CentraleSupélec

Version 011

Mineure

Télécommunications numériques – exercices d'application

Auteur : Jean-Louis Gutzwiller.

Dernière modification : 26 septembre 2019

Jean-Louis.Gutzwiller@centralesupelec.fr

Table des matières

Table des figures

1 Introduction

1.1 Objectif de ce document

Ce document propose une suite d'exercices à réaliser sur un environnement de simulation numérique (logiciel de calcul matriciel « julia ») permettant de mettre en évidence les différentes notions associées à ce cours, à savoir :

- chaîne de transmission idéale
- chaîne de transmission non idéale
- filtre adaptatif pour les environnement inconnus ou variables
- synchronisation d'horloge.

1.2 Logiciel de calcul vectoriel

Le logiciel « Julia », gratuit, est disponible sur les environnements Windows et Linux. Il s'agit d'un logiciel de calcul matriciel, analogue[1] à Matlab ou à Octave.

La version utilisée pour ce document est la version 1.2.0 qui peut être téléchargée pour Windows et pour Linux en [1][2].

Note : Sous certaines versions de Linux, il est possible d'installer julia par la commande « sudo apt-get install julia », mais il semble que la version installée ne soit pas la toute dernière. Sur d'autres versions, les tentatives d'installation de julia indiquent que le package est obsolète. Pour obtenir la toute dernière version de Julia, compatible avec le manuel décrit en [2], il faut la télécharger explicitement depuis [1]. Téléchargez la version correspondant à votre système, par exemple, la version générique pour Linux 64 bits. Décompressez l'archive obtenue et recopier le contenu du répertoire « julia-1.2.0 » de l'archive vers le répertoire « /usr/local » par la commande :

```
sudo cp -r julia-1.2.0/* /usr/local
```

Note : Julia est en pleine évolution. Les versions successives rajoutent des fonctionnalités nouvelles, mais malheureusement suppriment certaines fonctionnalités des versions précédentes. De ce fait, il est important que la même version soit utilisée par tout le monde. Veuillez donc utiliser la version indiquée dans ce document.

Par ailleurs, Julia est un langage extrêmement typé (contrairement à Matlab et Octave). Les types des opérandes doivent être strictement respectés, il faut éventuellement utilisé des transtypages pour corriger les défauts. Par exemple, pour convertir un nombre flottant en un nombre entier, il

1 Notons que la syntaxe de Julia n'est pas identique à celle de Matlab/Octave. De ce fait, les fichiers « .m » de matlab ne sont pas directement utilisable avec Julia.

2 Du fait de l'évolution très rapide de Julia, il est possible que la version disponible sur le site ne soit pas la même que celle indiquée dans ce document. Pour obtenir la version décrite dans ce document, merci de me contacter (mes coordonnées sont sur la page de titre).

faut écrire explicitement[3] *:* `i = Int(x).`

Les différents exercices proposés contiennent des exemples d'instructions à entrer sous Julia pour obtenir les résultats souhaités. On pourra, dans un premier temps, entrer ces instructions telles quelles et vérifier que les résultats sont bien ceux attendus. Dans un deuxième temps, chaque exercice propose, sous la rubrique marquée « À faire » une réalisation pour laquelle on créera un fichier script[4].

Pour obtenir un affichage graphique des résultats (courbes), il faut installer le package PyPlot. À cet effet, il convient de faire (sous Linux)[5] :

```
apt-get install python python-dev python-matplotlib
```

Lors du premier lancement de Julia, il faut activer les fonctionnalités supplémentaires nécessaires pour l'utilisateur en entrant les commandes suivantes :

```
import Pkg; Pkg.add("PyPlot")
import Pkg; Pkg.add("SpecialFunctions")
import Pkg; Pkg.add("DSP")
import Pkg; Pkg.add("FFTW")
```

Ces deux instructions doivent être entrée par chaque utilisateur (la configuration se fait sur le compte de l'utilisateur). En cas de changement de version de Julia, il faut refaire ces commandes.

Pour utiliser les fonctions disponibles, il faut faire, à chaque lancement de Julia :

```
using PyPlot
using SpecialFunctions
using DSP
using FFTW
```

Note : la commande « using PyPlot » peut produire une erreur du fait de la présence de deux versions de Python sur l'ordinateur. Lors de la première utilisation de cette commande, vous pouvez procéder comme suit :
Depuis le terminal, faire :

```
export PYTHON=/usr/bin/python2.7
```

Depuis le même terminal, lancer julia, puis :

```
import Pkg
Pkg.build("PyCall")
using PyPlot
```

3 Cette expression échoue si la valeur de x, codée en flottant, n'est pas un entier.
4 Les fichiers script sous Julia auront de préférence l'extension « .jl ».
5 Cette instruction doit être entrée en tant que « root ».

Note : beaucoup de fonctions sont disponibles pour Julia dans des packages. Pour trouver le bon package permettant d'obtenir la fonction souhaitée, on pourra consulter le site [9].

1.3 Documents à rendre

À la fin des séances du cours, il sera demandé de rendre :

- un compte-rendu
- les fichiers correspondant aux différentes simulations.

Les fichiers « Julia » devront être commentés[6]. Les commentaires doivent couvrir deux exigences :

- décrire l'utilisation de la fonction en entête de fichier
- expliquer les instructions utilisées pour réaliser les opérations (dans le corps de la fonction)

Le compte-rendu devra donner tous les éléments nécessaires à la compréhension de votre travail. En particulier, les références à vos fichiers « Julia », les références aux documents externes que vous pourriez consulter (sites internet, livres, polycopiés, énoncé du sujet). Il devra expliquer vos propres réflexions, c'est à dire décrire l'apport de votre travail relativement à ce qui est déjà connu (en particulier cet énoncé).

Le compte-rendu devra être remis sous forme électronique, au format PDF. Il devra comporter une page de titre et une table des matières ; les pages et les titres devront être numérotés. Les figures devront avoir un titre et être numérotées. La page de titre devra comporter les noms des auteurs et la date de la dernière modification du document.

L'ensemble de ces éléments devra être compressé dans une archive au format « zip » et envoyé par message électronique à « Jean-Louis.Gutzwiller@centralesupelec.fr ».

Remarque importante : *tous les fichiers (fichiers de simulation et compte-rendu) devront comporter les noms de participants (auteurs) ainsi que la date de dernière modification.*

1.4 Mode d'emploi du logiciel Julia

Le mode d'emploi du logiciel Julia est disponible en [2].

1.5 Format des données

L'ensemble de ce document est conçu selon la convention suivante : les signaux, formants et filtres sont représentés par des vecteurs à une dimension.

Sous « Julia », la génération d'un vecteur en une dimension utilise un paramètre qui est le nombre

6 Pour mettre un commentaire, utiliser le caractère #. Tout ce qui suit ce caractère sur la ligne sera ignoré.

de composantes. Par exemple, un vecteur nul de taille 10 se génère par :

```
nul = zeros(10);
```

Note : dans les exemples de codes à taper pour tester les fonctions, les codes sont toujours donnés complets, c'est à dire qu'on peut repartir de zéro à chaque exemple. Notons à ce propos que Julia ne propose pas de commande permettant de nettoyer l'espace de travail (comme « clear » en Matlab). Il est conseillé de quitter et de relancer Julia de temps en temps afin de faire le ménage. Pour quitter Julia, tapez : `exit()`

1.6 Méthodologie et preuves

Il est attendu, au cours du déroulement de ces exercices, de donner les preuves à chaque pas du bon fonctionnement de la simulation. Au début, les exercices sont très dirigistes (ils vous expliquent pas à pas comment procéder), mais au fur et à mesure de l'avancement, le sujet laisse des libertés sur la manière de procéder.

