Contents

1 Source files

1.1 codage.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

#include "../lib/BitArray/bit_array.h"
#include "../lib/BitArray/bar.h"

#include "turbocode.h"
#include "ldpc.h"
#include "list.h"
#include "random.h"
#include "basic.h"
#include "demo.h"


int main(int argc, char *argv[])
{
    // We only accept at most three arguments
    if (argc == 1 || argc > 4)
    {
        return 1;
    }

    int err = 0;
    int demos = 0;
    double s = 0.0;

    // Incorrect type for an argument
    sscanf(argv[1], "%d", &demos);
    sscanf(argv[3], "%lf", &s);

    char *file = argv[2];

    // Initialise the random generator
    srand((unsigned int)time(NULL));

    if (demos & 1)
    {
        demo_turbo_iter(file, s, 10);
    }
    if (demos & 2)
    {
        demo_ldpc_basic(file, s, 500);
    }
    if (demos & 4)
    {
        demo_ldpc_proba(file, s, 10);
    }
    if (demos & 8)
    {
        demo_turbo_graph(100, s);
    }
    if (demos & 16)
    {
        demo_ldpc_graph(100, s);
    }
    if (demos & 32)
    {
```

```
        demo_base_graph(100, s);
    }

    return 0;
}
```

1.2 image.py

```
import os
import matplotlib.pyplot as plt

from PIL import Image


def image_to_bytes(file_name):
    cwd = os.getcwd()
    file_path = os.path.join(cwd, '..', 'files', 'images', file_name) + '.png'
    image = Image.open(file_path)

    data = list(image.getdata(band=0))  # The image is in grayscale

    byte_path = os.path.join(cwd, '..', 'files', 'bytes', file_name) + '.bt'
    try:
        byte_file = open(byte_path, 'xb')
    except FileExistsError:
        byte_file = open(byte_path, 'wb')

    for x in data:
        byte_file.write(x.to_bytes(1, 'big'))

    byte_file.close()


def bytes_to_image(file_name, size):
    cwd = os.getcwd()
    byte_path = os.path.join(cwd, '..', 'files', 'bytes', file_name) + '.bt'
    byte_file = open(byte_path, 'rb')

    image = Image.frombytes('L', size, byte_file.read(), "raw")
    file_path = os.path.join(cwd, '..', 'files', 'images', file_name) + '.png'
    try:
        file = open(file_path, 'xb')
    except FileExistsError:
        file = open(file_path, 'wb')

    image.save(file)
    file.close()
    image.close()


def convert_all(file_name, size):
    bytes_to_image(file_name + '_n', size)
    bytes_to_image(file_name + '_d', size)


def graph_ber():
    cwd = os.getcwd()
    t_path = os.path.join(cwd, '../files/graph/turbo.grp')
    l_path = os.path.join(cwd, '../files/graph/ldpc.grp')
    b_path = os.path.join(cwd, '../files/graph/base.grp')

    t_file = open(t_path, 'r')
    l_file = open(l_path, 'r')
    b_file = open(b_path, 'r')

```

```
    t_snr, t_err = [], []
    l_snr, l_err = [], []
    b_snr, b_err = [], []

    for line in t_file:
        data = line.strip('\n').split(' ')
        t_snr.append(float(data[0]))
        t_err.append(100*float(data[1]))

    for line in l_file:
        data = line.strip('\n').split(' ')
        l_snr.append(float(data[0]))
        l_err.append(100*float(data[1]))

    for line in b_file:
        data = line.strip('\n').split(' ')
        b_snr.append(float(data[0]))
        b_err.append(100*float(data[1]))

    t_file.close()
    l_file.close()
    b_file.close()

    plt.plot(t_snr, t_err, label='Turbocode')
    plt.plot(l_snr, l_err, label='Code LDPC (Hard Decoding)')
    plt.plot(b_snr, b_err, label='Sans codage')

    plt.xlabel("$\\frac{E_b}{N_0}$ (dB)")
    plt.ylabel("Taux d'erreur (%)")

    plt.legend()
    plt.show()
```

1.3 basic.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <math.h>

#include "basic.h"
#include "random.h"


// Decode the s_list by comparing its values to 0
h_list * decode_h_basic(s_list *mes)
{
    h_list *res = chl(mes->n, mes->n);
    for (size_t i = 0; i < mes->n; i++)
    {
        res->list[i] = (mes->list[i] > 0);
    }
    return res;
}


// Add a white noise with variance s^2 to the map of mes where
// 0 -> -1
// 1 -> 1
s_list * add_noise(h_list *mes, double s)
{
    s_list *res = csl(mes->n, mes->n);
    for (size_t i = 0; i < res->n; i++)
    {
        res->list[i] = (2.0 * mes->list[i]) - 1.0 + box_muller(0.0, s);
    }
    return res;
}


// Computes the number of differences between og_mes and mes
int nb_errors(h_list *og_mes, h_list *mes)
{
    if (og_mes->n != mes->n)
    {
        return -1;
    }

    size_t d = 0;
    for (size_t i = 0; i < mes->n; i++)
    {
        d += (og_mes->list[i] != mes->list[i]);
    }
    return d;
}


// Returns the size of a file *fp
size_t file_size(FILE *fp)
{
    fseek(fp, 0, SEEK_END);
    size_t size = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    return size;
}
```

```
63
64
65 // Returns f(beta) as defined in Gallager's work
66 double f(double b)
67 {
68     return log((exp(-b) + 1.0) / (1.0 - exp(-b)));
69 }
```

1.4 list.c

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

#include "list.h"
#include "basic.h"


h_list * create_h_list(size_t n, size_t m_s)
{
    h_list *res = malloc(sizeof(h_list));
    res->n = n;
    res->m_s = m_s;
    res->list = calloc(m_s, sizeof(char));

    return res;
}


i_list * create_i_list(size_t n, size_t m_s)
{
    i_list *res = malloc(sizeof(i_list));
    res->n = n;
    res->m_s = m_s;
    res->list = calloc(m_s, sizeof(int));

    return res;
}


s_list * create_s_list(size_t n, size_t m_s)
{
    s_list *res = malloc(sizeof(s_list));
    res->n = n;
    res->m_s = m_s;
    res->list = calloc(m_s, sizeof(double));

    return res;
}


h_matrix * create_h_matrix(size_t n, size_t m)
{
    h_matrix *res = malloc(sizeof(h_matrix));
    res->n = n;
    res->m = m;
    res->mat = calloc(n * m, sizeof(char));

    return res;
}


a_matrix * create_a_matrix(size_t n, size_t m)
{
    a_matrix *res = malloc(sizeof(a_matrix));
    res-> n = n;
    res->m = m;
    res->list_m = malloc(m * sizeof(i_list *));
    res->list_n = malloc(n * sizeof(i_list *));

    return res;
}
```

```
63
64
65 hh_list * create_hh_list(size_t n, size_t m_e)
66 {
67     hh_list *res = malloc(sizeof(hh_list));
68     res->n = n;
69     res->m_e = m_e;
70     res->list = malloc(n * sizeof(h_list *));
71
72     return res;
73 }
74
75
76 char get_h_list(h_list *list_h, size_t i)
77 {
78     return list_h->list[i];
79 }
80
81
82 double get_s_list(s_list *list_s, size_t i)
83 {
84     return list_s->list[i];
85 }
86
87
88 char get_h_matrix(h_matrix *mat_h, size_t i, size_t j)
89 {
90     return mat_h->mat[(i * (mat_h->m)) + j];
91 }
92
93
94 void set_h_list(h_list *list_h, char x, size_t i)
95 {
96     list_h->list[i] = x;
97 }
98
99
100 void set_s_list(s_list *list_s, double x, size_t i)
101 {
102     list_s->list[i] = x;
103 }
104
105
106 void set_h_matrix(h_matrix *mat_h, char x, size_t i, size_t j)
107 {
108     mat_h->mat[i * (mat_h->m) + j] = x;
109 }
110
111
112 void write_char_h(h_list *list_h, char x, size_t p)
113 {
114     unsigned char t;
115     if (p < (list_h->n - 7))
116     {
117         t = x;
118         for (size_t i = 0; i < 8; i++)
119         {
120             list_h->list[p + i] = t % 2;
121             t = t / 2;
122         }
123     }
124 }
125
126
```

```
127 char read_char_h(h_list *list_h, size_t p)
128 {
129     if (p < (list_h->n - 7))
130     {
131         unsigned char x = 0;
132         for (size_t i = 8; i > 0; i--)
133         {
134             x *= 2;
135             x += list_h->list[p + i - 1];
136         }
137         return x;
138     }
139 }
140
141
142 void write_bit_h(h_list *list_h, unsigned char x, size_t p)
143 {
144     if (p < list_h->n)
145     {
146         list_h->list[p] = x % 2;
147     }
148 }
149
150
151 char read_bit_h(h_list *list_h, size_t p)
152 {
153     if (p < list_h->n)
154     {
155         return 255 * list_h->list[p];
156     }
157 }
158
159
160 void set_all_h_list(h_list *list_h, char x)
161 {
162     for (size_t i = 0; i < list_h->n; i++)
163     {
164         list_h->list[i] = x;
165     }
166 }
167
168
169 void set_all_i_list(i_list *list_i, int x)
170 {
171     for (size_t i = 0; i < list_i->n; i++)
172     {
173         list_i->list[i] = x;
174     }
175 }
176
177
178 void set_all_s_list(s_list *list_s, double x)
179 {
180     for (size_t i = 0; i < list_s->n; i++)
181     {
182         list_s->list[i] = x;
183     }
184 }
185
186
187 // Determines if the h_list is only 0
188 char is_all_nil(h_list *list_h)
189 {
190     char ok = 1;
```

