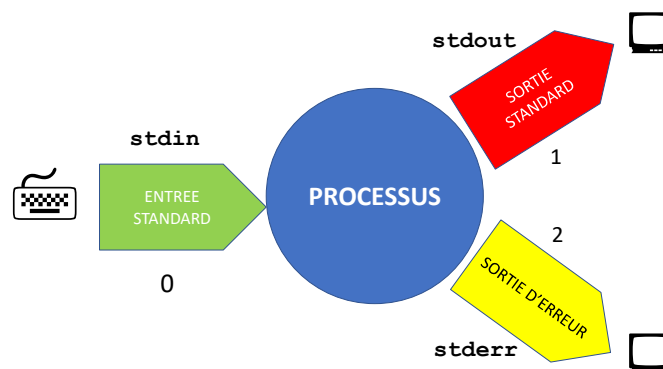


Atelier n°2 : Redirection des entrées sorties, « tubes anonymes » et « pipelines ».

Introduction.

Lorsqu'un programme est lancé, c'est à dire lorsqu'un processus est créé, il hérite par défaut de trois canaux de communication lui permettant *a priori* d'échanger avec « l'extérieur », c'est à dire l'utilisateur ou d'autres processus. Ces canaux sont bien connus puisqu'il s'agit essentiellement de ceux permettant d'afficher du texte sur la console et de saisir du texte (au clavier normalement). Il s'agit de « l'entrée standard », de la « sortie standard » et de la « sortie d'erreur ». Ces différents canaux portent des noms et des numéros précis. En fonction de la manière dont on les accède (format « texte » ou format « binaire »), on utilisera l'un ou l'autre. Comme suggéré précédemment, l'entrée standard opère par défaut sur la clavier, tandis que la sortie standard et la sortie d'erreur sont quant à elles orientées par défaut vers la fenêtre du terminal utilisé. Ceci peut se schématiser de la manière suivante :



Accès aux canaux standards avec les fonctions associées au flux formatés « texte » :

Les noms `stdin` (entrée standard), `stdout` (sortie standard) et `stderr` (sortie d'erreur) sont en fait des variables de type `FILE *` (pointeur sur flux formaté de type « texte ») qui sont utilisées avec le *corpus* de fonctions associées, soit `fopen()`, `fclose()`, `fprintf()` et `fscanf()`, entre autres.

Par ailleurs, comme on sait que l'entrée standard correspond par défaut au clavier et que la lecture d'une entrée clavier se fait par l'intermédiaire de la fonction `scanf()`, on en déduit que les appels `scanf(...)` et `fscanf(stdin, ...)` sont strictement équivalents.

Dans le même ordre d'idée, la sortie standard opérant par défaut sur la console (« écran »), on en déduit l'équivalence entre `fprintf(stdout, ...)` et `printf()`.

Ainsi, les lignes :

```
...
double x ;
printf(« entrer un reel : \n ») ;
scanf(« %lf », &x ) ;
...
```

sont équivalentes à :

```
...
double x ;
fprintf(stdout, « entrer un reel:\n ») ;
fscanf(stdin, « %lf », &x ) ;
...
```

Il faut noter que les variables `stdin`, `stdout` et `stderr` sont déjà déclarées (*via* le fichier « header » `stdio.h`) et n'ont donc pas besoin de l'être explicitement par la programmeur un fois ce fichier d'entête déclaré.

Accès au canaux standards avec les fonctions associées aux fichiers binaires.

De manière comparable, les entrées et sorties standards peuvent être manipulées par l'intermédiaire de leur numéro de canal respectif (**0, 1 ou 2**), ceci en utilisant le *corpus* de fonctions associées aux fichiers binaires, plus particulièrement `open()`, `close()`, `read()` et `write()`.


Le fichier source ci-dessous est un exemple de ce type d'accès :


```
#include <stdlib.h>
#include <string.h>      /* ->manipulation des chaines de caracteres          */
#include <unistd.h>      /* ->pour prototypes des fonctions read(), etc. sous linux */
/*.....*/
/* constantes */
/*.....*/
#define MESSAGE        "Bonjour!!!\n"
#define MAX_LEN        256          /* ->longueur maximale d'une chaine      */
int main( void )
{
    char *szOutMessage = MESSAGE;    /* ->a utiliser avec precaution...      */
    char szInString[MAX_LEN];        /* ->pour la saisie d'une chaine        */
    int n;                          /* ->longueur de la chaine a imprimer  */
    int iNbBytes;                   /* ->nombre d'octets saisis             */
    double dbValeur;                /* ->valeur reellev saisie             */
    /* affichage : */
    n = strlen( MESSAGE );
    write( 1, szOutMessage, n );
    /* saisie (chaine): */
    fflush( stdin );                /* ->nettoyage canal d'entree avant saisie */
    memset( szInString, 0, MAX_LEN );
    iNbBytes = read(0, szInString, MAX_LEN);
    printf("%d octets saisis \n", iNbBytes);
    /* tentative de conversion de la chaine saisie en un reel : */
    if( sscanf(szInString, "%lf", &dbValeur) == 1)
    {
        printf("valeur saisie = %lf\n", dbValeur);
    }
    else
    {
        printf("la chaine %s ne correspond pas a un reel\n", szInString);
    };
    return( 0 );
}
```