Il convient donc, à chaque simulation demandée, de fournir, non seulement les résultats simulés, mais des preuves suffisantes montrant que la simulation fournit bien les résultats attendus. Ces preuves peuvent être de deux ordres :

- prévision des résultats attendus par un autre méthode (pas seulement qualitative, mais également quantitative)
- mise en œuvre de la simulation avec des hypothèses simplifiées pour lesquelles on connaît la réponse attendue.

1.7 Nature du signal traité et hypothèses de transmission

Dans l'ensemble des exercices abordés dans ce document, nous supposons une transmission en bande de base d'un signal binaire. Les symboles sont notés -1 et +1 et correspondent directement à l'amplitude (facteur multiplicatif) du formant.

Ce choix apporte un certain nombre de simplification par rapport à des cas pouvant être rencontré :

- Pas de synchronisation de porteuse à envisager (chapitre 5)
- Les signaux sont réels (pas de modulation en phase et en quadrature)
- La simulation est plus aisée (le pas de simulation pour les signaux analogiques peut être plus grand sans que cela ne dégrade de manière significative les résultats obtenus)

Ainsi, les simulations envisagées sous Julia se déroulent en un temps raisonnable et il est possible d'atteindre les courbes donnant les taux d'erreurs en fonction du rapport E_b/N_0 durant les séances prévues à l'emploi du temps.

Les exercices proposés sont progressifs et abordent les problématiques suivantes :

- Simulation d'une chaîne de transmission idéale, permettant de prendre en main les outils de simulation et de comparer le résultat obtenu au résultat bien connu issu du cours de signal et

communication de deuxième année (chapitre 2).

- Simulation d'une chaîne de transmission sur un canal de bande passante finie. Les déformations introduites par le canal nécessitent d'adapter le filtre du récepteur (chapitre 3).

- Simulation d'une chaîne de transmission sur un canal de bande passante finie, mais en supposant inconnue la réponse du canal. L'utilisation d'un filtre adaptatif au récepteur permet de traiter ce cas de figure (chapitre 4).

- Prise en compte de l'erreur d'horloge en bande de base. La chaîne de transmission utilisée sera à nouveau la chaîne de transmission idéale, mais il sera supposé que les horloges de l'émetteur et du récepteur ne sont pas correctement calées (erreur à la fois sur la fréquence et sur la phase). Le récepteur devra alors réaliser la synchronisation d'horloge (chapitre 5).

2 Chaîne de transmission idéale

2.1 Introduction

L'objectif de cette simulation est de se remémorer les éléments théoriques vus en cours de deuxième année (signal et communication) et d'en faire la simulation à titre d'exercice afin d'apprendre à maîtriser l'outil de simulation sur un cas de figure connu.

2.2 Principe de simulation

La simulation est effectuée de manière numérique et donc, forcément échantillonnée. Afin de simuler convenablement un système analogique, il est donc nécessaire d'échantillonner ce système suffisamment finement pour faire apparaître les éléments intéressants. En particulier, le signal analogique évoluant entre les instants d'échantillonnage de la chaîne de transmission, il est nécessaire de choisir une fréquence d'échantillonnage de simulation relativement grande par rapport à la fréquence symbole. Un facteur multiplicatif sera donc appliqué à la fréquence de transmission symbole pour obtenir la fréquence d'échantillonnage pour la simulation.

Les signaux et les filtres seront représentés par des vecteurs sous Julia. Les opérations habituelles sont les opérations d'addition, de multiplication et de convolution.

Afin de rendre le code de simulation le plus réutilisable possible, on pensera à passer en paramètre aux différentes fonctions les éléments de réglage.

Remarque importante : pour toutes les simulations effectuées dans ces exercices, les signaux représentant des filtres sont supposés être centrés sur l'origine (ils s'étendent mathématiquement de $-\infty$ à $+\infty$). Nous conviendrons donc que les signaux occupent un intervalle temporel symétrique (de -t à +t) et de ce fait, le nombre d'échantillons de tous les signaux est impair.
Lors du tracé des courbes temporelles, l'échelle horizontale à utiliser sera l'échelle des échantillons du signal numérique. L'origine se trouve donc au milieu de l'échelle temporelle.
L'exemple donné ci-dessous pour le formant montre comment afficher correctement le signal.

Les différents fichiers intervenant dans les simulations devront être placés dans des répertoires séparés ayant les noms suivants :

- Commun : fichiers communs à toutes les simulations, à savoir : « formantrect.jl », « formantcos.jl », « bruit.jl », « emission.jl » et « reception.jl ».
- Taux_Erreurs_Canal : fichiers intervenant dans les calculs de taux d'erreurs pour les récepteurs idéaux, naifs, sous-optimaux (chapitres 2 et 3 de ce document).
- Filtre_adaptatif : fichiers intervenant dans les calculs concernant les filtres adaptatifs (chapitre 4 de ce document).
- Synchronisation : fichiers intervenant dans les calculs relatifs à la synchronisation (chapitre 5 de ce document).

2.3 Éléments de la chaîne de transmission

2.3.1 Formant

La chaîne de transmission idéale utilise un formant (voir cours de signal et communication). Ce formant peut-être généré par une fonction qui fournit le bon vecteur en fonction des paramètres.

Les deux formants envisagés (« formantrect » et « formantcos ») peuvent être fournis sous la forme de deux fonctions définies dans des scripts portant les noms respectifs des fonctions.

À faire : *créer les deux fichiers « formantrect.jl » et « formantcos.jl » avec le contenu ci-dessous. Tracer leurs formes et vérifier que vous obtenez bien le même résultat que sur les graphes ci-dessous.*

Fichier « **formantrect.jl** »[7] :

```
function formantrect(TAILLE, SURECHANTILLONNAGE)
    PI = 3.1415926535;
    formant = zeros(TAILLE);
    t = (collect(1:1:TAILLE) .- Int(floor((TAILLE+1)/2))) / SURECHANTILLONNAGE;
    for i=1:TAILLE
        if t[i] == 0
            formant[i] = 1;
        else
            formant[i] = sin(PI*t[i]) ./ (PI*t[i]);
        end
    end
    return formant;
end
```

Cette fonction définit un formant en racine de cosinus surélevé avec un coefficient β égal à 0. La réponse en fréquence de ce formant a une forme rectangulaire.

Pour vérifier que le formant fonctionne correctement, tapez[8] :

```
using PyPlot                # Une seule fois au lancement de Julia
include("../Commun/formantrect.jl");
SURECHANTILLONNAGE = 10;
TAILLE_FORMANT=10*SURECHANTILLONNAGE+1;
formant = formantrect(TAILLE_FORMANT,SURECHANTILLONNAGE);
x = collect(1:1:length(formant));
x = (x .- (length(formant)+1)/2) ./ SURECHANTILLONNAGE;
plot(x,formant)
```

7 La notation [A:B:C] compatible avec Matlab et Octave pour créer un vecteur existe sur la version 0.4.5, mais est marquée obsolète. La version 0.6.1 ne reconnaît plus cette notation pour créer un vecteur (la notation existe, mais produit autre-chose). La fonction équivalente sous Julia est : `collect(A:B:C)`.

