Séance 2: programmation de filtres IIR en virgule fixe, application du *noise-shaping*

Lors de cette séance, on va présenter une technique permettant d'améliorer les performances des filtres iir codés sur des nombres entiers (en virgule fixe). Le point de départ sera le programme de filtrage que vous avez rédigé lors du travail préparatoire, sur lequel sera appliquée cette technique...

1 test des performances sous netbeans

l'objectif de cette partie est de tester le fonctionnement de votre programme, en le comparant au filtre idéal, qui serait généré avec des nombres réels.

☐ sous netbeans, créer un nouveau fichier source ***my_iir.c***, et le header correspondant ***my_iir.h***. Ces fichiers contiendront respectivement l'implémentation et la déclaration des structures et fonctions correspondant au filtre *iir* codé dans la partie 4 du travail préparatoire, sous forme directe 2.

1.1 écriture des structures et fonctions

1.1.1 intégration du filtre iir idéal

dans le fichier *my_iir.c* , ajouter la structure et les fonctions permettant de simuler le filtre *iir* idéal (codé avec des réels double précision, sous forme directe 2). Pour mettre en évidence le seul effet des bruits de quantification de signal, on codera le filtre iir idéal en employant les coefficients quantifiés a_{iq}, b_{iq} , au lieu des coefficients réels a_i, b_i . (au moins 5 décimales par coefficient svp). Plus précisément, compléter le programme ci-après et l'insérer dans le fichier ***my_iir.c:***

```
/* structure correspondant au filtre iir ideal
 tous les coeffs et memoires nécessaires a la programmation du filtre iir
  doivent être codés dans cette structure */
typedef struct{
  double a1; /* à compléter , déclarer les autres coeffs et variables nécessaires*/
} s_my_iir_ideal;
typedef s_my_iir_ideal *p_my_iir_ideal;
/* fonction d'initialisation du filtre iir ideal , reserve memoire, init coeffs /
p_my_iir_ideal new_my_iir_ideal(double a1,...) {
    p_my_iir_ideal p_iir;
    p_iir=malloc ( sizeof(?)) ; /* a completer, reservation memoire */
    p_iir->a1= ?; /* a completer initialisation des coeffs de la structure */
    p_iir->x_n_1= ?; /* a completer, initialisation des variables passees de la structure */
}
/* fonction de destruction du filtre iir ideal , libere memoire et ressources */
void destroy_my_iir_ideal( p_my_iir_ideal p_iir) {
    free(p_iir->?) ; /* a completer eventuellement, liberation memoires internes si precedemment
reservees */
    free(?) ; /* a completer, liberation memoire de la structure */
}
/* fonction d'implementation du filtre iir ideal
calcule la nouvelle sortie sn, en fonction de la nouvelle entree en,
et des coeffs et variables passees contenus dans le pointeur de structure
p_iir /
double  one_step_my_iir_ideal( double en, p_my_iir_ideal p_iir) {
    double sn; /* variable interne correspondant a la nouvelle sortie */
/* calcul de sn, a effectuer */
    return(sn); /* renvoie la nouvelle sortie */
}
```

❒copier (dupliquer, ne pas détruire) la fonction *teste_fir* du fichier fichier *fir.c* vers le fichier *my_iir.c*, et la renommer *teste_my_iir_ideal*

❒ modifier le contenu de la fonction *teste_my_iir_ideal* de façon à ce qu'elle permette d'étudier les résultats obtenus avec le filtre iir idéal. Ajouter la déclaration de cette fonction au header, puis l'exécuter avec le programme principal (depuis la fonction main). On veillera avec cette fonction à vérifier les caractéritiques suivantes du filtre, lorsque l'amplitude de l'entrée est égale à 32767, sur une durée de simulation de 50000 échantillons :

1- gain statique (pour une fréquence d'entrée nulle)

2- gain à la fréquence de coupure théorique à -3dB

3- gain a la fréquence pour laquelle le gain théorique est égal à -6dB

(moyenne, ecart-type, minimum et maximum, valeur efficace, de la sortie pour chacun de essais)

1.1.2 intégration du filtre iir en nombres entiers

❒ En vous inspirant de la partie précédente, ajouter au fichier source *my_iir.c* les structures et fonctions correspondant au filtre *iir* codé dans la partie 4 du travail préparatoire, sous forme directe 2

pour cela procéder intelligemment :

1- sélectionner et copier à la fin du programme *my_iir.c* tout ce que vous avez écrit relativement au filtre iir idéal (dupliquer les structures et fonctions avec le copier coller).

