) (difference*GRANULARITY);
     if (intdiff < TABLESIZE)
       return maximum + log_exp_tab[intdiff];
     else
       return maximum;
   }
   else
     return maximum;
```

```
   //return (maximum+log(1+exp(-difference)));
}

inline double LogDec_code ::SISO(double *ApriCode,double *ApriInfo,
                                 double *OutInfo,double *OutCode,
                                 double rot, int computeOutCode)
{
   int i,state,j;
   int numberofstates = trellis->numberofstates;
   int numberofedges = trellis->numberofedges;
   double sum;
   double Infty=1000000000.0;
   double teller1 = 0; //NY
   double teller2 = 0; //NY


   // Initialize the probability before loop

   for (i=1; i<numberofstates*2; i++) Alpha[0][i]=-Infty;
Alpha[0][0]=0.0;
Alpha[0][numberofstates] = 0.0;


   for (i=1; i<numberofstates*2; i++) Beta[TrellisLengde][i]=-Infty;
   Beta[TrellisLengde][0]=0.0;
Beta[TrellisLengde][numberofstates] = 0.0;

    // Always terminated trellis..ending in zero state

rot = 0;

   // Computation of the probability Alpha

   for (i=1; i<=TrellisLengde; i++)
 {


       for (state=0; state<numberofstates*2; state++)
 Gamma[state]=-Infty;


        for (int edgeno=0; edgeno<numberofedges; edgeno++)
                {
                int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam = (Alpha[i-1][oldstate] + rot);

                for (int j = 0; j<InfoLengde; j++)
                  gam+=Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j])));

                for (int j = 0; j<n; j++)
```

```
                  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j]));

                Gamma[newstate]=Max2(gam,Gamma[newstate]);

                }


       for (int edgeno=0; edgeno<numberofedges; edgeno++)           //NY
              {
              int *infobits = trellis->edge[edgeno].u;             //NY
              int oldstate = trellis->edge[edgeno].start;          //NY
              int newstate = trellis->edge[edgeno].end;            //NY
              int *codebits = trellis->edge[edgeno].c;             //NY

              double gam = (Alpha[i-1][numberofstates + oldstate] - rot);  //NY

              for (int j = 0; j<InfoLengde; j++)                    //NY
                gam+=(Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j]))));//NY

              for (int j = 0; j<n; j++) {                             //NY
                gam+=( Mult((1-codebits[j]),(ApriCode[n*(i-1)+j]))); // Endre her for å inve
//printf("gam =%d", gam);
      }

               Gamma[newstate + numberofstates]=Max2(gam,Gamma[newstate + numberofstates]);

               }                                                     //NY


       for (state=0; state<numberofstates*2; state++)
              {
              Alpha[i][state]=Gamma[state];

              }


      sum=Alpha[i][0];
      for (j=1; j<numberofstates*2; j++) if (Alpha[i][j]>sum) sum=Alpha[i][j];
    for (j=0; j<numberofstates*2; j++) Alpha[i][j]=Alpha[i][j]-sum;

}


  // Computation of probability Beta

  for (i=TrellisLengde-1; i>=0; i--)
   {
   for (state=0; state<numberofstates*2; state++)
   Gamma[state]=-Infty;

  // for all edges:

       for (int edgeno=0; edgeno<numberofedges; edgeno++)
        {
```

```
        int *infobits = trellis->edge[edgeno].u;
        int oldstate = trellis->edge[edgeno].start;
        int newstate = trellis->edge[edgeno].end;
        int *codebits = trellis->edge[edgeno].c;

       double gam = (Beta[i+1][newstate] + rot);

               for (int j = 0; j<InfoLengde; j++)
  gam+=Mult(infobits[j],(ApriInfo[InfoLengde*i+j]));

               for (int j = 0; j<n; j++)
                 gam+=Mult(codebits[j],(ApriCode[n*i+j])); // Endre her for å invertere