8 Cette séquence assure d'afficher correctement l'échelle temporelle.

Vous devez obtenir une fenêtre graphique contenant la courbe suivante (Figure 1) :

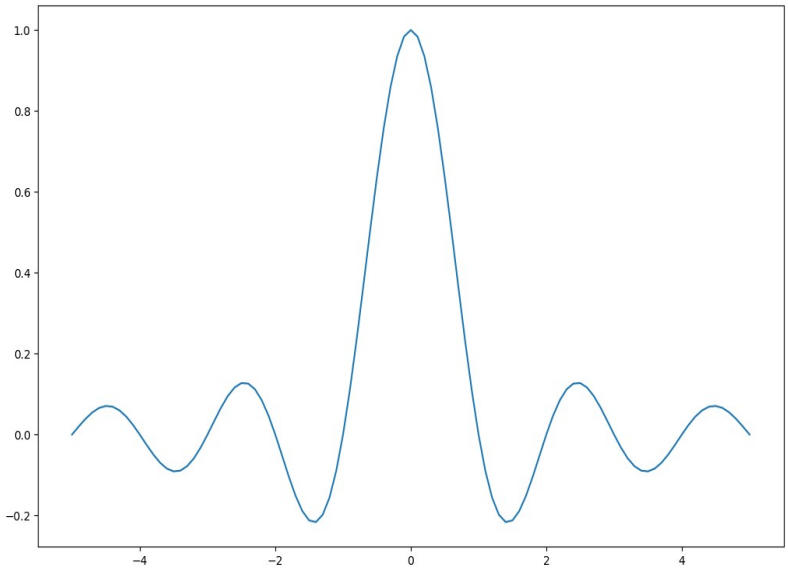

Figure 1 : représentation du formant rectangulaire

De même, le formant en racine de cosinus surélevé, avec β = 1 est défini dans le fichier suivant :

Fichier « **formantcos.jl** » :

```
function formantcos(TAILLE, SURECHANTILLONNAGE)
    PI = 3.1415926535;
    formant = zeros(TAILLE);
    t = (collect(1:1:TAILLE) .- floor((TAILLE+1)/2)) / SURECHANTILLONNAGE;
    for i=1:TAILLE
        if (t[i]*t[i]) == (1/16)
            formant[i] = PI / 4;
        else
            formant[i] = cos(2*PI*t[i]) ./ (1 - 16 * t[i] * t[i]);
        end;
    end
    return formant
end
```

De même, il est possible de vérifier le bon fonctionnement de ce formant par :

```
using PyPlot                # Une seule fois au lancement de Julia
include("../Commun/formantcos.jl");
SURECHANTILLONNAGE = 10;
TAILLE_FORMANT=10*SURECHANTILLONNAGE+1;
formant = formantcos(TAILLE_FORMANT,SURECHANTILLONNAGE);
x = collect(1:1:length(formant));
x = (x .- (length(formant)+1)/2) ./ SURECHANTILLONNAGE;
plot(x,formant)
```

Vous devez obtenir une fenêtre graphique contenant la courbe suivante (Figure 2) :

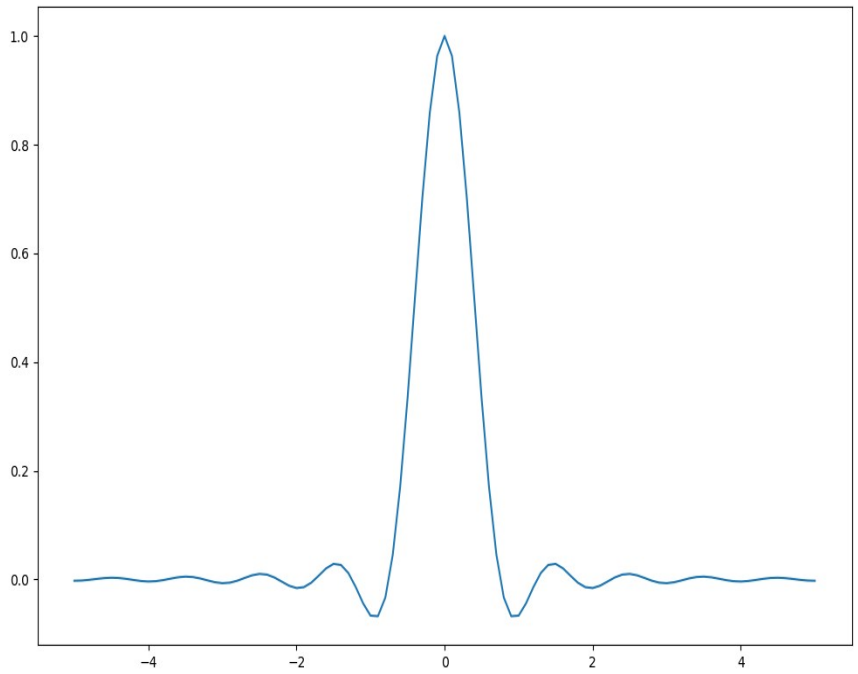

Figure 2 : représentation du formant cosinus

2.3.2 Émetteur

Nous convenons d'utiliser un émetteur binaire, c'est à dire fournissant des symboles sur deux niveaux. Les deux niveaux seront codés par leurs amplitudes respectives, à savoir -1 et +1. L'émetteur devra donc générer un message aléatoire contenant des symboles représentés par ces deux valeurs. L'instruction suivante permet de réaliser cela[9] :

```
TAILLE_MESSAGE = 1000;
message = 2 .* Int.(rand(TAILLE_MESSAGE) .> 0.5) .- 1;
```

Le message est un vecteur ligne contenant 1000 symboles prenant l'une des valeurs -1 ou +1.

Ce message doit être converti en analogique en utilisant un formant approprié. Afin de simuler une réalité suffisamment fine pour bien voir le signal analogique, il convient de sur-échantillonner le signal d'un facteur de sur-échantillonnage que nous nommerons « SURECHANTILLONNAGE ».

```
SURECHANTILLONNAGE = 30;
```

9 Le message binaire à transmettre n'est pas un signal analogique. La taille n'a pas besoin d'être impaire et le signal n'est pas centré temporellement sur 0.
À noter dans cet exemple : une fonction ou une conversion qui doit agir sur un tableau doit avoir un point ('.') après son nom. « Int.() » indique d'appliquer la fonction « Int() » à chacun des éléments du tableau.
De même, lorsqu'une opération applique un élément unique à un tableau, l'opérateur doit être précédé par un point. Un opérateur sans point indique la version matricielle et exige la compatibilité totale (addition/multiplication de vecteurs et/ou de matrices).

