

Návrh a implementácia testovacieho prostredia pre vizualizáciu a hodnotenie SDR komunikácie v reálnom čase na platforme GNU Radio

Bakalársky projekt 1

Obsah

Zoznam použitých skratiek a označení.....	3
1 GNU Radio.....	4
1.1 Študovanie projektov iných.....	4
1.2 Môj projekt.....	6
1.2.1 Výsledky projektu.....	7
2 Out Of Tree blok.....	10
2.1 Vstupy / Výstupy s rôznymi dátovými typmi.....	10
2.2 Zmena v header súbore.....	11
2.3 Inicializácia mimo funkcie "work".....	11
2.4 Vektor alebo zoznam.....	12
2.5 Výpis hodnôt do konzole.....	12
3 AWGN blok.....	13
3.1 Náhodná vzorka šumu.....	13
3.2 Normálový generátor.....	14
3.3 Hlavná časť bloku (work).....	15
3.3.1 Lineárne rozdelenie.....	15
3.3.2 Logika funkcie.....	15
4 BER blok.....	17
4.1 Inicializácia premenných.....	17
4.2 Logika bloku.....	18
4.2.1 Porovnanie vstupných signálov.....	18
4.2.2 Zistenie počet chýb.....	18
4.2.3 Hlavná časť bloku (work).....	19
5 Plán na letný semester.....	21
6 Literatúra.....	22
Odkaz na GitHub repository.....	22

Zoznam použitých skratiek a označení

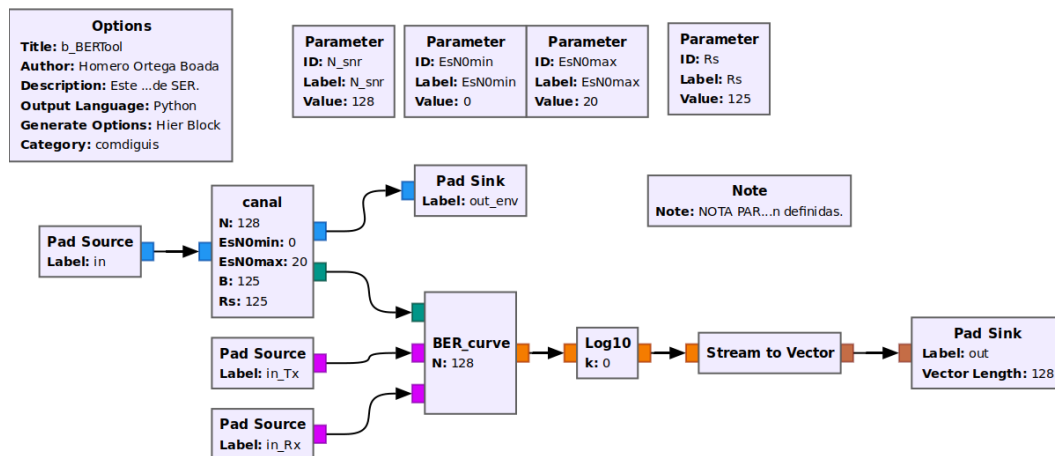
<i>GR</i>	GNU Radio
<i>OOT</i>	Out Of Tree
<i>I/O</i>	Vstup/Výstup
<i>QT GUI</i>	QT Graphical User Interface
<i>OFDM</i>	Ortogonal Frequency Division Multiplexing
<i>SNR</i>	Odstup signál/šum
E_b/N_0	Energia bitu na výkon šumu
E_s/N_0	Energia symbolu na výkon šumu
dB	decibel
EDB	E_b/N_0 v dB
<i>PSK</i>	Phase-Shift Keying
<i>BPSK</i>	Binary Phase-Shift Keying
<i>QPSK</i>	Quadrature Phase-Shift Keying
<i>DQPSK</i>	Differential Quadrature Phase-Shift Keying
<i>MPSK</i>	"M" Phase Shift Keying (M-stavov)
<i>QAM</i>	Quadrature Amplitude Modulation
<i>MQAM</i>	"M" Quadrature Amplitude Modulation (M-stavov)
<i>AWGN</i>	Additive White Gaussian Noise
<i>BER</i>	Bit Error Rate
<i>SER</i>	Symbol Error Rate
<i>FER</i>	Frame Error Rate
<i>YAML</i>	Yet Another Markup Language
<i>GCC</i>	GNU C Compiler
<i>RAM</i>	Random Access Memory

1 GNU Radio

Moje bloky boli vyvinuté pre GNU Radio 3.10 avšak dajú sa skompilovať aj pre 3.9. Testoval som môj projekt na Linux Mint 22, Ubuntu 24 a Ubuntu 20 a všade to bez problémov fungovalo. Cez leto sa mi taktiež podarilo rozbehať GR 3.7 a GR 3.8 na Ubuntu 20 cez kompiláciu zdrojového kodu [16].

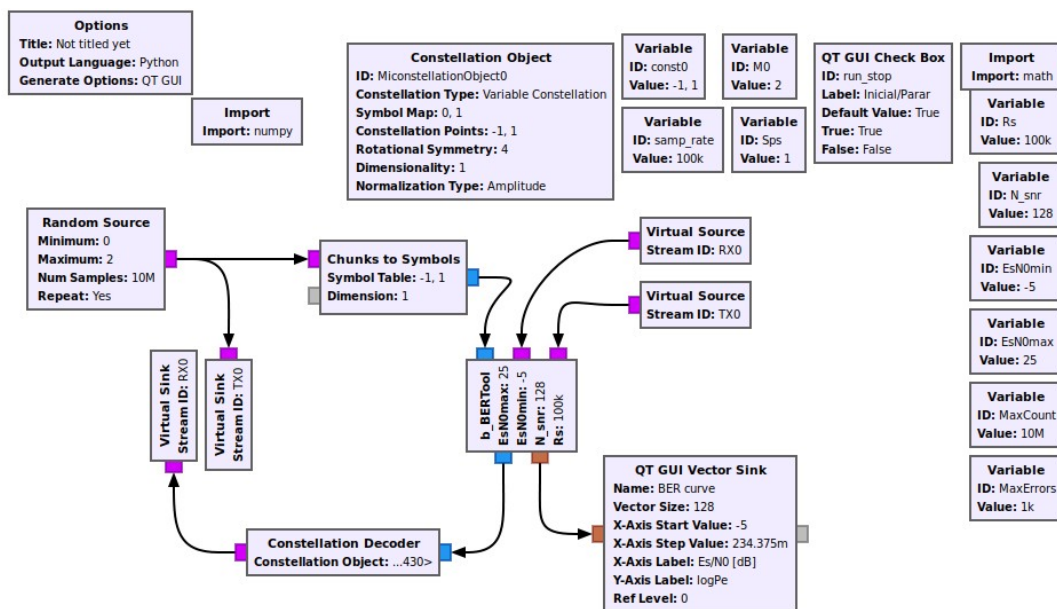
1.1 Študovanie projektov iných

Snažil som sa rozbehať dva projekty, jedno OFDM [11] a druhé SER [12]. OFDM sa mi nepodarilo lebo baličky, ktoré to žiadalo neboli už dostupné. Avšak SER sa mi podarilo rozbehať v Python3 OOT bloku. Celý tento projekt som musel prerobiť do GR 3.9, lebo flowgraphy boli robené pre GR 3.8 a samotné bloky pre GR 3.7. Nechal som sa inšpirovať týmto projektom [12], lebo ponúkal jednoduchšie zapojenie pre testovanie SER. K tomu bol ešte samostatný AWGN kanál, ktorý pripočítal rôzny šum podľa $E_s/N_0 <min; max>$ ku modulovanému signálu. Ako štartovací bod mi tento projekt postačil, ale mal veľa chýb, nedostatkov (Python). A samozrejme všetko bolo po španielsky písané, hlavne kod a komentý.

