SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

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

Bakalársky projekt 1

Bratislava 2025 Marek Hetteš

Obsah

Skratky

GR – GNU Radio
OOT – Out Of Tree
I/O – Vstup/Výstup
OFDM – Ortogonal Frequancy Division Multiplexing
PSK – Phase Shift Keying
QAM – Quadrature Amplitude Modulation
AWGN – Additive White Gaussian Noise
BER – Bit Error Rate
SER – Symbol Error Rate
YAML – Yet Another Markup Language
GCC – GNU C Compiler
SNR – Odstup signál/šum
RAM – Random Access Memory

SNR sa považuje v tomto dokumente ako Eb/N0.

Odkaz na GitHub repository pre môj projekt

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

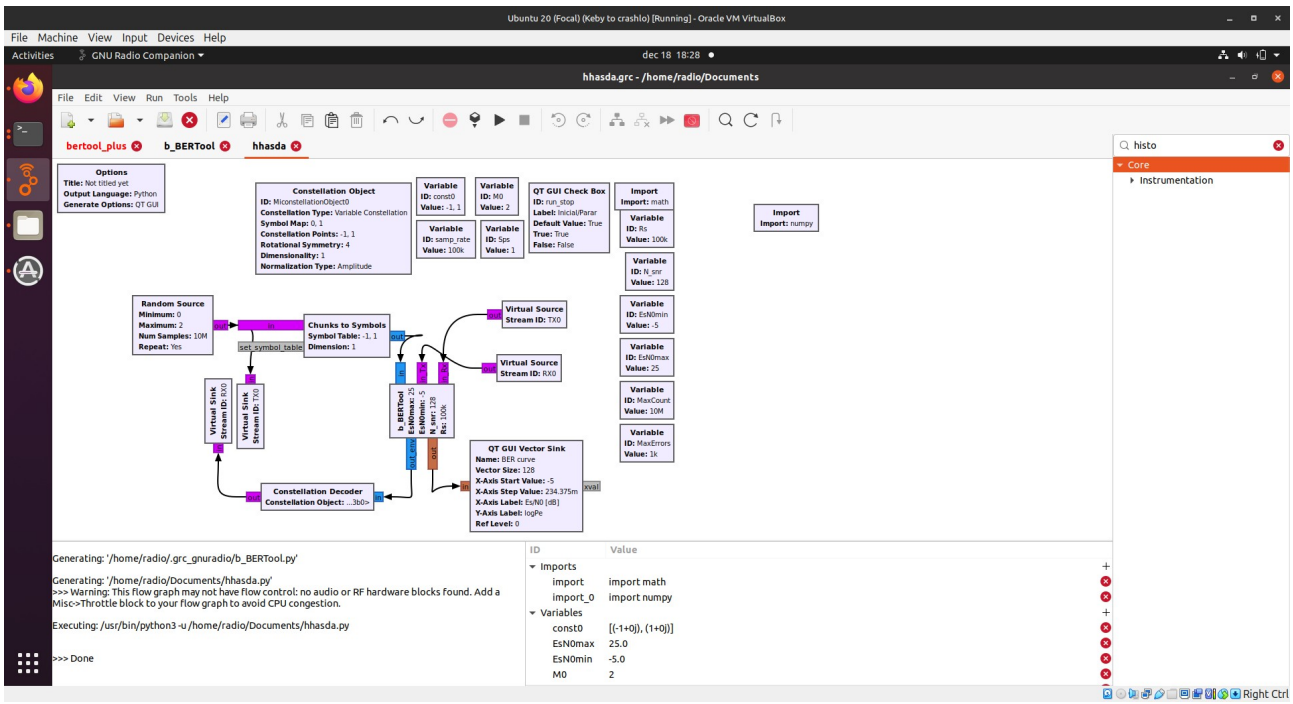
https://github.com/Aymo19/BP_2026

GNU Radio

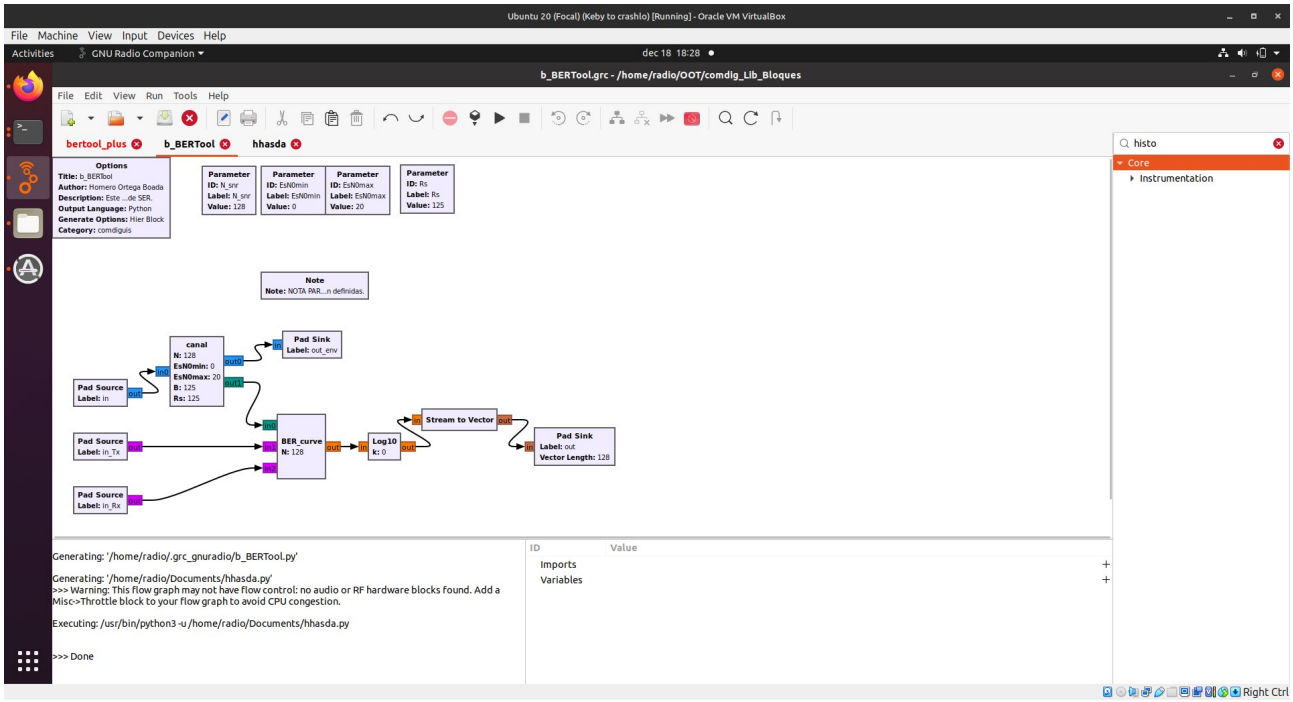
Moje bloky boli vyvinuté pre GNU Radio 3.10 avšak dajú sa skompilovať aj pre 3.9. Testoval som môj projekt na Linux Mint 22, Ubuntu 24 a Ubuntu 20 a všade to bez problémov fungovalo. Cez leto sa mi taktiež podarilo rozbehať GR 3.7 a GR 3.8 na Ubuntu 20 cez kompiláciu zdrojového kodu.