2- sélectionner la partie dupliquée en fin de programme, et utiliser le menu 'edition' pour remplacer (uniquement dans le bloc sélectionné)

- la chaîne de caractères *ideal* , par la chaîne de caractères *16bits*
- la chaîne de caractères *double* , par la chaîne de caractères *int_16*

3- modifier (compléter et adapter) les structures et fonctions

-*p_my_iir_16bits new_my_iir_16bits*();

int_16 one_step_my_iir_16bits(*int_16 en*, *p_my_iir_16bits* p_iir);

void *destroy_my_iir_16bits*(*p_my_iir_16bits p*);

de façon à ce qu'elles correspondent au codage de votre filtre iir en arithmétique 16/ 32 bits

❒ modifier à présent le contenu de la fonction *teste_my_iir_16bits* de façon à ce qu'elle permette d'étudier les résultats obtenus avec le filtre iir sur 16 bits. Ajouter la déclaration de cette fonction au header, puis l'exécuter avec le programme principal (depuis la fonction main). On veillera avec cette fonction à vérifier les caractéritiques suivantes du filtre, lorsque l'amplitude de l'entrée est égale à 32767 , sur une durée de simulation de 50000 échantillons:

1- gain statique (pour une fréquence d'entrée nulle)

2- gain à la fréquence de coupure théorique à -3dB

3- gain à la fréquence pour laquelle le gain théorique est égal à -6dB

(moyenne, écart-type, minimum et maximum, valeur efficace, de la sortie pour chacun de essais)

1.1.3 Comparaisons temporelles entre les 2 implémentations

❒Créer une fonction ***compare_my_iir*** , permettant de simuler et de comparer le comportement des 2 filtres, idéal et en nombres entiers, dans les mêmes conditions que pour les questions précédentes.

❒ pour chacune des simulations, relever les caractéristiques (moyenne, écart type, valeurs minimum et maximum, valeur efficace)

1- de la sortie idéale {= sortie utile }

2- de l'erreur entre la sortie idéale et la sortie entière.{ = bruit de sortie }

3- de la variable interne du filtre *iir* en nombres entiers.{ = variable interne x_n }

Ne pas reprendre le travail à zéro, s'inspirer de ce qui est déjà codé dans les fonctions *teste_my_iir_ideal*. et *teste_my_iir_16bits*

ne déclencher les mesures qu'à partir de l'échantillon 25000.

❒ comparer les résultats obtenus aux résultats théoriques...

❍d'après les simulations, aurait-il été possible de choisir un facteur d'échelle λ plus élevé?

❒Quelle que soit la réponse, expliquer pourquoi cela aurait été intéressant.

2 technique de *noise shaping*(modelage du bruit)

Une technique très souvent employée en audio est la technique du modelage de bruit au premier ou au second ordre (*first or second order noise shaping*). L'oreille humaine est en effet très sensible aux petits signaux (120db d'échelle dynamique aux alentours de 1Khz) .Cette technique permet de réduire grandement le niveau de bruit de sortie, lorsque les filtres ont des fréquences de coupure petites devant la fréquence d'échantillonnage. Dans cette partie, nous allons implémenter cette technique, et analyser les performances obtenues.

2.1 Écriture d'un programme scilab *my_iir.sce* permettant l'analyse de votre filtre

avant de commencer l'analyse du noise shaping, on va écrire un programme scilab permettant d'analyser et calculer les facteurs d'échelle de votre filtre.

