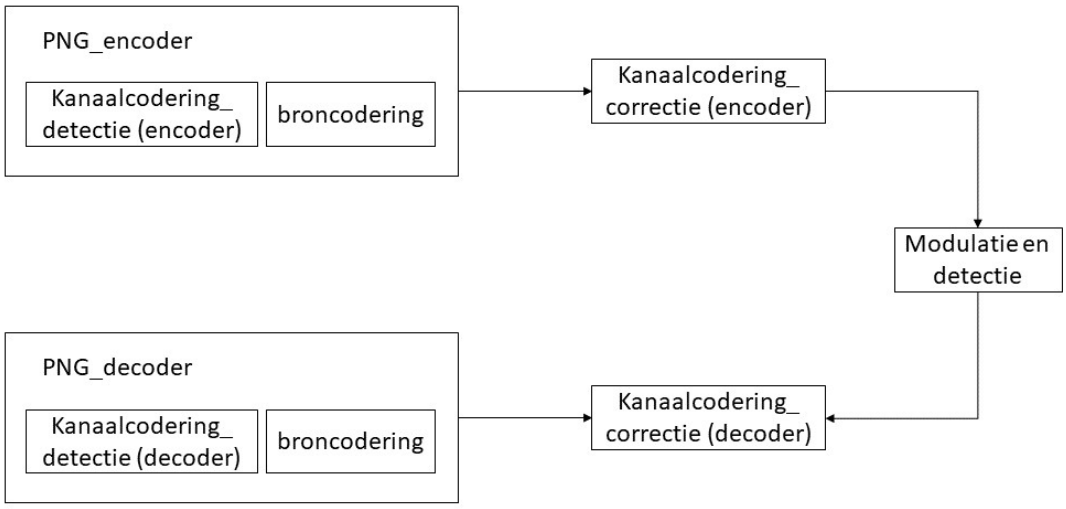
Groepswerk cursus Communicatietheorie: opgave

I. Inleiding

In dit project dat aansluit bij de cursus *Communicatietheorie* versturen we een PNG-afbeelding over een (gemodelleerd) communicatiekanaal. Hierbij zul je de verschillende segmenten van het communicatiesysteem moeten implementeren in de programmeertaal Python, waarna de eigenschappen en prestaties van elk segment worden onderzocht en besproken. Als hulp zijn er op Ufora enkele Python-bestanden beschikbaar waarin sommige functionaliteiten al geïmplementeerd zijn. Het is de bedoeling dat je deze bestanden aanvult waar nodig en deze op het eind van het project mee indient. In deze bestanden staat voor elke functie duidelijk vermeld welke input de functie krijgt en welke output de functie moet geven. Aangezien deze code automatische getest wordt, is het van groot belang om de input/output van reeds geïmplementeerde functies niet te wijzigen en de vereisten voor de te implementeren functies strikt te volgen. In Figuur 1 wordt de globale structuur van het project getoond waarin de verschillende Python-bestanden zijn in aangegeven. Merk dus op dat de PNG encoder en decoder gebruik maken van de code voor broncodering en kanaalcodering (voor foutdetectie).



Figuur 1. Structuur project

Verslag en peer assessment

Bij het project dient ook een verslag gemaakt te worden waarin je onderstaande vragen beantwoordt en jouw resultaten illustreert aan de hand van figuren. Merk op dat een verslag meer is dan zomaar een opsomming van antwoorden op de vragen en zorg er ook voor dat de figuren duidelijk zijn! Maak je verslag echter ook niet onnodig lang. Een elektronische versie (pdf) van het verslag moet samen met alle code ingediend worden **ten laatste op zondag 8 december 2024 voor 23u59** en dit via de opdracht van jullie groep op Ufora in een zipbestand met de naam *groepXX.zip*. Op Ufora zal er ook een peer assessment beschikbaar gesteld worden waarin jullie elkaar moeten beoordelen.

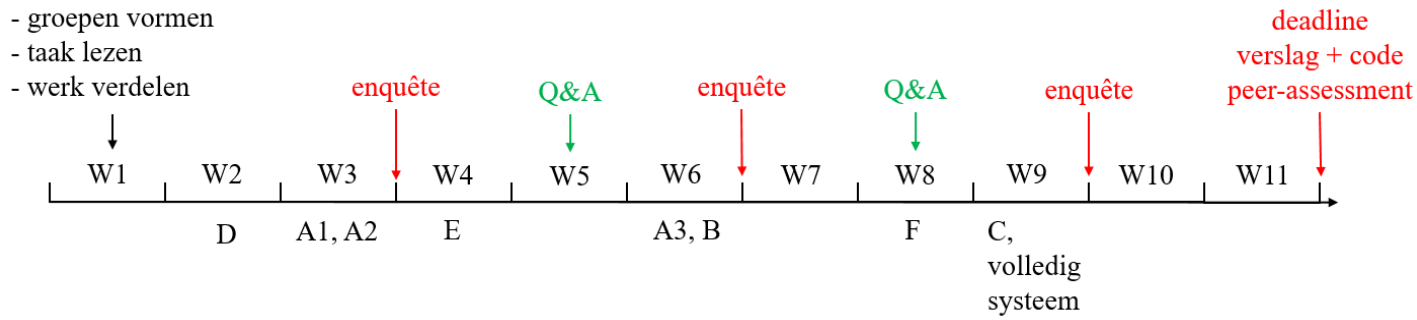
Q&A sessies, feedback en enquêtes

Tijdens het semester zijn er twee Q&A sessies voorzien waarop jullie tussentijdse feedback kunnen krijgen en vragen kunnen stellen. Deze sessies gaan door op 22 oktober van 10u tot 11u15 en 25 november

van 8u30 tot 11u15 in auditorium F, campus Plateau. Gelieve voorbereid naar deze sessies te komen en eventuele vragen al op voorhand door te sturen. Voor vragen in verband met het project tijdens het jaar kunnen jullie steeds mailen naar Tim Willems (tim.willems@ugent.be). De punten van het project samen met beknopte feedback zullen ten laatste 13 januari gepubliceerd worden. Voor extra feedback is het steeds mogelijk om, na de bekendmaking van de punten, een afspraak te maken bij Tim Willems met Prof. Noels (nele.noels@ugent.be) in cc.

Planning

In Figuur 2 wordt een suggestie gedaan wanneer jullie de verschillende onderdelen van dit project kunnen uitvoeren rekening houdende met de timing waarop de bijhorende theorie aan bod komt tijdens de lessen.



Figuur 2. Suggestie planning

II. Opmerkingen over het gebruik van Python

De programmeertaal Python aangevuld met enkele uitbreidingspakketen zoals 'NumPy', 'SciPy' en 'Matplotlib' kan gebruikt worden voor tal van wiskundige toepassingen. In dit project modelleren we er een communicatiesysteem in. 'NumPy' laat je toe om multi-dimensionale rijen en matrices te definiëren en stelt ook veel wiskundige functies ter beschikking. In 'SciPy' zijn verschillende basisfuncties geïmplementeerd, zoals bijvoorbeeld de fft. Tot slot zorgt de package 'Matplotlib' ervoor dat je jouw data op een overzichtelijke manier kunt voorstellen. Online is er uitgebreide documentatie beschikbaar[1] en op Ufora vind je een document waarin de basis is uitgelegd.

- In dit project krijg je een aantal files waarbij je enkele functies moet aanvullen. Het kan handig zijn om de code tijdens het testen in een afzonderlijk bestand te schrijven.
- Python zal sneller werken als je het aantal lussen in je code beperkt. Gebruik dus zoveel mogelijk matrices en vectoren.
- In Tabel 2 vind je enkele nuttige Python commando's die je kan gebruiken in het project. Je mag geen ingebouwde functies gebruiken wanneer er gevraagd wordt om zelf iets te implementeren. Bij twijfel kan je ons altijd contacteren.