Študovanie projektov iných

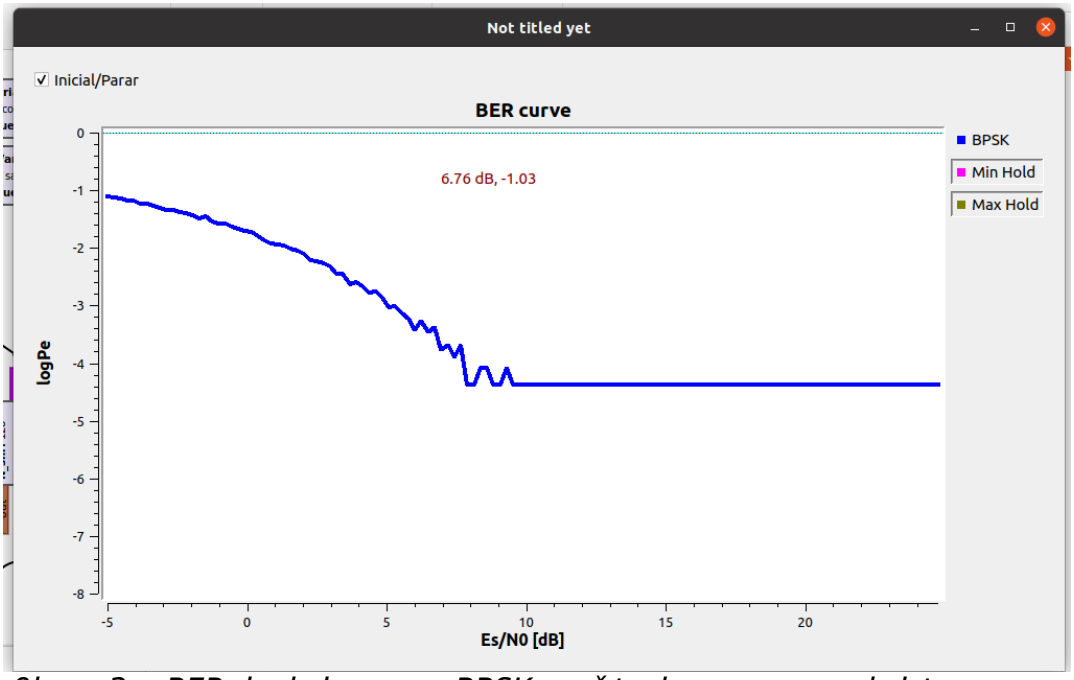
Snažil som sa rozbehať dva projekty, jedno OFDM a druhé SER. OFDM sa mi nepodarilo lebo baličky, ktoré to žiadalo neboli už dostupné. Avšak SER sa mi podarilo rozbehať v Python3 OOT bloku. Celý tento projekt som musel prerobiť do GR 3.9, lebo flowgraphy boli robené pre GR 3.8 a samotné bloky pre GR 3.7. Nechal som sa inšpirovať týmto projektom, lebo ponúkal jednoduchšie zapojenie pre testovanie SER. K tomu bol ešte samostatný AWGN kanál, ktorý pripočítal rôzny šum podľa Es/N0 <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 komenty.



Obr. 1: Flowpgragh študovaného projektu v GR 3.9



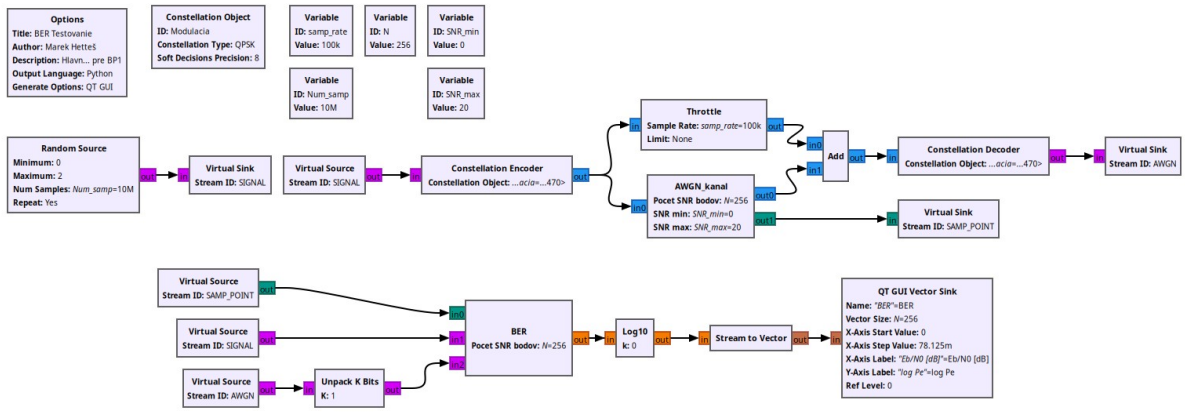
Obr. 2: Hiearchický blok študovaného projektu v GR 3.9