❒à l'aide d'un éditeur de texte, éditer le programme ***seance2_exemple_av_phase.sce*** et l'enregistrer sous ***my_iir.sce***.(*ce programme décrit la synthèse et l'analyse en arithmétique 8/16 bits, sous forme df2t, du correcteur à avance de phase traité en exemple*)

❒ modifier l'entête (première ligne), de façon à ce qu'elle corresponde au nom de votre programme.
(exécuter ensuite le programme ***my_iir.sce*** sous scilab, pour vérifier qu'il n'y a pas d'erreur)

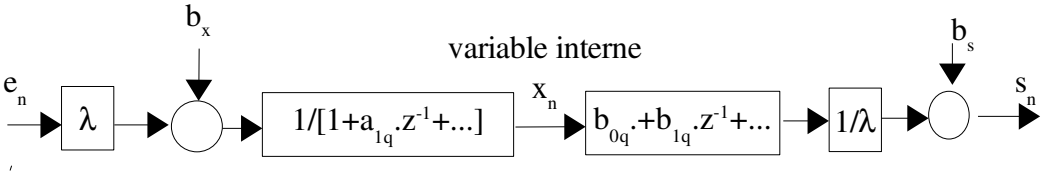
2.1.1 *PARTIE 1: description du filtre en w, en z et en z^{-1}*

❒Modifier la déclaration du filtre $F(w)$ dans le programme, de façon à ce qu'elle corresponde à votre filtre(partie 4 du travail préparatoire). Modifier également le nombre de bits de codage et la fréquence d'échantillonnage, conformément aux valeurs de la partie 4 du travail préparatoire.

❒Exécuter le programme, et vérifier que les valeurs des coefficients quantifiés, entiers, et des décalages à droite correspondent à celles que vous avez retenues.(lire la partie 4du programme: "*quantification des coeffs de F''*, pour chercher le nom des variables *scilab* correspondant aux coeffs en $q=z^{-1}$

2.1.2 PARTIE 2: calcul du facteur d'échelle λ à partir du schéma quantifié

on rappelle le schéma d'analyse standard de la forme directe 2 ci-dessous :



❒en raisonnant sur ce schéma d'analyse standard, modifier dans le programme l'initialisation de la variable *H_e_x_de_q,* représentant la fonction de transfert en $q=z^{-1}$ entre l'entrée e_n et la variable interne x_n , lorsque $\lambda=1$. (par défaut elle a été initialisée en raisonnant sur le schéma d'analyse de la forme ***df2t***, et non pas ***df2***)

❒Exécuter le programme, et en déduire la plus grande valeur de $\lambda=2^L$ que l'on peut retenir, et la valeur de L correspondante, pour empêcher les dépassements d'échelle de la variable interne x_n (normalement cette valeur devrait correspondre à celle du travail préparatoire).

❒Lire la partie 8 du programme, correspondant à la détermination des caractéristiques de la variable interne x_n , et en déduire si on a raisonné en nombres entiers ou en virgule fixe.

2.1.3 Partie 3 : calcul du bruit de sortie

on ignorera volontairement dans cette partie le bruit b_s , pour se focaliser sur l'effet du bruit b_x , qui est responsable de la majeure partie du bruit en sortie...

❒en raisonnant sur ce schéma d'analyse standard de la forme ***df2***, modifier dans le programme l'initialisation de la variable *H_bx_s_de_q,* représentant la fonction de transfert en $q=z^{-1}$ entre le bruit b_x et la sortie s , lorsque $\lambda=1$. (par défaut elle a été initialisée en raisonnant sur le schéma d'analyse de la forme ***df2t***, et non pas ***df2***).

❒modifier la partie 8.1 du programme, pour tenir compte du fait que le bruit b_x est un bruit d'arrondi,

et non pas de troncature.

❒Exécuter à nouveau le programme, et relever les caractéristiques du bruit de sortie {valeur absolue maximale, écart-type, valeur efficace maximale}.