III. Portable network graphics (PNG)

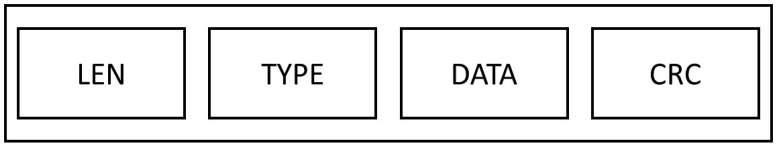
Portable network graphics (PNG) is een binair bestandsformaat dat gebruikt wordt om afbeeldingen digitaal op te slaan als één lange bytestream. Dit bestandsformaat laat toe om zowel afbeeldingen met grijswaarden als RGB kleuren voor te stellen met al dan niet informatie over de transparantie van elke pixel. Verder is het ook mogelijk om maar een beperkt aantal kleuren toe te laten in de afbeelding (door bv. kleurenkwantisatie) wat de bestandsgrootte drastisch kan verlagen. In dit project beperken we ons

[1] https://docs.scipy.org/doc/ en https://matplotlib.org/

Tabel I
ENKELE NUTTIGE PYTHON COMMANDO'S

functienaam	*omschrijving*
a/b en a*b	Matrices a en b puntsgewijs delen of vermenigvuldigen
integrate.quad	Numeriek berekenen van een integraal
np.linspace	Maak een rij met x punten tussen twee grenzen
np.reshape, np.kron, np.matlab.repmat	Functies om matrices te vervormen of uit te breiden
np.sort, np.where	Sorteer of zoek een vector in een matrix
np.random.rand,np.random.randn	Genereren van random data
plt.plot, plt.hist,...	Functies om data te plotten
np.mod, %	Bereken van de modulus
np.convolve	Bereken van de convolutie of product van veeltermen
np.max, np.min	Bereken van minimum en maximum
np.ones, np.zeros, np.eye	Genereer matrix met enen/nullen, eenheidsmatrix
np.sum, np.prod	Berekent som/product van alle vectorelementen

echter tot RGB afbeeldingen (ook wel truecolor PNG's genoemd) zonder transparantie waardoor de kleur van elke pixel kan voorgesteld worden door 3 bytes (een waarde tussen 0 en 255 per kleurkanaal). Om een afbeelding voor te stellen, bestaat de bytestream van het PNG-bestand uit verschillende delen. Voor elk PNG-bestand zijn de eerste 8 bytes (de signature) van de bytestream gelijk. Dit maakt het voor een PNG decoder mogelijk om te verifiëren of het effectief om een PNG-bestand gaat. Vervolgens wordt de overige data van het PNG-bestand ingedeeld in verschillende chunks. De algemene structuur van een chunk wordt weergegeven in Figuur 3, waarbij LEN de lengte van de chunk is, TYPE het type van de chunk is, DATA de eigenlijke data in de chunk is en CRC de CRC32-waarde is van de chunk voor foutdetectie door de decoder.

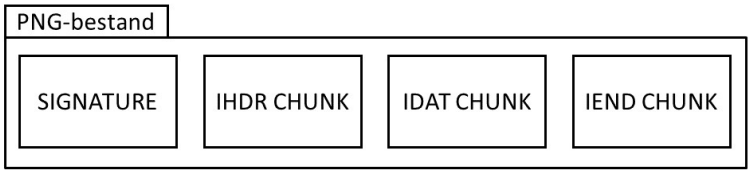


Figuur 3. Algemene structuur van een chunk

In dit project zijn enkel de strikt noodzakelijke chunks met TYPE $\in$ { IHDR, IDAT, IEND } geïmplementeerd. Een chunk van type IHDR is altijd de eerste chunk in een PNG-bestand en bevat data over onder andere de dimensies van de afbeelding, het kleurtype, enz. Een chunk van type IEND daarentegen is altijd de laatste chunk in het bestand en bevat geen data, maar wordt alleen gebruikt om aan te geven dat we aan het einde van de bytestream zijn gekomen. Tussen deze twee chunks zitten dan één of meerdere chunks van type IDAT (in dit project altijd maar één) die de eigenlijke data van de afbeelding bevatten (de RGB-waarden). Voor dit project is de structuur van elk PNG-bestand dus zoals voorgesteld in Figuur 4. Voor meer informatie over de PNG-standaard kan je op Ufora een pdf met deze standaard terugvinden.

De PNG-standaard is voor jullie reeds geïmplementeerd in *PNG.py*. Dit Python-bestand bevat twee klassen die een deel van jullie code zullen oproepen.

- De eerste klasse is de klasse *PNG_encoder* dewelke de functie *encode()* bevat. Deze functie verwacht als input een 3D matrix die overeenkomt met de te encoderen RGB-waarden van de afbeelding, en de bestandsnaam van het resulterende PNG-bestand. Om deze 3D matrix te generen voor een bepaalde



Figuur 4. Algemene structuur van een PNG-bestand

afbeelding kunnen jullie gebruik maken van de *read_RGB_values()* functie in *PNG.py* waarbij de naam van de afbeelding meegegeven moet worden. De *encode()* functie heeft als output de bitstream van het resulterende PNG-bestand.

- De tweede klasse in *PNG.py* is de *PNG_decoder* klasse. Deze klasse bevat de functie *decode()* dewelke gegeven de bitstream van het PNG-bestand, de RGB-waarden van de afbeelding probeert te extraheren en heeft als output een variabele die aangeeft of de decodering succesvol was. De geëxtraheerde RGB-waarden kunnen na het uitvoeren van de *decode()* functie opgevraagd worden via de *get_image()* functie van de *PNG_decoder* klasse (indien decodering succesvol was). Om deze afbeelding te plotten kan de *plot_afbeelding()* functie in *PNG.py* gebruikt worden.

IV. Opgave

Het doel van het beschouwde communicatiesysteem is het foutloos versturen van een PNG afbeelding van de zender naar de ontvanger.

A. Compressie en broncodering (broncodering.py)

Om een afbeelding te versturen, moet deze eerst omgezet worden naar een bitsequentie. De eenvoudigste/meest naïeve manier om dit te realiseren is door elke pixel voor te stellen door 3 bytes waarbij elke byte overeenstemt met de waarde van het rode, groene of blauwe kleurkanaal. Een afbeelding bevat echter vaak ook veel redundantie (bv. vlakken met eenzelfde kleur) waardoor deze methode leidt tot het versturen van veel onnodige data. Een oplossing is om de afbeelding eerst te comprimeren en vervolgens de gecomprimeerde data om te zetten naar een bitsequentie.

De PNG-standaard maakt hiervoor gebruik van een combinatie van Huffmancodering en Lempel-Ziv codering (LZ77). Het LZ77 algoritme benut herhalingen in sequentiële data zoals bv. een tekst of afbeelding om deze data te comprimeren. Om een herhaling te encoderen gebruikt het algoritme twee getallen: één om te specifiëren hoelang de herhaling is en één om te specifiëren hoe ver in het verleden we deze data al gezien hebben. Neem als voorbeeld de tekst 'abcabcdabcdeabc', welke duidelijk een aantal herhalingen bevat. Het LZ77 algoritme zal deze tekst comprimeren tot 'abc(3,3)d(4,4)e(5,3)' waarbij (x,y) aangeeft dat we de tekst van x posities terug kopiëren en dit met een lengte y.

