Aide-mémoire libtsd

La librairie `libtsd` est une collection de routines pour le traitement du signal, fondée sur la librairie Eigen. Les fonctions dont il est question concernent pour l'essentiel le traitement des signaux de dimension 1 (filtrage, transformée de Fourier, tracé de courbes, etc.), ainsi que quelques fonctions plus spécialisées pour les applications télécom.
Ce petit formulaire est un bref résumé des fonctions disponibles (la documentation complète est disponible ici : https://tsdconseil.github.io/libtsd/fr/).

1. Conventions

Les fréquences seront la plupart du temps exprimées de manière normalisée par rapport à la fréquence d'échantillonnage :

$$\nu = f/f_e$$

2. Fichiers d'en-tête et namespaces

Chaque module est définit par un fichier d'en-tête et un sous namespace du namespace principal `tsd`. Par exemple, pour utiliser le module de filtrage :

```
#include "tsd/filtrage.hpp"
using namespace tsd::filtrage;
```

Pour importer tous les modules et les namespaces automatiquement, vous pouvez aussi utiliser directement :

```
#include "tsd/tsd-all.hpp"
```

Tous les sous-namespace de tsd sont alors disponibles (`tsd`, `tsd::filtrage`, `tsd::fourier`, etc.).

3. Alias

Ces alias ont pour but d'alléger un peu le code.

Alias pour les nombres complexes :
`cfloat = std::complex<float>`
`cdouble = std::complex<double>`

Alias pour les tableaux :

Dims.	Type	Nom alias	Type Eigen
1d	float	`ArrayXf`	`Array<float,Dyn.,1>`
1d	cfloat	`ArrayXcf`	`Array<cfloat,Dyn.,1>`
1d	int	`ArrayXi`	`Array<int,Dyn.,1>`
1d	bool	`ArrayXb`	`Array<bool,Dyn.,1>`

Dans la suite, nous utiliserons la notation `Array...` pour signifier un de ces types de tableau.

Pointeurs partagés : `sptr<...> = std::shared_ptr<...>`

Constantes : π (double précision), π_f (simple précision)

4. Fonctions diverses

```
#include "tsd/tsd.hpp"
using namespace tsd;
```

`Array... h = vconcat(a,b) ou auto h = a | b` : concaténation verticale de deux vecteurs
`Array... y = rotation_vec(x, d)` : $y_k = x_{k+d[N]}$
`Array... y = diff(x)` : $y_k = x_{k+1} - x_k$
`Array... y = cumsum(x)` : $y_n = \sum_{k \leq n} x_k$
`Array... y = déplie_phase(x, r)` : $y_k \equiv x_k \mod r$, $|y_k - y_{k-1}| < r$
`auto lst = trouve(x)` : $\forall k \in \text{lst}, x_k$
`int k = trouve_premier(x)` : x_k, et $\forall i < k,\ !x_i$
`Array... y = sousech(x, R)` : $y_k = x_{Rk}$
`Array... y = surech(x, R)` : $y_{Rk} = x_k,\ y_{Rk+l} = 0$ $(0 < l < R)$
`auto y = pow2db(x)` : $y = 10 \log_{10} x$
`auto y = db2pow(x)` : $y = 10^{x/10}$
`int j = prochaine_puissance_de_2(i)` : $j = 2^k, j \geq i$
`Array... y = reechan(x, R)` : $y_k = x_{k/R}$
`auto y = modulo(x, m)` : $y \equiv x \mod m,\ 0 \leq y < m$
`auto y = modulo_2π(x)` : $y \equiv x \mod 2\pi,\ 0 \leq y < 2\pi$
`auto y = modulo_pm_π(x)` : $y \equiv x \mod 2\pi,\ -\pi \leq y < \pi$
`auto y = deg2rad(x)` : $y = \pi x/180$
`auto y = rad2deg(x)` : $y = 180x/\pi$
`ArrayXcf x = polar(theta)` : exponentielle complexe