2.1.4 analyses fréquentielles

⇨ Observer la figure scilab numéro 0, représentant les réponses fréquentielles du filtre et du filtre quantifié (en pseudo-pulsations). En déduire les modifications de comportement importantes induites par la quantification des coefficients.

⇨ Observer la figure scilab numéro 1, représentant les réponses fréquentielles des fonctions de transfert H_{e_x} et H_{bx_s} , avec et sans facteur d'échelle. Tracer rapidement leurs caractéristiques lorsque $\lambda = 1$ (allure générale, gains minimum et maximum)

⇨ Modifier le programme pour imposer un facteur d'échelle 4 fois plus grand (*ne pas supprimer les lignes de calcul de lambda, ajouter une ligne juste après la ligne :**lambda=2^L**).*

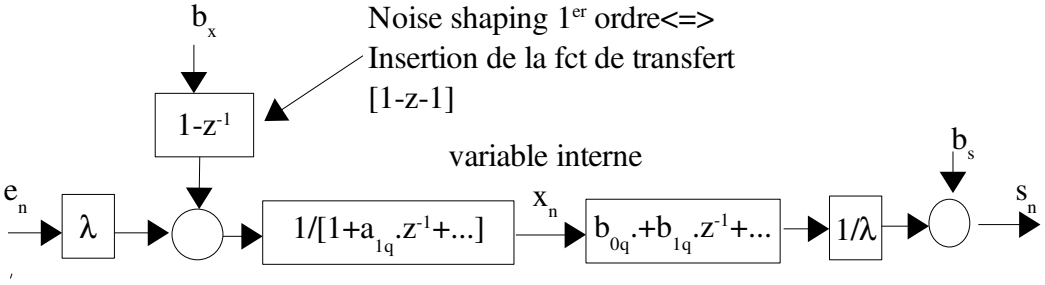
Exécuter le programme, relever les caractéristiques des variables internes et des bruits. Conclure sur les avantages et dangers de cette nouvelle valeur de lambda.

❒Ne surtout pas oublier de supprimer à présent la ligne de programme que vous venez d'ajouter!...

2.2 Principe du noise shaping, et analyse sous scilab

le principe du *noise shaping* consiste à dériver (une ou 2 fois selon l'ordre du filtre) le bruit b_x .

Pour un filtre du premier ordre sous forme *df2*, le schéma d'analyse correspondant est donné ci-après



2.2.1 mesure des effets du noise shaping

❒Au début de votre programme, choisir *switch_ns=%*t (*<=>true, ou encore vrai*). Rechercher l'endroit où est employée cette variable, et initialiser la variable H_{bx_s} conformément au schéma d'analyse avec noise shaping, lorsque *switch_ns* est vraie

❒exécuter le programme, et répondre aux questions suivantes, en fonction des résultats observés

- ☐le facteur d'échelle $\lambda = 2^L$, et les caractéristiques de la variable interne x_n ont-ils été modifiés?.. est-ce normal?...
- ☐les caractéristiques du bruit de sortie ont-elles été modifiées?
- ☐Observer la figure scilab numéro 1, représentant les réponses fréquentielles des fonctions de transfert H_{e_x} et H_{bx_s} , avec et sans facteur d'échelle. Tracer rapidement leurs caractéristiques lorsque $\lambda = 1$ (allure générale, gains minimum et maximum). En déduire la raison pour laquelle le noise shaping a permis de diminuer le bruit de sortie

2.2.2 un peu d'intuition sur le noise-shaping

<u>analyse en l'absence de noise shaping</u>

la fonction de transfert F_{bx_x} entre le bruit b_x et la variable interne x , *sans noise shaping*, s'écrit : $F_{bx\_x}(z^{-1}) = \dfrac{1}{1 + a_{1q}.z^{-1}}$,

le gain statique correspondant est donné par $F_{bx\_x}(z^{-1})\Big|_{en\ z=1} = \dfrac{1}{1+a_{1q}}$

le gain à la fréquence $\dfrac{f_e}{2}$ est donné par $F_{bx\_x}(z^{-1})\Big|_{en\ z=-1} = \dfrac{1}{1-a_{1q}}$

Lorsque la fréquence de coupure est petite devant la fréquence d'échantillonnage, le coefficient a_{1q} devient proche de -1.
dans ce cas F_{bx_x} est un filtre passe-bas, avec

un grand gain statique $\frac{1}{1+a_{1q}} \gg 1$

un gain en haute fréquence proche de $\frac{1}{2}$

Les normes de F_{bx_x} deviennent donc très grandes, essentiellement à cause du comportement statique { très grand gain en basse fréquence }.