```
191     for (size_t i = 0; i < list_h->n; i++)
192     {
193         if (list_h->list[i])
194         {
195             ok = 0;
196         }
197     }
198     return ok;
199 }
200 
201 
202 double min_s(s_list *list_s)
203 {
204     double s = INFINITY;
205 
206     for (size_t i = 0; i < list_s->n; i++)
207     {
208         if (fabs(list_s->list[i]) < s)
209         {
210             s = fabs(list_s->list[i]);
211         }
212     }
213 
214     return s;
215 }
216 
217 
218 double max_s(s_list *list_s)
219 {
220     double s = 0;
221 
222     for (size_t i = 0; i < list_s->n; i++)
223     {
224         if (fabs(list_s->list[i]) > s)
225         {
226             s = fabs(list_s->list[i]);
227         }
228     }
229 
230     return s;
231 }
232 
233 
234 int append_i(i_list *list_i, int x)
235 {
236     if (list_i->n == list_i->m_s)
237     {
238         // The list is maxed out
239         return 1;
240     }
241 
242     list_i->list[list_i->n] = x;
243     list_i->n ++;
244 
245     return 0;
246 }
247 
248 
249 // Shift the list l elements to the right
250 int shift_i(i_list *list_i, int l)
251 {
252     if (0 >= list_i->n + l >= list_i->m_s)
253     {
254         return 1;
```

```
255     }
256
257     list_i->n += l;
258     for (size_t i = (list_i->n - 1); i >=0; i--)
259     {
260         list_i->list[i + list_i->n] = list_i->list[i];
261     }
262     return 0;
263 }
264
265
266 int substract_s(s_list *list_a, s_list *list_b)
267 {
268     if (list_a->n != list_b->n)
269     {
270         return 1;
271     }
272
273     for (size_t i = 0; i < list_a->n; i++)
274     {
275         list_a->list[i] -= list_b->list[i];
276     }
277     return 0;
278 }
279
280
281 // Copy n elements of list_a into list_b if possible
282 int copy_h(h_list *list_a, h_list *list_b, size_t n, size_t s)
283 {
284     if (n > list_b->m_s || n * s >list_a->n)
285     {
286         return 1;
287     }
288
289     list_b->n = n;
290     for (size_t i = 0; i < n; i++)
291     {
292         list_b->list[i] = list_a->list[i * s];
293     }
294     return 0;
295 }
296
297
298 hh_list *deep_copy(hh_list *list_hh)
299 {
300     hh_list *res = chhl(list_hh->n, list_hh->m_e);
301     for (size_t i = 0; i < res->n; i++)
302     {
303         res->list[i] = chl(list_hh->list[i]->n, list_hh->list[i]->m_s);
304         copy_h(list_hh->list[i], res->list[i], res->list[i]->n, 1);
305     }
306     return res;
307 }
308
309
310 void max_i_list(i_list *list_i, i_list *res)
311 {
312     int m = list_i->list[0];
313     res->n = 0;
314
315     for (size_t i = 0; i < list_i->n; i++)
316     {
317         if (list_i->list[i] == m)
318         {
```

```
319             append_i(res, i);
320         }
321 
322         if (list_i->list[i] > m)
323         {
324             m = list_i->list[i];
325             res->n = 0;
326             append_i(res, i);
327         }
328     }
329 }
330 
331 
332 h_list * product_h(h_matrix *mat, h_list *vect)
333 {
334     if (mat->m != vect->n)
335     {
336         return NULL;
337     }
338 
339     h_list *res = chl(mat->n, mat->n);
340     for (size_t k = 0; k < mat->m; k++)
341     {
342         for (size_t i = 0; i < mat->n; i++)
343         {
344             if (ghm(mat, i, k) && ghl(vect, k))
345             {
346                 res->list[i] = 1 - res->list[k];
347             }
348         }
349     }
350     return res;
351 }
352 
353 
354 h_list * product_a(a_matrix *mat, h_list *vect)
355 {
356     if (mat->m != vect->n)
357     {
358         return NULL;
359     }
360 
361     h_list *res = chl(mat->n, mat->n);
362     for (size_t i = 0; i < mat->n; i++)
363     {
364         for (size_t k = 0; k < mat->list_n[i]->n; k++)
365         {
366             if (vect->list[mat->list_n[i]->list[k]])
367             {
368                 res->list[i] ^= 1;
369             }
370         }
371     }
372     return res;
373 }
374 
375 
376 int product_a_in_place(a_matrix *mat, h_list *vect, h_list *res)
377 {
378     if (mat->m != vect->n || mat->n > res->m_s)
379     {
380         return 1;
381     }
382 
```

```
383     res->n = mat->n;
384     set_all_h_list(res, 0);
385 
386     for (size_t i = 0; i < mat->n; i++)
387     {
388         for (size_t k = 0; k < mat->list_n[i]->n; k++)
389         {
390             if (vect->list[mat->list_n[i]->list[k]])
391             {
392                 res->list[i] ^= 1;
393             }
394         }
395     }
396     return 0;
397 }
398 
399 
400 int product_s_ip(a_matrix *mat, s_list *vect, s_list *res)
401 {
402     if (mat->m != vect->n || mat->n > res->m_s)
403     {
404         return 1;
405     }
406 
407     res->n = mat->n;
408     set_all_s_list(res, 0.0);
409 
410     for (size_t i = 0; i < mat->n; i++)
411     {
412         for (size_t k = 0; k < mat->list_n[i]->n; k++)
413         {
414             res->list[i] += vect->list[mat->list_n[i]->list[k]];
415         }
416     }
417     return 0;
418 }
419 
420 
421 // If dir = 0 -> [A | I]
422 // If dir = 1 -> [I
423 //                A]
424 h_matrix * juxtapose_h(h_matrix *mat, char dir)
425 {
426     int n = mat->n;
427     int m = mat->m;
428     h_matrix *res;
429 
430     if (dir)
431     {
432         res = chm(n + m, m);
433         // Copy mat into res and add identity
434         for (size_t j = 0; j < mat->m; j++)
435         {
436             shm(res, 1, j, j);
437             for (size_t i = 0; i < mat->n; i++)
438             {
439                 shm(res, ghm(mat, i, j), i + m, j);
440             }
441         }
442     }
443     else
444     {
445         res = chm(n, m + n);
446         // Copy mat into res and add identity
```

```
        for (size_t i = 0; i < mat->n; i++)
        {
            shm(res, 1, i, i + m);
            for (size_t j = 0; j < mat->m; j++)
            {
                shm(res, ghm(mat, i, j), i, j);
            }
        }
    }
    return res;
}


// If dir = 0 -> [A | I]
// If dir = 1 -> [I
//                A]
a_matrix * juxtapose_a(h_matrix *mat, char dir)
{
    int n = mat->n;
    int m = mat->m;
    a_matrix *tmp = convert_h(mat);
    a_matrix *res;

    if (dir)
    {
        res = cam(n + m, m);

        // Copy tmp into res and add identity
        for (size_t i = 0; i < mat->m; i++)
        {
            res->list_n[i] = cil(1, 1);
            res->list_n[i]->list[0] = i;

            res->list_m[i] = cil(tmp->list_m[i]->n + 1, tmp->list_m[i]->n + 1);
            res->list_m[i]->list[0] = i;
            for (size_t k = 1; k < res->list_m[i]->n; k++)
            {
                res->list_m[i]->list[k] = tmp->list_m[i]->list[k -1] + m;
            }
        }

        for (size_t i = 0; i < mat->n; i++)
        {
            res->list_n[i + m] = cil(tmp->list_n[i]->n, tmp->list_n[i]->n);
            for (size_t k = 0; k < res->list_n[i + m]->n; k++)
            {
                res->list_n[i + m]->list[k] = tmp->list_n[i]->list[k];
            }
        }
    }
    else
    {
        res = cam(n, m + n);

        // Copy tmp into res and add identity
        for (size_t i = 0; i < mat->n; i++)
        {
            res->list_m[i + m] = cil(1, 1);
            res->list_m[i + m]->list[0] = i;

            res->list_n[i] = cil(tmp->list_n[i]->n, tmp->list_n[i]->n + 1);
            for (size_t k = 0; k < res->list_n[i]->n; k++)
            {
                res->list_n[i]->list[k] = tmp->list_n[i]->list[k];
```

```
            }
            append_i(res->list_n[i], m + i);
        }

        for (size_t i = 0; i < mat->m; i++)
        {
            res->list_m[i] = cil(tmp->list_m[i]->n, tmp->list_m[i]->n);
            for (size_t k = 0; k < res->list_m[i]->n; k++)
            {
                res->list_m[i]->list[k] = tmp->list_m[i]->list[k];
            }
        }
    }
    free_a_matrix(tmp);
    return res;
}


a_matrix * convert_h(h_matrix *mat)
{
    a_matrix *res = cam(mat->n, mat->m);

    // Initialise list_m and list_n
    for (size_t i = 0; i < mat->m; i++)
    {
        res->list_m[i] = cil(0, mat->n);
    }

    for (size_t i = 0; i < mat->n; i++)
    {
        res->list_n[i] = cil(0, mat->m);
    }

    // Fill the matrix
    for (size_t i = 0; i < mat->n; i++)
    {
        for (size_t j = 0; j < mat->m; j++)
        {
            if (ghm(mat, i, j))
            {
                append_i(res->list_m[j], i);
                append_i(res->list_n[i], j);
            }
        }
    }
    return res;
}


hh_list * read_file_h(FILE *fp, size_t n)
{
    if (n % 8)
    {
        return NULL;
    }

    size_t s = file_size(fp);
    size_t m = (s % n) ? (s / n) + 1 : (s / n); // Size in bytes
    m *= 8;                                     // Size in bits

    hh_list *res = chhl(m, s);
    for (size_t i = 0; i < m; i++)
    {
        res->list[i] = chl(n, n);
```