De PNG-standaard comprimeert een afbeelding door het LZ77 algoritme toe te passen op de sequentie van RGB-waarden. Vervolgens gebruikt de standaard verschillende Huffmancodes om deze data nog verder te comprimeren: één Huffmancode om de RGB-waarden en de lengte van de herhaling in de data te encoderen en één Huffmancode om te encoderen hoever in het verleden de herhaling startte. Tot slot wordt er ook nog een derde Huffmancode gebruikt om de lengte van de codewoorden van de twee andere Huffmancodes te encoderen. Deze combinatie van LZ77 en Huffmancodering is ook wel beter bekend als het deflate algoritme. Voor meer informatie over het deflate algoritme zijn er op Ufora ook een aantal bestanden beschikbaar.

De Huffmancodes in de PNG-standaard zijn van een specifiek type, namelijk canonische Huffmancodes. Dit type Huffmancodes kan bijzonder efficiënt worden opgeslagen. Om deze canonische Huffmancode te bepalen, bepalen we eerst een normale Huffmancode zoals beschreven in de cursus. Vervolgens kan deze code omgevormd worden tot een canonische code met behulp van het algoritme op p.7-8 in *rfc1951.pdf*.

Voor dit deel van het communicatiesysteem zullen jullie een aantal functies moeten implementeren in *broncodering.py*.

1) Implementeer eerst de functie *maak_codetabel_Huffman()* die op basis van een set macrosymboolwaarschijnlijkheden met behulp van het Huffmanalgoritme de codewoorden voor elk symbool uit het alfabet berekent (zie ook **Appendix A**). Merk op dat het hierbij ook mogelijk is dat sommige symbolen van het alfabet een waarschijnlijkheid 0 hebben en deze symbolen dus niet voorkomen. Bijgevolg moet voor deze symbolen dus geen codewoord gevonden worden.
2) Implementeer vervolgens *genereer_canonische_Huffman()* die, gegeven de lengte van de codewoorden en het alfabet, de Huffmancode omzet naar een canonische vorm. Volg hiervoor het algoritme beschreven in *rfc1951.pdf*.
3) **Om de volgende vragen te beantwoorden zal je eerst ook de CRC code uit deel IV-B moeten implementeren.** Pas de *encode()* functie van de *PNG_encode* klasse toe op *afbeelding1.pkl* en *afbeelding2.pkl*. Dit zijn twee bestanden die de RGB-waarden bevatten van twee afbeeldingen. Om deze RGB-waarden te bekomen kunnen jullie gebruik maken van de *read_RGB_values()* functie welke als output de RGB-waarden in het *.pkl* bestand teruggeeft. Na het uitvoeren van de *encode()* functie kunnen de gegevens (gemiddelde codewoordlengte, entropie, ...) van de drie Huffmancodes geraadpleegd worden via de *Huffmancode*x variabele (met $x \in [1, 2, 3]$) van de *PNG_encode* klasse. Voor meer informatie zie de *init()* functie van deze klasse. Plot de afbeelding. Wat is het gemiddeld aantal bits per codewoord van de drie verschillende Huffmancodes in de PNG-afbeelding? Zijn de entropiegrenzen voldaan? Wat zou het gemiddeld aantal bits zijn per codewoord indien we een vaste-lengte code zouden gebruiken en vergelijk met de Huffmancodering? Wat kan een voordeel zijn van de vaste-lengte code? Vergelijk de resultaten voor beide afbeeldingen en verklaar.

B. Kanaalcodering: foutdetectie (kanaalcodering_foutdetectie.py)

Bij het versturen van een PNG-bestand over een kanaal kunnen er altijd fouten geïntroduceerd worden in de verzonden bitstream. Hierdoor is het mogelijk dat het PNG-bestand niet meer leesbaar is of erger dat de bijhorende afbeelding fouten bevat. Om deze problemen zoveel mogelijk te voorkomen, wordt er zoals vermeld in Sectie III aan elke chunk van een PNG-bestand een CRC32-waarde toegevoegd. Wanneer we uit deze waarde kunnen afleiden dat er fouten zijn opgetreden kan de ontvanger vervolgens een retransmissie aanvragen.

1) Waarom werd er gekozen om een CRC code te gebruiken voor foutdetectie?
2) CRC-codes werken met binaire veeltermen. Een binaire veelterm van graad $q-1$ of lager $a_1x^{q-1} + a_2x^{q-2} + ... + a_{q-1}x + a_q$ correspondeert met de q-bitsequentie $(a_1a_2...a_{q-1}a_q)$. Met een informatiewoord van lengte k komt dus een informatieveelterm $b(x)$ van graad $k-1$ overeen. De CRC-waarde voor dit informatiewoord is dan gelijk aan de bitsequentie die overeenkomt met de restveelterm $r(x)$ bij deling van $b(x)x^{n-k}$ door een CRC-veelterm $g(x)$ van graad $n-k$, waarbij n de lengte van het resulterende codewoord is. Het resulterende codewoord is gelijk aan de bitsequentie die overeenkomt met de codewoordveelterm $c(x)$ waarbij $c(x) = b(x)x^{n-k} + r(x)$. Bij zuivere foutdetectie aan de ontvanger wordt nagegaan of $c(x)$ een veelvoud is van $g(x)$ (rest nul bij deling!). Een alternatieve manier om een fout te detecteren bestaat er in de eerste k bits van het ontvangen codewoord opnieuw te coderen en de bekomen CRC-waarde te vergelijken met de laatste $n-k$ bits van het ontvangen codewoord. Meer informatie over CRC-codes kunnen jullie terugvinden in Sectie 9.6.4 van de cursus (bekijk ook de lijst met correcties op Ufora). Toon aan dat elke CRC code een lineaire code is (een code is lineair als en slechts als de modulo-2 som van elke twee codewoorden een codewoord is). Implementeer vervolgens ook de functie *determine_CRC_bits()* in *kanaalcodering_foutdetectie.py* welke voor een algemene generatorveelterm $g(x)$ de restveelterm $r(x)$ bepaalt. Voor het bepalen van de CRC-waarde is het best om de (binaire) polynoomdeling zelf te implementeren. **Opm.: De implementatie van de CRC code in de PNG-standaard wijkt licht af van bovenstaande beschrijving, maar de omzetting is reeds voor jullie geïmplementeerd in *PNG.py*.**

3) Test jullie code voor de CRC-5 code met $g(x) = (x+1)(x^4+x+1) = x^5+x^4+x^2+1$ en $k = 5$. Bepaal voor deze code:
   - de lengte n van de codewoorden
   - het gegarandeerd burstfoutdetecterend vermogen
   - de minimale Hammingafstand, foutcorrigerend en foutdetecterend vermogen (zie **Appendix B**)
   - een analytische uitdrukking voor de kans op een niet-detecteerbare fout (zie **Appendix B**) in het geval van een binair symmetrisch kanaal met bitovergangswaarschijnlijkheid *p* en geef een benadering voor kleine *p*. Om de gewichten van deze uitdrukking te bepalen mogen jullie eigen code implementeren. Vermeld in het verslag duidelijk hoe jullie te werk zijn gegaan. Plot de kans op een niet-detecteerbare fout (zowel de exacte waarde als de benadering) in functie van p, waarbij p gaat van 0.05 tot 0.5. Denk hierbij goed na over de schaal van de assen. Voer tevens simulaties uit om de kans op een niet-detecteerbare fout experimenteel te bepalen en voeg dit toe aan de figuur. **Opm.: Genereer hiervoor zelf een binaire sequentie. Gebruik vervolgens *determine_CRC_bits()* om de CRC bits te bepalen en voeg deze toe aan de codewoorden. Voeg hier fouten aan toe met waarschijnlijkheid p en gebruik vervolgens opnieuw *determine_CRC_bits()* om te bepalen of er fouten zijn opgetreden.**