Pour mémoire, on peut rappeler les points suivants :


✖ la fonction `write()` prend comme premier paramètre le numéro de canal (un entier) à utiliser. Dans cet exemple, on utilise le canal **1** correspondant à la **sortie standard** et dont on rappelle qu'il est ouvert par défaut. Le second paramètre est un pointeur sur une zone mémoire qui contient les données à écrire vers le canal spécifié. Dans la mesure où on manipule une chaîne (ici `szOutMessage`), celle-ci est nativement gérée par le **C** au travers d'un pointeur, et il n'est pas nécessaire (pour être plus précis, **IL NE FAUT PAS**) que son nom soit préfixé par le « symbole d'indirection mémoire » `&`. Le troisième paramètre est le nombre de caractères qu'il faut tenter d'écrire sur le canal. La fonction `write()` retourne **-1** en cas **d'erreur** ou le nombre de caractères effectivement écrits en cas de succès.

✖ La fonction `read()` utilise le même type de paramétrage. La différence se situe au niveau du sens de l'échange (on lit « depuis » le canal spécifié). Elle retourne elle aussi **-1** en cas **d'erreur** ou le nombre de caractères effectivement lus en cas de succès.


 Saisir, compiler et exécuter ce programme.

 Indiquer quel est le rôle de l'appel à la fonction `fflush()`.


 Indiquer quel peut-être l'utilité de l'appel préalable à la fonction `memset()`.


 Rappeler comment la fonction `sscanf()` permet de contrôler que la saisie est conforme au format spécifié (ici une valeur réelle).


Application 1 : un premier exemple de redirection manuelle.

 Saisir et compiler le programme ci-dessous (ES-redirection-1.c) :

```
/*=====*/
/* exemple de manipulation des entrees / sorties standard */
/*=====*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h> /* ->cf open() */
#include <sys/stat.h> /* ->cf open() */
#include <sys/types.h> /* ->cf open() */
/*.....*/
/* constantes */
/*.....*/
#define OUT_FILE_NAME "sortie.txt"
#define MAX_LEN 256 /* ->taille maximale pour une chaine */
/*=====*/
/* programme de test (sortie) */
/*=====*/
int main( void )
{
    int iOut; /* ->numero de canal */
    close(1);
    iOut = open( OUT_FILE_NAME, O_CREAT | O_RDWR );
    printf("ceci est un message.\n Est-il visible sur l'ecran de la console ?\n");
    return( 0 );
}
```


 Après l'avoir exécuté, note-t-on effectivement l'apparition du message indiqué dans le `printf()` ? **Rien ne s'affiche**

 Après exécution, on doit normalement noter la création d'un nouveau fichier nommé `sortie.txt`. Quel est le contenu de ce fichier (on peut le voir par la commande **cat** en ligne de commande) ? **Oui, le texte du PRINTF**

 Pourquoi peut-on parler d'une opération de « redirection de la sortie »?

La sortie n'est plus la console mais un fichier


Application 2 : second exemple de « redirection manuelle ».


 Saisir et compiler le programme ci-dessous (ES-redirection-2.c) :


```
/*=====*/
/* autre exemple de manipulation des entrees-sorties */
/* standards d'un processus. */
/*=====*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
/*.....*/
/* constantes */
/*.....*/
#define IN_FILE_NAME    "texte.txt"
#define MAX_LEN        256          /* ->taille maximale pour une chaine */
/*#####*/
/* programme de test (sortie) */
/*#####*/
int main( void )
{
    int iIn;                      /* ->canal d'entree */
    char szInStr[MAX_LEN];
    close(0);
    iIn = open(IN_FILE_NAME, O_RDONLY);
    do
    {
        /*saisie clavier */
        scanf("%s", szInStr);
        /* affichage saisie clavier */
        printf("%s\n", szInStr);
    }
    while( strcmp( szInStr, "STOP") != 0 ); /* arret si STOP saisi */

    return( 0 );
}
```

 Exécuter celui-ci. **ATTENTION** : pour un fonctionnement correct, il est nécessaire que le fichier `texte.txt` (récupérable sur la plateforme MLS / AMSE / OS) soit copié dans le répertoire d'exécution choisi.