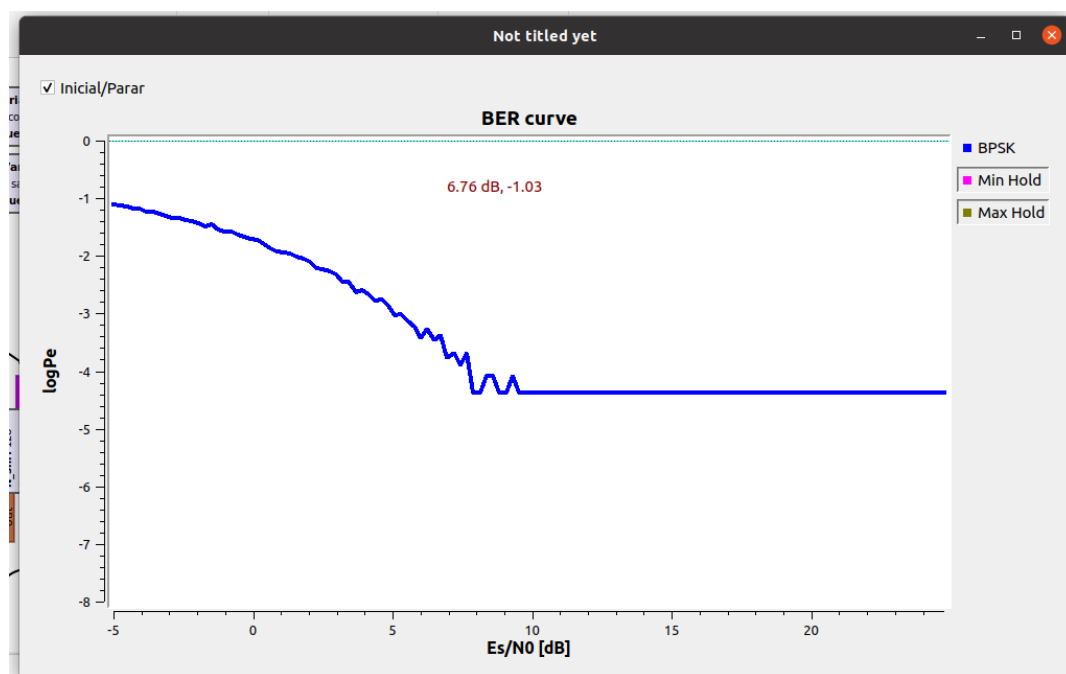


Obr. 1: Hierarchický blok "b_BERTool" v študovanom projekte [12]

Hierarchický blok sa používa ako obal na zjednodušenie viacerých blokov GNU Radio do jedného bloku. [5]



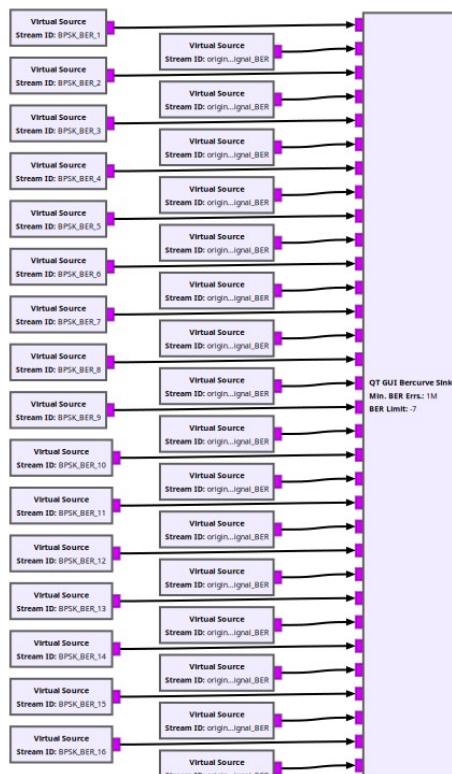
Obr. 2: Hlavný flowgraph "b_bertool_plus" v študovanom projekte [12]



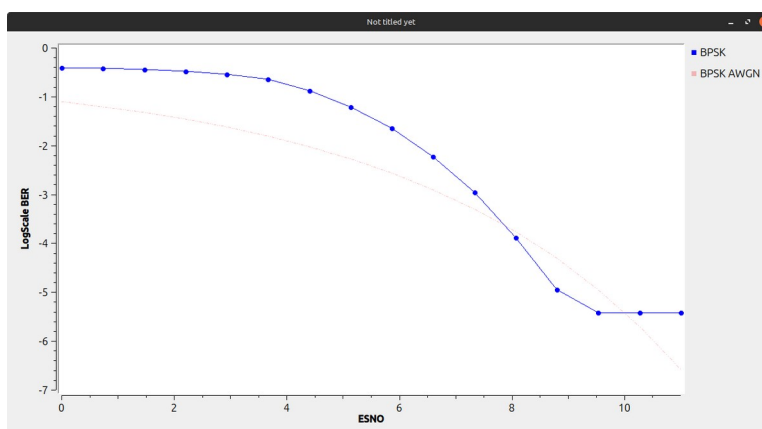
Obr. 3: BER krivka pre BPSK v študovanom projekte [12]

1.2 Aktuálna situácia v GR 3.10

GR 3.10 ponúka *QT GUI Bercurve Sink* blok na výpočet a vykreslenie BER krivky. Ako to vidno na Obr. 4, zapojenie tohto bloku je neefektívne z hľadiska rozširiteľnosti. Čím viac stavová je modulácia tak tým sa zvyšuje počet vstupov do bloku. [1]