Obr. 3: BER krivka pre BPSK v študovanom projekte

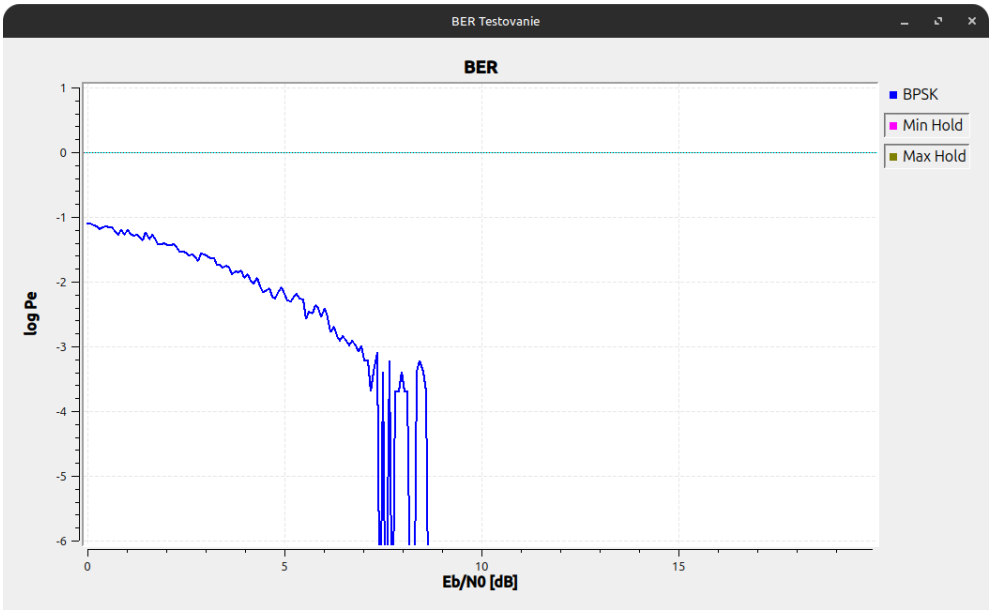
Môj projekt

V tomto odseku opíšem moje všeobecné riešenie problematiky, kde to porovnám 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 =/= 2). 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 hiearchickom 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 konštelačnom objekte, t.j. MPSK, DQPSK, MQAM.

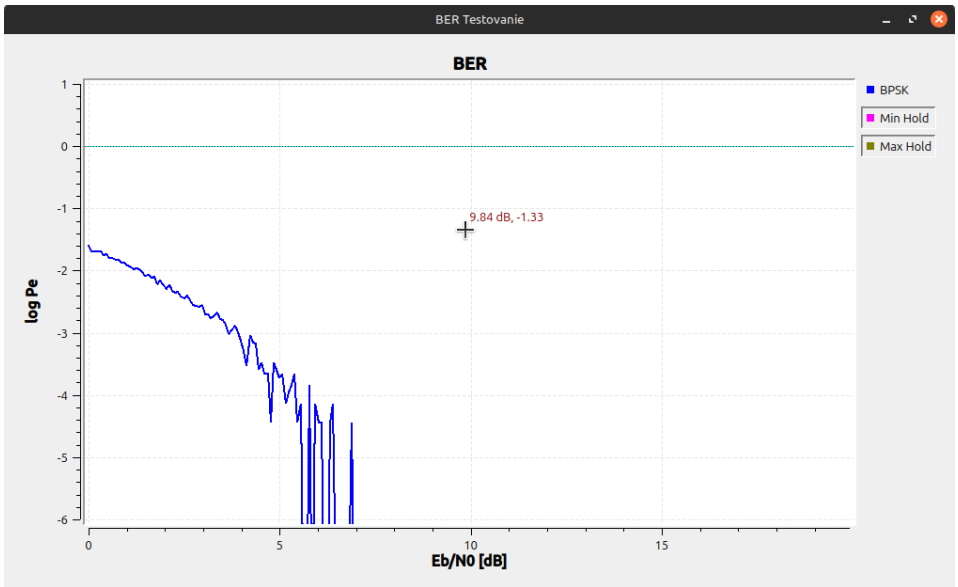


Obr. 4: Flowgraph môjho projektu

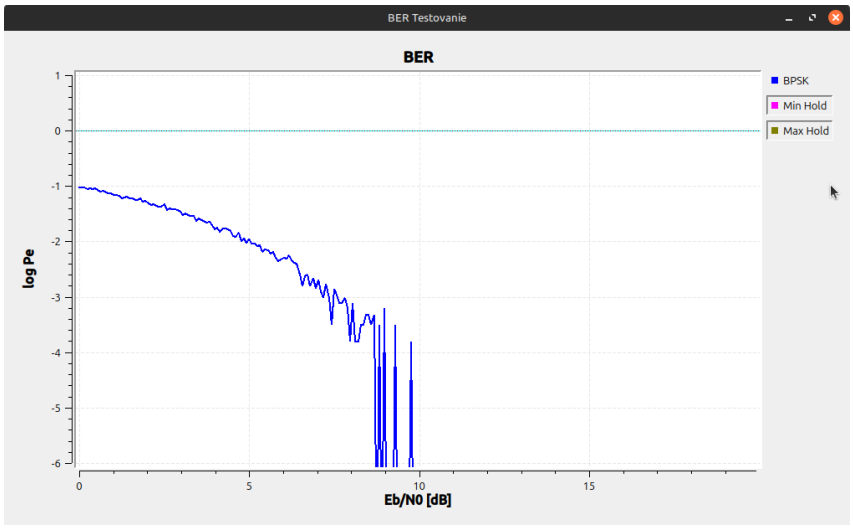
Nasledujúce obrázky sú BER krivky z môjho projektu.