 Sur quoi le processus lit-il à la place du clavier ? [Le fichier "texte.txt"](#)

 Tenter d'expliquer le principe de fonctionnement des deux exemples précédents. Quelques indications utiles :

➡ la fonction `scanf()` correspond à une lecture sur le canal **0** (si elle pouvait avoir un avis, elle se ficherait complètement de savoir ce qui s'y trouve « connecté »...).


➡ La fonction `printf()` correspond à une écriture sur le canal **1** (même remarque que pour la fonction `scanf()`).

➡ Quand on ferme un canal (fonction `close()`), le numéro de canal correspondant est rendu disponible.

➡ Quand on crée un nouveau canal (~ fichier), on utilise systématiquement le plus petit numéro de canal disponible.

Application 3 : Redirection avec le shell.

Les redirections d'entrées-sorties peuvent être réalisées au lancement d'un processus en utilisant les opérateurs de redirection appropriés. Ceux-ci sont : `>` et `>>` pour la redirection de la sortie standard ; `2>` et `2>>` pour la redirection de la sortie d'erreur. La différence entre `>` et `>>` (dans les deux cas) est que `>` efface systématiquement le fichier de sortie s'il pré-existe, tandis que `>>` ajoute la nouvelle sortie à la fin du fichier existant (mode « *append* »), le cas échéant.

 A des fins de test, saisir et compiler le fichier source suivant (on suppose que le fichier

exécutable généré s'appelle **redirection**):

```
/*=====*/
/* exemple de programme simple écrivant sur la sortie standard et sur la */
/* sortie d'erreur                                                         */
/*=====*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*#####*/
/* exemple */
/*#####*/
int main( void )
{
    printf("ceci est un message affiche sur stdout\n");
    fprintf(stderr,"ceci est un message affiche sur stderr\n");
    return( 0 );
}
```

Lancer les commandes suivantes :

```
$ redirection
$ redirection >sortie-standard.txt
$ cat sortie-standard.txt
$ redirection 2>sortie-erreur.txt
$ cat sortie-erreur.txt
$ rm sortie-erreur.txt sortie-standard.txt
$ redirection >sortie-standard.txt 2>sortie-erreur.txt
$ cat sortie-erreur.txt
$ cat sortie-standard.txt
$ redirection >>sortie-standard.txt 2>>sortie-erreur.txt
$ redirection >>sortie-standard.txt 2>>sortie-erreur.txt
$ redirection >>sortie-standard.txt 2>>sortie-erreur.txt
$ cat sortie-erreur.txt
$ cat sortie-standard.txt
```

✍ Donner un intérêt potentiel pour :

Le mécanisme de redirection séparé entre la sortie d'erreur et la sortie standard.

La possibilité de faire des redirection en mode « ajout » (« *append* ») avec >>.

Application 4 : faire un pipeline de traitement avec le shell.

Introduction.

Les shells disponibles sont capables de faire un peu plus que les simples redirections évoquées plus haut. Bien que cela ne soit pas finalement très complexe à mettre en œuvre, le résultat est un mécanisme puissant qui permet de réaliser des traitements répartis entre différents processus sans devoir mobiliser un arsenal technique trop ardu. Les quelques exemples de code qui suivent sont destinées à en illustrer le principe.

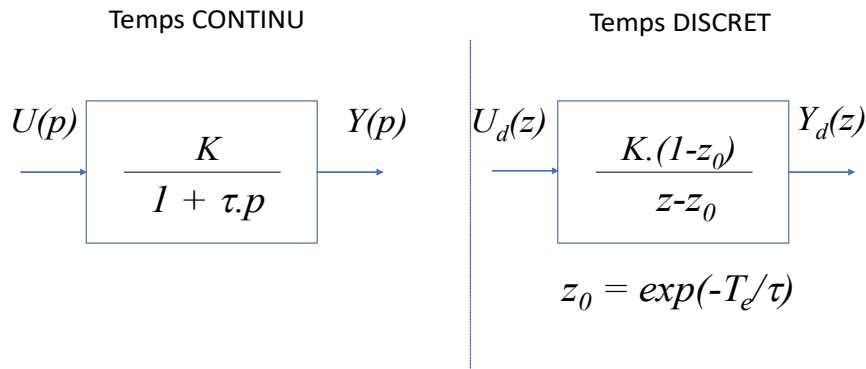
A 4.1 : un programme simple de filtrage numérique.