Le message sur-échantillonné donne lieu à un signal comportant des impulsions de Dirac de la manière suivante :

```
signal = zeros((TAILLE_MESSAGE-1)*SURECHANTILLONNAGE+1);
signal[1:SURECHANTILLONNAGE:end] = message;
```

Ce signal est convolué avec la réponse impulsionnelle du formant pour obtenir le signal à émettre :

```
TAILLE_FORMANT = 30;
include("../Commun/formantrect.jl");
using DSP                # Une seule fois au lancement de Julia
signal = conv(signal, formantrect(SURECHANTILLONNAGE*TAILLE_FORMANT+1, SURECHANTILLONNAGE));
```

Il est possible d'afficher la forme de ce signal, synchronisé sur les instants des symboles (le premier symbole ayant le numéro 0) par l'instruction suivante :

```
using PyPlot                    # Une seule fois au lancement de Julia
plot(collect(-TAILLE_FORMANT/2:1/SURECHANTILLONNAGE:TAILLE_FORMANT/2+TAILLE_MESSAGE-1), signal);
```

<u>À faire</u> : *écrire une fonction prenant en paramètre le message, le formant et le coefficient de sur-échantillonnage, qui génère le signal de sortie de l'émetteur. Cette fonction prendra le nom « emission » et devra être écrite dans le fichier script «emission.jl ».*

Test : pour tester votre fonction, entrer les instructions suivantes :

```
using PyPlot                    # Une seule fois au lancement de Julia
include("../Commun/emission.jl")
include("../Commun/formantrect.jl")
TAILLE_MESSAGE = 1000;
SURECHANTILLONNAGE = 30;
TAILLE_FORMANT = 30;
message = 2 .* Int.(rand(TAILLE_MESSAGE) .> 0.5) .- 1;
formant = formantrect(SURECHANTILLONNAGE*TAILLE_FORMANT+1, SURECHANTILLONNAGE);
signal = emission(message, formant, SURECHANTILLONNAGE);
plot(collect(-TAILLE_FORMANT/2:1/SURECHANTILLONNAGE:TAILLE_FORMANT/2+TAILLE_MESSAGE-1), signal);
```

Le résultat doit être similaire[10] à celui obtenu ci-dessus.

2.3.3 Récepteur

Le récepteur de la chaîne de transmission idéale utilise comme filtre de récepteur le formant retourné dans le temps. Il s'agit donc de convoluer le signal reçu par ce filtre, puis d'échantillonner le résultat aux instants des temps d'horloge. Bien entendu, il faudra correctement synchroniser le récepteur, afin que les instants d'échantillonnages soient bien placés.

10 Dans la mesure où une partie du signal est généré de manière aléatoire, le fait de relancer une manipulation donnera bien entendu un résultat différent. Il s'agit ici de vérifier que la forme de la sortie obtenue ressemble à celle obtenue précédemment selon des critères qu'on pourra exprimer.

À faire : *écrire une fonction « reception » dans le fichier script « reception.jl » qui prendra en entrée le signal reçu, le filtre du récepteur et le facteur de sur-échantillonnage, ainsi qu'un paramètre de synchronisation qui indique, dans le signal d'entrée, le numéro de la position du premier échantillon. La fonction devra renvoyer le message décodé. Note : le message décodé pourra éventuellement contenir des symboles supplémentaires à la fin, qui devront, en principe, être tous nuls.*

Note : on fera les tests avec les deux formants.

Note : *le paramètre de synchronisation est défini tel que sa valeur indique la position de l'échantillon correspondant à l'instant* $t=0$ *dans le signal fourni à l'entrée du récepteur. Rappelons que dans Julia, les éléments sont numérotés à partir de 1. Ainsi, si le premier échantillon correspond à l'instant* $t=0$ *, le paramètre de synchronisation doit être égal à 1.*

Test : la séquence suivante émet un message, le reçoit, compte le nombre d'erreurs et affiche ce nombre. Le résultat doit être 0. Note : justifiez la ligne qui fournit le filtre du récepteur.

```
using PyPlot                  # Une seule fois au lancement de Julia
include("../Commun/reception.jl")
include("../Commun/emission.jl")
include("../Commun/formantrect.jl")
TAILLE_MESSAGE = 1000;
SURECHANTILLONNAGE = 30;
TAILLE_FORMANT = 100;
message = 2.0 .* Int.(rand(TAILLE_MESSAGE) .> 0.5) .- 1;
formant = formantrect(SURECHANTILLONNAGE*TAILLE_FORMANT+1, SURECHANTILLONNAGE);
signal = emission(message, formant, SURECHANTILLONNAGE);
filtre = formant[end:-1:1] / (formant'*formant);
filtre = filtre[1:end,1];
recu = reception(signal, filtre, SURECHANTILLONNAGE, 1+TAILLE_FORMANT*SURECHANTILLONNAGE/2);
sum(abs.(recu-message)/2)
```

Note : on pourra effectuer des manipulations intermédiaires en vue de vérifier le bon fonctionnement de la fonction.

2.3.4 Ajout d'un bruit de canal

La chaîne de transmission idéale est le siège d'un bruit gaussien additif qui manifeste sa présence à l'entrée du récepteur. On convient de définir la puissance de ce bruit par le rapport E_b/N_0. Nous allons donc simuler une présence de bruit, calibrée par ce paramètre.

D'après [3], page 20, un bruit blanc gaussien additif de densité spectrale bilatérale N_0 produit, aux instants d'échantillonnages, des valeurs caractérisées par une espérance nulle et une variance de $N_0/2$.

Il nous faut donc estimer la valeur de N_0. Ce paramètre est déterminé simplement par :

$$N_0 = \frac{E_b}{E_b/N_0}$$

ou encore, si le rapport E_b/N_0 est exprimé en dB :

$$N_0 = \frac{E_b}{10^{\frac{(E_b/N_0)_{dB}}{10}}}$$

Le paramètre E_b lui-même est l'énergie par bit. Dans le cas d'une chaîne de transmission idéale respectant donc le critère de Nyquist, il s'agit de l'énergie du formant multipliée par la moyenne des carrés des symboles émis. En l'occurrence, les symboles émis valant toujours -1 ou 1, ce paramètre vaut donc l'énergie du formant, à savoir :

```
Eb = formant' * formant;
```

À faire : *écrire une fonction « bruit » dans le fichier « bruit.jl » prenant comme paramètre la valeur du rapport E_b/N_0 exprimé en dB, la valeur de l'énergie par bit (E_b) et la taille du vecteur de bruit souhaité. La fonction renvoie le vecteur de bruit à rajouter au signal avant passage dans le récepteur.*

Note : selon les estimations des questions précédentes, l'un des formants se comporte mieux que l'autre. On utilisera pour la suite des opérations le formant qui a le meilleur comportement.

Rappelons que la fonction « randn » génère des nombres aléatoires selon une distribution normale d'espérance nulle et de variance 1.

Test : vérifier que le taux d'erreur devient non nul lorsque le niveau de bruit augmente :

```
include("../Commun/bruit.jl")
include("../Commun/formantcos.jl")
include("../Commun/emission.jl")
include("../Commun/reception.jl")
TAILLE_MESSAGE = 1000;
SURECHANTILLONNAGE = 30;
TAILLE_FORMANT = 100;
message = 2 .* Int.(rand(TAILLE_MESSAGE) .> 0.5) .- 1;
formant = formantcos(SURECHANTILLONNAGE*TAILLE_FORMANT+1, SURECHANTILLONNAGE);
signal = emission(message, formant, SURECHANTILLONNAGE);
signal = signal + bruit(5, (formant'*formant)[1], size(signal,1));
filtre = formant[end:-1:1] / (formant'*formant);
filtre = filtre[1:end,1];
recu = reception(signal, filtre, SURECHANTILLONNAGE, 1+TAILLE_FORMANT*SURECHANTILLONNAGE/2);
sum(abs.(recu-message)/2)
```

Le résultat doit être non nul. Bien entendu, comme le bruit est aléatoire, plusieurs simulations successives peuvent donner des résultats différents.