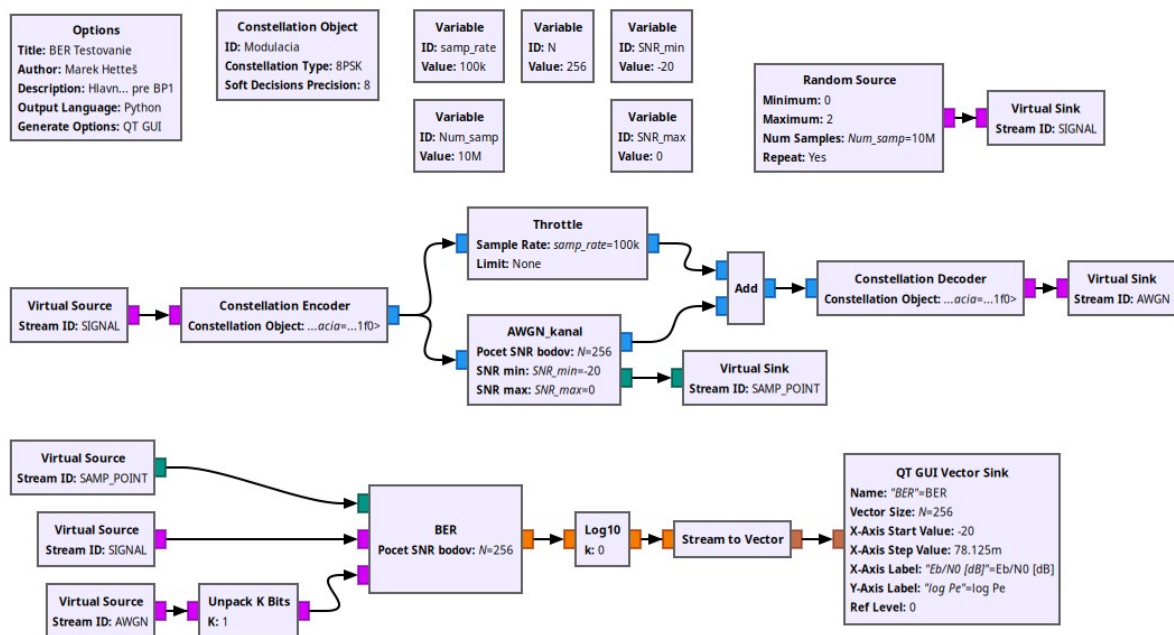
Obr. 4: *QT GUI Bercurve Sink* blok úplne napravo



Obr. 5: BER krivka pre MPSK od GR

1.3 Môj projekt

V tomto odseku opíšem moje všeobecné riešenie problematiky, kde to porovnam voči projektu, ktorý som skúšal v GR 3.9 a BER nástroje, ktoré ponúka GR v základnej inštalácii. Ako som spomínal, nechal som sa inšpirovať projektom, kde BER a AWGN bloky majú menej vstupov/výstupov než čo je v GR a zobrazuje sa to do *QT GUI Vector Sink*. Ďalej stačí pridať šum na celý signál a netreba rozkladať na jednotlivé bity a potom ich dať zase dokopy (maximálne pri vstupe do BER bloku, keď M sa nerovná 2, kde M je počet stavov v modulácii). Všetky bloky som písal ako C++ OOT. Toto nám zaručuje väčšiu kontrolu nad spracovaním signálu a zároveň vieme písať optimálnejší kod. V projekte boli tieto bloky v hierarchickom bloku, čož zbytočne komplikuje náš flowgraph, ja som to zapojil ako jeden flowgraph a využil som GR modulátor namiesto Chunks to Symbols. V tomto zapojení som skúšal všetky modulácie, ktoré GR ponúka v konštelacom objekte, t.j. *MPSK*, *DQPSK*, *MQAM*.

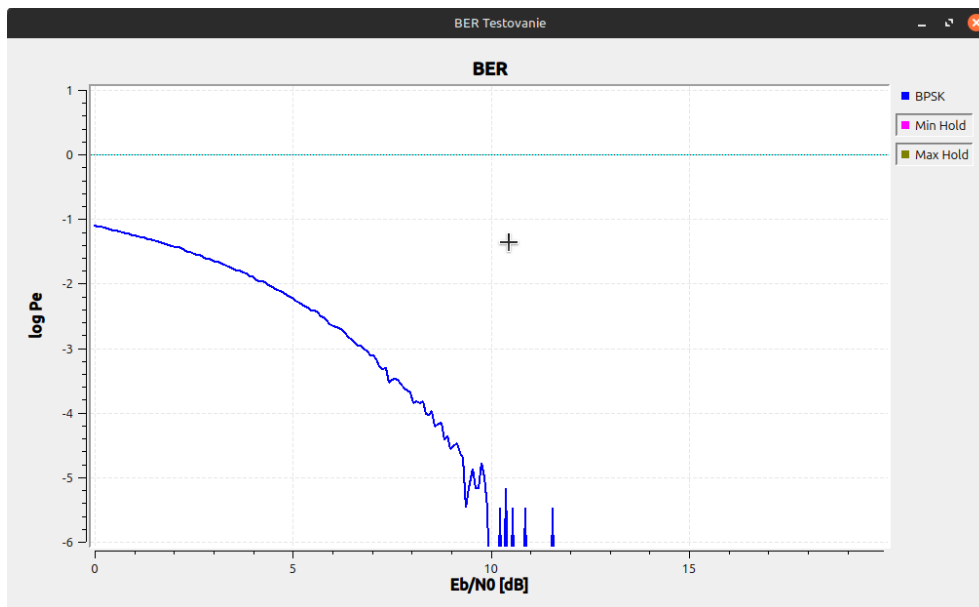


Obr. 6: Flowgraph môjho projektu

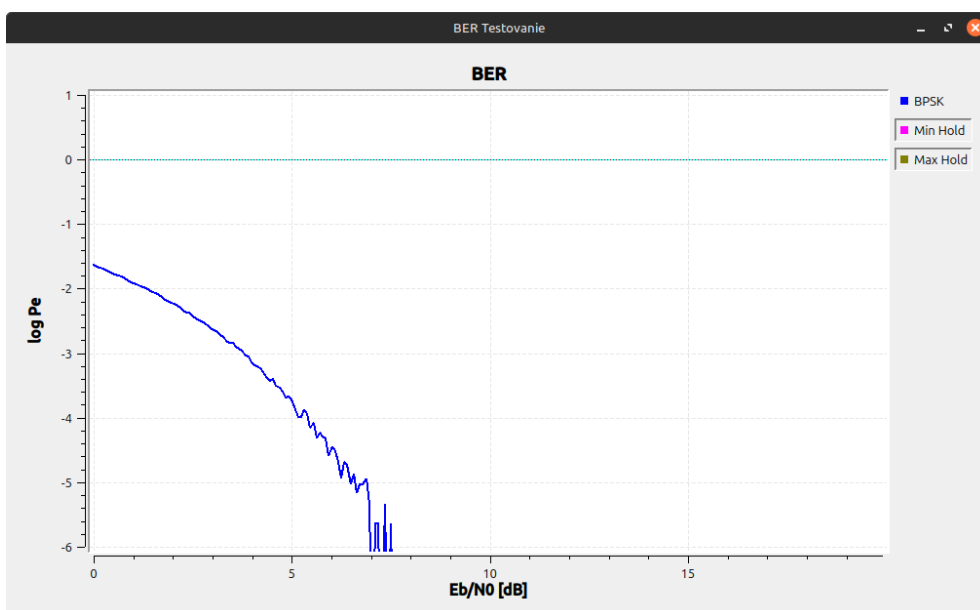
1.3.1 Výsledky projektu

Všetky testy boli v rozmedzí 0 až 20 dB E_b/N_0 pri vzorkovacej frekvencii 10 Mhz a pri 50 M vzoriek zo zdroja. Počet E_b/N_0 bodov bolo 256 .