               Gamma[oldstate]=Max2(gam,Gamma[oldstate]);
               }



    for (int edgeno=0; edgeno<numberofedges; edgeno++)             //NY
       {
        int *infobits = trellis->edge[edgeno].u;             //NY
        int oldstate = trellis->edge[edgeno].start;          //NY
        int newstate = trellis->edge[edgeno].end;            //NY
        int *codebits = trellis->edge[edgeno].c;             //NY

       double gam = (Beta[i+1][numberofstates + newstate] - rot);   //NY

               for (int j = 0; j<InfoLengde; j++)                   //NY
  gam+=(Mult(infobits[j],(ApriInfo[InfoLengde*i+j]))); //NY

               for (int j = 0; j<n; j++)
                 gam+=(Mult((1 - codebits[j]),(ApriCode[n*i+j])));          // Endre her for

               Gamma[oldstate + numberofstates]=Max2(gam,Gamma[oldstate + numberofstates]);/
               }

       for (state=0; state<numberofstates*2; state++)
      {
         Beta[i][state]=Gamma[state];
      }

sum=Beta[i][0];
       for (j=1; j<numberofstates*2; j++) if (Beta[i][j]>sum) sum=Beta[i][j];
       for (j=0; j<numberofstates*2; j++) Beta[i][j]=Beta[i][j]-sum;
    }



   // Computation of the soft output, likelihood ratio of symbols in the frame
   for (i=1; i<=TrellisLengde; i++)
   {
        for (int infobitno=0;infobitno<InfoLengde;infobitno++)
    {
               gamma1[infobitno][0]=-Infty;
```

```
gamma2[infobitno][0]=-Infty;
   gamma1[infobitno][1]=-Infty;
gamma2[infobitno][1]=-Infty;
   }

   //gamma[0]=-Infty;
   //gamma[1]=-Infty;

   for (int edgeno=0; edgeno<numberofedges; edgeno++)
   {
   int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam= (Alpha[i-1][oldstate]+Beta[i][newstate] + rot);

                //for (int j = InfoLengde; j<n; j++)
                //  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j]));

                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for å invertere

                for (int infobitno=0;infobitno<InfoLengde;infobitno++)
                  {
  double gam1 = gam;
                  int infobit = infobits[infobitno];
                  for (int j = 0; j<InfoLengde; j++)
                    if (j != infobitno)
    {
                      gam1+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j]));
                      //gam1+=Mult(codebits[j],ApriCode[n*(i-1)+j]);
    }

  //teller1+=gam1;

                    gamma1[infobitno][infobit]=
                 Max2(gam1,gamma1[infobitno][infobit]);  //
  }
                }


   for (int edgeno=0; edgeno<numberofedges; edgeno++)          //NY
   {
   int *infobits = trellis->edge[edgeno].u;            //NY
                int oldstate = trellis->edge[edgeno].start;         //NY
                int newstate = trellis->edge[edgeno].end;           //NY
                int *codebits = trellis->edge[edgeno].c;            //NY

                double gam=(Alpha[i-1][numberofstates + oldstate]+Beta[i][numberofstates + ne

                for (int j = 0; j<n; j++)
                  gam+=(Mult((1-codebits[j]),(ApriCode[n*(i-1)+j]))); // Endre her for å inve
```

```
                for (int infobitno=0;infobitno<InfoLengde;infobitno++) //NY
                  {
  double gam1 = gam;                                    //NY
                  int infobit = infobits[infobitno];                 //NY
                  for (int j = 0; j<InfoLengde; j++)                 //NY
                    if (j != infobitno)                              //NY
    {                                                   //NY
                      gam1+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j])); //NY
    }                                                   //NY