```
575     }
576
577     unsigned char *buffer = malloc(s * sizeof(char));
578     fread(buffer, sizeof(char), s, fp);
579
580     for (size_t i = 0; i < s; i++)
581     {
582         write_char_h(res->list[8 * i / n], buffer[i], (8 * i) % n);
583     }
584
585     return res;
586 }
587
588
589 int write_file_h(FILE *fp, hh_list *list_hh)
590 {
591     size_t c = 0;
592     unsigned char x;
593
594     for (size_t i = 0; i < list_hh->n; i++)
595     {
596         if (list_hh->list[i]->n % 8)
597         {
598             return 1;
599         }
600
601         for (size_t j = 0; j < (list_hh->list[i]->n / 8); j++)
602         {
603             if (c < list_hh->m_e)
604             {
605                 x = read_char_h(list_hh->list[i], 8 * j);
606                 fwrite(&x, sizeof(char), 1, fp);
607             }
608             c ++;
609         }
610     }
611     return 0;
612 }
613
614
615 hh_list * read_bit_file_h(FILE *fp, size_t n)
616 {
617
618     size_t s = file_size(fp);
619     size_t m = (s % n) ? (s / n) + 1 : (s / n); // Number of lists to create
620
621     hh_list *res = chhl(m, s);
622     for (size_t i = 0; i < m; i++)
623     {
624         res->list[i] = chl(n, n);
625     }
626
627     unsigned char *buffer = malloc(s * sizeof(char));
628     fread(buffer, sizeof(char), s, fp);
629
630     for (size_t i = 0; i < s; i++)
631     {
632         write_bit_h(res->list[i / n], buffer[i], i % n);
633     }
634
635     return res;
636 }
637
638
```

```
int write_bit_file_h(FILE *fp, hh_list *list_hh)
{
    size_t c = 0;
    unsigned char x;

    for (size_t i = 0; i < list_hh->n; i++)
    {
        for (size_t j = 0; j < (list_hh->list[i]->n); j++)
        {
            if (c < list_hh->m_e)
            {
                x = read_bit_h(list_hh->list[i], j);
                fwrite(&x, sizeof(char), 1, fp);
            }
            c ++;
        }
    }
    return 0;
}


void print_h_list(h_list *list_h)
{
    printf("[");
    for (size_t i = 0; (i + 1) < list_h->n; i++)
    {
        printf("%d, ", list_h->list[i]);
    }
    printf("%d]\n", list_h->list[list_h->n - 1]);
}


void print_i_list(i_list *list_i)
{
    int n = list_i->n;
    printf("[");
    for (size_t i = 0; i < (n - 1); i++)
    {
        printf("%d, ", (list_i->list[i]));
    }
    printf("%d]\n", list_i->list[list_i->n - 1]);
}


void print_s_list(s_list *list_s)
{
    printf("[");
    for (size_t i = 0; (i + 1) < list_s->n; i++)
    {
        printf("%f, ", list_s->list[i]);
    }
    printf("%f]\n", list_s->list[list_s->n - 1]);
}


void print_h_matrix(h_matrix *mat_h)
{
    int n = mat_h->n;
    int m = mat_h->m;

    printf("[");
    for (size_t i = 0; (i + 1) < n; i++)
    {
        printf("[");
```

```
703         for (size_t j = 0; (j + 1) < m; j++)
704         {
705             printf("%d, ", mat_h->mat[(i * m) + j]);
706         }
707         printf("%d],\n", mat_h->mat[(i * m) + m - 1]);
708     }
709 
710     printf("[");
711     for (size_t j = 0; (j + 1) < m; j++)
712     {
713         printf("%d, ", mat_h->mat[((n - 1) * m) + j]);
714     }
715     printf("%d]]\n", mat_h->mat[(n * m) - 1]);
716 }
717 
718 
719 void print_a_matrix(a_matrix *mat_a)
720 {
721     // Print the m vertical lists
722     for (size_t i = 0; i < mat_a->m; i++)
723     {
724         print_i_list(mat_a->list_m[i]);
725     }
726     printf("\n");
727 
728     // Print the n horizontal lists
729     for (size_t i = 0; i < mat_a->n; i++)
730     {
731         print_i_list(mat_a->list_n[i]);
732     }
733 }
734 
735 
736 void print_hh_list(hh_list *list_hh)
737 {
738     for (size_t i = 0; i < list_hh->n; i++)
739     {
740         print_h_list(list_hh->list[i]);
741     }
742 }
743 
744 
745 void free_h_list(h_list *list_h)
746 {
747     free(list_h->list);
748     free(list_h);
749 }
750 
751 
752 void free_i_list(i_list *list_i)
753 {
754     free(list_i->list);
755     free(list_i);
756 }
757 
758 
759 void free_s_list(s_list *list_s)
760 {
761     free(list_s->list);
762     free(list_s);
763 }
764 
765 
766 void free_h_matrix(h_matrix *mat_h)
```

```
767 {
768     free(mat_h->mat);
769     free(mat_h);
770 }
771 
772 
773 void free_a_matrix(a_matrix *mat_a)
774 {
775     for (size_t i = 0; i < mat_a->m; i++)
776     {
777         free_i_list(mat_a->list_m[i]);
778     }
779     free(mat_a->list_m);
780 
781     for (size_t i = 0; i < mat_a->n; i++)
782     {
783         free_i_list(mat_a->list_n[i]);
784     }
785     free(mat_a->list_n);
786 
787     free(mat_a);
788 }
789 
790 
791 void free_hh_list(hh_list *list_hh)
792 {
793     for (size_t i = 0; i < list_hh->n; i++)
794     {
795         free_h_list(list_hh->list[i]);
796     }
797     free(list_hh->list);
798 
799     free(list_hh);
800 }
```

1.5 random.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "bit_array.h"
#include "bar.h"

#include "random.h"
#include "list.h"


// Fills the h_list with random 0 and 1 with uniform proba p
void random_h(h_list *list_h, double p)
{
    for (size_t i = 0; i < list_h->n; i++)
    {
        list_h->list[i] = (rand() / (double) RAND_MAX) < p;
    }
}


// Returns a permutation of 0..n-1 using the Fisher-Yates shuffle
void permutation(i_list *list_i)
{
    int j;
    for (size_t i = 0; i < list_i->n; i++)
    {
        j = rand() % (i + 1);
        if (j != i)
        {
            list_i->list[i] = list_i->list[j];
        }
        list_i->list[j] = i;
    }
}


// Add a White Gaussian Noise with mean 0 and standard deviation s to the
// encoded data
void addNoise(bar *message, double s, double *noisy)
{
    size_t n = barlen(message);

    for(size_t i = 0; i < n; i++)
    {
        noisy[i] = (2 * barget(message, i) - 1.0) + box_muller(0.0, s);
    }
}


// Compute the transition probability of the channel
double pTransition(double x, char d, double s)
{
    char mu = 2 * d - 1; // -1 if d = 0, 1 if d = 1
    double res;

    if (isfinite(x))
    {
        if (s == 0)
        {
            return 1.0 * (x == mu);
```

```
63          }
64
65          if (fabs(x) / s > 26)   // If |x| is too big, the probability returns 0
66                                  // which would be incorrect
67          {
68              return x * mu > 0;
69          }
70
71          return exp(- pow((x - mu) / s, 2) / 2.0) / (s * sqrt(2.0 * M_PI));
72      }
73
74      if (isnan(x))
75      {
76          return 0;
77      }
78
79      if (x == inf)
80      {
81          return mu > 0;
82      }
83
84      if (x == -inf)
85      {
86          return mu < 0;
87      }
88
89      return 0;
90  }
91
92
93  // P(x) with a normal distribution of mean m and variance s^2
94  double normal(double x, double m, double s)
95  {
96      return exp(- pow((x - m) / s, 2) / 2.0) / (s * sqrt(2.0 * M_PI));
97  }
98
99
100 // Compute the mean of an array
101 double mean(double *Z, size_t n)
102 {
103     double mu;
104     for(size_t i = 0; i < n; i++)
105     {
106         mu += Z[i];
107     }
108
109     return mu / (double) n;
110 }
111
112
113 // Compute the variance of an array
114 double variance(double *Z, size_t n)
115 {
116     double s;
117     double m = mean(Z, n);
118     for(size_t i = 0; i < n; i++)
119     {
120         s += pow((Z[i] - m), 2.0);
121     }
122
123     return s / (double) n;
124 }
125
126
```

```
// Create the pseudorandom generator
generator * initGenerator(void)
{
    bar *generator = barcreate(8);
    barfill(generator);

    return generator;
}


// Return a random value and update the memory state
char yield(generator *gen)
{
    char res = barget(gen, 0);
    char a = res;
    a = a ^ barget(gen, 3);
    a = a ^ barget(gen, 5);
    a = a ^ barget(gen, 7);
    barshr(gen, 1, a);

    return res;
}


// Generate a sequence of n random values
bar * sequence(generator *gen, size_t n)
{
    bar *res = barcreate(n);

    for(size_t i = 0; i < n; i++)
    {
        barmake(res, i, yield(gen));
    }

    return res;
}


// XOR a bit array with a random sequence
bar * combine(generator *gen, bar *message)
{
    resetGenerator(gen);
    int n = barlen(message);
    bar *seq = sequence(gen, n);
    bar *res = barcreate(n);
    barxor(res, seq, message);

    return res;
}


// Reset the memory state of the generator
void resetGenerator(generator *gen)
{
    barfill(gen);
}


void freeGenerator(generator *gen)
{
    bardestroy(gen);
}
```