C. Kanaalcodering: foutcorrectie (kanaalcodering_foutcorrectie.py)

Met behulp van de CRC code is het mogelijk om te detecteren of een PNG-bestand fouten bevat en of het bestand (deels) opnieuw verzonden moet worden. Bij een lange bitsequentie is de kans echter zeer klein dat het kanaal geen fouten introduceert waardoor het aantal retransmissies bijzonder hoog zal liggen. Om dit te vermijden, passen we een foutcorrigerende code toe op de bitstream van het PNG-bestand die toelaat om een aantal van de fouten geïntroduceerd door het kanaal te verbeteren. We onderzoeken de prestaties van de (14,8) lineaire blokcode met checkmatrix

$$\boldsymbol{H} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

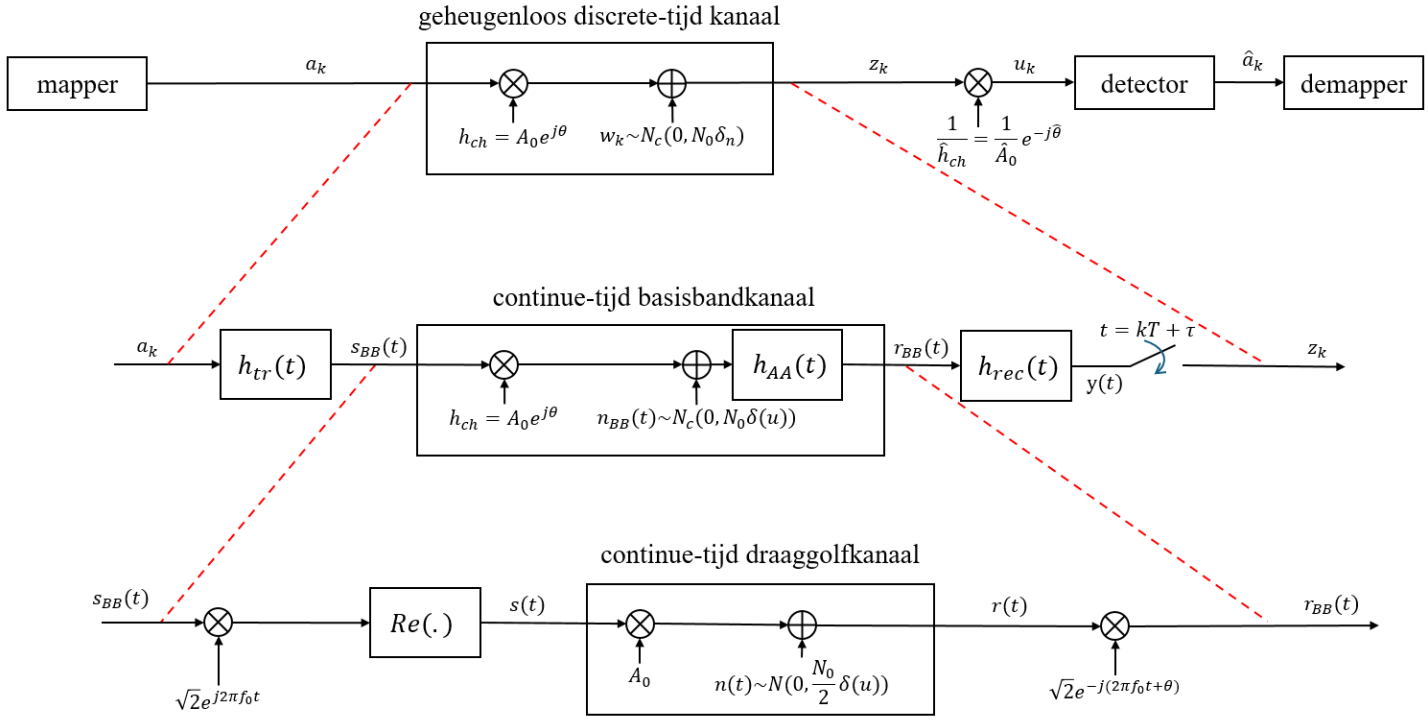
1) Geef voor deze code (zie **Appendix B**):
   - een generatormatrix $\boldsymbol{G}$ en de generator- en checkmatrix $\boldsymbol{G}_{sys}$ en $\boldsymbol{H}_{sys}$ in standaard systematische vorm. Genereren $\boldsymbol{G}$ en $\boldsymbol{G}_{sys}$ dezelfde code? Controleer of $\boldsymbol{G}\boldsymbol{H}^T = \boldsymbol{G}_{sys}\boldsymbol{H}_{sys}^T = 0$.
   - de minimale Hammingafstand
   - het foutdetecterend en foutcorrigerend vermogen
   - de gewichten van de cosetleaders in de syndroomtabel

   Voor het uitvoeren van deze taken mogen jullie eigen code gebruiken. Vermeld in het verslag duidelijk hoe jullie te werk zijn gegaan.
2) Implementeer nu de encoder en de decoder van deze code in respectievelijk *encodeer_lineaire_blokcode()* en *decodeer_lineaire_blokcode()* (zie **Appendix B**) in *kanaalcodering_foutcorrectie.py*. De encodering (zie cursus Sectie 10.3) gebeurt aan de hand van een generatormatrix ($\boldsymbol{c} = \boldsymbol{b}\boldsymbol{G}_{sys}$) en de decodering (zie cursus Sectie 10.5) via het syndroom ($\boldsymbol{s} = \boldsymbol{r}\boldsymbol{H}_{sys}^T$). In de decoder is het de bedoeling dat je zowel volledige decodering als zuivere foutdetectie implementeert. Hierbij stelt de outputvector *bitdec* de gedecodeerde bits na volledige decodering voor en is het $i$de element van vector *bool_fout* gelijk aan 1 als er in het $i$de codewoord een fout is gedetecteerd bij zuivere foutdetectie en gelijk aan 0 als er geen fout is gedetecteerd. In het laatste geval, kan het corresponderende codewoord dan gevonden worden in de sequentie *bitdec*.

3) Bepaal in het geval van volledige decodering een analytische uitdrukking voor de kans op een decodeerfout p_e (zie **Appendix B**). Je mag hier opnieuw code schrijven om manuele berekeningen te vermijden. Geef zowel de exacte formule als een benadering voor $p \ll 1$. Voer tevens simulaties uit om de kans op een decodeerfout experimenteel te bepalen en ga hierbij gelijkaardig te werk als bij de simulatie voor een niet-detecteerbare fout bij de CRC code. Maak een figuur waarin de gesimuleerde resultaten en de exacte en benaderde uitdrukking met elkaar vergeleken worden als functie van p, waarbij p gaat van 0.05 tot 0.5.
4) Geef een analytische uitdrukking (exact + benadering) voor de kans op decodeerfalen p_f, en voor de kans op een niet-detecteerbare fout p_m, in het geval van zuivere foutdetectie met deze lineaire blokcode. Hoe groot zijn beide kansen voor $p = 0.05$?

*D. Mapping en detectie (*moddet.py*)*

In dit deel modelleren we de transmissie over een geheugenloos discrete-tijd kanaal (bovenste blokdiagram in Figuur 5). Om je code te testen en onderstaande vragen te beantwoorden werk je best met een random gegenereerde (ongecodeerde) bitsequentie.