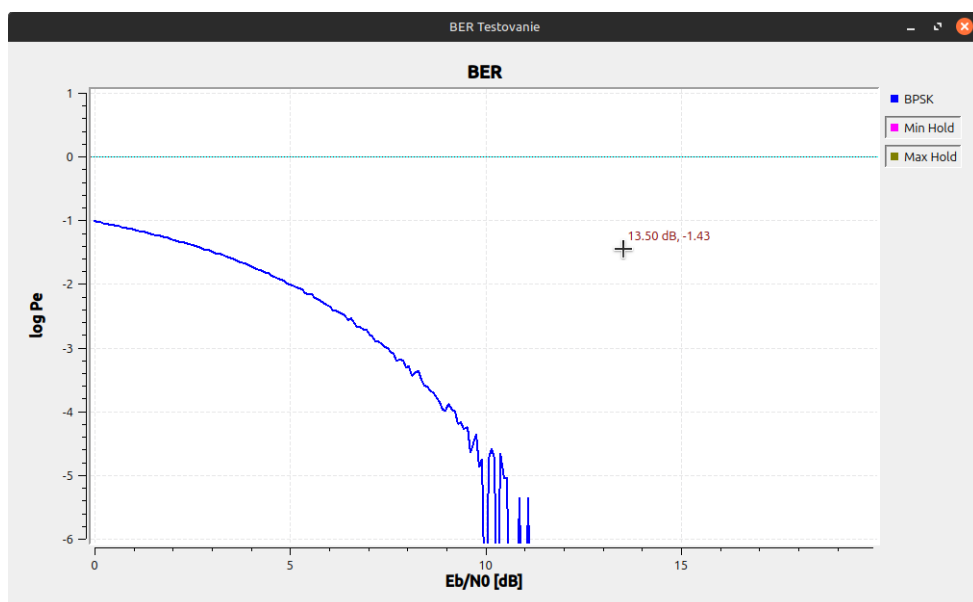
Nasledujúce obrázky výstupy v QT GUI Vector Sink (BER krivky):



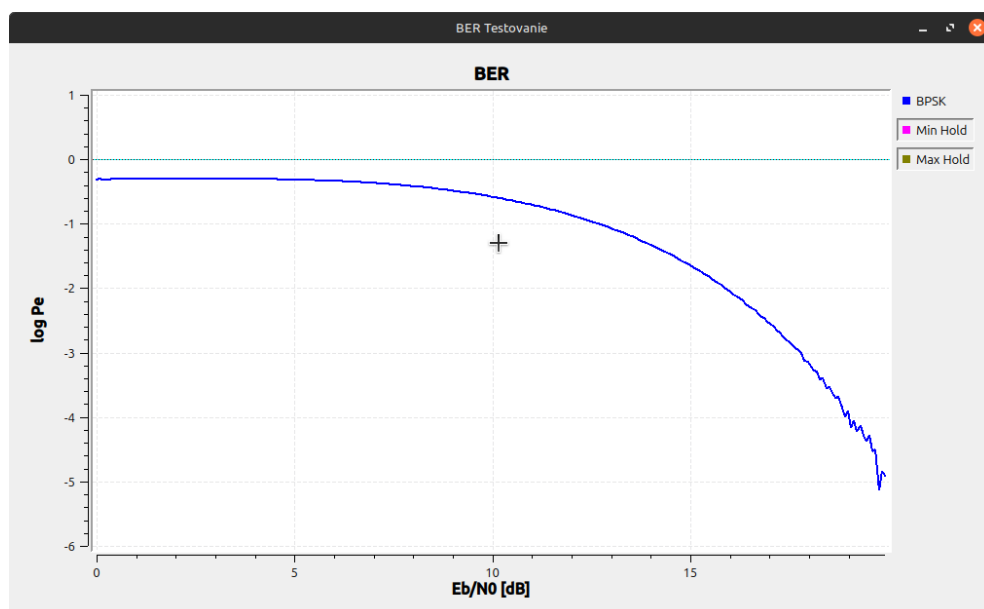
Obr. 7: BER pre BPSK



Obr. 8: BER pre QPSK

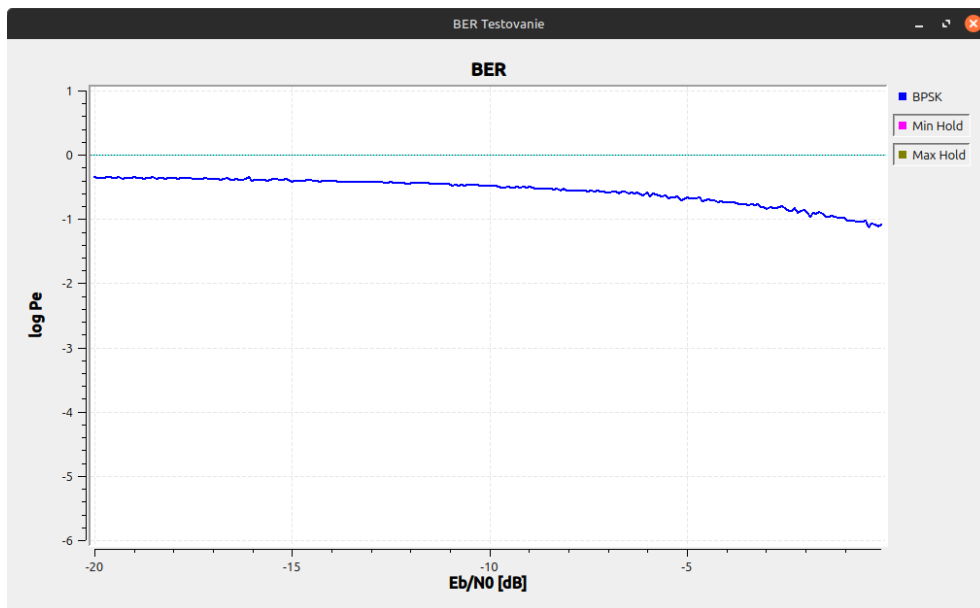


Obr. 9: BER pre 8PSK



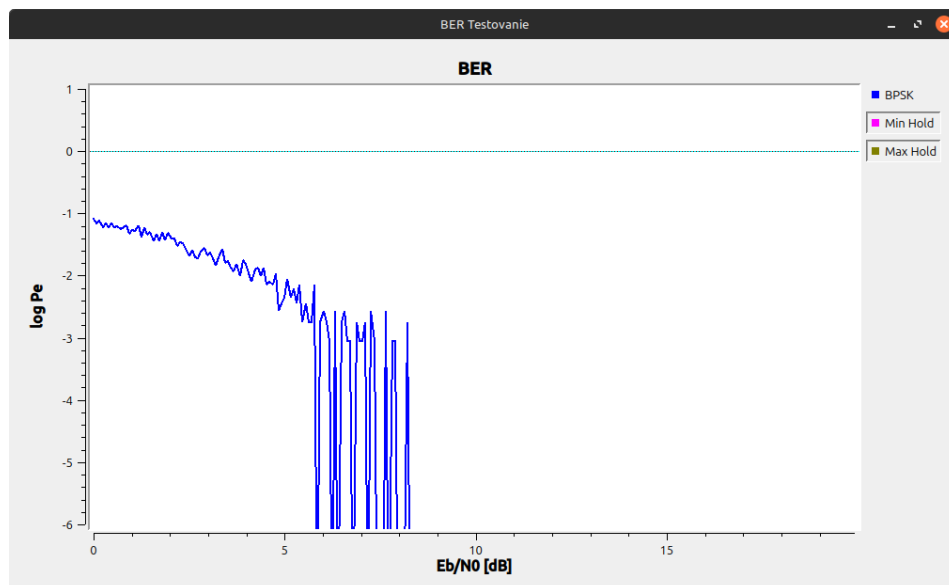
Obr. 10: BER pre 16-QAM

Nasledujúce testy boli v rovnakých podmienkach ako ostatné, avšak boli zmenené niektoré parametre aby sa demonštrovalo ako sa správajú moje bloku v rôznych prípadoch. BPSK modulácia bola použitá v oboch prípadoch. Pri Obr. 9 bolo zmenené rozpätie E_b/N_0 na interval od -20 až 0 dB, teda šum má vyšší výkon než signál.