 // teller2+=gam1;

                    gamma1[infobitno][infobit]=
                 Max2(gam1,gamma1[infobitno][infobit]); //NY NÅ ER DET BARE EN GAMMA TABELL I
  }
                }                                                    //NY



if(true) {
  for (int infobitno=0;infobitno<InfoLengde;infobitno++)
   OutInfo[InfoLengde*(i-1)+infobitno]=
     gamma1[infobitno][1]-gamma1[infobitno][0];   //"Return" statement 1
}

else {
 for (int infobitno=0;infobitno<InfoLengde;infobitno++)
   OutInfo[InfoLengde*(i-1)+infobitno]=
      gamma2[infobitno][1]-gamma2[infobitno][0];   //"Return" statement 2 BRUKES IKKE
}
}


   if (computeOutCode) // computeOutCode ==1: Brukes i SISOouter
    for (i=1; i<=TrellisLengde; i++)
    {
    for (int codebitno=0;codebitno<n;codebitno++)
    {
                gammac[codebitno][0]=-Infty;
   gammac[codebitno][1]=-Infty;
   }

   for (int edgeno=0; edgeno<numberofedges; edgeno++)
   {
   int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam=Alpha[i-1][oldstate]+Beta[i][newstate];

                       for (int j=0;j<InfoLengde;j++)
                 gam+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j]));
```

```
                for (int codebitno=0;codebitno<n;codebitno++)
                {
double gam1 = gam;
                        int codebit = codebits[codebitno];
                 for (int j = 0; j<n; j++)
                   if (j != codebitno)
                        gam1+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for å inv

                 gammac[codebitno][codebit]=
                      Max2(gam1,gammac[codebitno][codebit]);

                 }
    }
    for (int codebitno=0;codebitno<n;codebitno++)
    OutCode[n*(i-1)+codebitno]=
                 gammac[codebitno][1]-gammac[codebitno][0];
    }

  // End Computation of OutCode


#if defined(___TEST__SKRIVUT)
   FILE *fp = fopen("Alpha_org.txt","w");
   fprintf(fp,"ny SISO\n");
   for (i=1; i<=TrellisLengde; i++)
    {
   for (state=0; state<numberofstates; state++)
    {
    fprintf(fp,"i=%d,state=%d: Alpha=%1.5f Beta=%1.5f \n",
    i,state,Alpha[i][state],
    Beta[i][state]);
    }
    //printf("OutInfo[%d]=%1.5f,OutCode=%1.5f,%1.5f\n",
    // i-1,OutInfo[i-1],OutCode[n*(i-1)],OutCode[n*(i-1)+1]);
//fprintf(fp,"ApriInfo[%d]=%1.5f,ApriCode=%1.5f\n",
    //       i-1,ApriInfo[i-1],ApriCode[n*(i-1)]);
    }
   fclose(fp);

#endif

   return rot;

}
```

B.5 Modifisert LogDec_code med rotasjonsparameter

```
#define _LOGDEC_CODE__H
```

```
#if !defined(_LOGDEC__H)
#include "LogDec.h"
#endif

#if !defined(_TRELLIS_CODE__H)
#include "Trellis_code_ext.h"
#endif

class LogDec_code: public LogDec
{
   public:

   LogDec_code(int _k,int _n,int trellis_lengde,Trellis_code *_trellis);
   ~LogDec_code();

   double SISO(double *ApriCode,double *ApriInfo,
     double *OutInfo,double *OutCode,
             double rot, int computeOutCode);

   double Max2(double tall1,double tall2);

   private:
Trellis_code *trellis;
       double **gamma1,**gamma2, **gammac, *Delta_G, **Delta_A;
};


LogDec_code :: LogDec_code(int _k,int _n,int trellis_lengde,
                           Trellis_code *_trellis):LogDec()
{
int i;
n=_n;
    InfoLengde=_k;
    TrellisLengde = trellis_lengde;
    trellis = _trellis;

gamma1 = new double*[InfoLengde];
gamma2 = new double*[InfoLengde];

   gammac = new double*[n];
   for (i=0; i<n; i++)
      gammac[i] = new double[2];

        for (i=0; i<InfoLengde; i++)
      gamma1[i] = new double[2];

for (i=0; i<InfoLengde; i++)
  gamma2[i] = new double[2];