La brique élémentaire que nous allons utiliser est un programme se comportant comme un filtre passe-bas du premier ordre (échantillonné) : l'utilisateur saisie une valeur qui correspond à celle du signal d'entrée du filtre à l'instant considéré. A la suite de l'appui sur entrée, le programme affiche la valeur de la sortie du filtre correspondante et attend l'entrée suivante. Pour quitter le programme, il suffit que l'utilisateur saisisse une chaîne qui ne soit pas interprétable comme une valeur réelle (par exemple la chaîne « **STOP** »). Le cas échéant, notre programme filtre affiche « **STOP** » et se termine.

Pour ne pas perturber le mécanisme que nous allons tenter de mettre en place, il ne faut ajouter aucun message d'invite (comme « veuillez saisir une valeur »). On supposera donc que l'utilisateur sait ce qu'il a à faire.


Pour mémoire, le schéma-bloc d'un tel filtre est représenté sur la figure suivante (la partie gauche illustre la fonction de transfert en temps continu, tandis que celle de droite correspond à celle en temps discret, qui nous intéresse davantage) :

Filtre passe-bas du premier ordre



La sortie de ce système se calcule de proche en proche, en utilisant une relation de récurrence. Ainsi, si y_k représente la sortie du filtre numérique à « l'instant » k et u_{k-1} son entrée à l'instant précédent, celle-ci est donnée par :

$y_k = z_0 y_{k-1} + K \cdot (1 - z_0) \cdot u_{k-1}$, avec $z_0 = e^{\frac{-T_e}{\tau}}$. Ici T_e est la « période d'échantillonnage » utilisée (en s) tandis que τ est la constante du temps du filtre. De manière « traditionnelle », K représente le gain statique.

 La première tâche à réaliser consiste à écrire le programme filtre (le fichier source C correspondant sera nommé `filtre.c`). Pour vous y aider, l'embryon de code ci-dessous vous est proposé (les emplacements à compléter sont indiqués en commentaires) :

```
/*=====*/  
/* un filtre numerique pour illustrer le principe des pipelines */  
/*=====*/  
  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <ctype.h>  
#include <string.h>  
#include <string.h>  
#include <math.h>                /* ->cf exp()                  */  
/* declarations */  
void usage( char *);              /* ->aide de ce programme    */  
/* definitions */  
/*&&&&&&&&&&&&&&&&&&&&&&&&/  
/* aide de ce programme */  
/*&&&&&&&&&&&&&&&&&&&&&&&/  
void usage( char *szPgmName)  
{  
    if( szPgmName == NULL)  
    {  
        exit( -1 ); /* ->ca va mal... */  
    };  
    printf("%s <gain statique> <constante de temps> <période d'ech.>\n", szPgmName);
```

```

    printf("saisir STOP (ou qq chose qui n'est pas un nombre) stoppe le programme\n");
}
/*#####*/
/* programme principal : on passe en argument et dans cet ordre */
/* - le gain statique K */
/* - la constante de temps tau (s) */
/* - la periode d'echantillonnage Te (s) */
/*#####*/
int main( int argc, char *argv[])
{
    double k; /* ->gain statique */
    double tau; /* ->constante de temps */
    double Te; /* ->periode d'echantillonnage */
    double z0; /* ->pole discret (pour la recurrence) */
    double y; /* ->sortie "courante" (instant k) */
    double y1; /* ->sortie precedente (instant k-1) */
    double u; /* ->entree courante (saisie utilisateur) */
    /*.....*/
    /* Verification du nombre d'arguments passes a la ligne de commande */
    /*.....*/
    if( argc != 4)
    {
        usage( argv[0] );
        return(0);
    };
    /*.....*/
    /* Recuperation des arguments (un par un en verifiant le format). */
    /* En cas d'erreur, un message s'affiche sur la sortie d'ERREUR */
    /* et l'aide s'affiche sur la sortie standard : */
    /* A COMPLETETER */
    /*.....*/
    /*.....*/
    /* initialisations : */
    /* -poles discret */
    /* -y1 (a 0.0) */
    /* -nettoyage buffer d'entree : fflush( stdin ) */
    /*.....*/
    z0 = exp(-Te / tau);
    y1 = 0.0;
    fflush( stdin );
    /*.....*/
    /* boucle de saisie : */
    /*.....*/
    do
    {
        /* saisie de la chaine utilisateur */
        /* (sortie si format incorrect) */
        /* A COMPLETER */

        /* calcul de la sortie courante: */
        /* (application de la recurrence) */
        /* A COMPLETER

        /* on decale pour le "tour" suivant : */
        /* A COMPLETER

        /* affichage de la sortie courante : */
        printf("%lf\n", y);
        fflush( stdout ) ; /* ->forçage de l'affichage */

    } while (1);
    /* sortie de bouccle : */
    printf("STOP\n");
    /* FIN */
    return( 0 );
}