1.6 turbocode.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <math.h>

#include "../lib/BitArray/bit_array.h"
#include "../lib/BitArray/bar.h"
#include "random.h"
#include "turbocode.h"
#include "list.h"
#include "tests.h"


const size_t p[] = {0, 31, 37, 43, 47, 53, 59, 61, 67};


// In the CCSDS Standard, bits are numbered from 1 but we really start at 0
// so we need to replace pi(s) by pi(s + 1) - 1
// Instead, we return pi(s) - 1 and call pi(k + 1)
size_t pi(size_t s)
{
    size_t m = s % 2;
    size_t i = s / (2 * k2);
    size_t j = (s / 2) - i * k2;
    size_t t = (19 * i + 1) % (k1 / 2);
    size_t q = (t % 8) + 1;
    size_t c = (p[q] * j + 21 * m) % k2;

    return 2 * (t + c * (k1 / 2) + 1) - m - 1;
}


// We use the connection vector G1 = 11011 for the message
// and G0 = 10011 for the component code
char yieldEncode(char d, bar *m)
{
    char a = d ^ barget(m, 0) ^ barget(m, 2) ^ barget(m, 3);
    char g = d ^ barget(m, 2) ^ barget(m, 3);

    barshl(m, 1, g);

    return a;
}


bar * initMemState(size_t n)
{
    return barcreate(n);
}


h_list * encode_turbo(h_list *buf)
{
    // We use the rate 1/3 for convenience
    h_list *buf_e = chl(3 * (buf->n + 4), 3 * (buf->n + 4));

    // Create the shift-registers
    bar *registerA = initMemState(4);
    bar *registerB = initMemState(4);

    char a;
```

```
63      char b;
64      char d;
65
66      // Encode the buffer
67      for(size_t i = 0; i < buf->n; i++)
68      {
69          d = buf->list[i];
70          buf_e->list[3 * i] = d;
71
72          a = yieldEncode(d, registerA);
73          buf_e->list[3 * i + 1] = a;
74
75          d = buf->list[pi(i)];
76          b = yieldEncode(d, registerB);
77          buf_e->list[3 * i + 2] = b;
78      }
79
80      // Clean the registers
81      for(size_t i = buf->n; i < (buf->n + 4); i++)
82      {
83          d = barget(registerA, 3) ^ barget(registerA, 2);
84          buf_e->list[3 * i] = d;
85
86          a = yieldEncode(d, registerA);
87          buf_e->list[3 * i + 1] = a;
88
89          d = barget(registerB, 3) ^ barget(registerB, 2);
90          b = yieldEncode(d, registerB);
91          buf_e->list[3 * i + 2] = b;
92      }
93
94      bardestroy(registerA);
95      bardestroy(registerB);
96
97      return buf_e;
98  }
99
100
101 // Computes the llr given the arrays X and Y
102 int decode_part(s_list *X, s_list *Y, s_list *llr, double s)
103 {
104     // Same as in decodeStream
105     if (X->n != Y->n || X->n != llr->n)
106     {
107         return 1;
108     }
109
110     size_t n = X->n; // The original message length plus the
111                              // padding bits at the end
112
113     s_list *alpha = csl(32*(n + 1), 32*(n + 1));    // alpha(i, k, m)
114                                                     // = alpha[i + 2*m + 32*k]
115
116     s_list *beta = csl(16*(n + 1), 16*(n + 1));     // beta(k, m)
117                                                     // = beta[m + 16*k]
118
119     s_list *gamma = csl(32*(n + 1), 32*(n + 1));    // gamma(i, R_k, m', m)
120                                                     // = gamma[i + 2*m + 32*k]
121                                                  // We only have one choice for
122                                                  // m' knowing i and m
123
124     s_list *lambda = csl(32*(n + 1), 32*(n + 1));   // lambda(i, k, m)
125                                                     // = lambda[i + 2*m + 32*k]
126
```

```
    s_list *a = csl(n + 1, n + 1);                // Used for normalization

    if (alpha == NULL || beta == NULL || gamma == NULL || lambda == NULL ||
        a == NULL)
    {
        return 1;
    }

    double tmp[2];

    size_t d; // The value of the k-th bit
    size_t b; // The value of the k-th encoded bit
    size_t m; // The previous state of the register
    size_t i;
    double x;
    double y;

    // Compute recursively alpha, gamma, beta and lambda for y1 and y2
    // For rate 1/3, the received bits are x_0, y1_0, y2_0, x_1, ...

    // Initialize alpha
    alpha->list[0] = 1.0;
    alpha->list[1] = 1.0;

    // Initialize the norm of alpha
    a->list[0] = 1.0;
    a->list[n] = 1.0;

    // Initialize beta
    beta->list[16*n] = 1.0;

    for(size_t k = 1; k <= n; k++) // k-th bit of the message
    {
        x = X->list[k - 1];
        y = Y->list[k - 1];

        for(size_t S = 0; S < 16; S++) // Register state of the encoder
        {
            // Knowing d_k = i and S_k = S,
                // m = S_{k-1} = S/2 + (S & 8) ^ 8*(i ^ (S & 1))
                // b = i ^ (m & 1) ^ (m & 4)/4 ^ (m & 8)/8

            // d_k = 0
            d = 0;
            m = S/2 + (S & 8) ^ 8*((S & 1) ^ d);
            b = d ^ (m & 1) ^ (m & 4)/4 ^ (m & 8)/8;
            gamma->list[2*S + 32*k] = pTrans(x, d, s) * pTrans(y, b, s);
            i = 2*m + 32*(k-1);
            alpha->list[2*S + 32*k] = gamma->list[2*S + 32*k] *
                                      (alpha->list[i] + alpha->list[1 + i]);

            // d_k = 1
            d = 1;
            m = S/2 + (S & 8) ^ 8*((S & 1) ^ d);
            b = d ^ (m & 1) ^ (m & 4)/4 ^ (m & 8)/8;
            gamma->list[1 + 2*S + 32*k] = pTrans(x, d, s) * pTrans(y, b, s);
            i = 2*m + 32*(k-1);
            alpha->list[1 + 2*S + 32*k] = gamma->list[1 + 2*S + 32*k] *
                                          (alpha->list[i] + alpha->list[1 + i]);
            a->list[k] += alpha->list[2*S + 32*k] + alpha->list[1 + 2*S + 32*k];
        }

        if (isnan(a->list[k]))
        {
```

```
            return 2;
        }

        for(size_t S = 0; S < 16; S++)
        {
            // Normalize alpha
            alpha->list[2*S + 32*k] /=  a->list[k];
            alpha->list[1 + 2*S + 32*k] /= a->list[k];
        }
    }

    // lambda(i, n, 0) = alpha(i, n, 0)
    // lambda(i, n, m) = 0 if m != 0
    lambda->list[32 * n] = alpha->list[32 * n];
    lambda->list[1 + 32 * n] = alpha->list[1 + 32 * n];

    for(int k = (n - 1); k > 0; k--)    // Compute the probabilities beta
    {
        for(size_t S = 0; S < 16; S++)
        {
            // Knowing d_k = i and S_{k-1} = S,
                // m = S_k = (2*S & 15) + (S & 8)/8 ^ (i ^ (S & 4)/4)

            // d_k = 0
            m = (2*S & 15) + ((S & 8)/8) ^ ((S & 4)/4);
            beta->list[S + 16*k] = beta->list[m + 16*(k + 1)] *
                                                gamma->list[2*m + 32*(k + 1)];

            // d_k = 1
            m = (2*S & 15) + ((S & 8)/8) ^ (1 - (S & 4)/4);
            i = 2*m + 32*(k + 1);
            beta->list[S + 16*k] += beta->list[i / 2] * gamma->list[1 + i];
        }

        for(size_t S = 0; S < 16; S++)
        {
            // Normalize beta
            beta->list[S + 16*k] /= a->list[k + 1];

            // Compute lambda
            lambda->list[2*S + 32*k] = alpha->list[2*S + 32*k] *
                                                    beta->list[S + 16*k];
            lambda->list[1 + 2*S + 32*k] = alpha->list[1 + 2*S + 32*k] *
                                                    beta->list[S + 16*k];
        }

        tmp[0] = 0.0;
        tmp[1] = 0.0;

        for(size_t S = 0; S < 16; S++)
        {
            tmp[0] += lambda->list[2*S + 32*k];
            tmp[1] += lambda->list[1 + 2*S + 32*k];
        }

        llr->list[k - 1] = log(tmp[1] / tmp[0]);
    }

    // Free the arrays
    free_s_list(alpha);
    free_s_list(beta);
    free_s_list(gamma);
    free_s_list(lambda);
    free_s_list(a);
```

```
255
256     return 0;
257 }
258
259
260 // If  f = 1 then we must deinterleave the array
261 h_list * recreate(s_list *mes, char f)
262 {
263     char debug = 0;
264
265     if (f != 0 && f != 1)
266     {
267         if (debug)
268         {
269             printf("Incorrect value for f\n");
270         }
271         return NULL;
272     }
273     size_t i;
274
275     h_list *res = chl(mes->n, mes->n);
276     if (debug)
277     {
278         printf("Created recipient\n");
279         printf("Sizes : %d, %d\n", mes->n, res->n);
280     }
281
282     for(size_t k = 0; k < mes->n; k++)
283     {
284         i = f ? pi(k) : k;
285         if (debug)
286         {
287             //printf("%d, %d\n", i, k);
288         }
289         res->list[i] = (mes->list[k] > 0);
290     }
291     if (debug)
292     {
293         printf("Copied data\n");
294     }
295     return res;
296 }
297
298
299 // Divides the rate 1/3 stream into three arrays
300 int split_s(s_list *buf, s_list *X, s_list *Y1, s_list *Y2)
301 {
302     if (buf->n % 3 || X->n != Y1->n || X->n != Y2->n || X->n != (buf->n / 3))
303     {
304         return 1;
305     }
306
307     for(size_t i = 0; i < X->n; i++)
308     {
309         X->list[i] = buf->list[3 * i];
310         Y1->list[i] = buf->list[3*i + 1];
311         Y2->list[i] = buf->list[3*i + 2];
312     }
313     return 0;
314 }
315
316
317 // Interleave the array Y and puts the result in X
318 int interleave(s_list *X, s_list *Y)
```