Gamma = new double[(trellis->numberofstates)*2];
Delta_G =  new double[(trellis->numberofstates)*2];
```

```
   Alpha=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Alpha[i]=new double[(trellis->numberofstates)*2];

Delta_A = new double*[TrellisLengde+1];
for (i=0; i<TrellisLengde+1; i++)
  Delta_A[i]=new double[(trellis->numberofstates)*2];

    Beta=new double*[TrellisLengde+1];
    for (i=0; i<TrellisLengde+1; i++)
    Beta[i]=new double[(trellis->numberofstates)*2];
}

LogDec_code :: ~LogDec_code()
{
   int i;

   for (i=0; i<TrellisLengde+1; i++)
    {
    delete []Alpha[i];
       delete []Beta[i];
delete []Delta_A[i];
    }
   delete []Alpha;
   delete []Beta;
   delete []Gamma;
   delete []Delta_A;
   delete []Delta_G;


   for (i=0; i<n; i++)
     delete []gammac[i];

   for (i=0; i<InfoLengde; i++)
     delete []gamma1[i];

   for (i=0; i<InfoLengde; i++)
     delete []gamma2[i];


   delete []gammac;
   delete []gamma1;
   delete []gamma2;
}

inline double LogDec_code :: Max2(double tall1,double tall2)
{
   double maximum, difference,dummy;
   int intdiff;

   if (tall1 > tall2)
    {
    maximum=tall1;
    difference = tall1-tall2;//+maximum;
```

```
    }
   else
      {
    maximum=tall2;
    difference = tall2-tall1;//+maximum;
    }

   //intdiff = (int) floor(difference*GRANULARITY);
   if (difference < DIFFERENCE_THRESHOLD)
   {
     intdiff = (int) (difference*GRANULARITY);
     if (intdiff < TABLESIZE)
       return maximum + log_exp_tab[intdiff];
     else
       return maximum;
   }
   else
     return maximum;

   //return (maximum+log(1+exp(-difference)));
}

inline double LogDec_code ::SISO(double *ApriCode,double *ApriInfo,
                                 double *OutInfo,double *OutCode,
                                 double rot, int computeOutCode)
{
   int i,state,j;
   int numberofstates = trellis->numberofstates;
   int numberofedges = trellis->numberofedges;
   double sum;
   double Infty=1000000000.0;
   double teller1 = 0; //NY
   double teller2 = 0; //NY


   // Initialize the probability before loop

   for (i=1; i<numberofstates*2; i++) Alpha[0][i]=-Infty;
Alpha[0][0]=0.0;
Alpha[0][numberofstates] = 0.0;

   for (i=1; i<numberofstates*2; i++) Delta_A[0][i]=-Infty; //NY
Delta_A[0][0]=0.0;                                          //NY
Delta_A[0][numberofstates] = 0.0;                           //NY


   for (i=1; i<numberofstates*2; i++) Beta[TrellisLengde][i]=-Infty;
   Beta[TrellisLengde][0]=0.0;
Beta[TrellisLengde][numberofstates] = 0.0;

    // Always terminated trellis..ending in zero state
```

```
   // Computation of the probability Alpha


  for (i=1; i<=TrellisLengde; i++)
    {
        // for all edges:

       for (state=0; state<numberofstates*2; state++)
                Delta_G[state]=-Infty;


//-----------------------------------------------------------------
// Dette er ren .rot beregning. .rot parameteret brukes ikke i denne beregningen. //NY

      for (int edgeno=0; edgeno<numberofedges; edgeno++)
                {
                int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam = (Delta_A[i-1][oldstate]);

                for (int j = 0; j<InfoLengde; j++)
                  gam+=Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j])));

                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for Ã invertere

                Delta_G[newstate]=Max2(gam,Delta_G[newstate]);