Figuur 5. blokdiagram transmissie kanaal

1) Implementeer de mapper die een bitsequentie omzet in een sequentie van datasymbolen $\{a_k\}$. De mapper dient de volgende genormeerde constellaties te ondersteunen: **BPSK, 4QAM en 4PAM**. Implementeer hiervoor de functie *mapper()* en maak hierbij gebruik van Gray-mapping. Implementeer tevens de demapper (*demapper()*) die de inverse operatie uitvoert.
2) Het kanaal schaalt de verstuurde symboolsequentie $\{a_k\}$ met een complexe (tijdsonafhankelijke) factor $h_{\text{ch}} = A_0 \cdot e^{j\theta}$, met amplitude A_0 en fase θ, en voegt witte Gaussiaanse ruis toe. Stel in het volledige project $A_0 = 1$ en $\theta = \frac{\pi}{6}$. Voor de samples z_k aan de uitgang van het kanaal geldt:

$$z_k = h_{\text{ch}} \cdot a_k + w_k \tag{1}$$

met $\{w_k\}$ onafhankelijk en gelijk verdeelde complexe, circulair symmetrische, Gaussiaanse ruissamples met gemiddelde nul en variantie N_0. Implementeer de functie *discreet_geheugenloos_kanaal()* die de sequentie van datasymbolen $\{a_k\}$ omzet in de sequentie van ontvangen samples $\{z_k\}$ uit (1).

3) Implementeer nu de functie *maak_decisie_variable()* die de samples z_k schaalt met een factor $\frac{1}{\hat{A}_0}$ en de fase compenseert met een rotatie over een hoek van $\hat{\theta}$, wat resulteert in de decisievariabele $u_k = \frac{z_k}{\hat{A}_0} e^{-j\hat{\theta}}$.
4) Veronderstel voorlopig dat $\hat{A}_0 = A_0$ en $\hat{\theta} = \theta$. Maak een scatterdiagram voor de 4QAM constellatie van het signaal $\{u_k\}$ zonder ruis en onderzoek de invloed van E_b/N_0 met

$$E_s = A_0^2 = m\frac{k}{n}E_b. \tag{2}$$

In deze vergelijking is m het aantal bits dat elk symbool van de constellatie voorstelt, en zijn n en k respectievelijk het aantal codebits en informatiebits. Aangezien we in dit deel een ongecodeerde bitsequentie gebruiken is $k = n$. Voor de scatterplot kan je gebruik maken van een ingebouwde functie *scatter()* van matplotlib.

5) Op basis van de decisievariabele u_k zal de ontvanger het symbool a_k schatten. Dit doet hij door het dichtstbijzijnde constellatiepunt $\hat{a}_k$ te bepalen. Implementeer dit in *decisie()* en verifieer dat voor $N_0 = 0$ de juiste constellatiepunten worden teruggevonden.
6) Bepaal nu de BER voor alle drie de constellaties (BPSK, 4QAM en 4PAM) aan de hand van simulaties voor verschillende waarden van E_b/N_0 en maak een figuur waarbij je de BER logaritmisch uitzet als functie van E_b/N_0 in dB. Genereer voor elk punt in deze curve zo veel random realisaties van u_k als nodig is om minstens 100 keer een foutief constellatiepunt te detecteren (m.a.w. $\hat{a}_k \neq a_k$). Beperk je figuur tot het interval van E_b/N_0 waarvoor $10^{-1} > BER > 10^{-4}$ (ongeveer). Vergelijk het bekomen resultaat met de curves uit de cursus en waar mogelijk met de analytische uitdrukking.
7) In de praktijk kent de ontvanger de parameters (A_0, θ) niet en zullen deze geschat moeten worden. In wat volgt bestuderen we het effect van schattingsfouten. Om te beginnen bestuderen we de invloed van een amplitudeschattingsfout bij zowel de BPSK als de 4PAM constellatie. Maak hiervoor een scatterdiagram in afwezigheid van ruis voor beide constellaties bij $\epsilon = 0.1$ met $\epsilon = \frac{\hat{A}_0 - A_0}{A_0}$. Wat bemerk je? Maak opnieuw voor beide constellaties een figuur met daarop de BER-curves voor volgende waarden van $\epsilon = 0, 0.1, 0.2$. Verklaar het resultaat.
8) Vervolgens gaan we na wat het effect is van een faseschattingsfout als we de 4QAM constellatie gebruiken. Maak ook hier een scatterdiagram in afwezigheid van ruis bij $\phi = \frac{\pi}{16}$ met $\phi = \hat{\theta} - \theta$. Wat bemerk je? Maak vervolgens opnieuw een figuur met de BER-curves voor de volgende waarden van $\phi = 0, \frac{\pi}{16}, \frac{\pi}{8}, \frac{\pi}{4}$. Verklaar het resultaat.

*E. Basisbandmodulatie (*moddet.py*)*

Vervolgens modelleren we de transmissie over een niet-dispersief complex continue-tijd basisbandkanaal. De bovenste twee blokdiagrammen in Figuur 5 tonen het verband tussen het hier beschouwde basisbandsysteem, en het equivalente discrete-tijd systeem uit deel IV-D.

In tegenstelling tot eerder werken we met continue-tijd signalen. In een computersimulatie moeten continue-tijd signalen steeds worden voorgesteld door middel van een sequentie van samples. Beschouw het continue-tijd signaal $x(t)$ en de bijhorende sample-sequentie $x(lT_s) = x(l\frac{T}{N_s})$, waarbij T_s het sample-interval is, N_s het aantal samples per symboolinterval voorstelt en $l \in \mathbb{Z}$. In het algemeen is het niet mogelijk het oorsponkelijke continue-tijd signaal $x(t)$ perfect te reconstrueren op basis van de samples $x(lT_s)$ door middel van interpolatie. De reden hiervoor is dat de samples $x(lT_s)$ niet alle informatie over $x(t)$ bevatten. De interpolatie wordt wel nauwkeuriger naarmate de bemonsteringsfrequentie $\frac{1}{T_s}$ toeneemt. De bemonsteringstheorie van Nyquist/Shannon stelt echter dat een continu-tijdsignaal $x(t)$ met eindige bandbreedte B (d.w.z. $X(f) = 0$ voor $|f| > B$) perfect gereconstrueerd kan worden uit de samples $x(lT_s)$, op voorwaarde dat de bemonsteringssnelheid voldoet aan $\frac{1}{T_s} \leq 2B$. Onder deze voorwaarde kan niet alleen

perfecte interpolatie worden bereikt, maar kunnen de signalen ook verwerkt, opgeslagen en gesimuleerd worden op basis van hun samples.

Kies voor de rest van het project $N_s = 6$. We zullen later verifiëren dat hiermee inderdaad voldaan is aan het bemonsteringstheorema van Nyquist/Shannon.

1) Om transmissie over een continue-tijd basisbandkanaal mogelijk te maken wordt de symboolsequentie eerst $\{a_k\}$ omgezet in een continue-tijd basisbandsignaal:

$$s_{BB}(t) = \sum_{k=0}^{K-1} a_k h_{tr}(t - kT), \tag{3}$$

waarbij a_k het k^{de} data symbool voorstelt, $h_{tr}(t)$ de zenderpuls en T het symboolinterval. Voor $h_{tr}(t)$ maken we in dit project gebruik van een afgeknotte versie van de square-root raised cosine puls waarbij het symboolinterval $T = 1\mu s$ en de roll-off factor α tussen 0 en 1 ligt. De (enkelzijdige) bandbreedte van zo een puls $p(t)$ is $\frac{1+\alpha}{2T}$. Deze pulsen voldoen ook aan $\int_{-\infty}^{\infty} p\,(t)\,p\,(t - kT)\,\mathrm{d}t = \delta_k$. Square-root raised cosine pulsen hebben een oneindige duur, maar aangezien deze pulsen uitsterven in de tijd, kunnen we $h_{tr}(t)$ afknotten tot $2L_F$ symboolperioden zonder dat dit grote gevolgen heeft op de prestatie van het systeem, op voorwaarde dat L_F voldoende groot is. De gebruikte puls $h_{tr}(t)$ is dan gelijk aan $p(t)$ voor $|t| \leq L_F T$ en gelijk aan 0 voor $|t| > L_F T$. Stel in het volledige project L_F gelijk aan 10, wat resulteert in $20N_s + 1$ monsterwaarden.
Het effect van het afknotten van de puls is het best te observeren in het frequentiedomein. Maak daarom voor $\alpha = 0.05$, $\alpha = 0.5$ en $\alpha = 0.95$ een figuur van $|H_{tr}(f)|$ als functie van fT, met $H_{tr}(f)$ de FT van de afgeknotte square-root raised cosine puls $h_{tr}(t)$. Gebruik voor $|H_{tr}(f)|$ een logaritmische schaal. Vergelijk met de theoretische vorm van $|P(f)|$, met $P(f)$ de FT van de niet-afgeknotte square-root raised cosine puls $p(t)$ (zie cursus). Wat is het effect van het afknotten? Hoe komt dit? Hoe kan je verklaren dat de impact groter is voor kleinere waarden van α?
2) In de veronderstelling dat het effect van het afknotten van de puls verwaarloosbaar is, waaraan moet N_s voldoen opdat het bemonsteringstheorema van Nyquist/Shannon is voldaan voor $s_{BB}(t)$? Controleer dat $N_s = 6$ inderdaad voldoet aan deze voorwaarde.
3) Implementeer de basisbandmodulatie (3) in ***moduleerBB()***. Om $h_{tr}(t)$ te genereren kan je gebruik maken van de ***pulse()*** functie in ***moddet.py*** die reeds voor jullie is geprogrammeerd.
4) Het niet-dispersief basisbandkanaal schaalt het verstuurde signaal $s_{BB}(t)$ met een complexe (tijdsonafhankelijke) factor $h_{\mathrm{ch}} = A_0 \cdot e^{j\theta}$, met amplitude A_0 en fase θ, en voegt witte Gaussiaanse ruis toe. Voor het signaal $r_{BB}(t)$ aan de uitgang van het kanaal geldt:

$$z_k = h_{ch} \cdot s_{BB}(t) + n_{BB}(t) \tag{4}$$

met $n_{BB}(t)$ complexe, circulair symmetrische, witte Gaussiaanse ruis met gemiddelde nul en spectrale dichtheid N_0. Als gevolg van de witte ruis heeft het ontvangen signaal $r_{BB}(t)$ een oneindige bandbreedte. Leg uit waarom we in onze computersimulatie toch kunnen werken met de samples $r_{BB}\Big(l\frac{T}{N_s}\Big)$.
5) De samples van het ontvangen signaal worden gegeven door

$$r_{BB}\Big(l\frac{T}{N_s}\Big) = h_{\mathrm{ch}} s_{BB}\Big(l\frac{T}{N_s}\Big) + n_{BB,l}, \tag{5}$$

met $n_{BB,l} = n_{BB,AA}(l\frac{T}{N_s})$ en $n_{BB,AA}(t)$ het antwoord van een ideaal laag-doorlaat filter $H_{AA}(f)$ op $n_{BB}(t)$. Hierbij wordt $H_{AA}(f)$ gedefinieerd als

$$H_{AA}(f) = \begin{cases} 1 & |f| < \frac{1}{2T_s} \\ 0 & anders \end{cases}. \tag{6}$$

De ruissamples $n_{BB,l}$ zijn onafhankelijk en gelijk verdeelde, complexe, circulair symmetrische, Gaussiaanse toevalsgrootheden met gemiddelde nul en variantie $\frac{N_0 N_s}{T}$. Toon aan dat $\mathbb{E}[n_{BB,l_1} n^*_{BB,l_2}] = \frac{N_0 N_s}{T}\delta_{l_1-l_2}$.

6) Implementeer de functie *basisband_kanaal()* die de samples van het verstuurde signaal $s_{BB}\left(l\frac{T}{N_s}\right)$ omzet in de samples van het ontvangen signaal $r_{BB}\left(l\frac{T}{N_s}\right)$.
7) Het signaal aan de uitgang van het continue-tijd basisbandsignaal wordt door de ontvanger opnieuw omgezet in een sequentie van complexe waarden $\{z_k\}$. Daartoe wordt het ontvangen signaal $r_{BB}(t)$ eerst aangelegd aan een ontvangerfilter $h_{rec}(t)$, gematched aan de zenderpuls. In het geval van een square-root raised cosine zenderpuls, geldt er dat dit ontvangerfilter identiek is aan de zenderpuls, i.e., $h_{rec}(t) = h_{tr}(t)$. De uitgang van het ontvangerfilter wordt gegeven door

$$y(t) = \int h_{rec}(u) r_{BB}(t-u) du. \tag{7}$$

Deze operatie kan gemakkelijk geïmplementeerd worden in Python met behulp van de functie *np.convolve()*:

$$\boldsymbol{y} = T_s \cdot np.convolve(\boldsymbol{r}_{BB}, \boldsymbol{h}_{rec}) \tag{8}$$

waarbij $\boldsymbol{y}$, $\boldsymbol{r}_{BB}$ en $\boldsymbol{h}_{rec}$ respectievelijk de samples van $y(t)$, $r_{BB}(t)$ en $h_{rec}(t)$ bevatten. Implementeer de functie *demoduleerBB()*, die $r_{BB}(t)$ omzet in $y(t)$.
8) Verwacht je dat $y(t)$ een reëel of een complex signaal is in het geval van de BPSK constellatie? Leg uit.
9) Maak een oogdiagram over twee symboolperioden voor $\alpha = 0.05$, $\alpha = 0.5$ en $\alpha = 0.95$ op basis van het reële deel van het signaal $y(t)$, zonder ruis en voor de BPSK constellatie. Hiertoe genereer je een voldoende lang signaal $y(t)$. Vertrek van een random bitsequentie en gebruik de functies *mapper()*, *moduleerBB()*, *basisband_kanaal()* en *demoduleerBB()* om (de samples van) $y(t)$ te berekenen. Laat het overgangsverschijnsel aan het begin en einde van $y(t)$ weg, en plot vervolgens de waarde van $y(t)$ in opeenvolgende intervallen van twee symboolperioden boven op elkaar. Benoem je assen en duidt de schaal correct aan. Om het oogdiagram te plotten mag je GEEN gebruik maken van functies/packages die dit voor jou doen. Is de verticale oogopening gelijk voor alle α? Is dit in overeenkomst met de theorie? Zo niet, waarom niet (verklaar)?
10) Maak nu ook een oogdiagram voor de 4PAM constellatie met $\alpha = 0.5$. Wat is het verschil met BPSK? Leg uit en verklaar.
11) De uitgang van het ontvangerfilter $y(t)$ wordt gesampled aan een snelheid van 1 sample per symbool. Dit resulteert in de sequentie $\{z_k\}$ met $z_k = y(kT + \tau)$. Bespreek het verband tussen de intersymboolinterferentie en de keuze van τ (gebruik hiervoor het oogdiagram) en bepaal het optimale sample tijdstip τ_{opt}.
12) Stel voor de rest van dit project $\tau = \tau_{opt}$. Implementeer de functie *decimatie()* die $\{z_k = y(kT+\tau_{opt})\}$ bepaalt op basis van $y(t)$.
13) Controleer je code door de scatterdiagrammen uit deel D opnieuw te generen waarbij je *discreet_geheugenloos_kanaal()* vervangt door de cascade *mapper()*, *moduleerBB()*, *basisband_kanaal()*, *demoduleerBB()* en *decimatie()*. Aangezien beide systemen equivalent zijn, zou je nagenoeg hetzelfde resultaat moeten bekomen.