La solution logique pour diminuer ces normes, et donc le niveau de bruit de sortie, consiste à essayer de diminuer le gain de F_{bx_x} essentiellement en basse fréquence, lorsque $z \to 1$, d'où l'idée d'insérer la fonction de transfert $1-z^{-1}$

analyse de l'effet du noise shaping

avec noise-shaping , on obtient: $F_{bx\_x}(z^{-1}) = \frac{1-z^{-1}}{1+a_{1q}.z^{-1}}$:

le gain statique correspondant est donné par $F_{bx\_x}(z^{-1})\Big|_{en\ z=1} = 0$

le gain à la fréquence $\frac{f_e}{2}$ est donné par $F_{bx\_x}(z^{-1})\Big|_{en\ z=-1} = \frac{2}{1-a_{1q}}$

lorsque $a_{1q} \approx -1$, cette fonction de transfert est un passe-haut, de gain statique nul(nette amélioration) , et de gain haute-fréquence proche de 1(petite dégradation). On peut donc espérer obtenir des normes correspondantes beaucoup plus faibles que précédemment...

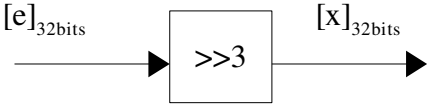
2.2.3 implémentation (codage) du noise-shaping

on va à présent présenter l'implémentation du *noise-shaping* en langage c
- 1- pour comprendre comment on l'implémente
- 2- pour comprendre la raison pour laquelle on doit se limiter à l'insertion de la fonction de transfert trè simple($1-z^{-1}$)
- 3- pour vous montrer que l'emploi des mathématiques pour l'analyse des phénomènes conduit à des solutions beaucoup plus efficaces que la bidouille...

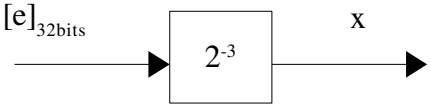
2.2.3.1 *mesure du bruit b_x , pour une quantification par troncature*

le bruit b_x est dû au décalage à droite L en sortie de l' additionneur du schéma d'implémentation détaillé.
On exposera la méthode d'implémentation pour $L=3$, et vous l'adapterez à la valeur de L que vous avez utilisée...

le bruit b_x est dû au morceau de schéma correspondant au décalage à droite $L=3$



ce schéma correspond à l'implémentation du schéma idéalisé suivant



le bruit b_x est la différence entre la sortie entière $[x]_{32bits}$ et la sortie idéalisée *x*:
$b_x = [x]_{32bits} - x$ (puisque $[x]_{32bits} = x + b_x$)

la sortie *x* (réelle) n'est pas mesurable. La seule donnée dont on dispose est $[e]_{32bits}=2^3.x$.

on ne donc peut mesurer directement que la quantité : $2^3.b_x=2^3.[x]_{32bits}-\underbrace{2^3.x}_{[e]_{32bits}}=2^3.[x]_{32bits}-[e]_{32bits}$

il reste à déterminer comment on peut évaluer $2^3.[x]_{32bits}$.

la sortie $[x]_{32bits}$ est caractérisée par le fait que tous ses bits correspondant à des puissances négatives sont égaux à 0 (voir exemple ci-après, pour $[e]_{32bits}=8+2+1=11$).
donc les 3 bits de poids faible de la quantité $2^3.[x]_{32bits}$ sont égaux à 0...