                }



        for (int edgeno=0; edgeno<numberofedges; edgeno++)           //NY
               {
               int *infobits = trellis->edge[edgeno].u;             //NY
               int oldstate = trellis->edge[edgeno].start;          //NY
               int newstate = trellis->edge[edgeno].end;            //NY
               int *codebits = trellis->edge[edgeno].c;             //NY

               double gam = (Delta_A[i-1][numberofstates + oldstate]);  //NY

               for (int j = 0; j<InfoLengde; j++)                    //NY
                 gam+=(Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j]))));//NY

               for (int j = 0; j<n; j++)                             //NY
                 gam+=(Mult((1 -codebits[j]),(ApriCode[n*(i-1)+j]))); // Endre her for Ã inve

                Delta_G[newstate + numberofstates]=Max2(gam,Delta_G[newstate + numberofstates

                }                                                    //NY
```

```
         for (state=0; state<numberofstates*2; state++)
                 {
                 Delta_A[i][state]=Delta_G[state];

                }

//Normalisering over hele trellisen....
        sum=Delta_A[i][0];
        for (j=1; j<numberofstates*2; j++) if (Delta_A[i][j]>sum) sum=Delta_A[i][j];
       for (j=0; j<numberofstates*2; j++) Delta_A[i][j]=Delta_A[i][j]-sum;

}

        //Verdien til rot parameteret beregnes fra siste kolonne i tabellen
        // rot = teller1 - teller2

sum=0;
        for (j=0; j<numberofstates; j++) sum+=Delta_A[TrellisLengde][j];
   teller1 = sum;


sum=0;
        for (j=0; j<numberofstates; j++) sum+=Delta_A[TrellisLengde][j+numberofstates];
   teller2 = sum;

sum = 0;



//***********************************************

//   RE_INITIALIZATION   Vanlig alpha bregning, MED rot.

   for (i=1; i<=TrellisLengde; i++)
     {

       for (state=0; state<numberofstates*2; state++)
 Gamma[state]=-Infty;


//------------------------------------------------------------------- NORMAL:::

        for (int edgeno=0; edgeno<numberofedges; edgeno++)
                {
                int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam = (Alpha[i-1][oldstate] + rot);

                for (int j = 0; j<InfoLengde; j++)
                  gam+=Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j])));
```

```
                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for Ã invertere

                Gamma[newstate]=Max2(gam,Gamma[newstate]);

                }


       for (int edgeno=0; edgeno<numberofedges; edgeno++)           //NY
              {
              int *infobits = trellis->edge[edgeno].u;             //NY
              int oldstate = trellis->edge[edgeno].start;          //NY
              int newstate = trellis->edge[edgeno].end;            //NY
              int *codebits = trellis->edge[edgeno].c;             //NY

              double gam = (Alpha[i-1][numberofstates + oldstate] - rot);  //NY

              for (int j = 0; j<InfoLengde; j++)                    //NY
                gam+=(Mult(infobits[j],((ApriInfo[InfoLengde*(i-1)+j]))));//NY

              for (int j = 0; j<n; j++) {                             //NY
                gam+=( Mult((1-codebits[j]),(ApriCode[n*(i-1)+j]))); // Endre her for Ã inve
//printf("gam =%d", gam);
      }

                Gamma[newstate + numberofstates]=Max2(gam,Gamma[newstate + numberofstates]);

                }                                                     //NY


       for (state=0; state<numberofstates*2; state++)
                {
                Alpha[i][state]=Gamma[state];

                }


      sum=Alpha[i][0];
      for (j=1; j<numberofstates*2; j++) if (Alpha[i][j]>sum) sum=Alpha[i][j];
    for (j=0; j<numberofstates*2; j++) Alpha[i][j]=Alpha[i][j]-sum;