```
{
    if (Y->n > X->m_s)
    {
        return 1;
    }

    X->n = Y->n;
    double t;

    for(size_t i = 0; i < k1 * k2; i++)
    {
        // If the result is inf or -inf we replace it to prevent errors
        t = Y->list[pi(i)];
        if (isfinite(t))
        {
            X->list[i] = t;
        }
        if (isinf(t))
        {
            X->list[i] = 100 * (t == inf ? 1.0 : -1.0);
        }
    }
    return 0;
}


// Deinterleave the array Y and puts the result in X
int deinterleave(s_list *X, s_list *Y)
{
    if (Y->n > X->m_s)
    {
        return 1;
    }

    double t;

    for(size_t i = 0; i < k1 * k2; i++)
    {
        // If the result is inf or -inf we replace it to prevent errors
        t = Y->list[i];
        if (isfinite(t))
        {
            X->list[pi(i)] = t;
        }
        if (isinf(t))
        {
            X->list[pi(i)] = 100 * (t == inf ? 1.0 : -1.0);
        }
    }
    return 0;
}


// Compute the max negative and min positive number of a s_list
void min_max(double mM[2], s_list *X)
{
    double x;
    double m = -inf;
    double M = inf;
    for(size_t i = 0; i < X->n; i++)
    {
        x = X->list[i];
        if (x <= 0 && x > m)
        {
```

```
383             m = x;
384         }
385 
386         if (x >= 0 && x < M)
387         {
388             M = x;
389         }
390     }
391     mM[0] = m;
392     mM[1] = M;
393 }
394 
395 
396 // Compute the min negative and max positive number of an array
397 void max_min(double mM[2], s_list *X)
398 {
399     double x;
400     double m = 0;
401     double M = 0;
402     for(size_t i = 0; i < X->n; i++)
403     {
404         x = X->list[i];
405         if (x < m)
406         {
407             m = x;
408         }
409 
410         if (x > M)
411         {
412             M = x;
413         }
414     }
415     mM[0] = m;
416     mM[1] = M;
417 }
418 
419 
420 // Executes a single pass through both decoders on the stream
421 h_list * decode_turbo_basic(s_list *buf, double s)
422 {
423     char debug = 0;
424 
425     char interleaved = 0;
426     size_t n = buf->n / 3;
427 
428     s_list *llr = csl(n, n);    // The 0-th bit is not considered
429     s_list *X1 = csl(n, n);
430     s_list *X2 = csl(n, buf->n);
431     s_list *Y1 = csl(n, n);
432     s_list *Y2 = csl(n, n);
433 
434     if (debug)
435     {
436         printf("Created the s_lists\n");
437         printf("Sizes :\n");
438         printf("\t- buf : %d\n", buf->n);
439         printf("\t- llr : %d\n", llr->n);
440         printf("\t- X1 : %d\n", X1->n);
441         printf("\t- X2 : %d\n", X2->n);
442         printf("\t- Y1 : %d\n", Y1->n);
443         printf("\t- Y2 : %d\n", Y2->n);
444     }
445 
446     double t;
```

```
447     int k;
448 
449     split_s(buf, X1, Y1, Y2);
450     if (debug)
451     {
452         printf("Split incomming message\n");
453     }
454 
455     k = decode_part(X1, Y1, llr, s);
456     if (debug)
457     {
458         printf("First pass on the decoder\n");
459         printf("\t- Error code : %d\n", k);
460     }
461 
462     // If the values in llr are sufficiently big, there is no need to do a
463     // second pass, thus we compute the min of positives and max of negatives
464     // to check this situation
465     double mM[2];
466     double Mm[2];
467     min_max(mM, llr);
468     max_min(Mm, llr);
469 
470     if (Mm[0] > -26 || Mm[1] < 26)
471     {
472         interleaved = 1;
473 
474         // We need to interleave the llr to match the pattern of Y2
475         interleave(X2, llr);
476         if (debug)
477         {
478             printf("Interleaved the data\n");
479         }
480 
481         k = decode_part(X2, Y2, llr, s);
482         if (debug)
483         {
484             printf("Second pass on the decoder\n");
485             printf("\t- Error code : %d\n", k);
486         }
487     }
488 
489 
490 
491     h_list *res = recreate(llr, interleaved);
492 
493     // Free all the arrays
494     free_s_list(llr);
495     free_s_list(X1);
496     free_s_list(X2);
497     free_s_list(Y1);
498     free_s_list(Y2);
499     if (debug)
500     {
501         printf("Freed structures\n");
502     }
503 
504     return res;
505 }
506 
507 
508 // Executes at most i_max passes through both decoders on the stream
509 h_list * decode_turbo_iter(s_list *buf, double s, size_t i_max)
510 {
```

```
511     char debug = 0;
512
513     char done = 0;
514     char interleaved = 0;
515     size_t iter = 0;
516
517     double mM[2];
518     double Mm[2];
519
520     size_t n = buf->n / 3;
521
522     s_list *llr = csl(n, n);    // The 0-th bit is not considered
523     s_list *X1 = csl(n, n);
524     s_list *X2 = csl(n, buf->n);
525     s_list *Y1 = csl(n, n);
526     s_list *Y2 = csl(n, n);
527
528     if (debug)
529     {
530         printf("Created the s_lists\n");
531         printf("Sizes :\n");
532         printf("\t- buf : %d\n", buf->n);
533         printf("\t- llr : %d\n", llr->n);
534         printf("\t- X1 : %d\n", X1->n);
535         printf("\t- X2 : %d\n", X2->n);
536         printf("\t- Y1 : %d\n", Y1->n);
537         printf("\t- Y2 : %d\n", Y2->n);
538     }
539
540     double t;
541     int k;
542
543     split_s(buf, X1, Y1, Y2);
544     if (debug)
545     {
546         printf("Split incomming message\n");
547     }
548
549     while (!done && iter < i_max)
550     {
551         iter ++;
552
553         if (debug)
554         {
555             printf("Iteration %d / %d\n", iter, i_max);
556         }
557
558         interleaved = 0;
559
560         k = decode_part(X1, Y1, llr, s);
561         substract_s(llr, X1);
562         if (debug)
563         {
564             printf("\tFirst pass on the decoder\n");
565             printf("\t\t- Error code : %d\n", k);
566         }
567
568         // If the values in llr are sufficiently big, there is no need to do a
569         // second pass, thus we compute the min of positives and max of
570         // negatives to check for this situation
571         max_min(Mm, llr);
572
573         if (Mm[0] > -5000000 || Mm[1] < 5000000)
574         {
```