5. Génération de signaux

```
#include "tsd/tsd.hpp"
using namespace tsd;
```

5.1 Signaux de durée finie

`ArrayXf x = linspace(a,b,n)` : $x_k = a + k(b-a)/(n-1)$
`ArrayXf x = logspace(a,b,n)` : $x_k = 10^{a+k(b-a)/(n-1)}$
`ArrayXf x = randn(n)` : $x_k : \mathcal{N}(0,1)$
`ArrayXf x = randu(n)` : $x_k : \mathcal{U}(0,1)$
`ArrayXb x = randb(n)` : $x_k = 0$ ou $1,\ p[x_k = 1] = 1/2$
`ArrayXi x = randi(M,n)` : $x_k \in [0, M-1],\ p[x_k = i] = 1/M$
`ArrayXf x = sigexp(f,n)`, `sigcos(f,n)`, `sigsin(f,n)`, `sigtri(p,n)`, `sigcar(p,n)`, `sigimp(n,p)`, `sigscie(p,n)`, `sigchirp(`f_0, f_1, n`)`, `sigchirp2(`f_0, f_1, n`)` : signaux divers

5.2 Sources « temps réel » (streaming)

`auto src = source_ohc(`ν`)`, `ArrayXcf x = src->step(n)` : Oscillateur harmonique (complexe) : $x_k = e^{2\pi i k \nu}$
`auto src = source_ohr(`ν`)`, `ArrayXf x = src->step(n)` : Idem, réel : $x_k = \cos(2\pi k \nu)$

`auto t = tampon_création<T>(n, [](const Vecteur<T> &y) {}); t->step(x);` : Découpage des paquets entrants (de dimensions arbitraires) en paquets de dimension fixe n (la call-back passée en paramètre est appelée avec des vecteurs y de longueur n).

6. Graphiques

```
#include "tsd/vue.hpp"
using namespace tsd::vue;
```

La classe `Figure` permet de tracer des courbes simples :
`Figure f;` (création d'une figure)
`f.plot(x,y,[opt,titre,...])` : tracé de y en fonction de x, opt = chaine de caractères avec options de couleur, de type de trait, et de curseur :

Couleurs : `"b"`, `"g"`, `"r"`, `"m"`, `"c"`, **etc.**

Traits : "-" (lignes), "|" (barres), "" (aucun)

Curseurs : "s" (carré), "d" (diamant), "*" (étoile), "." (point), "o" (rond), "" (aucun)

`f.plot(`y`,[opt,titre,...])` : idem, $x = 0 \ldots n-1$
`f.plot_iq(`z`,[opt,titre,...])` : idem, $x = \text{real}(z), y = \text{imag}(z)$
`f.plot_psd(`x`[,`f_e`=1])` : tracé du spectre
`f.plot_img(Z, format = "c[jet]")` : Dessin d'une image (Z est un tableau 2d)

La classe `Figures` permet de tracer plusieurs figures placées sur une grille :
`Figures f;` (création)
`f.subplot()` : Renvoie une nouvelle figure.

Exemple de figure

```
Figure f;

f.plot(t, x, "og-", "x");
f.plot(t, y, "ob-", "y");
f.plot_iq(z);
f.plot_psd(x, fe);
f.titre("Titre principal");

// pour afficher une fenêtre à l'écran
f.afficher("Ma figure");
// ou pour générer un fichier image
f.enregistrer("chemin.jpg");
```

7. Types génériques

La classe abstraite `Filtre<Te,Ts,C>` (où T_e est le type d'entrée (float, cfloat, double, etc.), T_s celui de sortie, et C celui de la configuration) sera très utilisée, car elle permet de cacher les détails d'implémentation des différents blocs.

En général, c'est un pointeur partagé vers un filtre (`sptr<Filtre<...>>`) que l'on pourra récupérer. Les actions possibles sur un filtre sont :

`filtre->configure(c)` : configuration du filtre,
`y = filtre->step(x)` : calcul du filtre sur une séquence de données.

Exemple d'utilisation d'un filtre

```
auto filtre = filtre_rif<float>(
  design_rif_fen(31, "lp"));
ArrayXf x = randn(1000);
ArrayXf y = filtre->step(x);
```

8. Module filtrage

```
#include "tsd/filtrage.hpp"
using namespace tsd::filtrage;
```

Les filtres LTI (linéaire invariant dans le temps) sont spécifiés par leur fonction de transfert $h(z)$ (simple vecteur de coefficients pour un filtre RIF, ou plus généralement fraction rationnelle avec le type `Frat<T>`, où T = `float` pour des coefficients flottants, `complex<float>` pour des complexes, etc.), puis implémentés suivant une des différentes structures possibles.