//***********************************************

}


   // Computation of probability Beta

   for (i=TrellisLengde-1; i>=0; i--)
```

```
    {
    for (state=0; state<numberofstates*2; state++)
    Gamma[state]=-Infty;

   // for all edges:

        for (int edgeno=0; edgeno<numberofedges; edgeno++)
         {
         int *infobits = trellis->edge[edgeno].u;
         int oldstate = trellis->edge[edgeno].start;
         int newstate = trellis->edge[edgeno].end;
         int *codebits = trellis->edge[edgeno].c;

        double gam = (Beta[i+1][newstate] + rot);

                for (int j = 0; j<InfoLengde; j++)
  gam+=Mult(infobits[j],(ApriInfo[InfoLengde*i+j]));

                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],(ApriCode[n*i+j])); // Endre her for Ã invertere

                Gamma[oldstate]=Max2(gam,Gamma[oldstate]);
                }



     for (int edgeno=0; edgeno<numberofedges; edgeno++)             //NY
        {
         int *infobits = trellis->edge[edgeno].u;             //NY
         int oldstate = trellis->edge[edgeno].start;          //NY
         int newstate = trellis->edge[edgeno].end;            //NY
         int *codebits = trellis->edge[edgeno].c;             //NY

        double gam = (Beta[i+1][numberofstates + newstate] - rot);   //NY

                for (int j = 0; j<InfoLengde; j++)                   //NY
  gam+=(Mult(infobits[j],(ApriInfo[InfoLengde*i+j]))); //NY

                for (int j = 0; j<n; j++)
                  gam+=(Mult((1 - codebits[j]),(ApriCode[n*i+j]))); // Endre her for Ã invert

                Gamma[oldstate + numberofstates]=Max2(gam,Gamma[oldstate + numberofstates]);
                }




        for (state=0; state<numberofstates*2; state++)
       {
          Beta[i][state]=Gamma[state];
       }

sum=Beta[i][0];
```

```
      for (j=1; j<numberofstates*2; j++) if (Beta[i][j]>sum) sum=Beta[i][j];
      for (j=0; j<numberofstates*2; j++) Beta[i][j]=Beta[i][j]-sum;
    }



   // Computation of the soft output, likelihood ratio of symbols in the frame
   for (i=1; i<=TrellisLengde; i++)
   {
        for (int infobitno=0;infobitno<InfoLengde;infobitno++)
    {
                gamma1[infobitno][0]=-Infty;
gamma2[infobitno][0]=-Infty;
   gamma1[infobitno][1]=-Infty;
gamma2[infobitno][1]=-Infty;
   }

   //gamma[0]=-Infty;
   //gamma[1]=-Infty;

   for (int edgeno=0; edgeno<numberofedges; edgeno++)
   {
   int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam= (Alpha[i-1][oldstate]+Beta[i][newstate] + rot);

                //for (int j = InfoLengde; j<n; j++)
                //  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j]));

                for (int j = 0; j<n; j++)
                  gam+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for Ã invertere

                for (int infobitno=0;infobitno<InfoLengde;infobitno++)
                  {
  double gam1 = gam;
                  int infobit = infobits[infobitno];
                  for (int j = 0; j<InfoLengde; j++)
                    if (j != infobitno)
    {
                      gam1+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j]));
                      //gam1+=Mult(codebits[j],ApriCode[n*(i-1)+j]);
    }

  //teller1+=gam1;

                      gamma1[infobitno][infobit]=
                 Max2(gam1,gamma1[infobitno][infobit]);
  }
                }
```