	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
$[e]_{32bits}=11$	0	1	0	1	1			
$x=2^{-3}.e$	0	0	0	0	1	0	1	1
$[x]_{32bits}$	0	0	0	0	1	0	0	0
$2^3.[x]_{32bits}$	0	1	0	0	0			

la quantité $2^3.[x]_{32bits}$ peut donc être mesurée directement
- en forçant à zéro les 3 bits de poids faible (bits 0,1,2) de l'entrée $[e]_{32bits}$

cas général :
Pour un décalage à droite de L bits, la quantité $2^L.[x]_{32bits}$ est évaluée en forçant à 0 les L bits de poids faible (*0* à *L-1*) de $[e]_{32bits}$
on peut ensuite évaluer la quantité $2^L.b_x=2^L[x]_{32bits}-\underbrace{2^L.x}_{[e]_{32bits}}=2^L.[x]_{32bits}-[e]_{32bits}$

le petit bout de programme en langage c correspondant est donné ci-dessous
```
bx3=e_32
bx3&=0xfffffff8; // force à 0 les 3 bits de poids faible, avec un ET logique
bx3=bx3-e_32; //   bx3=2^3.b_x=2^3.[x]_32bits-[e]_32bits
```
($bx3=\underbrace{2^3.[x]_{32bits}}_{e\ \&\ 0xfffffff8}-[e]_{32bits}$)

- Dans le cas d'une troncature, on peut encore optimiser le code, on sait en effet que la quantité $2^L.[x]_{32bits}$ est toujours inférieure à la quantité $[e]_{32bits}$ ...
On peut donc directement évaluer $-2^L.[b_x]$, qui est toujours positif, en forçant à zéro tous les bits de $[e]_{32bits}$, sauf les L bits de poids faible (bits $0,..,L-1$).
le petit bout de programme en langage c correspondant est donné ci-dessous
```
moins_bx3=e_32
moins_bx3&=0x00000007; // force à 0 des L-3 bits de poids fort, avec un ET logique
```

2.2.3.2 *mesure du bruit b_x , pour une quantification par arrondi*

la même technique peut être appliquée dans le cas d'une quantification par arrondi, à condition de modifier légèrement le code...
plutôt que de coder une multiplication par 2^{-L} avec arrondi, de la façon suivante (qui ne permet pas facilement la mesure du bruit b_x)
1- on décale à droite de $L-1$ bits
2- on ajoute 1
3- on décale à droite de 1 bit
On la code de la façon suivante (qui revient au même, mais permet la mesure de b_x)
1- on ajoute 2^{L-1}
// a ce stade on peut directement évaluer $-2^L.[b_x]$ avec un ET logique
2- on décale à droite de L bits

le petit bout de programme en langage c correspondant est donné ci-dessous pour L=3
x_32= e_32+0x00000004 // on ajoute 2^(L-1)=2^2 =0x04 en hexadécimal
moins_bx3=x_32 &0x00000007; // force à 0 des *L*-3 bits de poids fort, avec un ET logique
x_32>>=3; // on décale de 3 bits à droite

2.2.3.3 implémentation complète du noise-shaping

La partie difficile du *noise-shaping* consiste à comprendre qu'il est impossible de mesurer directement b_x . On ne peut mesurer que la quantité $-2^L.[b_x]$, si on veut un code efficace.
Cette étape passée, l'implémentation de la méthode est assez triviale

pour appliquer le *noise shaping*, on doit soustraire à la variable $[x_{32}]_n$ la valeur précédente du bruit :
on doit en effet programmer : $x_{32}(z)=[1-z^{-1}].b_x+2^{-L}.e_{32}(z)$
ce qui dans le domaine temporel donne : $[x_{32}]_n=[2^{-L}.[e_{32}]_n+bx_n]-bx_{n-1}$
soit encore : $[x_{32}]_n=[2^{-L}.[e_{32}]_n+\left[-bx_{n-1}\right]]+bx_n$
Pour en déduire le programme correspondant, on multiplie par 2^L , ce qui donne :