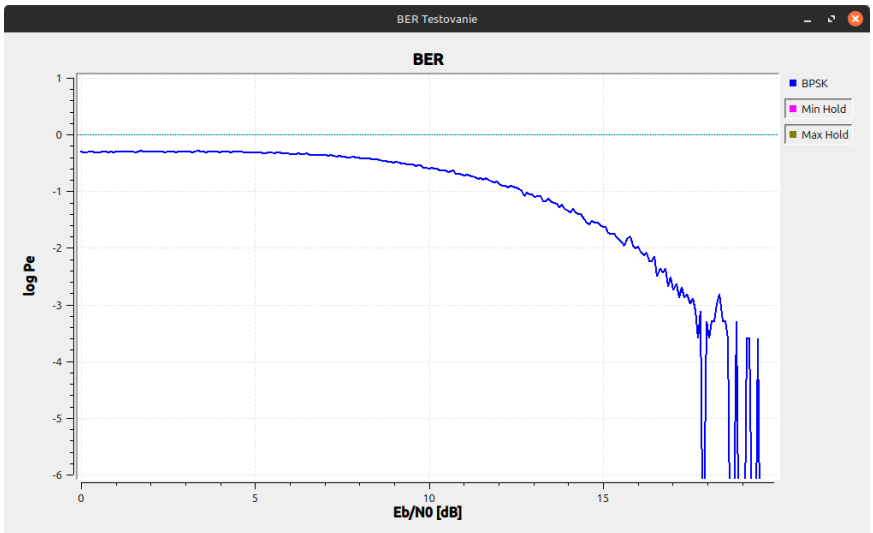
Obr. 5: BER pre BPSK



Obr. 6: BER pre QPSK



Obr. 7: BER pre 8PSK



Obr. 8: BER pre 16-QAM

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.

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" metodou som prešiel všetky bloky v GR a v dostupných zdrojových kodoch na GitHube pre bloky, ktoré mali rôzne vstupy/výstupy som "reverse engineer"-ol logiku.

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 ľubovolný mix dátových typov.

```
...

// Output vector
static int os[] = { sizeof(gr_complex), sizeof(int) };
static std::vector<int> osig(os, os + sizeof(os) / sizeof(int));

//The private constructor
AWGN_kanal_impl::AWGN_kanal_impl(int N, int EbN0min, int EbN0max)
  : gr::sync_block("AWGN_kanal",
                   gr::io_signature::make(1, 1, sizeof(gr_complex)),
                   gr::io_signature::makev(1, 2, osig))

...
```

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

```
...

//-----------------------Inicializacia-I/O-------------------------|
  // INPUT
  gr_complex *in0 = (gr_complex *) input_items[0];

  // OUTPUT
  gr_complex *out0 = (gr_complex *) output_items[0];
  int *out1 = (int *) output_items[1];

...
```

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

Zmena v header súbore

Keď súbor "*gr-modul/include/gnuradio/modul/blok.h*" akokolvek upravíme, nemôže hneď skompilovať projekt. Ak sa o to pokúsime, dostaneme chybovú hlášku.

Na toto je relatívne jednoduché ale pracné riešenie. Najprv si zmeníme verziu *GCC* a *G++* z 13 na 11, prípadne ak ju nemáme tak si ju aj nainštalujeme.

```
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/gcc gcc /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 vratiť do pôvodnej verzii, t.j. 13-tej.

```
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.