8.1 Analyse des fonctions de transfert

`ArrayXf x = repimp(h, n)` : réponse impulsionnelle (n points)
`ArrayXcf X = repfreq(h, f)` : réponse fréquentielle (f = vecteur de fréquences)
`auto [fr, mag] = frmag(h, n)` : réponse fréquentielle (en magnitude), n points entre 0 et 1/2
`auto [fr, phase] = frphase(h, n)` : réponse fréquentielle (en phase)
`auto [fr, tps] = frgroup(h, n)` : temps de groupe
`float τ = rif_delais(h)` : Délais d'un filtre RIF de type I ou II.
`affiche_filtre(h)` : tracé réponse impulsionnnelle et fréquentielle
`analyse_filtre(h[, fe = 1])` : tracé des différentes courbes caractérisant le filtre (réponses fréquentielles, temporelles, pôles et zéro, temps de groupe).
`plot_plz(fig, h)` : tracé des pôles / zéros

8.2 Conception des fonctions de transfert

`h = design_rif_fen(n, type, fc, fen[, fc'])` : design par sinus-cardinal fenêtré. `type` = "lp" (passe-bas), "hp" (passe-haut), "bp" (passe-bande), ou "sb" (stoppe-bande).
`h = design_rif_fen_kaiser(type, fc, atten, df[, fc'])` : Idem, fenêtre de Kaiser (paramètres = atténuation en dB, et largeur du lobe principal).
`h = design_rif_fen_chebychev(n, type, fc, atten[, fc'])` : Idem, fenêtre de Chebychev (paramètres = nombre de coefficients, et atténuation).
`h = design_rif_freq(n, d)` : design par échantillonnage fréquentiel (d est un vecteur définissant la réponse fréquentielle souhaitée).
`h = design_rif_eq(n, d, w)` : design équiripple / Remez (d : réponse souhaitée, w : poids de pondération).
`h = design_rif_cs(n, α, fc)` : filtre en cosinus sur-élevé (RC)
`h = design_rif_rcs(n, α, fc)` : filtre en racine de cosinus sur-élevé (SRRC)
`h = design_rif_gaussien(n, σ)` : Gaussien (approximation RIF)
`h = design_rif_gaussien_telecom(n, BT, osf)` : Gaussien + moyenne glissante (approximation RIF)
`h = design_rif_hilbert(n, fen)` : filtre de Hilbert (approximation RIF)
`h = design_riia(n, type, proto, fcut)` : design RII analogique (type = "lp", "hp", ..., proto = "butt", "cheb1", "cheb2", ...)
`h = design_cic({R,M,N})` : Filtre CIC (fonction de transfert théorique, pour l'implémentation utiliser `filtre_cic(...)` décris ci-dessous), R = facteur de décimation, N = nombre d'étages, M = facteur de design.
`h = design_cic_comp({R,M,N},fin,R2,fcut,n)` : Conception d'un filtre de compensation pour filtrage CIC.
`h = design_rif_prod(h1,h2)` : Coefficients d'un filtre RIF équivalent à la cascade de 2 filtres RIF.
`h = design_bloqueur_dc(fc)` : Filtre passe-haut récursif.
`h = design_rii1(fc)` : filtre exponentiel
`hd = trf_bilineaire(ha, fe)` : Fonction de transfert analogique vers digitale.

8.3 Création de filtres

Les fonctions suivantes renvoient un pointeur vers un filtre générique (`sptr<Filtre<T>>`), `T` étant le type de données à traiter (float, cfloat, etc.) :
`auto f = filtre_id<T>()` : élément neutre (filtre sans effet)
`auto f = filtre_rif<T>(h)` : structure RIF
`auto f = filtre_rif_fft<T>(h)` : idem, domaine fréquentiel
`auto f = filtre_rii<T>(h)` : filtre RII (forme directe)
`auto f = filtre_sois<T>(h)` : filtre RII (chaine de sections du second ordre)
`auto f = filtre_cic<T,Ti>({R, N, M}, [mode = 'd'];` : filtre CIC (Cascade Integrated Comb), T = type externe, T_i = type interne, mode = 'd' (décimation) ou 'i' (interpolation).
`auto f = filtre_rii1(γ)` : filtre RII du premier ordre (dit ≪ RC numérique ≫).
`auto f = filtre_mg(n)` : moyenne glissante
`auto f = filtre_dc(fc)` : coupe le DC.
`auto f = ligne_a_retard(n)` : retarde le signal d'entrée de n échantillons.
`auto f = hilbert_transformeur(n[,fen])` : définit un transformeur de Hilbert, qui convertit un signal réel en un signal analytique (complexe).