$$2^L.[x_{32}]_n=\left[[e_{32}]_n+\underbrace{\left[-2^L.bx_{n-1}\right]}_{quantité\ mesurable}\right]+2^L.bx_n$$

en multipliant à présent par 2^{-L} , on obtient finalement

$$[x_{32}]_n=2^{-L}.\left[[e_{32}]_n+\underbrace{\left[-2^L.bx_{n-1}\right]}_{quantité\ mesurable}\right]+bx_n$$

cette équation correspond à une multiplication par 2^{-L} ,de la quantité $\left[[e_{32}]_n+\underbrace{\left[-2^L.bx_{n-1}\right]}_{quantité\ mesurable}\right]$, avec ajout d'un bruit de troncature ou d'arrondi bx_n , selon la façon dont on programme le décalage à droite de L bits .
Remarque: L'addition du bruit bx_n est implicite=> ce bruit représente les bits perdus après le décalage à droite de L bits

On peut donc en déduire l'implémentation du *noise-shaping*, qui consiste à
1- ajouter à l'entrée la quantité $\left[-2^L.bx_{n-1}\right]$, précédemment calculée
2- ajouter éventuellement 2^{L-1} , si on veut un arrondi
3- mesurer la quantité $\left[-2^L.bx_n\right]$
4 -décaler à droite de L bits
5- garder en mémoire la quantité $\left[-2^L.bx_n\right]$, qui sera employé à la phase ***1*** du prochain pas d'échantillonnage..

Le bout de programme correspondant en langage C est donné ci-après
// la variable *masque_L* a été initialisée à $2^0+2^1+2^2+2^{L-1}$
// la variable *deux_pow_L_1* a été initialisée à 2^{L-1} (1<<(L-1) en langage C)
// la variable moins_bx_L a été initialisée à 0, au tout premier pas d'échantillonnage
x_32=e_32+moins_bx_L; // *moins_bx_L* est ici $\left[-2^L.bx_{n-1}\right]$, calculé précédemment
x_32+=deux_pow_L_1; // arrondi, on ajoute 2^{L-1}
moins_bx_L=x_32 & *masque_L;* // mesure de $\left[-2^L.bx_n\right]$ par application du masque
x_32>>=L; // décalage à droite de *L* bits

2.2.3.4 réalisation du noise shaping en langage c sous *netbeans*

Pour intégrer le noise-shaping au projet netbeans, on applique le même principe que précédemment : on ne détruit pas son travail, on l'enrichit en profitant de ce qui a déjà été réalisé =>

❒créer un source ***my_iir_ns.c***, et le header ***my_iir_ns.h*** *{ns:abbréviation de noise-shaping}*

❒ copier le contenu de ***my_iir.c*** dans ***my_iir_ns.c***

❒ dans ***my_iir_ns.c,*** remplacer partout la chaîne de caractères *iir* par :*iir_ns*, et sauver le programme *my_iir_ns.c.*

❒ ajouter au header ***my_iir_ns.h*** la déclaration de la fonction externe *teste_iir_ns*

❒ modifier le programme principal de façon à inclure ce header, et à exécuter la fonction *teste_iir_ns()*

❒modifier le source ***my_iir_ns.c*** pour y intégrer l'utilisation du *noise-shaping*, il faudra

1- modifier la structure *my_iir_ns*, pour y ajouter

-la mémoire associée à $\left[-2^{L}.bx_{n}\right]$

- le masque et la puissance de 2 associés au décalage commun

2- modifier la routine d'initialisation, pour initialiser ces 2 quantités

3- modifier la routine appelée à chaque pas d'échantillonnage, pour y intégrer le *noise- shaping*

2.2.3.5 *simulation et test*

❒exécuter le programme dans les mêmes conditions qu'en *1.1.3,* et comparer les niveaux de bruits à ceux précédemment obtenus.

⇨ ces résultats correspondent-t-ils à ceux prévus par l'analyse théorique en 1.1.3

⇨ dans le cas contraire, expliquer d'où peuvent provenir les différences observées