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.

```
...
 private:
     int _N;

     // Deklarovanie vektorov
     std::vector<int> _count;
     std::vector<double> _pamat_BER;
     std::vector<int> _pocet_chyb;
...
```

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

```
...
 _N = N; // Pocet vzoriek EbN0 (kolko bodov na X-osi)

 // Vektory
 _count = std::vector<int>(N, 1);        // Celkovy pocet bitov co sme spracovali
 _pocet_chyb = std::vector<int>(N, 0);     // Celkovy pocet zistenych chyb
 _pamat_BER = std::vector<double>(N, 1.0); // Vystupne hodnoty BER

...
```

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 lahšia a môžem si pri nich viac dovoliť.

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 nalepšie robia kombinovane.

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

AWGN kanál

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

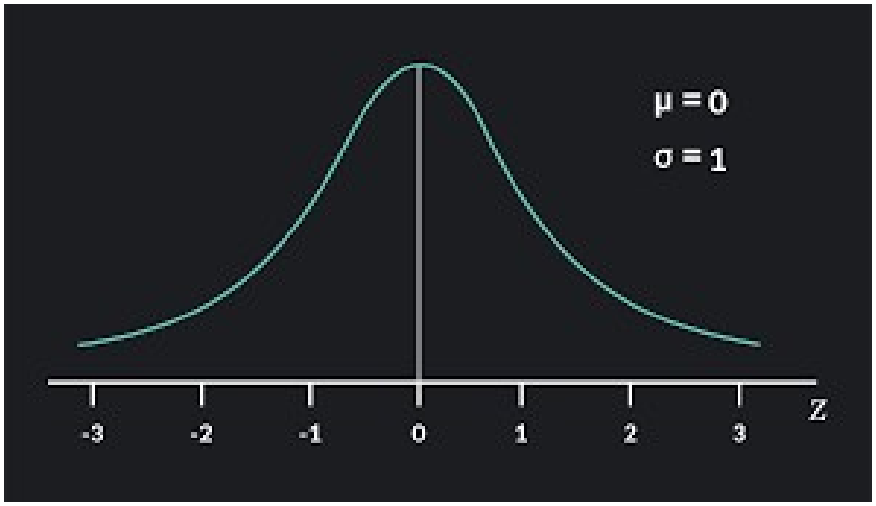
Začal by som asi s pomocnými funkciami. Prvá funkcia slúži na premenu SNR z decibelov na bezrozmerný pomer

$$EbN0 = 10^{\frac{EDB}{10}}$$

a získanie náhodnej vzorky z normáloveho rozdelenia, ktoré potom vloží do komplexnej premennej. Výstupný šum vypočítame ako

$$\text{šum} = \frac{SumVypocet}{\sqrt{2 \cdot EbN0}}$$

čitatel je len náhodná vzorka zo štandardnej normálovej distribúcie:



menovatel je rovnaky pre obe časti komplexného čísla ale čitatel zavolá znova funkciu na náhodnú vzorku.

```
...

 //-------------------Odvodenie-varianci-esumu-z-EbN0-[db]---------------------|
 gr_complex Sum(float EDB) {
      double EbN0, N;
      double REAL, IMAG;

      // Premena z dB na pomer
      EbN0 = pow(10.0, EDB/10.0);

      double menovatel = sqrt(2 * EbN0);

      REAL = Sum_vypocet() / menovatel;
      IMAG = Sum_vypocet() / menovatel;

      gr_complex n(REAL, IMAG);

      return n;
 }

...
```

Táto funkcia nie je moc zaujímava, keďže je to podľa vzoru na dokumentačnej stránke C++.

```
...

  //-------------------Tvorba-Gaussovky-a-random-bodu----------------------|
  double Sum_vypocet() {
      double GR;

      // Generovanie nahodneho cisla podla seed
      std::random_device rd;
      std::mt19937 R(rd());

      // Tvorba Gaussovky podla odchylky
      std::normal_distribution<double> Gauss{0, 1}; //mi = 0 lebo AWGN

      // Vyberieme nahodne hodnoty z Gaussovky
      GR = Gauss(R);

      return GR;
  }
...
```

Nasleduje funckia "*work*", veľmi rád by som to prerobil ešte, ale zatial to funguje tak sa toho nedotýkam.

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

$$rozpätia = \frac{EbN0max - EbN0min}{N-1}$$

Do vektora *EDB* sa postupne potom kumulatívne pridáva SNR hodnota. Začne sa s *EbN0min* a za každou iteráciou sa pripočíta rozpätie, ktoré som si vypočítal. Toto sa robí až po *EbN0max*.