```
575             interleaved = 1;
576 
577             // We need to interleave the llr to match the pattern of Y2
578             interleave(X2, llr);
579             if (debug)
580             {
581                 printf("\tInterleaved the data\n");
582             }
583 
584             k = decode_part(X2, Y2, llr, s);
585             substract_s(llr, X2);
586             if (debug)
587             {
588                 printf("\tSecond pass on the decoder\n");
589                 printf("\t\t- Error code : %d\n", k);
590             }
591 
592             // Update X1 with the new values
593             deinterleave(X1, llr);
594         }
595 
596         if (Mm[0] < -5000000 && Mm[1] > 5000000)
597         {
598             done = 1;
599             if (debug)
600             {
601                 printf("Decoding complete\n");
602             }
603         }
604     }
605 
606     h_list *res = recreate(llr, interleaved);
607 
608     // Free all the arrays
609     free_s_list(llr);
610     free_s_list(X1);
611     free_s_list(X2);
612     free_s_list(Y1);
613     free_s_list(Y2);
614     if (debug)
615     {
616         printf("Freed structures\n");
617     }
618 
619     return res;
620 }
```

1.7 ldpc.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "ldpc.h"
#include "list.h"
#include "random.h"
#include "basic.h"

// Create a ldpc matrix as described in Gallager's paper from 1963
h_matrix * create_base(size_t n, size_t j, size_t k)
{
    if (n % k)
    {
        return NULL;
    }

    size_t m = (n * j) / k;
    h_matrix *res = chm(m, n);
    i_list *perm = cil(n, n);
    // Fill the first horizontal part of the matrix
    for (size_t i = 0; i < n; i++)
    {
        shm(res, 1, (i / k), i);
    }

    // Fill the j-1 other bands
    for (size_t i = 1; i < j; i++)
    {
        permutation(perm);
        for (size_t x = 0; x < n; x++)
        {
            shm(res, 1, ((perm->list[x] + i * n) / k), x);
        }
    }
    return res;
}


// Create the encoder matrix with the base matrix
h_matrix * create_generator_matrix_h(h_matrix *mat)
{
    return juxtapose_h(mat, 1);
}


// Create the encoder matrix with the base matrix
a_matrix * create_generator_matrix_a(h_matrix *mat)
{
    return juxtapose_a(mat, 1);
}


// Create the decoder matrix with the base matrix
h_matrix * create_decoder_matrix_h(h_matrix *mat)
{
    return juxtapose_h(mat, 0);
}


// Create the decoder matrix with the base matrix
a_matrix * create_decoder_matrix_a(h_matrix *mat)
```

```
63 {
64     return juxtapose_a(mat, 0);
65 }
66
67
68 // Encode a message mes with the generator matrix gen
69 h_list * encode_ldpc_h(h_matrix *gen, h_list *mes)
70 {
71     return product_h(gen, mes);
72 }
73
74
75 // Encode a message mes with the generator matrix gen
76 h_list * encode_ldpc_a(a_matrix *gen, h_list *mes)
77 {
78     return product_a(gen, mes);
79 }
80
81
82 // Decode the received message res with the decoding matrix mat
83 // Returns the number of iterations
84 int decode_ldpc_a_basic(a_matrix *mat, h_list *mes, size_t nb_max)
85 {
86     h_list *verif = product_a(mat, mes);
87     i_list *count = cil(mes->n, mes->n);
88     i_list *max_errors = cil(mes->n, mes->n);
89     size_t iter = 0;
90
91     char correct = is_all_nil(verif);
92
93     while (!correct && (iter < nb_max))
94     {
95         set_all_i_list(count, 0);
96
97         // Count the number of errors for each bit
98         for (size_t i = 0; i < verif->n; i++)
99         {
100             if (verif->list[i])
101             {
102                 for (size_t j = 0; j < mat->list_n[i]->n; j++)
103                 {
104                     count->list[mat->list_n[i]->list[j]] ++;
105                 }
106             }
107         }
108
109         // Flip the bits with the most errors
110         max_i_list(count, max_errors);
111         for (size_t i = 0; i < max_errors->n; i++)
112         {
113             mes->list[max_errors->list[i]] ^= 1;
114         }
115
116         iter ++;
117
118         product_a_in_place(mat, mes, verif);
119         correct = is_all_nil(verif);
120     }
121
122     // Free used lists
123     free_h_list(verif);
124     free_i_list(count);
125     free_i_list(max_errors);
126
```

```
127     return iter;
128 }
129 
130 
131 h_list * decode_ldpc_proba(a_matrix *mat, s_list *mes, double s, size_t nb_max)
132 {
133     // Used to store the sign of the llr
134     h_list *alpha = chl(mes->n, mes->n);
135 
136     // Used to store the absolute value of the llr
137     s_list *beta = csl(mes->n, mes->n);
138 
139     // To store f(beta)
140     s_list *f_beta = csl(mes->n, mes->n);
141 
142     // To store the temporary results
143     s_list *tmp = csl(mes->n, mes->n);
144 
145     // To store the sum of f(beta_{i,l})
146     s_list *f_sum = csl(mat->n, mat->n);
147 
148     // Initialize alpha and beta
149     double l;
150 
151     for (size_t d = 0; d < mes->n; d++)
152     {
153         l = log(p_trans(mes->list[d], 1, s) / p_trans(mes->list[d], 0, s));
154         alpha->list[d] = (l > 0 ? 1 : -1);
155         beta->list[d] = fabs(l);
156     }
157 
158     size_t iter = 0;
159     double m = min_s(beta);
160 
161     char a;
162     double b;
163 
164     while (iter < nb_max && m < 5000)
165     {
166         iter ++;
167 
168         // Fill f_beta
169         for (size_t i = 0; i < beta->n; i++)
170         {
171             f_beta->list[i] = f(beta->list[i]);
172         }
173 
174         // Compute the right sums
175         product_s_ip(mat, f_beta, f_sum);
176 
177         for (size_t d = 0; d < mes->n; d++)
178         {
179             //if (beta->list[d] < 5000)
180             //{
181                 tmp->list[d] = alpha->list[d] * beta->list[d];
182 
183                 // Compute f of the sum and the product of alpha_{i,l}
184                 for (size_t i = 0; i < mat->list_m[d]->n; i++)
185                 {
186                     a = alpha->list[d];
187                     for (size_t k = 0; k < mat->list_n[i]->n; k++)
188                     {
189                         a *= alpha->list[mat->list_n[i]->list[k]];
190                     }
```

```
191
192                     tmp->list[d] += a * f(f_sum->list[mat->list_m[d]->list[i]] -
193                                           f_beta->list[d]);
194                 }
195             //}
196
197
198         }
199
200         // Update alpha and beta
201         for (size_t d = 0; d < mes->n; d++)
202         {
203             alpha->list[d] = (tmp->list[d] > 0 ? 1 : -1);
204             beta->list[d] = fabs(tmp->list[d]);
205         }
206
207         m = min_s(beta);
208     }
209
210     // Put the results in tmp
211     for (size_t d = 0; d < mes->n; d++)
212     {
213         tmp->list[d] = alpha->list[d] * beta->list[d];
214     }
215
216
217     h_list *res = decode_h_basic(tmp);
218
219     //print_s_list(tmp);
220
221     free_h_list(alpha);
222     free_s_list(beta);
223     free_s_list(f_beta);
224     free_s_list(f_sum);
225     free_s_list(tmp);
226
227     return res;
228 }
```

1.8 demo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "demo.h"
#include "list.h"
#include "ldpc.h"
#include "turbocode.h"
#include "random.h"
#include "basic.h"


int demo_turbo_basic(char *file_name, double s)
{
    char debug = 0;

    char path[1024];        // The source file
    char path_n[1024];      // The noisy file
    char path_d[1024];      // The decoded file

    getcwd(path, 1024);
    strcat(path, "/../files/bytes/");
    strcat(path, file_name);

    strcpy(path_n, path);
    strcpy(path_d, path);

    strcat(path, ".bt");
    strcat(path_n, "_turbo_n.bt");
    strcat(path_d, "_turbo_d.bt");

    FILE *fp = fopen(path, "r");
    FILE *fn = fopen(path_n, "w");
    FILE *fd = fopen(path_d, "w");

    if (debug)
    {
        printf("Opened files\n");
    }

    hh_list *mes_d = read_bit_file_h(fp, 8920);
    hh_list *mes_n = deep_copy(mes_d);

    if (debug)
    {
        printf("Created hh_lists\n");
    }

    printf("Number of slices : %d\n", mes_d->n);

    for (size_t i = 0; i < mes_d->n; i++)
    {
        printf("\nSlice number %d :\n", i);

        h_list *res = encode_turbo(mes_d->list[i]);
        printf("\t- Encoded\n");

        s_list *noisy = add_noise(res, s);
        printf("\t- Added gaussian noise\n");

        h_list *noisy_h = decode_h_basic(noisy);
        if (debug)
```

```
63          {
64              printf("\t- Performed basic decoding\n");
65          }
66
67          copy_h(noisy_h, mes_n->list[i], 8920, 3);
68          if (debug)
69          {
70              printf("\t- Copied basic decoding\n");
71          }
72
73          h_list *dec = decode_turbo_basic(noisy, s);
74          printf("\t- Decoded\n");
75
76          copy_h(dec, mes_d->list[i], 8920, 1);
77          if (debug)
78          {
79              printf("\t- Copied decoding\n");
80          }
81
82          free_h_list(dec);
83          free_h_list(noisy_h);
84          free_s_list(noisy);
85          free_h_list(res);
86      }
87
88      write_bit_file_h(fn, mes_n);
89      write_bit_file_h(fd, mes_d);
90
91      fclose(fp);
92      fclose(fn);
93      fclose(fd);
94
95      free_hh_list(mes_d);
96      free_hh_list(mes_n);
97
98      return 0;
99  }
100
101
102 int demo_turbo_iter(char *file_name, double s, size_t i_max)
103 {
104     char debug = 1;
105
106     char path[1024];        // The source file
107     char path_n[1024];      // The noisy file
108     char path_d[1024];      // The decoded file
109
110     getcwd(path, 1024);
111     strcat(path, "/../files/bytes/");
112     strcat(path, file_name);
113
114     strcpy(path_n, path);
115     strcpy(path_d, path);
116
117     strcat(path, ".bt");
118     strcat(path_n, "_turbo_n.bt");
119     strcat(path_d, "_turbo_d.bt");
120
121     FILE *fp = fopen(path, "r");
122     FILE *fn = fopen(path_n, "w");
123     FILE *fd = fopen(path_d, "w");
124
125     if (debug)
126     {
```

```
127         printf("Opened files\n");
128     }
129 
130     hh_list *mes_d = read_bit_file_h(fp, 8920);
131     hh_list *mes_n = deep_copy(mes_d);
132 
133     if (debug)
134     {
135         printf("Created hh_lists\n");
136     }
137 
138     printf("Number of slices : %d\n", mes_d->n);
139 
140     for (size_t i = 0; i < mes_d->n; i++)
141     {
142         printf("\nSlice number %d :\n", i);
143 
144         h_list *res = encode_turbo(mes_d->list[i]);
145         printf("\t- Encoded\n");
146 
147         s_list *noisy = add_noise(res, s);
148         printf("\t- Added gaussian noise with s = %f\n", s);
149 
150         h_list *noisy_h = decode_h_basic(noisy);
151         if (debug)
152         {
153             printf("\t- Performed basic decoding\n");
154         }
155 
156         copy_h(noisy_h, mes_n->list[i], 8920, 3);
157         if (debug)
158         {
159             printf("\t- Copied basic decoding\n");
160         }
161 
162         h_list *dec = decode_turbo_iter(noisy, s, i_max);
163         printf("\t- Decoded\n");
164 
165         dec->n = 8920;
166         printf("\t\tNumber of errors : %d\n", nb_errors(mes_d->list[i], dec));
167 
168         copy_h(dec, mes_d->list[i], 8920, 1);
169         if (debug)
170         {
171             printf("\t- Copied decoding\n");
172         }
173 
174         free_h_list(dec);
175         free_h_list(noisy_h);
176         free_s_list(noisy);
177         free_h_list(res);
178     }
179 
180     write_bit_file_h(fn, mes_n);
181     write_bit_file_h(fd, mes_d);
182 
183     fclose(fp);
184     fclose(fn);
185     fclose(fd);
186 
187     free_hh_list(mes_d);
188     free_hh_list(mes_n);
189 
190     return 0;
```