*F. Draaggolfmodulatie (*moddet.py*)*

Tot slot modelleren we de transmissie over een niet-dispersief reëel continue-tijd draaggolfkanaal (fysisch kanaal). De blokdiagrammen in Figuur 5 tonen het verband tussen de drie equivalente systemen uit delen IV-D, IV-E en IV-F.

1) Het complexe basisbandsignaal $s_{BB}(t)$ uit (3) wordt op een draaggolf geplaatst met frequentie $f_0 = 2\,\mathrm{MHz}$. Dit gebeurt door middel van de volgende operatie

$$s(t) = \sqrt{2}\Re[s_{BB}(t)e^{j2\pi f_0 t}]. \tag{9}$$

Druk de Fouriertransformatie (FT) van $s(t)$ uit als functie van de FT van $s_{BB}(t)$. Waaraan moet N_s voldoen opdat aan het bemonsteringstheorema van Nyquist/Shannon is voldaan voor $s(t)$? Voldoet $N_s = 6$ hieraan?

2) Implementeer (9) in *moduleer()*.
3) Het banddoorlaatsignaal $s(t)$ wordt verstuurd over een kanaal (schalingsfactor A_0 en reële additieve witte Gaussiaanse ruis $n(t)$ met spectrale dichtheid $\frac{N_0}{2}$). Net als in deel IV-E, kunnen we in onze computersimulatie, werken met de samples $r\left(l\frac{T}{N_s}\right)$ van het ontvangen signaal. We hebben:

$$r\left(l\frac{T}{N_s}\right) = A_0 \cdot s\left(l\frac{T}{N_s}\right) + n_l, \tag{10}$$

met $n_l = w_{AA}(l\frac{T}{N_s})$, $n_{AA}(t)$ het antwoord van een ideaal laag-doorlaat filter $H_{AA}(f)$ op $n(t)$ en $H_{AA}(f)$ gedefinieerd als in (6). Er kan worden aangetoond dat de ruissamples n_l onafhankelijk reëel Gaussiaans verdeeld zijn met gemiddelde 0 en variantie $\frac{N_0 N_s}{2T}$. Implementeer de functie *kanaal()*, die $r\left(l\frac{T}{N_s}\right)$ berekent uit $s\left(l\frac{T}{N_s}\right)$.

4) Het ontvangen signaal $r(t)$ wordt gedemoduleerd met de draaggolffrequentie $f_0 = 2\,\mathrm{MHz}$. In ons model veronderstellen we dat de demodulator niet perfect is en een onbekende rotatie θ introduceert. In dat geval is de (complexwaardige) uitgang van de demodulator gelijk aan

$$r_l' = \sqrt{2}r_l e^{-j(2\pi f_0 l T_s + \theta)}. \tag{11}$$

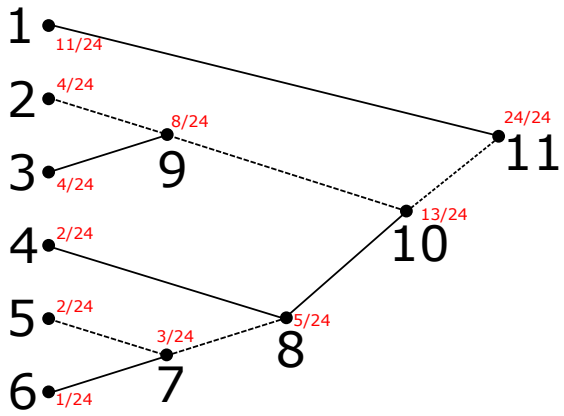
Implementeer (11) in de functie *demoduleer()*.

5) Controleer je code door de scatterdiagrammen uit deel D een derde keer te generen waarbij je *discreet_geheugenloos_kanaal()* deze keer vervangt door de cascade *mapper()*, *moduleerBB()*, *moduleer()*, *kanaal()*, *demoduleer()*, *demoduleerBB()* en *decimatie()*. Opnieuw zou je nagenoeg hetzelfde resultaat moeten bekomen.

V. VOLLEDIG SYSTEEM

In Secties IV-A tot IV-F werden de verschillende bouwblokken van het digitaal communicatiesysteem afzonderlijk bestudeerd. In deze sectie zullen we nu deze verschillende blokken combineren en hun gezamenlijke prestatie onderzoeken.

1) Vertrek opnieuw van een random gegenereerde (ongecodeerde) bitsequentie. In tegenstelling tot eerder passen we nu eerst de (14,8) lineaire blokcode uit Sectie IV-B toe op deze bitsequentie. Vervolgens moduleren/demoduleren we deze gecodeerde bitsequentie zoals besproken in Secties IV-D, IV-E en IV-F. Zender: *mapper()*, *moduleerBB()*, *moduleer()*. Ontvanger: *demoduleer()*, *demoduleerBB()*, *decimatie()*, *maak_decisie_variabele()*, *decisie()*, *demapper()*. Kanaal: *kanaal()*. Je mag veronderstellen dat de ontvanger de correcte parameters $(h_{\mathrm{ch}}, \theta)$ kent. De bitsequentie aan de uitgang van de demapper decoderen we vervolgens met de lineaire blokcode.
   - Bepaal de BER voor alle drie de constellaties (BPSK, 4QAM en 4PAM) aan de hand van simulaties voor verschillende waarden van E_b/N_0 en maak een figuur. Beschouw enkel het interval van E_b/N_0 waarvoor $10^{-1} > BER > 10^{-4}$ (ongeveer). Vergelijk het bekomen resultaat met de curves voor het ongecodeerde geval.
   - Wat zijn de bandbreedte-efficiëntie en vermogensefficiëntie van dit systeem in het geval van een BPSK constellatie? De bandbreedte-efficiëntie in geval van draaggolfmodulatie wordt gegeven door $r_c log_2(M)$, waarbij $r_c = \frac{k}{n}$ en $M = 2^m$ het aantal constellatiepunten voorstelt. De vermogensefficiëntie van een communicatiesysteem wordt gegeven door de minimale waarde



Figuur 6. Voorbeeld codeboom met genummerde knopen.

van $\frac{E_b}{N_0}$ waarvoor de BER van het communicatiesysteem een bepaalde referentiewaarde BER_{ref} bereikt. Voor de vermogensefficiëntie is de referentie in dit geval $BER_{ref} = 0.01$. Vergelijk deze efficiënties met het ongecodeerde geval.

- Wat is de codeerwinst $G_c = \frac{(E_b/N_0)_{unc}}{(E_b/N_0)_{cod}}$? In deze formule stellen $(E_b/N_0)_{unc}$ en $(E_b/N_0)_{cod}$ de vermogensefficiëntie voor in het geval van respectievelijk ongecodeerde en gecodeerde transmissie.

2) We willen nu *afbeelding1.pkl* over het kanaal verzenden door gebruik te maken van de (14,8) blokcode en de BPSK constellatie. Gebruik, aan de zenderzijde, de *encode()* functie van *PNG_encoder* (in *PNG.py*) om de bitstream van de afbeelding te genereren. Gebruik, aan de ontvangerzijde, de *decode()* functie van *PNG_decoder* (in *PNG.py*) om uit de gedetecteerde bitstream de afbeelding te reconstrueren. De *decode()* functie geeft een output terug die aangeeft of de decodering succesvol was. In het geval de decodering niet succesvol was vraagt de ontvanger een retransmissie aan. Bepaal aan de hand van simulaties de kans dat de afbeelding niet correct ontvangen wordt na maximaal 10 retransmissies voor verschillende waarden van E_b/N_0 en maak een figuur.