2.4 Courbes de taux d'erreur

2.4.1 Taux d'erreur théorique de la chaîne de transmission idéale

Le taux d'erreur binaire de la chaîne de transmission idéale est donnée dans le cas général par la

formule 1.4 page 14 du polycopié « Récepteur numérique ». Dans le cas binaire, cette formule devient :

$$TEB = \frac{1}{2} * erfc\left(\sqrt{\frac{E_b}{N_0}}\right)$$

À faire : *écrire un script « courbe_ideale.jl » qui réalise le tracé de cette courbe pour des rapports E_b/N_0 variant (en dB) de 0 à 5.*

2.4.2 Taux d'erreur simulé de la chaîne de transmission idéale

Le principe des simulations précédentes (paragraphe 2.3) permet d'obtenir le taux d'erreur de la chaîne de transmission en fonction du rapport E_b/N_0.

À faire : *écrire une fonction « erreur » dans le fichier « erreur.jl » qui réalise la simulation et fournit le taux d'erreur pour un rapport E_b/N_0 donné. La fonction prendra en entrée un rapport E_b/N_0, une taille de message (nombre de symboles à émettre), un facteur de sur-échantillonnage, un formant utilisé pour l'émission et un formant (filtre) utilisé pour la réception.*

Écrire ensuite un script « courbe_ideale_simulee.jl » qui réalise le tracé de la courbe de taux d'erreur. Notons que le résultat de la simulation dépend du tirage aléatoire de certaines valeurs. Le résultat peut donc varier selon ce tirage. Il est conseillé, pour chaque point, d'effectuer une dizaine de calculs et de tracer les deux courbes correspondant au minimum et au maximum afin de déterminer un ordre de grandeur de la précision de la simulation elle-même.

Note : afin d'éviter des temps de calculs trop long, on utilisera une taille de formant correspondant à 30 période d'horloge numérique. Le nombre de points du message devra être suffisamment grand pour obtenir une simulation précise, par exemple 10000. La simulation se fera pour des rapports Eb/N0 entre 0 et 8, par pas de 0,5.

3 Chaîne de transmission non idéale

3.1 Introduction

Nous introduisons ici la notion d'un canal de transmission non idéal, c'est à dire un canal de transmission qui déforme le signal par une opération de filtrage linéaire.

Cette notion est abordée dans [3] à partir de la page 36 (*1.3.2 Transmission sur un canal à bande limitée*). Le filtre du récepteur doit être adapté, non seulement au formant utilisé par l'émetteur, mais également à la déformation (filtrage) introduite par le canal.

Dans le cas général, si le comportement du canal est connu, une réalisation simple permet d'obtenir un récepteur dit « sous-optimal », car la réalisation du récepteur optimal est plus complexe. Cette réalisation simple est décrite en [3] à partir de la page 54 (*1.4 Récepteurs sous-optimaux*) et donne lieu à deux réalisations possibles qui sont résumées à la page 64. Ces deux réalisations sont :

- récepteur sous-optimal avec suppression de l'interférence entre symboles,
- récepteur sous-optimal sans suppression de l'interférence entre symboles.

Notons que dans le cas d'une chaîne de transmission idéale, la première réalisation est strictement équivalente à l'expression habituellement utilisée pour la chaîne de transmission idéale. Rappelons que le paramètre $G(f)$ utilisé dans ces équations est la combinaison du formant à l'émission et de la réponse du canal[11].

3.2 Introduction d'une déformation par le canal

Nous allons introduire une déformation du signal par le canal entre la sortie de l'émetteur et l'ajout du bruit à l'entrée du récepteur. À cet effet, il faudra ajouter, dans la fonction « erreur » la ligne suivante :

```
using DSP;
signal = conv(signal, canal);
```

Note : dans cette simulation, le récepteur n'est pas modifié. Autrement dit, le filtre utilisé est toujours le même, à savoir le formant de l'émetteur retourné dans le temps.

À faire : *créer une fonction « erreur_canal » dans le fichier « erreur_canal.jl » qui reprend la fonction « erreur » du fichier « erreur.jl » en ajoutant la ligne ci-dessus. Cette fonction prendra un paramètre supplémentaire « canal » afin de recevoir les caractéristiques du canal.*

Écrire dans le fichier « courbe_canal_simulee.jl » le script « courbe_canal_simulee » qui est une modification de la fonction « courbe_ideale_simulee » permettant de prendre en compte le comportement du canal. Ce script devra utiliser un canal dont le comportement temporel est celui de la figure 3, ci-dessous.

11 Dans le cas de la chaîne de transmission idéale, la réponse du canal est égale à 1, et donc *G(f)* devient le formant de l'émetteur.

Remarque : la déformation du signal par le canal implique deux modifications dans la fonction « erreur_canal » :

- l'énergie par bit n'est plus donnée par l'énergie du formant ; le calcul de cette énergie doit être effectué globalement sur le signal transmis (et modifier en conséquence le deuxième paramètre de la fonction « bruit » lors de son appel depuis cette fonction) ;
- le coefficient de synchronisation pour la fonction « réception » est modifié (prendre en compte la taille de la réponse impulsionnelle du canal).

Afin de simplifier le problème, on effectuera les tests avec une réponse impulsionnelle du canal sous la forme d'un filtre passe-bas du premier ordre, de constante de temps la période d'échantillonnage des symboles. Le tracé de cette réponse pour un coefficient de sur-échantillonnage quelconque se trouve à la figure 3, ci-dessous.

Si vous avez créé un fichier « canal.jl » pour une fonction « canal » prenant comme paramètre la taille du canal (en nombre de points) et le coefficient du sur-échantillonnage, le code ci-dessous permet d'afficher cette courbe :

```
using PyPlot
include("canal.jl");
TAILLE_CANAL=10;
SURECHANTILLONNAGE=10;
lecanal = canal(TAILLE_CANAL*SURECHANTILLONNAGE+1, SURECHANTILLONNAGE);
x=collect(0:TAILLE_CANAL*SURECHANTILLONNAGE);
x=(x.-TAILLE_CANAL*SURECHANTILLONNAGE/2)/SURECHANTILLONNAGE;
plot(x, lecanal)
```

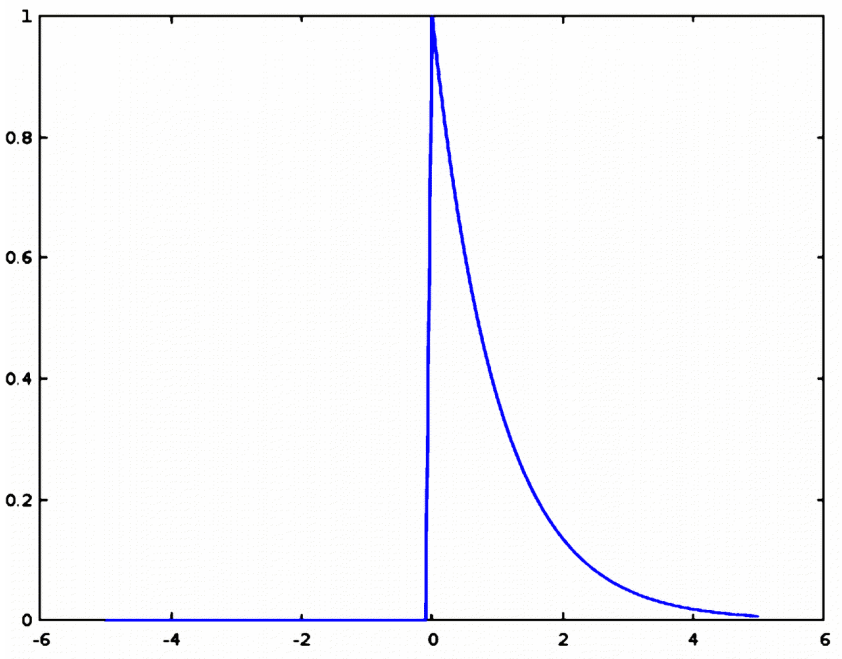

Figure 3 : réponse attendue du canal

Note : cette réponse est une exponentielle décroissante de constante de temps $\tau = 1$.