```
}


int demo_ldpc_basic(char *file_name, double s, size_t i_max)
{
    char debug = 1;
    int err;

    char path[1024];        // The source file
    char path_n[1024];      // The noisy file
    char path_d[1024];      // The decoded file

    getcwd(path, 1024);
    strcat(path, "/../files/bytes/");
    strcat(path, file_name);

    strcpy(path_n, path);
    strcpy(path_d, path);

    strcat(path, ".bt");
    strcat(path_n, "_ldpc_basic_n.bt");
    strcat(path_d, "_ldpc_basic_d.bt");

    FILE *fp = fopen(path, "r");
    FILE *fn = fopen(path_n, "w");
    FILE *fd = fopen(path_d, "w");
    if (debug)
    {
        printf("Opened files\n");
    }

    hh_list *mes_d = read_bit_file_h(fp, 8920);
    hh_list *mes_n = deep_copy(mes_d);
    if (debug)
    {
        printf("Created hh_lists\n");
    }

    h_matrix *base = create_base(8920, 40, 20);
    a_matrix *gen = cgm_a(base);
    a_matrix *dec = cdm_a(base);
    if (debug)
    {
        printf("Created LDPC matrices\n");
    }

    printf("Number of slices : %d\n", mes_d->n);

    for (size_t i = 0; i < mes_d->n; i++)
    {
        printf("\nSlice number %d :\n", i);

        h_list *res = encode_ldpc_a(gen, mes_d->list[i]);
        printf("\t- Encoded\n");

        s_list *noisy = add_noise(res, s);
        printf("\t- Added gaussian noise\n");

        h_list *noisy_h = decode_h_basic(noisy);
        if (debug)
        {
            printf("\t- Performed basic decoding\n");
        }

```

```
255         copy_h(noisy_h, mes_n->list[i], 8920, 1);
256         if (debug)
257         {
258             printf("\t- Copied basic decoding\n");
259         }
260 
261         err = decode_ldpc_a_basic(dec, noisy_h, i_max);
262         printf("\t- Decoded\n");
263         if (debug)
264         {
265             printf("\t\t- Iterations : %d\n", err);
266         }
267 
268         copy_h(noisy_h, mes_d->list[i], 8920, 1);
269         if (debug)
270         {
271             printf("\t- Copied decoding\n");
272         }
273 
274         free_h_list(noisy_h);
275         free_s_list(noisy);
276         free_h_list(res);
277     }
278 
279     write_bit_file_h(fn, mes_n);
280     write_bit_file_h(fd, mes_d);
281 
282     fclose(fp);
283     fclose(fn);
284     fclose(fd);
285 
286     free_hh_list(mes_d);
287     free_hh_list(mes_n);
288 
289     free_h_matrix(base);
290     free_a_matrix(gen);
291     free_a_matrix(dec);
292 
293     return 0;
294 }
295 
296 
297 int demo_ldpc_proba(char *file_name, double s, size_t i_max)
298 {
299     char debug = 1;
300     int err;
301 
302     char path[1024];        // The source file
303     char path_n[1024];      // The noisy file
304     char path_d[1024];      // The decoded file
305 
306     getcwd(path, 1024);
307     strcat(path, "/../files/bytes/");
308     strcat(path, file_name);
309 
310     strcpy(path_n, path);
311     strcpy(path_d, path);
312 
313     strcat(path, ".bt");
314     strcat(path_n, "_ldpc_proba_n.bt");
315     strcat(path_d, "_ldpc_proba_d.bt");
316 
317     FILE *fp = fopen(path, "r");
318     FILE *fn = fopen(path_n, "w");
```

```
319     FILE *fd = fopen(path_d, "w");
320     if (debug)
321     {
322         printf("Opened files\n");
323     }
324 
325     hh_list *mes_d = read_bit_file_h(fp, 8920);
326     hh_list *mes_n = deep_copy(mes_d);
327     if (debug)
328     {
329         printf("Created hh_lists\n");
330     }
331 
332     h_matrix *base = create_base(8920, 40, 20);
333     a_matrix *gen = cgm_a(base);
334     a_matrix *dec = cdm_a(base);
335     if (debug)
336     {
337         printf("Created LDPC matrices\n");
338     }
339 
340     printf("Number of slices : %d\n", mes_d->n);
341 
342     for (size_t i = 0; i < mes_d->n; i++)
343     {
344         printf("\nSlice number %d :\n", i);
345 
346         h_list *res = encode_ldpc_a(gen, mes_d->list[i]);
347         printf("\t- Encoded\n");
348 
349         s_list *noisy = add_noise(res, s);
350         printf("\t- Added gaussian noise\n");
351 
352         h_list *noisy_h = decode_h_basic(noisy);
353         if (debug)
354         {
355             printf("\t- Performed basic decoding\n");
356         }
357 
358         copy_h(noisy_h, mes_n->list[i], 8920, 1);
359         if (debug)
360         {
361             printf("\t- Copied basic decoding\n");
362         }
363 
364         h_list *dem = decode_ldpc_proba(dec, noisy, s, i_max);
365         printf("\t- Decoded\n");
366 
367         copy_h(dem, mes_d->list[i], 8920, 1);
368         if (debug)
369         {
370             printf("\t- Copied decoding\n");
371         }
372 
373         free_h_list(noisy_h);
374         free_s_list(noisy);
375         free_h_list(res);
376         free_h_list(dem);
377     }
378 
379     write_bit_file_h(fn, mes_n);
380     write_bit_file_h(fd, mes_d);
381 
382     fclose(fp);
```

```
383     fclose(fn);
384     fclose(fd);
385 
386     free_hh_list(mes_d);
387     free_hh_list(mes_n);
388 
389     free_h_matrix(base);
390     free_a_matrix(gen);
391     free_a_matrix(dec);
392 
393     return 0;
394 }
395 
396 
397 int demo_turbo_graph(size_t nb_iter, double p)
398 {
399     char path[1024];
400 
401     getcwd(path, 1024);
402     strcat(path, "/../files/graph/turbo.grp");
403 
404     FILE *fp = fopen(path, "w");
405 
406     double noise[] = {2, 1, 0.9, 0.8, 0.7, 0.6, 0.5,
407                                 0.4, 0.3, 0.2, 0.1, 0.05, 0.02, 0.01};
408     double s;
409     size_t nb_err;
410 
411     double ber;
412 
413     h_list *mes = chl(8920, 8920);
414     random_h(mes, p);
415 
416     h_list *enc = encode_turbo(mes);
417 
418     for (size_t i = 0; i < 14; i++)
419     {
420         nb_err = 0;
421         s = pow(10, -0.1*noise[i]);
422 
423         for (size_t k = 0; k < nb_iter; k++)
424         {
425             s_list *sig = add_noise(enc, s);
426             h_list *res = decode_turbo_iter(sig, s, 20);
427             res->n = 8920;
428 
429             nb_err += nb_errors(res, mes);
430 
431             free_s_list(sig);
432             free_h_list(res);
433         }
434         printf("%f -- %d / %d\n", s, nb_err, 8920*nb_iter);
435 
436         ber = ((double) nb_err) / (8920.0 * nb_iter);
437         fprintf(fp, "%f %f\n", noise[i], ber);
438     }
439 
440 
441     fclose(fp);
442 
443     free_h_list(mes);
444 }
445 
446 
```

```
447 int demo_ldpc_graph(size_t nb_iter, double p)
448 {
449     char path[1024];
450 
451     getcwd(path, 1024);
452     strcat(path, "/../files/graph/ldpc.grp");
453 
454     FILE *fp = fopen(path, "w");
455 
456     double noise[] = {5, 4, 3, 2, 1, 0.9, 0.8, 0.7, 0.6, 0.5,
457                                 0.4, 0.3, 0.2, 0.1};
458     double s;
459     size_t nb_err;
460 
461     double ber;
462 
463     h_list *mes = chl(8920, 8920);
464     random_h(mes, p);
465 
466     h_matrix *base = create_base(8920, 40, 20);
467     a_matrix *gen = cgm_a(base);
468     a_matrix *dec = cdm_a(base);
469 
470     h_list *enc = encode_ldpc_a(gen, mes);
471 
472     for (size_t i = 0; i < 14; i++)
473     {
474         nb_err = 0;
475         s = pow(10, -0.1*noise[i]);
476 
477         for (size_t k = 0; k < nb_iter; k++)
478         {
479             s_list *sig = add_noise(enc, s);
480             h_list *sig_h = decode_h_basic(sig);
481             decode_ldpc_a_basic(dec, sig_h, 100);
482             sig_h->n = 8920;
483 
484             nb_err += nb_errors(sig_h, mes);
485 
486             free_s_list(sig);
487             free_h_list(sig_h);
488         }
489         printf("%f -- %d / %d\n", s, nb_err, 8920*nb_iter);
490 
491         ber = ((double) nb_err) / (8920.0 * nb_iter);
492         fprintf(fp, "%f %f\n", noise[i], ber);
493     }
494 
495 
496     fclose(fp);
497 
498     free_h_list(mes);
499     free_h_matrix(base);
500     free_a_matrix(gen);
501     free_a_matrix(dec);
502 
503 }
504 
505 
506 int demo_base_graph(size_t nb_iter, double p)
507 {
508     char path[1024];
509 
510     getcwd(path, 1024);
```