Obr. 11: BER pre záporné hodnoty E_b/N_0 [dB]

Pri Obr. 10 bol vykonaný test v prípade, že sa neprenesie dostatočný počet bitov a tým pádom BER krivka nie je "vodopádový" ako sa očakáva. Počet vzoriek zo zdroja bolo znížených na 500.



Obr. 12: BER v prípade nedostatnu prenesených bitov

2 Out Of Tree blok

V tejto časti opíšem všeobecné problémy a moje riešenia na rôzne časti tvorby OOT bloku. Pri oboch blokoch som musel riešiť tieto veci, takže som uznal za vhodné ich dať ako jednu kapitolu.

2.1 Vstupy / Výstupy s rôznymi dátovými typmi

Jeden z mojich dlhodobých problémov bolo ako pridať viac vstupov alebo výstupov s rôznymi dátovými typmi, napr. VSTUP: `int`, `float`, `float`; VYSTUP: `byte`, `int`. Nikde som nevedel nájsť dokumentáciu k tomuto, avšak "brute force" metódou som prešiel všetky bloky v GR a v dostupných zdrojových kodoch na GitHubu pre bloky, ktoré mali rôzne vstupy/výstupy som "reverse engineer"-ol logiku. [13]

Riešenie spočíva v tom, že namiesto jednej premennej v `"io_signature"` dáme vektor premenných. Zároveň zmeníme `"io_signature::make"` na `"io_signature::makev"`. Táto jediná zmena mi dovolila použiť ľubovoľný mix dátových typov. [13]

```
22 ...
23
24 // Output vector
25 static int os[] = { sizeof(gr_complex), sizeof(int) };
26 static std::vector<int> osig(os, os + sizeof(os) / sizeof(int));
27
28 //The private constructor
29 AWGN_kanal_impl::AWGN_kanal_impl(int N, int EbN0min, int EbN0max)
30 : gr::sync_block("AWGN_kanal",
31                 gr::io_signature::make(1, 1, sizeof(gr_complex)),
32                 gr::io_signature::makev(1, 2, osig))
33
34 ...
```

Avšak v neposlednom rade treba aj zmeniť inicializáciu I/O vo "work" funkcii a to tak, že pre každý vstup/výstup staticky pridáme jeho dátový typ (napr. z `auto` na `int *`).

```
93 ...
94
95 //-----Inicializacia-I/O-----|
96 // INPUT
97 gr_complex *in0 = (gr_complex *) input_items[0];
98
99 // OUTPUT
100 gr_complex *out0 = (gr_complex *) output_items[0];
101 int *out1 = (int *) output_items[1];
102
103 ...
```

Samozrejme aj v YAML súbore treba zmeniť dátový typ vstupov/výstupov ale tento krok je už triviálny.

2.2 Zmena v header súbore

Keď súbor `"gr-modul/include/gnuradio/modul/blok.h"` drasticky upravíme, nemôže CMake hneď skompilovať projekt. Ak sa o to pokúsime, dostaneme chybovú hlášku. Avšak toto neplatí pri tvorbe nového bloku!

Na toto je relatívne jednoduché ale pracné riešenie. Vytvoril som bash script aby sme nemuseli furt písať príkazy ale logika zmeny je nasledujúca: najprv si zmeníme verziu GCC a G++ z 13 na 11, prípadne ak ju nemáme tak si ju aj nainštalujeme. [17]

```
sudo apt update
sudo apt install gcc-11 g++-11
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-11 11
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-11 11
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
```

Ďalej musíme bind-núť *Python* s *C++* kodom. Na to existuje príkaz, ale musíme ho zadať v začiatočnom priečinku pre modul.

```
gr_modtool bind blok
```

A na koniec, stačí sa vrátiť do pôvodnej verzii, t.j. 13-tej. [17]

```
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
```

Toto nám opraví prepoj medzi jazykmi, ak teraz dáme kompilovať, tak nám to prejde tak ako má.

2.3 Inicializácia mimo funkcie "work"

Niektoré premenné vyžadovali aby boli buď deklarované alebo inicializované iba raz. To znamená, že som ich musel deklarovať a inicializovať mimo funkcie `"work"`. Nikde nie je vysvetlené ako to vlastne treba spraviť, takže po viacerých pokusoch sa mi to podarilo zistiť.

V súbore `"gr-modul/lib/blok.h"` si všetky potrebné premenné len deklarujem.

```
17 ...
18 private:
19     int _N;
20
21     // Deklarovanie vektorov
22     std::vector<int> _count;
23     std::vector<double> _pamat_BER;
24     std::vector<int> _pocet_chyb;
25 ...
```

V súbore "gr-modul/lib/blok.h" si všetky premenné, ktoré som deklaroval v header súbore teraz inicializujem v constructor.

```
35 ...
36 _N = N; // Pocet vzoriek EbN0 (kolko bodov na X-osi)
37
38 // Vektory
39 _count = std::vector<int>(N, 1); // Celkovy pocet bitov co sme spracovali
40 _pocet_chyb = std::vector<int>(N, 0); // Celkovy pocet zistenych chyb
41 _pamat_BER = std::vector<double>(N, 1.0); // Vystupne hodnoty BER
42
43 ...
```

2.4 Vektor alebo zoznam

Na začiatku som všetko robil cez zoznamy. Najprv to stačilo ale po čase mi to robilo problémy. Asi najväčší problém mi robila inicializácia bez pevne danej veľkosti. V C++ zoznamy toto nedokážu, ale vektory áno. Tak som všade kde som mohol dal vektory namiesto zoznamov. Funkcionalitu majú tú istú ale práca s nimi je oveľa ľahšia a môžem si pri nich viac dovoliť. [8]

2.5 Výpis hodnôt do konzole

Ak som chcel niečo vypísať, napr. hodnotu slova, nemohol som len tak dať niekde *printf()* alebo *cout*. Tieto príkazy keď zavolám vo funkcii "work", môže mi to zahltiť procesor a celé GR "zamrzne" v najhoršom prípade padne bez uložených zmien. Na toto som našiel dve riešenia, ktoré sa najlepšie robia kombinovane.