8.4 Fenêtres

`ArrayXf x = fenetre(type, n, sym)` : création d'une fenêtre, avec type = "re", "hn", "hm", "tr", ... Fenêtre symétrique ou périodique.
`ArrayXf x = fenêtre_chebychev(n, atten_db, sym)`
`ArrayXf x = fenêtre_slepian(n, B)`
`ArrayXf x = fenêtre_kaiser(n, atten_db, sym)`

8.5 Filtrage de signaux de durée finie

`auto y = filtrer(h, x)` : Filtrage du signal x à partir de la fonction de transfert h.
`auto y = convol(h, x)` : Idem, à partir de coefficients.
`Array...y = filtfilt(h, x)` : filtrage bidirectionnel (zéro-phase)
`ArrayXcf z = hilbert(x)` : calcule le signal analytique complexe, à partir d'un signal réel.
`ArrayXcf z = hilbert_tfd(x)` : idem, calcul via la TFD (sans délais).

8.6 Interpolation

`auto f = itrp_linéaire<T>()`
`auto f = itrp_lagrange<T>(d)` : interpolation polynomiale, degré d.
`auto f = itrp_sinc<T>(ncoefs, nphase, fc, fenêtre)` : interpolation par sinc fenêtré.
`auto f = itrp_cspline<T>()` : interpolation par splines cardinales.

8.7 Changement de fréquence d'échantillonnage

`auto f = decimateur<T>(R)` : garde un échantillon sur R.
`auto f = filtre_rif_ups<T>(h, R)` : filtre d'interpolation polyphase RIF (R = facteur entier de décimation)
`auto f = filtre_rif_decim<T>(h, R)` : Idem, pour la décimation.
`auto f = filtre_rif_demi_bande<T>(h)` : Idem, version spécialisée pour $R = 2$ (suppose que le filtre est tel que un échantillon sur deux est nul).
`auto f = filtre_itrp(ratio[, itrp])` : Interpolation, ratio arbitraire.
`auto f = filtre_reechan(ratio)` : Ré-échantillonnage, ratio arbitraire (cascade de filtres demi-bande et d'un interpolateur).

9. Module fourier

```
#include "tsd/fourier.hpp"
using namespace tsd::fourier;
```

9.1 Signaux de durée finie

`ArrayXcf` X `= fft(`x`)` : transformée de Fourier discrète
`ArrayXcf` x `= ifft(`X`)` : transformée inverse
`ArrayXcf` X_2 `= fftshift(`X`)` : centrage TFD
`ArrayXf` f `= tfd_freqs(`n`[, avec_shift])` : calcul des fréquences normalisées associées à chaque bin de TFD.
`float freq = freqestim(`z`)` : Estimation de fréquence, pour une exponentielle ou sinusoïde pure
z `= czt(`x`,` m`,` W`[,` $z_0 = 1$`])` : transformée en chirp-Z
`Array... y = rééchan_freq(x, R)` : $y_k = x_{k/R}$
`auto [l, c] = ccorr(`x`[,` y`])` : Corrélation ou auto-corrélation circulaire : $c_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} x_k y^\star_{k+n[N]}$
`auto [l, c] = xcorr(`x`[,` y`])` : Corrélation ou auto-corrélation linéaire (non biaisée) : $c_n = \frac{1}{N-n} \cdot \sum_{k=0}^{N-n-1} x_k y^\star_{k+n}$
`auto [l, c] = xcorrb(`x`[,` y`])` : Corrélation ou auto-corrélation linéaire (biaisée) : $c_n = \frac{1}{N} \cdot \sum_{k=0}^{N-n-1} x_k y^\star_{k+n}$
`Array... y = délais(`x, τ`)` : Délais entier ou fractionnaire : $y_k = x_{k+\tau}$
`auto [délais,score] = estimation_délais(`x, y`)` : Calcul délais fractionnaire entre deux signaux.
`auto [fr, mag] = psd(x)` : Estimation spectale, simple corrélogramme, signal pondéré avec une fenêtre de Hann.
`auto [fr, mag] = psd_welch(x,` N`, fen)` : Estimation spectale, méthode de Welch (moyennage).
`auto [fr, mag] = psd_sousesp(x,` N_s`,` N_f`)` : Estimation spectale, méthode des sous-espaces
`float f = goertzel(`x, f`)` : Filtre de Goertzel