```
511     strcat(path, "/../files/graph/base.grp");
512 
513     FILE *fp = fopen(path, "w");
514 
515     double noise[] = {5, 4, 3, 2, 1, 0.9, 0.8, 0.7, 0.6, 0.5,
516                                 0.4, 0.3, 0.2, 0.1, 0.05, 0.02, 0.01};
517     double s;
518     size_t nb_err;
519 
520     double ber;
521 
522     h_list *mes = chl(8920, 8920);
523     random_h(mes, p);
524 
525     for (size_t i = 0; i < 17; i++)
526     {
527         nb_err = 0;
528         s = pow(10, -0.1*noise[i]);
529 
530         for (size_t k = 0; k < nb_iter; k++)
531         {
532             s_list *sig = add_noise(mes, s);
533             h_list *res = decode_h_basic(sig);
534 
535             nb_err += nb_errors(res, mes);
536 
537             free_s_list(sig);
538             free_h_list(res);
539         }
540         printf("%f -- %d / %d\n", s, nb_err, 8920*nb_iter);
541 
542         ber = ((double) nb_err) / (8920.0 * nb_iter);
543         fprintf(fp, "%f %f\n", noise[i], ber);
544     }
545 
546 
547     fclose(fp);
548 
549     free_h_list(mes);
550 
551 }
```

2 Header files

2.1 basic.h

```
#include <stdio.h>

#include "list.h"

#ifndef BASIC_H
#define BASIC_H

h_list * decode_h_basic(s_list *mes);
s_list * add_noise(h_list *mes, double s);
int nb_errors(h_list *og_mes, h_list *mes);

size_t file_size(FILE *fp);

double f(double b);

#endif
```

2.2 list.h

```
#include <stdio.h>

#ifndef LIST_H
#define LIST_H

typedef struct h_list {
  size_t n;      // Number of elements
  size_t m_s;    // Maximum number of elements
  char *list;
} h_list;

typedef struct s_list {
  size_t n;      // Number of elements
  size_t m_s;    // Maximum number of elements
  double *list;
} s_list;

typedef struct i_list {
  size_t n;      // Number of elements
  size_t m_s;    // Maximum number of elements
  int *list;
} i_list;

typedef struct h_matrix {
  size_t n;  // n lignes
  size_t m;  // m colonnes
  char *mat;
} h_matrix;

// Use a sparse matrix structure similar to that of David MacKay's
typedef struct a_matrix {
  size_t n;            // n lignes
  size_t m;            // m colonnes
  i_list **list_m;    // Liste des coordonÃ©es verticales non nulles
  i_list **list_n;    // Liste des coordonÃ©es horizontales non nulles

} a_matrix ;

typedef struct hh_list {
  size_t n;
  size_t m_e;         // Number of elements to write into file
```

```
  h_list **list;
} hh_list;

h_list * create_h_list(size_t n, size_t m_s);
i_list * create_i_list(size_t n, size_t m_s);
s_list * create_s_list(size_t n, size_t m_s);
h_matrix * create_h_matrix(size_t n, size_t m);
a_matrix * create_a_matrix(size_t n, size_t m);
hh_list * create_hh_list(size_t n, size_t m_e);

char get_h_list(h_list *list_h, size_t i);
double get_s_list(s_list *list_s, size_t i);
char get_h_matrix(h_matrix *mat_h, size_t i, size_t j);

void set_h_list(h_list *list_h, char x, size_t i);
void set_s_list(s_list *list_s, double x, size_t i);
void set_h_matrix(h_matrix *mat_h, char x, size_t i, size_t j);

void write_char_h(h_list *list_h, char x, size_t p);
char read_char_h(h_list *list_h, size_t p);

void write_bit_h(h_list *list_h, unsigned char x, size_t p);
char read_bit_h(h_list *list_h, size_t p);

void set_all_h_list(h_list *list_h, char x);
void set_all_i_list(i_list *list_i, int x);
void aet_all_s_list(s_list *list_s, double x);

char is_all_nil(h_list *list_h);

double min_s(s_list *list_s);
double max_s(s_list *list_s);

int append_i(i_list *list_i, int x);
int shift_i(i_list *list_i, int l);

int substract_s(s_list *list_a, s_list *list_b);

int copy_h(h_list *list_a, h_list *list_b, size_t n, size_t s);
hh_list *deep_copy(hh_list *list_hh);

void max_i_list(i_list *list_i, i_list *res);

h_list * product_h(h_matrix *mat, h_list *vect);
h_list * product_a(a_matrix *mat, h_list *vect);
int product_a_in_place(a_matrix *mat, h_list *vect, h_list *res);
int product_s_ip(a_matrix *mat, s_list *vect, s_list *res);
h_matrix * juxtapose_h(h_matrix *mat, char dir);
a_matrix * juxtapose_a(h_matrix *mat, char dir);

a_matrix * convert_h(h_matrix *mat);

hh_list * read_file_h(FILE *fp, size_t n);
int write_file_h(FILE *fp, hh_list *list_hh);

hh_list * read_bit_file_h(FILE *fp, size_t n);
int write_bit_file_h(FILE *fp, hh_list *list_hh);

void print_h_list(h_list *list_h);
void print_i_list(i_list *list_i);
void print_s_list(s_list *list_s);
void print_h_matrix(h_matrix *mat_h);
void print_a_matrix(a_matrix *mat_a);
void print_hh_list(hh_list *list_hh);
```

```
106
107 void free_h_list(h_list *list_h);
108 void free_i_list(i_list *list_i);
109 void free_s_list(s_list *list_s);
110 void free_h_matrix(h_matrix *mat_h);
111 void free_a_matrix(a_matrix *mat_a);
112 void free_hh_list(hh_list *list_hh);
113
114 // AbrÃ©viations
115 #define chl create_h_list
116 #define cil create_i_list
117 #define csl create_s_list
118 #define chm create_h_matrix
119 #define cam create_a_matrix
120 #define chhl create_hh_list
121
122 #define ghl get_h_list
123 #define gsl get_s_list
124 #define ghm get_h_matrix
125
126 #define shl set_h_list
127 #define ssl set_s_list
128 #define shm set_h_matrix
129
130 #define ph product_h
131
132 #endif
```

2.3 random.h

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>
4
5 #include "../lib/BitArray/bit_array.h"
6 #include "../lib/BitArray/bar.h"
7
8 #include "list.h"
9
10 #ifndef RANDOM_H
11 #define RANDOM_H
12
13 #define M_PI 3.14159265358979323846 /* pi */
14 #define inf INFINITY
15
16 typedef bar generator;
17
18 void random_h(h_list *list_h, double p);
19
20 void permutation(i_list *list_i);
21 double box_muller(double m, double s);
22 void addNoise(bar *message, double s, double noisy[]);
23 double pTransition(double x, char d, double s);
24 double normal(double x, double m, double s);
25 double mean(double *Z, size_t n);
26 double variance(double *Z, size_t n);
27
28 bar * initGenerator(void);
29 char yield(generator *gen);
30 bar * sequence(generator *gen, size_t n);
31 bar * combine(generator *gen, bar *message);
32 void resetGenerator(generator *gen);
33 void freeGenerator(generator *gen);
34
35 #define p_trans pTransition
```

```

#endif
```

2.4 turbocode.h

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

#include "../lib/BitArray/bit_array.h"
#include "../lib/BitArray/bar.h"
#include "basic.h"
#include "list.h"

#ifndef TURBOCODE_H
#define TURBOCODE_H

#define k1 8
#define k2 (223 * 5)

#define pTrans pTransition

extern const size_t p[];

typedef bar buffer;

size_t pi(size_t s);
char yieldEncode(char d, bar *memState);
bar * initMemState(size_t n);
h_list * encode_turbo(h_list *buf);
int decode_part(s_list *X, s_list *Y, s_list *llr, double s);

h_list * recreate(s_list *mes, char f);
int split_s(s_list *buf, s_list *X, s_list *Y1, s_list *Y2);
int interleave(s_list *X, s_list *Y);
int deinterleave(s_list *X, s_list *Y);
void min_max(double mM[2], s_list *X);
void max_min(double mM[2], s_list *X);
h_list * decode_turbo_basic(s_list *buf, double s);
h_list * decode_turbo_iter(s_list *buf, double s, size_t i_max);

#endif
```

2.5 ldpc.h

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "list.h"
#include "random.h"
#include "basic.h"

#ifndef LDPC_H
#define LDPC_H

h_matrix * create_base(size_t n, size_t j, size_t k);
h_matrix * create_generator_matrix_h(h_matrix *mat);
a_matrix * create_generator_matrix_a(h_matrix *mat);
h_matrix * create_decoder_matrix_h(h_matrix *mat);
a_matrix * create_decoder_matrix_a(h_matrix *mat);
h_list * encode_ldpc_h(h_matrix *gen, h_list *mes);
h_list * encode_ldpc_a(a_matrix *gen, h_list *mes);

int decode_ldpc_a_basic(a_matrix *mat, h_list *mes, size_t nb_max);
h_list * decode_ldpc_proba(a_matrix *mat, s_list *mes, double s, size_t nb_max);
```

```
22
23 #define cgm_h create_generator_matrix_h
24 #define cgm_a create_generator_matrix_a
25 #define cdm_a create_decoder_matrix_a
26
27 #endif
```

2.6 demo.h

```
1 #include <stdio.h>
2
3 #include "list.h"
4 #include "ldpc.h"
5 #include "turbocode.h"
6
7 #ifndef DEMO_H
8 #define DEMO_H
9
10 int demo_turbo_basic(char *file_name, double s);
11 int demo_turbo_iter(char *file_name, double s, size_t i_max);
12
13 int demo_ldpc_basic(char *file_name, double s, size_t i_max);
14 int demo_ldpc_proba(char *file_name, double s, size_t i_max);
15
16 int demo_turbo_graph(size_t nb_iter, double p);
17 int demo_ldpc_graph(size_t nb_iter, double p);
18 int demo_base_graph(size_t nb_iter, double p);
19
20 #endif
```