Dans un premier temps, la fonction « erreur_canal » pourra être testée avec une réponse impulsionnelle du canal sous la forme d'une impulsion de Dirac (`canal = 1`), afin de vérifier qu'on retrouve bien les mêmes résultats que précédemment. Effectuer ceci dans un premier script nommé « courbe_canal_simulee_test1.jl ». Le résultat obtenu doit être similaire à celui de la question précédente.

Dans un second temps, la vérification consiste à fournir comme canal un vecteur d'une certaine taille pour lequel seul le coefficient du milieu est non nul et vaut 1. Cela teste le positionnement correct de la synchronisation. Faire ceci dans le script « courbe_canal_simulee_test2.jl ».

Dans un troisième temps, la vérification assure qu'une modification de l'amplitude provoquée par le canal ne modifie pas les résultats, en fixant par exemple, l'amplitude du canal à 10. Les résultats obtenus doivent toujours être les mêmes. Nommer ce script : « courbe_canal_simulee_test3.jl ».

Enfin, mettre en œuvre la version définitive sous le nom « courbe_canal_simulee.jl ». Cette version utilise le canal correspondant à la figure 3.

3.3 Récepteur adapté au canal déformé

3.3.1 Introduction

Il y a deux points importants à prendre en compte lorsque le canal introduit une déformation du signal :

- le filtre adapté de chaîne de transmission idéale s'écrivait $\lambda\overline{G(f)}$: il s'agissait dans le cas d'une chaîne de transmission idéale du formant à l'émission $G(f)$ qui servait à déterminer le filtre adapté du récepteur ; dans le cas où le canal introduit une distorsion, $G(f)$ ne représente plus seulement le formant à l'émission, mais représente le formant à l'émission combiné avec la distorsion du canal ;
- dans le cas où l'ensemble « formant + canal » respecte toujours le critère de Nyquist, le filtre adapté du récepteur prend toujours la forme $\lambda\overline{G(f)}$. Cependant, dans le cas général, cela n'est pas respecté. Le récepteur optimal n'est plus le récepteur de la chaîne de transmission idéale, mais devient plus complexe, comme cela est expliqué à partir de la page 36 du polycopié [3]. Il existe cependant deux récepteurs linéaires sous-optimaux relativement faciles à mettre en œuvre (page 54 et résumés à la page 64) :
  → le récepteur avec élimination de l'interférence entre symboles
  → le récepteur sans élimination de l'interférence entre symboles.

3.3.2 Récepteur naïf avec filtre adapté de la chaîne de transmission idéale

Dans cette première expérimentation, nous allons faire comme si le canal introduisait une distorsion qui ne met pas en péril le critère de Nyquist. En d'autres termes, nous allons faire comme si le canal faisait en sorte que la chaîne de transmission reste idéale.
Le point à prendre en compte ici est que le filtre adapté du récepteur, même dans le cas d'un système idéal, doit prendre en compte l'effet du canal. Le filtre adapté du récepteur se met sous la forme $\overline{G(f)}$, où $G(f)$ ne représente plus seulement le formant de l'émetteur, mais inclut également la réponse du canal.

À faire : *Modifier le script « `courbe_canal_simulee.jl` » en « `courbe_canal_simulee_naive.jl` » pour ce cas de figure. Comparez les performances obtenues aux cas traités précédemment.*

3.3.3 Récepteur avec élimination des interférences entre symboles

À faire : *Modifier le script « `courbe_canal_simulee.jl` » en « `courbe_canal_simulee_sansinterferences.jl` » pour ce cas de figure.*

Utiliser la formule du filtre qui se trouve dans le résumé à la page 64 du polycopié [3].

Note : le filtre inverse qui est à calculer dans cette formule a une réponse impulsionnelle infinie. Il est probable qu'il faille augmenter la durée temporelle avant d'effectuer l'inversion. Pensez à rendre votre simulation compatible avec ce point.

3.3.4 Récepteur sans élimination des interférences entre symboles

À faire : *Modifier le script « `courbe_canal_simulee.jl` » en « `courbe_canal_simulee_avecinterferences.jl` » pour ce cas de figure.*

Utiliser la formule du filtre qui se trouve dans le résumé à la page 64 du polycopié [3].

4 Égalisation : récepteur à filtre adaptatif

4.1 Introduction

Nous nous trouvons maintenant dans une situation encore plus difficile que la situation précédente : non seulement, le canal n'est pas un canal de transmission idéal (il provoque donc de l'interférence entre symboles), mais en plus la réponse du canal n'est pas connue[12]. Le récepteur doit alors s'adapter automatiquement à la réponse du canal. C'est l'objet du filtre adaptatif.

Comme indiqué dans le polycopié à la page 74 du polycopié, le filtre adaptatif peut être utilisé dans deux modes :

- mode d'apprentissage : une séquence connue est émise par l'émetteur et le récepteur en profite pour régler correctement son filtre ;
- mode poursuite : le réglage du filtre se poursuit alors que l'émetteur transmet le vrai message qui n'est donc pas connu du récepteur.

4.2 Filtre adaptatif

Ce premier exercice permet de se faire la main sur le filtre adaptatif lui-même, sans prendre en compte les problématiques de la transmission numérique. Nous allons simuler un système dans lequel nous demandons au filtre de s'adapter automatiquement au fait de fournir en sortie un signal *y* lorsqu'on lui place en entrée en signal *x*.

À faire : *Mettre en œuvre les deux algorithmes (méthode du gradient stochastique et méthode des moindres carrés) dont le résumé se trouve à la page 104 du polycopié. En déduire les performances attendus de ce type de méthodes. On prendra pour ces tests un signal d'entrée « x » aléatoire normal et blanc. Le signal « y » sera la sortie d'un filtre de réponse impulsionnelle donnée qui prend en entrée le signal « x ». Dans un premier temps, utiliser l'impulsion de Direct ([1]), puis le filtre [1 2 1] / 4 .*

Tracer la courbe d'erreur en fonction du nombre de points du signal (convergence). En déduire la vitesse de convergence et optimiser le paramètre du filtre pour obtenir la meilleure vitesse de convergence possible.

Observer la forme du filtre H obtenu après convergence.

12 Deux raisons peuvent conduire à cela : le fait que le canal n'est tout simplement pas connu à la conception du système, ou le fait que la réponse du canal puisse varier au cours du temps. Dans le second cas, particulièrement connu pour la transmission radio mobile, le récepteur doit s'adapter en permanence aux variations du comportement du canal.

Note : pour réaliser un filtre, il faut écrire deux fonctions. Par exemple, pour le filtre gradient, les deux fonctions seront les suivantes :

- `function adaptatif_gradient_init (taille, mu)::Filtre_gradient`
- `function adaptatif_gradient(x, y, filtre::Filtre_gradient)::Float64`

La première fonction initialise le filtre et fournit une variable de type « filtre_gradient » qui contient l'état du filtre.