Appendix A: Huffmancodering en-decodering

Huffmancodering en -decodering maakt gebruik van een *binaire codeboom* (zie Figuur 6). Het opstellen van deze codeboom gaat als volgt:

- Beschouw een lijst van de macrosymboolwaarschijnlijkheden.
- Schrijf de macrosymbolen onder elkaar. Schrijf de bijhorende macrosymboolwaarschijnlijkheden er naast.
- Associeer aan elk macrosymbool een *knoop*. Dit worden de *bladeren* van de boom.
- Herhaal tot alle bladeren verbonden zijn met de boom:
  - Selecteer de twee knopen met de laagste waarschijnlijkheden in de lijst met waarschijnlijkheden.
  - Verbind deze twee knopen met een nieuwe knoop. [Teken twee *takken* en één nieuwe knoop, rechts van de bestaande knopen.]
  - Associeer aan de nieuwe knoop de som van de waarschijnlijkheden van de knopen waarmee je hem hebt verbonden.
  - Verwijder in de lijst met waarschijnlijkheden de twee laagste waarschijnlijkheden en voeg de som van deze twee waarschijnlijkheden toe aan de lijst. [De lijst bevat nu 1 waarschijnlijkheid minder.]
- Als alles goed gaat, eindig je met een knoop met waarschijnlijkheid 1. Dit is de *wortel* van de boom.
- Associeer aan elke tak een bitwaarde. Twee takken die verbonden zijn met eenzelfde knoop hebben een verschillende bitwaarde.

- Coderen: Vorm de informatiebitsequentie die hoort bij een bepaald macrosymbool door het pad te volgen van de wortel van de boom naar het bijhorende blad en de corresponderende bitsequentie op te schrijven.
- Decoderen: Vind het eerstvolgende macrosymbool terug door in de boom het pad te volgen vanaf de wortel dat overeenkomt met de ontvangen bitsequentie. Lees het macrosymbool af zodra je een blad hebt bereikt. Vind een volgend macrosymbool door opnieuw aan de wortel te beginnen, enz...

APPENDIX B: SYNDROOMDECODERING

Om een checkmatrix $\boldsymbol{H}$ van een lineaire blokcode te bepalen op basis van een generatormatrix $\boldsymbol{G}$ ga je als volgt te werk:

- Voer rijbewerkingen en/of kolompermutaties uit op $\boldsymbol{G}$ om een matrix te bekomen van de vorm $[\boldsymbol{IP}]$, met $\boldsymbol{I}$ de eenheidsmatrix.
- Vorm de matrix $\boldsymbol{X} = \left[\boldsymbol{P}^T \boldsymbol{I}\right]$.
- Pas op $\boldsymbol{X}$ de inverse kolompermutaties toe. De matrix die je bekomt is een checkmatrix $\boldsymbol{H}$ van de code.

Om $\boldsymbol{G}$ te bepalen op basis van $\boldsymbol{H}$, ga je op een gelijkaardige manier te werk.
Een belangrijke parameter van een lineaire blockcode is de *minimale Hammingafstand* $d_{\min}$. Dit is het minimaal aantal bitfouten dat nodig is om één geldig codewoord om te zetten in een ander geldig codewoord. Voor lineaire blokcodes is d_{min} gelijk aan het minimum aantal 1-bits in een van nul verschillend codewoord. Het *gegarandeerd foutcorrigerend vermogen* van een code is t (bitfouten per codewoord), met

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor. \tag{12}$$

Het *gegarandeerd foutdetecterend vermogen* van een code is $\mathcal{L}$ (bitsfouten per codewoord), met

$$\mathcal{L} = d_{\min} - 1. \tag{13}$$

Foutdetectie en foutcorrectie van lineaire blokcodes kan efficiënt worden geimplementeerd met behulp van een *syndroomtabel*. Het *syndroom* van een ontvangen woord $\boldsymbol{\beta}$ is een bitsequentie $\boldsymbol{s}$ van $n-k$ bits. Het syndroom van een n-bit sequentie $\boldsymbol{\beta}$ wordt berekend als

$$\boldsymbol{s} = \boldsymbol{\beta}\boldsymbol{H}^T, \tag{14}$$

met $\boldsymbol{H}$ een *checkmatrix* van de code (met als eigenschap dat $\boldsymbol{G}\boldsymbol{H}^T = \boldsymbol{0}$). Bijgevolg is het syndroom $\boldsymbol{s} = \boldsymbol{\beta}\boldsymbol{H}^T$ een nulwoord als en slechts als $\boldsymbol{\beta}$ een codewoord is.

Om een syndroomtabel op te stellen, ga je als volgt te werk:

- Overloop de mogelijke ontvangen woorden in volgorde van toenemend Hamminggewicht (aantal 1-en in de sequentie).
- Bepaal de bijhorende syndromen (14).
- Staat dit syndroom reeds in de tabel?
  - JA: ga verder met het volgende codewoord.
  - NEE: Schrijf het syndroom in de eerste kolom van de tabel. Schrijf het bijhorende woord op dezelfde lijn in de tweede kolom van de tabel.

De woorden in de tweede kolom van de syndroomtabel fungeren als foutpatronen $\boldsymbol{e}$ die gebruikt worden in het geval van foutcorrectie (syndroomdecodering). De ontvanger berekent eerst het syndroom van het ontvangen woord. Vervolgens telt de ontvanger het foutpatroon dat hoort bij dit syndroom (zie syndroomtabel) binair op bij het ontvangen woord. Het resultaat is het gecorrigeerde codewoord.

In het geval van transmissie over een BSC met bitovergangswaarschijnlijkheid p, kan de syndroomtabel ook gebruikt worden om snel de kans op een decodeerfout in geval van volledige decodering $p_{e,COD}$, de

kans op een niet-detecteerbare fout p_m en de kans op decodeerfalen in het geval van zuiver foutdetectie $p_{f,DET}$ te bepalen.

$$\begin{aligned} p_{e,COD} &= 1 - \sum_{\boldsymbol{e}\in\mathcal{T}} p^{w(\boldsymbol{e})} \left(1-p\right)^{n-w(\boldsymbol{e})} \\ &\approx N_{e,t+1} p^{t+1} \end{aligned}$$

$$\begin{aligned} p_m &= \sum_{\boldsymbol{c}\in\mathcal{C}_\mathbf{0}} p^{w(\boldsymbol{c})} \left(1-p\right)^{n-w(\boldsymbol{c})} \\ &\approx N_{c,\mathcal{L}+1} p^{\mathcal{L}+1} \end{aligned}$$

$$\begin{aligned} p_{f,DET} &= \sum_{\boldsymbol{e}\in\mathcal{T}_\mathbf{0}} \sum_{\boldsymbol{c}\in\mathcal{C}} p^{w(\boldsymbol{e}+\boldsymbol{c})} \left(1-p\right)^{n-w(\boldsymbol{e}+\boldsymbol{c})} \\ &\approx Np \end{aligned}$$

met $\mathcal{C}_\mathbf{0}$ de verzameling van geldige codewoorden behalve het nulwoord, $\mathcal{T}$ de verzameling van de foutvectoren in de syndroomtabel, $\mathcal{T}_\mathbf{0}$ de verzameling $\mathcal{T}$ zonder het nulwoord, n de bloklengte van de codewoorden, $w\left(\cdot\right)$ een functie die het Hamminggewicht van een bitsequentie bepaalt. De benaderingen zijn geldig voor kleine waarden van p, waarbij $t > 0$ het gegarandeerd foutcorrigerend en $\mathcal{L} > 0$ het gegarandeerd foutdetecterend vermogen van de code voorstelt, en met

- $N_{e,t+1}$: het aantal foutpatronen van gewicht $t+1$ die niet in de syndroomtabel staan.
- $N_{c,\mathcal{L}+1}$: het aantal codewoorden van gewicht $\mathcal{L}+1$.