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

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

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

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

```
...
   // INPUT
   gr_complex *in0 = (gr_complex *) input_items[0];

   // OUTPUT
   gr_complex *out0 = (gr_complex *) output_items[0];
   int *out1 = (int *) output_items[1];

   //----------------------LOGIKA-------------------------|
   //------------------Prvotne-vypocty--------------------|
   float EDB[_N];

   // Linearne rozlozenie EbN0db bodov
   rozpatie = float((_EbN0max - _EbN0min)) / float((_N-1));
   rozpatiePostup = float(_EbN0min);

   for(int i = 0; i < _N; i++) {
     EDB[i] = rozpatiePostup;
     rozpatiePostup += rozpatie;
   }

   //STARE
   // Vypocet vykonu vstupneho signalu Ps = E(x)
   /*for(int a = 0; a < noutput_items; a++)
     sumPs += pow(abs(in0[a]), 2);

   Ps = sumPs / float(noutput_items);
   */

   //----------------------Prejdeme-vsetkymi-I/O-items-------------------------|
   for(int b = 0; b < noutput_items; b++) {

     // Ziskanie komplexneho sumu
     gr_complex sg_n = Sum(EDB[k]);

     // Tuto by sa malo scitat ale C++ neznasa komplexne cisla, alebo mna...
     //gr_complex spolu(in0[b].real() + sg_n.real(), in0[b].imag() + sg_n.imag());

     // Komplexny vystup = AWGN sum
     out0[b] = sg_n;

     // Int vystup = N-ta vzorka EbN0, ktoru sme pocitali
     out1[b] = k;

     // Iterujeme po vsetkych vzorkach EbN0, t.j. od 0 do N-1
     if(k < _N-1) {
       k += 1;
     }else {
       k = 0;
     }
   }
...
```

BER blok

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

Začal by som s inicializáciou vektorov a premenných. Premenná *N* je len veľkosť vektorov. Teda pre koľko bodov SNR budeme počítať.

Vektor *_count* pozoruje a zapisuje koľko bitov prešlo pre každý SNR bod. Toto by sa podľa mna dalo upraviť na jednu premennú namiesto vektora, lebo v podstate cez všetky SNR 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 ale to nám pri logaritme neprejde. GR QT GUI prvky totiž nedokážu zobraziť mínus nekonečno, a kvôli tomu ďalej v kode toto ošetrujem aby sa priradila radšej malá avšak konečná hodnota.

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.

```
...
 _N = N; // Pocet vzoriek EbN0 (kolko bodov na X-osi)

 // Vektory
 _count = std::vector<int>(N, 1);        // Celkovy pocet bitov co sme spracovali
 _pocet_chyb = std::vector<int>(N, 0);     // Celkovy pocet zistenych chyb
 _pamat_BER = std::vector<double>(N, 1.0); // Vystupne hodnoty BER

...
```

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

Prvá časť je dosť okomentovaná ale z toho dôvodu, že som experimentoval s logaritmickým výstupom, tým pádom ušetríme jeden blok vo flowgraphe. 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, vratíme späť do "*work*" funkcie.

```
...

 //-------------------Zisti-ci-nastala-chyba---------------------|
 int BER_calc(unsigned char S1, unsigned char S2) {
      unsigned char XORnute;
      int e_sum;

      // Logaritmicke premenne
      //float BER, log_BER;

      // XOR-neme slovo zo signalu 1 a slovo zo signalu 2, vieme ze je chyba, ale
         nevieme kde a kolko chyb
      XORnute = S1 ^ S2;

      // Ak hej, tak nie je nulova hodnota e_sum
      e_sum = BitCounter(XORnute, 1);

      // Logaritmicky vystup
      //log_BER = logf(e_sum);

      //return log_BER;*/

      return e_sum;
 }
...
```

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. Cely 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 vratí sumu chýb.

```
...

 //--------------------Zisti-kolko-chyb-je-v-slove---------------------|
 int BitCounter(unsigned char slovo, int N) {
      int sum = 0, sign = 0;
      unsigned char b_jedna = 1;

      // Ci hodnota slova je zaporna
      if(slovo < 0)
        sign = 1;

      for(int i = 0; i < N; i++) {
             if((slovo & b_jedna) > 0)
               sum++;

             if(sign) { //ked plus
               slovo >>= 1;
             }else { //ked minus
               b_jedna <<= 1;
             }
      }

      return sum;
 }
...
```

Teraz ideme na funkciu "*work*". Prvý vstup pochádza z AWGN kanála a je to bod s určitou SNR 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 SNR bode, ktorú určuje prvý vstup zapísaný v premennej *k*.