9.2 Traitements de flux

`auto` p `= fftplan_création(); p-¿step(`x`, `y`[, forward]);` : Plan pour calcul de plusieurs FFT de mêmes dimensions.
`auto` p `= rfftplan_création(); ArrayXcf y = p-¿step(`x`);` : Idem, FFT avec vecteur réel en entrée.
`auto` f `= filtre_goertzel(`f`,`n`)` : f = fréquence à détecter, n = longueur fenêtre
`auto` f `= filtre_fft(config)` : Filtrage dans le domaine fréquentiel (OLA avec ou sans fenêtrage)
`auto d = detecteur_création(config)` : Détecteur automatique de motif sur un flux, avec estimation position, gain, phase, etc.
`auto` f `= rt_spectrum(config)` : Calcul d'un spectre en temps réel

10. Module audio

```
#include "tsd/audio.hpp"
using namespace tsd::audio;
```

`auto [x, fe] = wav_charge(chemin)` : lecture fichier mono (x = signal réel)
`auto [z, fe] = wav_charge_stereo(chemin)` : lecture fichier stéréo (z = signal complexe)

11. Module telecom

```
#include "tsd/telecom.hpp"
using namespace tsd::telecom;
```

11.1 Formes d'onde

M = nombre de valeurs possibles / symbole ($\log_2(M)$ bits / symbole).
`auto fo = forme_onde_psk(`M`[,filtre])`
`auto fo = forme_onde_bpsk([filtre])`
`auto fo = forme_onde_qpsk([filtre])`
`auto fo = forme_onde_`π`4_qpsk([filtre])`
`auto fo = forme_onde_ask(`M, K_1, K_2`[,filtre])`
`auto fo = forme_onde_qam(`M`[,filtre])`
`auto fo = forme_onde_fsk(`M`,index[,filtre])`

11.2 Mise en forme

Création d'un filtre :
`auto f = SpecFiltreMiseEnForme::aucun()`
`auto f = SpecFiltreMiseEnForme::nrz()`
`auto f = SpecFiltreMiseEnForme::gaussien(BT)`
`auto f = SpecFiltreMiseEnForme::rcs(`β`)`

Méthodes (`ncoefs` = nb coefs filtre RIF, R = facteur de sur-échantillonnage) :
`auto h = f.get_coefs(ncoefs,` R`)` : Coefficients filtre RIF
`auto flt = f.filtre_mise_en_forme(ncoefs,` R`)` : Filtre + sur-échantillonnage
`auto flt = f.filtre_adapté(ncoefs,` R`)` : Filtre (réception)
`auto res = f.analyse(ncoefs,` R`)` : Analyse du spectre

Autres fonctions :
`ArrayXf h = sah(`x`,` R`)` : Sample and hold

11.3 Modulation / démodulation

`auto f = modulateur_création(config)` : création d'un modulateur, pour une modulation numérique. La structure de configuration permet de spécifier la forme d'onde, la fréquence d'échantillonnage, la fréquence symbole et éventuellement une fréquence intermédiaire.
`auto f = démodulateur_création(config_mod, config_démod)`
`auto f = émetteur_création(format_trame)` : Mise en forme de trames (ajout d'en-tête, padding, etc.))
`auto f = récepteur_création(config)` : Récepteur de trames (avec en-tête spécifié).

11.4 Codes de synchronisation

`BitStream code_Barker(n)`
`BitStream code_mls(n)`

11.5 Canaux de propagation

`ArrayXcf y = bruit_awgn(`x`,` σ`)`
`auto c = canal_dispersif(type,` f_d`,` f_e`,` K`)`