Treba využiť *GR_LOG_INFO(d_logger, STRING);*, je to optimalizovaný nástroj na výpis do GR konzole. Následne odporúča sa volať túto funkciu na výpis len každých *M* vzoriek, a nie každú vzorku. *M* sa určí podľa vzorkovacej frekvencie ale v ja som určil hodnotu 10 000. [4]

```
142 ...
143
144 if(k%10000 == 0)
145     GR_LOG_INFO(d_logger, std::string("b: ") + std::to_string(in2[i]) +
146                                     + std::string("\n"));
146
147 ...
```

3 AWGN blok

Na to aby som mal BER krivku po rôznych E_b/N_0 , vytvoril som si vlastný generátor šumu AWGN. Na rozdiel od GR generátora šumu, môj blok dokáže generovať rôzne úrovne šumu v zadanom rozsahu E_b/N_0 minimum a E_b/N_0 maximum (lineárne rozdelené).

3.1 Náhodná vzorka šumu

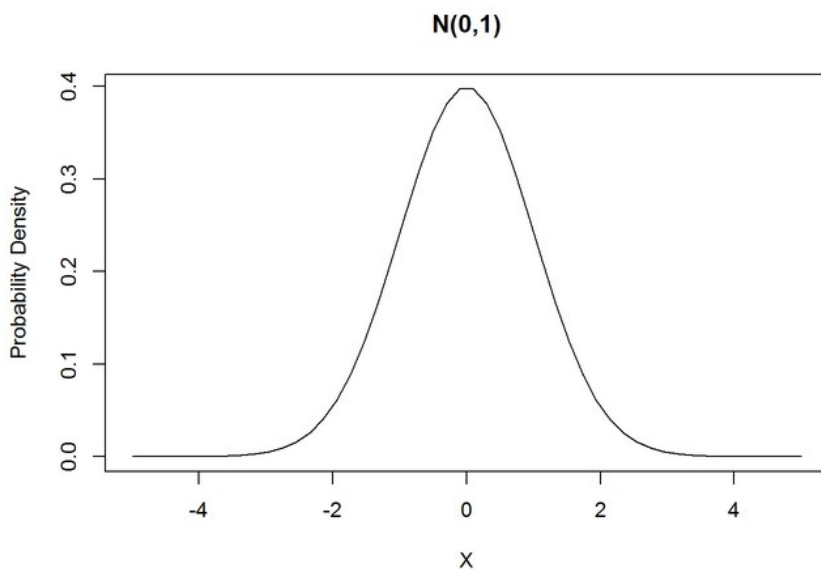
Začal by som asi s pomocnými funkciami. Prvá funkcia slúži na premenu E_b/N_0 z decibelov na bezrozmerný pomer (1), [19]

$$\frac{E_b}{N_0} = 10^{\frac{EDB}{10}} \quad (1)$$

a získanie náhodnej vzorky z normáloveho rozdelenia (2), ktoré potom vloží do komplexnej premennej. Výstupný šum vypočítame ako [19]

$$n(t) = \frac{N(0,1)}{\sqrt{2 \cdot \frac{E_b}{N_0}}} \quad (2)$$

čítateľ v (2) je len náhodná vzorka zo štandardnej normálovej distribúcie:



Obr. 13: Normálova distribúcia [15]

Menovateľ v (2) je rovnaký pre obe časti komplexného čísla ale čítateľ zavolá znova funkciu na náhodnú vzorku.

```
60 ...
61
62 //-----Odvodenie-varianci-esumu-z-EbN0-[db]-----|
63 gr_complex Sum(float EDB) {
64     double EbN0, N;
65     double REAL, IMAG;
66
67     // Premena z dB na pomer
68     EbN0 = pow(10.0, EDB/10.0);
69
70     double menovatel = sqrt(2 * EbN0);
71
72     REAL = Sum_vypocet() / menovatel;
73     IMAG = Sum_vypocet() / menovatel;
74
75     gr_complex n(REAL, IMAG);
76
77     return n;
78 }
79
80 ...
```

3.2 Normálový generátor

Nasledujúca funkcia je podľa vzoru na dokumentačnej stránke C++. [6] [7] [9]

```
42 ...
43
44 //-----Tvorba-Gaussovky-a-random-bodu-----|
45 double Sum_vypocet() {
46     double GR;
47
48     // Generovanie nahodneho cisla podľa seed
49     std::random_device rd;
50     std::mt19937 R(rd());
51
52     // Tvorba Gaussovky podľa odchyľky
53     std::normal_distribution<double> Gauss{0, 1}; //mi = 0 lebo AWGN
54
55     // Vyberieme nahodne hodnoty z Gaussovky
56     GR = Gauss(R);
57
58     return GR;
59 }
60 ...
```

3.3 Hlavná časť bloku (work)

Nasleduje funkcia "work", veľmi rád by som to prerobil ešte, ale zatiaľ to funguje tak sa toho nedotýkam.

3.3.1 Lineárne rozdelenie

Lineárne rozdelenie som si musel spraviť sám. Vzorec som si odvodil empiricky a dostal som:

$$R_z = \frac{\left(\frac{E_b}{N_0}\right)_{max} - \left(\frac{E_b}{N_0}\right)_{min}}{N - 1} \quad (3)$$

Do vektora *EDB* sa postupne potom kumulatívne pridáva E_b/N_0 hodnota. Začne sa s *EbN0min* a za každou iteráciou sa pripočíta rozpätie R_z (3), ktoré som si vypočítal. Toto sa robí až po *EbN0max*.

V hlavnom cykle, kde prechádza všetky I/O prvky (*items*), zavolá prvú pomocnú funkciu a ako argument použije jednu hodnotu E_b/N_0 z lineárneho rozdelenia. Keď sa vytvorí požadovaný šum, vyšle sa na prvý komplexný výstup *out0*.

Druhý výstup je bod, ktorá ukazuje na danú hodnotu E_b/N_0 , takto vedľa iné bloky sledovať presne, ktorá vzorka je s akým E_b/N_0 . Táto hodnota ide po *N-1*, potom sa vynuluje a ide odznova. Táto hodnota sa pošle na celočíselný (*int*) výstup *out2*.

3.3.2 Logika funkcie

Vzorec na AWGN (2) funguje určite na BPSK moduláciu. Podľa experimentov funguje aj na iné modulácie, ale to je len môj predpoklad, pokiaľ nenájdem lepší vzorec na to, ostáva tento.

Snažil som sa implementovať aj sčítanie signálu so šumom, ale rozhodol som sa to nespraviť aby sa zanechala modulárnosť bloku.

```

94  ...
95
96  // INPUT
97  gr_complex *in0 = (gr_complex *) input_items[0];
98
99  // OUTPUT
100 gr_complex *out0 = (gr_complex *) output_items[0];
101 int *out1 = (int *) output_items[1];
102
103 //-----LOGIKA-----|
104 //-----Prvotne-vypocty-----|
105 float EDB[_N];
106
107 // Linearne rozlozenie EbN0db bodov
108 rozpatie = float((_EbN0max - _EbN0min)) / float((_N-1));
109 rozpatiePostup = float(_EbN0min);
110
111 for(int i = 0; i < _N; i++) {
112     EDB[i] = rozpatiePostup;
113     rozpatiePostup += rozpatie;
114 }
115
116 //STARE
117 // Vypocet výkonu vstupneho signalu Ps = E(x)
118 /*for(int a = 0; a < noutput_items; a++)
119     sumPs += pow(abs(in0[a]), 2);
120
121 Ps = sumPs / float(noutput_items);
122 */
123
124 //-----Prejdeme-všetkými-I/O-items-----|
125 for(int b = 0; b < noutput_items; b++) {
126
127     // Ziskanie komplexneho sumu
128     gr_complex sg_n = Sum(EDB[k]);
129
130     // Tuto by sa malo scitat ale C++ neznasa komplexne cisla, alebo mna...
131     //gr_complex spolu(in0[b].real() + sg_n.real(), in0[b].imag() + sg_n.imag());
132
133     // Komplexny vystup = AWGN sum
134     out0[b] = sg_n;
135
136     // Int vystup = N-ta vzorka EbN0, ktoru sme pocitali
137     out1[b] = k;
138
139     // Iterujeme po všetkých vzorkách EbN0, t.j. od 0 do N-1
140     if(k < _N-1) {
141         k += 1;
142     }else {
143         k = 0;
144     }
145 }
146 ...