La seconde fonction réalise le calcul effectif du fonctionnement du filtre, pas à pas. Un appel consiste donc à prendre un échantillon en compte en entrée et à fournir la valeur de l'erreur en sortie. Il est nécessaire de conserver l'état du filtre entre deux appels de la seconde fonction, puisqu'à chaque pas de calcul, l'état du filtre est modifié. Cela est effectué en Julia en déclarant le type du filtre « mutable ». Ainsi, à chaque appel de la fonction d'adaptation, le filtre est mis à jour.

Note : sous Julia, contrairement à Matlab, il est possible de définir les deux fonctions dans un seul et même fichier nommé « adaptatif_gradient.jl ».

De même, pour le filtre utilisant la méthode des moindres carrés, les deux fonctions sont :

- `function adaptatif_carre_init (taille, w)::Filtre_carre`
- `function adaptatif_carre(x, y, filtre::Filtre_carre)::Float64`

On pourra alors tester ces filtres avec les deux fonctions suivantes (à écrire) :

- `function test_gradient(taille_filtre, taille_signal, mu, filtre)`
- `function test_carre(taille_filtre, taille_signal, w, filtre)`

Les paramètres sont les suivants :

- taille_filtre : taille pour le filtre adaptatif.
- taille_signal : taille du signal à tester. Il faudra générer un signal aléatoire dans la fonction pour envoyer ce signal dans le filtre.
- mu ou w : les paramètres de réglage de la convergence respectivement pour le filtre gradient et pour le filtre à moindres carrés.
- filtre : le filtre canal à tester. Pour ce test, on passera comme filtre la valeur : « [1 2 1] ./ 4 ».

Le résultat de ces fonctions, sous le nom « convergence » est la courbe de l'évolution de l'erreur au cours du temps. Cette courbe doit tendre vers 0. *Note : vous pouvez faire afficher à l'écran le filtre obtenu après convergence. Il doit être proche du filtre passé en paramètre.*

4.3 Chaîne de transmission avec égalisation

4.3.1 Introduction

On utilisera les routines des exercices précédents permettant d'obtenir les filtres. Notons que ces routines fournissent le filtre retourné dans le temps, car l'expression qui intervient dans les formules est un produit scalaire à la place du produit de convolution. En tenir compte lors de l'utilisation des filtres dans les exercices qui suivent.

4.3.2 Récepteur sous-optimal adaptatif à compensation du canal

Nous reprenons la déformation du canal introduite en 3.2, mais nous supposons maintenant que le récepteur ne connaît pas cette déformation. L'utilisation du filtre envisagée dans les paragraphes 3.3.3 et 3.3.4 n'est plus possible.

Il nous faut donc utiliser un filtre adaptatif avec le principe suivant :

- transmission, dans un premier temps, d'une séquence d'apprentissage connue ; cela permet à l'algorithme de déterminer le filtre ;
- transmission du message et utilisation du filtre déterminé précédemment.

Remarque importante : pour les courbes de taux d'erreur, il est important de refaire la détermination du filtre pour chaque point de calcul. En effet, la précision avec laquelle le filtre est obtenu dépend également du niveau de bruit.

Deux structures sont proposées à la page 74 du polycopié. On se propose d'utiliser la première qui est représentée à la figure 2.4 (un seul filtre adaptatif). Notons que dans cette structure, le filtre se trouve avant la décimation (échantillonnage des symboles). Cela est rendu nécessaire par le fait que le filtre envisagé précédemment aux paragraphes 3.3.3 ou 3.3.4 de cet énoncé et représenté à la page 55 du polycopié intègre dans la partie analogique ($\overline{g(f)}$) une composante du filtre du canal. Le filtre adaptatif ne peut donc pas agir uniquement sur les échantillons au rythme des symboles.

A faire : *Établir une chaîne de transmission fonctionnant sur ce principe. Note : du fait de l'augmentation de la taille du filtre, les temps de calculs peuvent devenir dissuasifs. Cela se produit particulièrement pour le filtre utilisant la méthode des moindres carrés. On ne fera donc les essais que sur le filtre adaptatif selon la méthode du gradient.*

4.3.3 Récepteur sous-optimal à suppression d'interférence entre symboles

Le récepteur envisagé au paragraphe précédent a comme principal inconvénient de nécessiter un filtre ayant beaucoup de coefficients, car le filtre se trouve dans la zone « analogique », donc numériquement sur-échantillonné. Le temps de calcul devient donc très long.

En plaçant le filtre après l'échantillonnage, on espère supprimer l'interférence entre symboles dans la partie « numérique » du récepteur, c'est à dire non sur-échantillonnée. Le nombre de coefficients du filtre est alors bien plus faible.

A faire : *Établir une chaîne de transmission fonctionnant sur ce principe. Note : les calculs sont beaucoup plus rapides que dans le cas précédent. Il est possible d'envisager de faire cette réalisation avec le filtre adaptatif par la méthode du gradient et aussi par la méthode des moindres carrés.*

5 Synchronisation

5.1 Introduction

On s'intéresse ici à la problématique de synchronisation de l'horloge. Si l'horloge du récepteur n'est pas synchronisée correctement dans le temps avec l'horloge de l'émetteur, le taux d'erreur augmente. Nous allons successivement :

- Mettre de phénomène en évidence
- Rechercher une solution afin d'assurer la synchronisation.

5.2 Mise en évidence de la nécessité de synchroniser correctement le récepteur

A faire : *Pour la chaîne de transmission idéale, tracer le taux d'erreur en fonction de la désynchronisation, en l'absence de bruit. Interpréter le résultat obtenu. Tracer la même courbe en présence de bruit.*

5.3 Chaîne de transmission avec erreur d'horloge.

Afin de tester l'efficacité d'un dispositif de synchronisation, il nous fait tout d'abord simuler une chaîne de transmission dans laquelle il se produit une erreur d'horloge. Cette erreur d'horloge est aléatoire et variable au cours du temps.

Les variables des fréquences d'horloge étant souvent lentes au cours du temps, nous pouvons simuler une simple erreur de fréquence d'horloge. Nous supposons donc que les rythmes d'horloge de l'émetteur et du récepteur ne sont pas les mêmes, mais que ces rythmes restent constants durant le temps de transmission du paquet.

Note : cet exercice nécessite de réaliser une interpolation d'un signal. À cet effet, on peut utiliser le package de Julia prévu à cet effet, en faisant (une seule fois pour une personne donnée) :

```
import Pkg
Pkg.add("Interpolations")
```

La description de ce package est accessible en [6].

A faire : *Pour la chaîne de transmission idéale, faire en sorte que les horloges de l'émetteur et du récepteur diffèrent d'un rapport relatif qu'on pourra spécifier en paramètre, et tracer la courbe du taux d'erreur obtenu en fonction du rapport* E_b/N_0 *, pour différentes valeurs de ce paramètre.*

5.4 Correction de l'erreur d'horloge

Le but de cette correction est de déterminer automatiquement l'erreur d'horloge et de la compenser. Il est bien entendu que le récepteur ne connaît pas cette erreur et doit la déterminer par lui-même.

A faire : *Ecrire un script qui effectue cela et tracer la courbe de taux d'erreur obtenue.*

Note : dans le cadre de cet énoncé, on peut utiliser la version simplifiée de la récupération de rythme seul, décrite à partir de la page 122 de [3]. Par exemple, le retour de décision décrit à partir de la page 123 au paragraphe 3.3.3.3.2 peut être utilisé.