```
   for (int edgeno=0; edgeno<numberofedges; edgeno++)          //NY
   {
   int *infobits = trellis->edge[edgeno].u;            //NY
                int oldstate = trellis->edge[edgeno].start;         //NY
                int newstate = trellis->edge[edgeno].end;           //NY
                int *codebits = trellis->edge[edgeno].c;            //NY

                double gam=(Alpha[i-1][numberofstates + oldstate]+Beta[i][numberofstates + ne

                for (int j = 0; j<n; j++)
                  gam+=(Mult((1-codebits[j]),(ApriCode[n*(i-1)+j]))); // Endre her for Ã inve

                for (int infobitno=0;infobitno<InfoLengde;infobitno++) //NY
                  {
  double gam1 = gam;                                  //NY
                  int infobit = infobits[infobitno];                //NY
                  for (int j = 0; j<InfoLengde; j++)                //NY
                    if (j != infobitno)                             //NY
    {                                                 //NY
                      gam1+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j])); //NY
    }                                                 //NY

 // teller2+=gam1;

                    gamma1[infobitno][infobit]=
                 Max2(gam1,gamma1[infobitno][infobit]); //NY NÃ ER DET BARE EN GAMMA TABELL I
  }
                }                                                   //NY



  rot = teller1-teller2;

if(true) {
  for (int infobitno=0;infobitno<InfoLengde;infobitno++)
   OutInfo[InfoLengde*(i-1)+infobitno]=
     gamma1[infobitno][1]-gamma1[infobitno][0];   //"Return" statement 1
}

else {
 for (int infobitno=0;infobitno<InfoLengde;infobitno++)
   OutInfo[InfoLengde*(i-1)+infobitno]=
      gamma2[infobitno][1]-gamma2[infobitno][0];   //"Return" statement 2 BRUKES IKKE
}
}

   teller1 = 0;
   teller2 = 0;



   if (computeOutCode) // computeOutCode ==1: Brukes i SISOouter
    for (i=1; i<=TrellisLengde; i++)
    {
```

```
    for (int codebitno=0;codebitno<n;codebitno++)
    {
                gammac[codebitno][0]=-Infty;
   gammac[codebitno][1]=-Infty;
   }

   for (int edgeno=0; edgeno<numberofedges; edgeno++)
   {
   int *infobits = trellis->edge[edgeno].u;
                int oldstate = trellis->edge[edgeno].start;
                int newstate = trellis->edge[edgeno].end;
                int *codebits = trellis->edge[edgeno].c;

                double gam=Alpha[i-1][oldstate]+Beta[i][newstate];

                       for (int j=0;j<InfoLengde;j++)
                 gam+=Mult(infobits[j],(ApriInfo[InfoLengde*(i-1)+j]));

                for (int codebitno=0;codebitno<n;codebitno++)
                {
double gam1 = gam;
                       int codebit = codebits[codebitno];
                for (int j = 0; j<n; j++)
                  if (j != codebitno)
                       gam1+=Mult(codebits[j],(ApriCode[n*(i-1)+j])); // Endre her for Ã inv

                gammac[codebitno][codebit]=
                     Max2(gam1,gammac[codebitno][codebit]);

                }
    }
    for (int codebitno=0;codebitno<n;codebitno++)
    OutCode[n*(i-1)+codebitno]=
                gammac[codebitno][1]-gammac[codebitno][0];
    }

  // End Computation of OutCode


#if defined(___TEST__SKRIVUT)
   FILE *fp = fopen("Alpha_org.txt","w");
   fprintf(fp,"ny SISO\n");
   for (i=1; i<=TrellisLengde; i++)
    {
   for (state=0; state<numberofstates; state++)
    {
    fprintf(fp,"i=%d,state=%d: Alpha=%1.5f Beta=%1.5f \n",
    i,state,Alpha[i][state],
    Beta[i][state]);
    }
    //printf("OutInfo[%d]=%1.5f,OutCode=%1.5f,%1.5f\n",
    // i-1,OutInfo[i-1],OutCode[n*(i-1)],OutCode[n*(i-1)+1]);
//fprintf(fp,"ApriInfo[%d]=%1.5f,ApriCode=%1.5f\n",
    //       i-1,ApriInfo[i-1],ApriCode[n*(i-1)]);
```

```
    }
   fclose(fp);

#endif

   return rot;

}
```