```

4 BER blok

V tejto časti opíšem môj algoritmus na zistenie bitovej chyby, ako som postupoval a s čím som nespokojný. Navrhнем tu aj možné vylepšenia, každopádne súčasná verzia v čase písanie reportu funguje ako má.

4.1 Inicializácia premenných

Začal by som s inicializáciou vektorov a premenných. Premenná N je len veľkosť vektorov. Teda pre koľko bodov E_b/N_0 budeme počítat'.

Vektor `_count` pozoruje a zapisuje koľko bitov prešlo pre každý E_b/N_0 bod. Toto by sa podľa mňa dalo upraviť na jednu premennú namiesto vektora, lebo v podstate cez všetky E_b/N_0 body prechádza rovnaký počet bitov, t.j. jeden za cyklus. Takto by som ušetril RAM a výpočtovú silu. Inicializuje sa, ako ostatné vektory, na už spomínaných N veľkosť a pri tomto vektore na počiatočnú hodnotu 1. Kebyže je to 0 tak by sme potom delili nulou.

Vektor `_pocet_chyb` robí viacmenej to isté čo `_count`, avšak teraz sleduje a počíta počet nastaných bitových chýb. Tuto už môže byť počiatočná hodnota 0.

Posledný vektor je interná pamäť pre vypočítane hodnoty BER `_pamat_BER`. Aby sa samozdokonalovala BER krivka, musí si blok zapamätať svoje hodnoty, na to slúži tento vektor. Hodnoty vektora sa nekumulujú ale prepisujú.

```
35 ...
36 _N = N; // Pocet vzoriek EbN0 (kolko bodov na X-osi)
37
38 // Vektory
39 _count = std::vector<int>(N, 1); // Celkovy pocet bitov co sme spracovali
40 _pocet_chyb = std::vector<int>(N, 0); // Celkovy pocet zistenych chyb
41 _pamat_BER = std::vector<double>(N, 1.0); // Vystupne hodnoty BER
42
43 ...
```

4.2 Logika bloku

Teraz by som opísal hlavnú logiku môjho BER bloku. Sú dve hlavné časti algoritmu, jeden pre porovnanie vstupných signálov a druhá je zistenie počet chýb.

4.2.1 Porovnanie vstupných signálov

Prvá časť je dosť okomentovaná ale z toho dôvodu, že som experimentoval s logaritmickým výstupom, tým pádom ušetríme jeden blok vo flowgrape. Nakoniec som sa rozhodol to takto nechať aby to bolo modulárne. Argument bere pôvodný signál *S1* a signál so šumom *S2*. Tieto dva signály *XOR*-ne a ostane len jedno kodové slovo, kde sa vyskytnú chyby ako jednotky. Zavoláme funkciu *BitCounter()*, čo je druhá časť algoritmu. Hodnotu, ktorú nám vráti, vrátime späť do "work" funkcie. [18]

```
76 ...
77
78 //-----Zisti-ci-nastala-chyba-----|
79 int BER_calc(unsigned char S1, unsigned char S2) {
80     unsigned char XORnute;
81     int e_sum;
82
83     // Logaritmické premenne
84     //float BER, log_BER;
85
86     // XOR-neme slovo zo signalu 1 a slovo zo signalu 2, vieme ze je chyba,
87     // ale nevieme kde a koľko chyb
88     XORnute = S1 ^ S2;
89
90     // Ak hej, tak nie je nulova hodnota e_sum
91     e_sum = BitCounter(XORnute, 1);
92
93     // Logaritmický vystup
94     //log_BER = logf(e_sum);
95
96     //return log_BER;*/
97     return e_sum;
98 }
99 ...
```

4.2.2 Zistenie počet chýb

Druhá časť zobere ako argument *XOR*-nuté signály a veľkosť kodového slova. Najprv sa zistí či je slovo záporné, ak je tak sa porovnávacia jednoka posúva doľava po celom slove, a ak je kladé, posúva sa slovo doprava. V tomto prípade posun je *Bit-Shift* doprava alebo doľava. Argument funkcie je *unsigned char*, tým pádom nenastajú záporné hodnoty. Celý tento algoritmus som vyvynul a skúšal mimo bloku a v jazyku C. Túto nezhodu by som veľmi rád ešte potom upravil, buď so alebo bez záporných hodnôt slova. Následne cyklom prejde celým slovom a každý bit *AND*-ne s jednotkou. Ak výsledok logickej operácie je jedna, našiel chybu a pripočíta sa k sume chýb. Po skončení cyklu vráti sumu chýb.

```

52 ...
53
54 //-----Zisti-koľko-chyb-je-v-slove-----|
55 int BitCounter(unsigned char slovo, int N) {
56     int sum = 0, sign = 0;
57     unsigned char b_jedna = 1;
58
59     // Ci hodnota slova je zaporna
60     if(slovo < 0)
61         sign = 1;
62
63     for(int i = 0; i < N; i++) {
64         if((slovo & b_jedna) > 0)
65             sum++;
66
67         if(sign) { //ked plus
68             slovo >>= 1;
69         }else { //ked minus
70             b_jedna <<= 1;
71         }
72     }
73
74     return sum;
75 }
76 ...

```

4.2.3 Hlavná časť bloku (work)

Teraz ideme na funkciu "work". Prvý vstup pochádza z AWGN kanála a je to bod s určitou

E_b/N_0 hodnotou. Využil som to ako index vo vektoroch aby sa správne potom zobrazilo BER v "QT GUI Vector Sink". V tomto bloku teda slúži len ako index pre vektory. Taktiež jeho hodnota sa rovná prvého vstupu bloku *in0* (int).

Prvý vstup by mal byť použitý na pôvodný signál a druhý na signál so šumom, ale nemalo by nič stať kebyže vymeníme poradie, lebo XOR nerieši poradie. Tieto vstupy vložíme ako argumenty do prvej funkcie kde sa zo-XOR-ujú. Vrátenú hodnotu (počet chýb) pripočítame k vektoru chýb v danom E_b/N_0 bode, ktorú určuje prvý vstup zapísaný v premennej *k*.

$$P_b = \frac{N_{chyb}}{N_{bitov}} \quad (4)$$

Tuto P_b je pravdepodobnosť chyby na bit od 0 po 1. Keď tieto hodnoty zlogaritmujeme tak dostaneme záporné hodnoty.