Ako som spomínal, GR nedokáže vypísať nekonečnú hodnotu. Preto ak nenastala chyba, nastaví sa dostatočne malá hodnota ale konečná, ide o to aby logaritmus vedelo spracovať hodnotu. Ak počet chýb je nenulová tak sa ide podla vzorca:

$$Pb = \frac{PočetChýb}{CelkovýPočetBitov}$$

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

Na jediný výstup ide vypočítaná pravdepodnosť chyby na bit (BER) ako desatinné čislo (double) do logaritmického bloku.

Na koniec sa ešte inkrementuje počet prejdenných bitov cez blok o jedna.

```
...

//----------------------Inicializacia-I/O-------------------------|
   // INPUT
   int *in0 = (int *)input_items[0];
   unsigned char *in1 = (unsigned char *)input_items[1];
   unsigned char *in2 = (unsigned char *)input_items[2];

   // OUTPUT
   auto out = static_cast<output_type*>(output_items[0]);

   //----------------------LOGIKA-------------------------|
   //-------------------Prvotne-vypocty--------------------|

   /*long long int pocet_chyb[_N], count[_N];
   float pamat_SER[_N];
   std::fill_n(pocet_chyb, _N, 0);
   std::fill_n(count, _N, 8);
   std::fill_n(pamat_SER, _N, 1.0);*/

   // Bod v SNR
   int k;

   //----------------------Prejdeme-vsetkymi-I/O-items-------------------------|
   for(int i = 0; i < noutput_items; i++) {
     k = in0[i];

     // Zistime, ci nastala chyba v slove
     _pocet_chyb.at(k) += BER_calc(in1[i], in2[i]);

     // Nastavenia pre logaritmus v pripade ze nenastala chyba
     // GR nedokaze zobrazit v QT GUI nekonecna
     if(_pocet_chyb.at(k) == 0) {
       _pamat_BER.at(k) = 0.00000001; // log cisla je -8
     }else {
       _pamat_BER.at(k) = double(_pocet_chyb.at(k)) / double(_count.at(k)); //
Tuto nastala chyba, nemenime umelo hodnotu, pocet chyb deleno pocet bitov za celu
dobu
     }

    /* if(k%10000 == 0)
       GR_LOG_INFO(d_logger, std::string("b: ") + std::to_string(in2[i]) +
std::string("\n"));
     */

     // Float vystup = pravdepodobnost chyby na bit v danom Eb/N0
     out[i] = _pamat_BER.at(k);

     // Inkrementujeme celkovy pocet bitov
     _count.at(k) += 1;
   }
...
```

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, tak aj možno FER blok. 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 zapnuť.

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 SNR rozpätie počas behu programu (callback).

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

Referencie

GR:
https://wiki.gnuradio.org/index.php?title=Tutorials
https://wiki.gnuradio.org/index.php?title=LinuxInstall
https://wiki.gnuradio.org/index.php?title=Simulation_example:_BPSK_Demodulation
https://wiki.gnuradio.org/index.php?title=Legacy_Logging

C++:
https://www.pcg-random.org/using-pcg-cpp.html
https://cplusplus.com/reference/random/
https://en.cppreference.com/w/cpp/container/vector.html
https://en.cppreference.com/w/cpp/numeric/random/normal_distribution.html
https://en.cppreference.com/w/cpp/numeric/complex.html
/fll_band_edge_cc_impl.cc

GitHub:
https://github.com/rwth-ti/gr-ofdm
https://github.com/hortegab/comdig_su_software_libro3.8?tab=readme-ov-file
https://github.com/gnuradio/gnuradio/blob/main/gr-digital/lib

Teoria:
https://dsplog.com/2007/08/05/bit-error-probability-for-bpsk-modulation/

Rôzne:
https://mattgibson.ca/compiling-downgrading-to-gnuradio-3-7-x-on-ubuntu-20-x/
https://webhostinggeeks.com/howto/how-to-downgrade-gcc-version-on-ubuntu/
https://www.ascii-code.com/

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

Report z extra zadania z predmetu IPV6
https://github.com/Aymo19/ExtraZadanie_IPv6