6 Notions de Julia, fonctions utiles

6.1 Scalaires, vecteurs, matrices

Sous Julia, il est possible de définir des éléments scalaires, vecteurs ou matrice. Les opérations possibles sur ces éléments sont les opérations habituelles comme l'addition, la soustraction, la multiplication et la division selon la définition mathématique de ces opérations.

Exemple :

```
A = 10                          # Est un scalaire
B = [ 1 2 3 4 5 6 ]             # Est un vecteur horizontal
C = [ 1; 2; 3; 4; 5 ]           # Est un vecteur vertical
M = [ 1 2 ; 3 4 ]               # Est une matrice 2 par 2
```

Pour multiplier une matrice par un vecteur : exemple :

```
M = [ 1 2 ; 3 4 ]     # M est une matrice 2 par 2
V = [ 5 ; 6]          # V est un vecteur vertical à 2 composantes
M * V                 # M * V est un vecteur vertical à 2 composantes
```

Les opérations sont possibles si elles sont conformes à la définition mathématique habituelle de ces opérations (multiplication de deux matrices de dimensions compatibles, multiplication d'une matrice par un vecteur de dimension compatible…).

Note : il est possible de transposer un vecteur ou une matrice par :

```
X = Y'                # X est le transposé de Y
```

On peut utilise cela pour écrire le produit scalaire de deux vecteurs verticaux compatibles :

```
P = X' * Y            # Est le produit scalaire des deux vecteurs
```

6.2 Opération élément par élément

Dans certains cas, on souhaite effectuer des opérations élément par élément. Par exemple, si on considère deux vecteurs horizontaux de dimension 2 :

```
X = [ 1 2 ]
Y = [ 3 4 ]
```

L'addition `X + Y` produit ce qu'on attend classiquement de l'addition de deux vecteurs.

Mais la multiplication `X * Y` n'est pas une opération mathématique possible (en terme de multiplication de matrices, deux vecteurs horizontaux sont incompatibles). Si on souhaite effectuer l'opération élément par élément et obtenir un vecteur horizontal de dimension 2 en sortie (comme ce qui est fait pour l'addition), on écrira : `X .* Y`

Notons que « `X .+ Y` » produit dans ce cas le même résultat que « `X + Y` ».

Pour obtenir une opération élément par élément sur deux objets (vecteur ou matrice) de même taille, on pourra donc utiliser l'un des opérateurs existants, précédé d'un point (« . »). Le résultat est un objet vecteur ou matrice de même taille.

6.3 Opération élément par scalaire

Une autre situation classique est l'opération élément par scalaire. Par exemple, la multiplication d'un vecteur ou d'une matrice par un scalaire produit un vecteur ou une matrice de même dimension que l'original, dont chaque élément est multiplié par le scalaire.

Considérons `X = [ 1 2 ]`

L'opération « `2 * X` » produit ce qui est habituellement attendu.

Il est aussi possible de demander une addition d'un scalaire. Dans ce cas, il faudra écrire « `2 .+ X` », car l'opération « `2 + X` » n'est pas possible.

Notons que « `2 .* X` » produit le même résultat que « `2 * X` ».

Pour obtenir une opération scalaire par élément sur un objet vecteur ou matrice, on pourra donc utiliser l'un des opérateurs existants, précédé d'un point (« . »). Le résultat est un objet vecteur ou matrice de même taille.

6.4 Fonctions

Les fonctions s'appliquent à des scalaires.

Par exemple :

```
X = cos( 3 )
```

Pour appliquer une fonction à un vecteur ou à une matrice, élément par élément, il faut faire suivre le nom de la fonction par un point (« . »). Le résultat sera un vecteur ou une matrice de même taille.

Par exemple :

```
Y = cos.( [ 1 2 3 4 ] )
```

6.5 Packages

Toutes les possibilités de Julia ne sont pas disponibles au lancement. Il faudra parfois préciser d'utiliser un package comme dans l'exemple :

```
using DSP       # Fournit la fonction « conv ».
```

Ceci devra être fait à chaque lancement du programme. Il donc conseillé de l'écrire au début des

scripts qui utilisent ces fonctions afin de garantir que la fonction demandée a bien été chargée.

En outre, tous les packages ne sont pas installés lors de l'installation du Julia. Chaque utilisateur devra émettre, après installation de Julia, la suite de commande ci-dessous. Le package est alors installé sur son compte et il ne sera pas nécessaire de refaire ces commandes à chaque lancement de Julia :

```
import Pkg; Pkg.add("DSP")
```

6.6 Fonctions utiles

Les fonctions ci-dessous sont utiles dans le cadre de ce cours :

Fonction	Description	Package
exit()	Quitte Julia	
zeros(n)	Génère un vecteur vertical de n valeurs nulles	
zeros(m,n)	Génère une matrice de m par n valeurs nulles (m lignes, n colonnes)	
collect(x:y:z)	Génère un vecteur vertical dont la première valeur est x, les valeurs suivantes sont à chaque fois augmentées de y jusqu'à atteindre z (la dernière valeur est inférieure ou égale à z).	
floor(x)	Calcule la partie entière mathématique de x	
plot(x,y)	Dessine une courbe avec x en abscisse et y en ordonnée. x et y sont deux vecteurs verticaux de même dimension.	PyPlot
cos(x)	Calcule le cosinus de x.	
rand(n)	Fourni un vecteur vertical contenant n valeurs aléatoires selon une répartition uniforme entre 0 et 1.	
randn(n)	Fourni un vecteur vertical contenant n valeurs aléatoires selon une loi gaussienne centrée et de variance 1.	
length(x)	x étant un vecteur ou une matrice, fournit le nombre total d'éléments de x.	
conv(x,y)	x et y étant deux vecteurs verticaux, fournit la convolution de ces deux vecteurs. Note : contrairement à Matlab ou Octave, la fonction ne prend que les vecteurs verticaux.	DSP
abs(x)	Valeur absolue de x	
sum(x)	x étant un vecteur ou une matrice, calcule la somme de ses éléments.	
FFT	Calcule la transformée de Fourier d'un vecteur ou d'une matrice.	FFTW

diagm(x)	x étant un vecteur vertical de m éléments, produit une matrice diagonale de m par m en plaçant les m éléments du vecteur sur la diagonale.	LinearAlgebra
@printf	Accès à la fonction « printf » du langage C.	Printf

7 Bibliographie

[1] Site de téléchargement de Julia : https://julialang.org/downloads/

[2] Mode d'emploi du logiciel Julia : https://docs.julialang.org/en/v1/

[3] Polycopié du cours : Récepteur numérique, Jean-Louis Gutzwiller, 2008

[4] Polycopié du cours : Signal et communication, Jean-Claude Dany, Jean-Louis Gutzwiller, Lionel Husson, Pierre Leray, Armelle Wautier, 2012

[5] Tracer des courbes en Julia : https://julialang.org/downloads/plotting.html

[6] Package « Interpolations » pour Julia : https://github.com/JuliaMath/Interpolations.jl

[7] Mode d’emploi de PyPlot : https://matplotlib.org/users/pyplot_tutorial.html

[8] Interface Julia vers PyPlot : https://github.com/JuliaPy/PyPlot.jl

[9] Rechercher des packages pour Julia (où se trouve telle ou telle fonction?) : https://pkg.julialang.org/docs/