Na jediný výstup ide vypočítaná pravdepodobnosť chyby na bit (BER) ako desatinné číslo (double) do logaritmickeho bloku.

Na koniec sa ešte inkrementuje počet prejdenných bitov cez blok o jedna. Pri iných moduláciach ako BPSK je viacbitové slovo po demodulovaní. Aby blok prešiel všetky bity musel som rozdeliť to

slovo na jednotlivé bity. Vyriešil som to pomocou bloku *Unpack K Bits*, tým pádom na vstupe BER bloku bude vždy jeden bit.

```
106 ...
107
108 //-----Inicializacia-I/O-----|
109 // INPUT
110 int *in0 = (int *)input_items[0];
111 unsigned char *in1 = (unsigned char *)input_items[1];
112 unsigned char *in2 = (unsigned char *)input_items[2];
113
114 // OUTPUT
115 auto out = static_cast<output_type*>(output_items[0]);
116
117 //-----LOGIKA-----|
118 //-----Prvotne-vypocty-----|
119
120 /*long long int pocet_chyb[_N], count[_N];
121 float pamat_SER[_N];
122 std::fill_n(pocet_chyb, _N, 0);
123 std::fill_n(count, _N, 8);
124 std::fill_n(pamat_SER, _N, 1.0);*/
125
126 // Bod v SNR
127 int k;
128
129 //-----Prejdeme-vsetkymi-I/O-items-----|
130 for(int i = 0; i < noutput_items; i++) {
131     k = in0[i];
132
133     // Zistime, ci nastala chyba v slove
134     _pocet_chyb.at(k) += BER_calc(in1[i], in2[i]);
135
136     // Nastavenia pre logaritmus v pripade ze nenastala chyba
137     // GR nedokaze zobrazit v QT GUI nekonečna
138     if(_pocet_chyb.at(k) == 0) {
139         _pamat_BER.at(k) = 0.00000001; // log cisla je -8
140     } else {
141         _pamat_BER.at(k) = double(_pocet_chyb.at(k)) / double(_count.at(k));
142         // Tuto nastala chyba, nemenime umelo hodnotu, pocet chyb deleno
143         // pocet bitov za celu dobu
144
145     }
146
147     /* if(k%10000 == 0)
148     GR_LOG_INFO(d_logger, std::string("b: ") + std::to_string(in2[i]) +
149         std::string("\n"));
150     */
151
152     // Float vystup = pravdepodobnost chyby na bit v danom Eb/N0
153     out[i] = _pamat_BER.at(k);
154
155     // Inkrementujeme celkovy pocet bitov
156     _count.at(k) += 1;
157 }
158 ...
```

5 Plán na letný semester

Určite by som pridal ešte SER OOT blok, lebo to je len upravený BER, ktorý už mám. A ak mi ostane čas. Každopádne moja hlavná priorita by malo byť stále BER blok.

Pridal by som určite aj teoretickú krivku ku reálnej aby sa dalo porovnávať. Táto krivka by fungovala pre rôzne modulácie a dalo by sa vypnúť a zapnúť.

Na zobrazovanie vodopádovej krivky BER by som buď použil samotný BER block, t.j. bude typu "Sink". Alebo pridám ďalší výstup, ktorý pošle rovnako do "*QT GUI Vector Sink*" teoretickú krivku pre danú moduláciu.

Ak by AWGN blok ostal, tak by som určite najprv zmenil jeho typ zo "*sync*" na "*source*", lebo jeho jediný komplexný vstup už nepotrebuje (v minulosti som to využil). Zároveň pri som pridal k tomuto bloku aj možnosť meniť E_b/N_0 rozpätie počas behu programu (callback). [20]

Ako som písal dokumentáciu blokov tak som pridal rovno aj nejaké návrhy na vylepšenia. Písal som to tak aby bolo zrozumiteľné podľa kontextu a nie všetko na koniec.

6 Literatúra

GR:

- [1] <https://wiki.gnuradio.org/index.php?title=Tutorials>
- [2] <https://wiki.gnuradio.org/index.php?title=LinuxInstall>
- [3] https://wiki.gnuradio.org/index.php?title=Simulation_example:_BPSK_Demodulation
- [4] https://wiki.gnuradio.org/index.php?title=Legacy_Logging
- [5] https://wiki.gnuradio.org/index.php/Hier_Blocks_and_Parameters

C++:

- [6] <https://www.pcg-random.org/using-pcg-cpp.html>
- [7] <https://cplusplus.com/reference/random/>
- [8] <https://en.cppreference.com/w/cpp/container/vector.html>
- [9] https://en.cppreference.com/w/cpp/numeric/random/normal_distribution.html
- [10] <https://en.cppreference.com/w/cpp/numeric/complex.html>

GitHub:

- [11] <https://github.com/rwth-ti/gr-ofdm>
- [12] https://github.com/hortegab/comdig_su_software_libro3.8?tab=readme-ov-file
- [13] <https://github.com/gnuradio/gnuradio/blob/main/gr-digital/lib>

Teoria:

- [14] <https://dsplog.com/2007/08/05/bit-error-probability-for-bpsk-modulation/>
- [15] <https://rpubs.com/nitinseelam/NormalDistribution>

Rôzne:

- [16] <https://mattgibson.ca/compiling-downgrading-to-gnuradio-3-7-x-on-ubuntu-20-x/>
- [17] <https://webhostinggeeks.com/howto/how-to-downgrade-gcc-version-on-ubuntu/>
- [18] <https://www.asci-code.com/>

- [19] *DK1 a DK2 poznámky*
https://www.mangoud.com/EENG373_files/Book-Sklar.pdf

- [20] *Report z extra zadania z predmetu IPV6*
https://github.com/Aymo19/ExtraZadanie_IPv6

Odkaz na GitHub repository

Tu sa dá stiahnúť celý môj projekt, dokumenty a flowpgraphy:

https://github.com/Aymo19/BP_2026

Zadanie BP

Téma bakalárskej práce je zameraná na vývoj testovacieho prostredia v GNU Radio, ktoré umožní monitorovanie a hodnotenie SDR simulácie alebo reálneho prenosu v reálnom čase.

1. Preskúmajte architektúru GNU Radio, so zameraním na možnosti spracovania dát v reálnom čase.
2. Navrhňte vlastný komponent pre zistenie BER, SER pre rôzne druhy modulácie v GNU Radio v jazyku C++.
3. Implementujte prostredie, ktoré bude počas simulácie alebo reálneho prenosu sledovať a vizualizovať výstupné parametre ako BER, SER, SNR, spektrum, konštelačný diagram a iné.
4. Rozšírte vaše riešenie o používateľské rozhranie, ktoré umožní sledovanie priebehu simulácie ako aj ovládanie jej parametrov počas behu.
5. Zdokumentujte implementáciu, možnosti využitia a návrhy na ďalší rozvoj vášho riešenia.

Literatúra:

1. GNU Radio Tutorials. Dostupné online: <https://wiki.gnuradio.org/index.php/Tutorials>
2. GNU Radio Project. Dostupné online: <https://www.youtube.com/@GNURadioProject/videos>
3. GNU Radio: Creating C++ OOT with gr-modtool. Dostupné online: https://wiki.gnuradio.org/index.php?title=Creating_C%2B%2B_OOT_with_gr-modtool