```

```
}
```

✂ Pour la compilation, il faudra prendre garde au fait que nous utilisons une fonction de la librairie mathématique, en l'occurrence `exp()`. Il est nécessaire alors d'en indiquer l'usage avec l'option **-lm** lors de l'invocation de `gcc`. Pour compiler le fichier source `filtre.c` en un exécutable nommé `filtre`, on pourra par exemple entrer :

```
$ gcc filtre.c -lm -o filtre
```

Pour vérifier le fonctionnement de `filtre`, on peut lui faire calculer la suite de valeurs correspondant à la réponse indicielle d'un système du premier ordre de gain statique 1 et de constante de temps 0,1 s échantillonné avec un pas temporel de 0,01 s, en tapant :

```
$ filtre 1 0.1 0.01
```

puis en saisissant à chaque fois la valeur 1 en entrée. Au bout de la **dixième** saisie, on a atteint la date 0,1 s (c'est à dire la constante de temps) et la valeur de la sortie qui s'affiche doit valoir sensiblement **0,63** (résultat « classique » pour un tel système avec ces valeurs de paramètre et à cette date).

A.4.3 : Création d'un pipeline pour réaliser un traitement plus complexe.

Supposons que notre problème ne soit plus de calculer la sortie d'un filtre du premier ordre, mais celle du filtre dont la fonction de transfert est la suivante :

$$H(p) = \frac{2}{(1+0,1 \cdot p) \cdot (1+0,15 \cdot p) \cdot (1+0,2 \cdot p) \cdot (1+0,25 \cdot p) \cdot (1+0,3 \cdot p)}$$

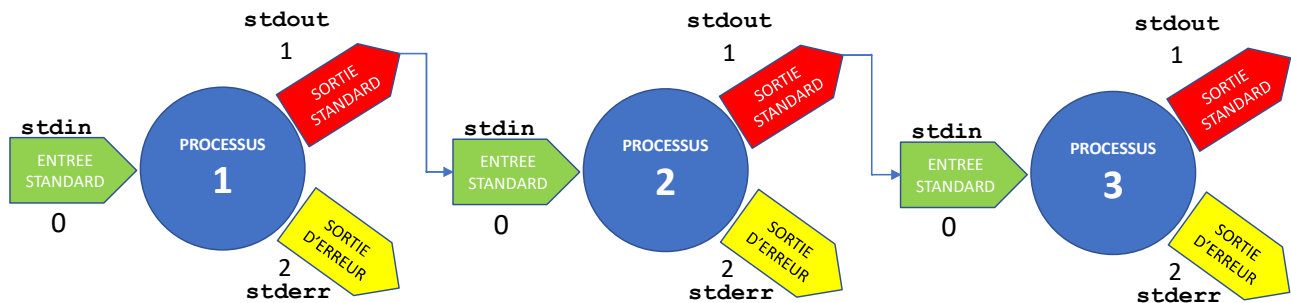
Deux solutions s'offrent à nous : faire appel à des souvenirs de traitement du signal ou utiliser le mécanisme de « pipeline » proposé par le système d'exploitation. Nous opterons ici pour la seconde approche.

Mécanisme de pipeline.

Le shell est en fait capable de lancer plusieurs processus depuis la ligne de commande, en faisant en sorte que ceux-ci travaillent « à la chaîne » dans la mesure où les programmes correspondants sont écrits de manière compatible avec un tel mécanisme. C'est normalement le cas de notre filtre. Pour réaliser le chaînage, le shell réalise des opérations de redirection des entrées sorties des processus concernés. Le principe en est que la sortie standard du processus « amont » est redirigée vers l'entrée standard du processus « aval » (et *vice-versa*). Du point de vue des processus, ceux-ci continuent à utiliser les fonctions `scanf()` et `printf()` « comme si de rien n'était » mais, ce faisant, c'est un véritable mécanisme de « communication interprocessus » qui est mis en œuvre. L'entité qui sert de lien de communication entre les processus s'appelle un « tube » (ou « *pipe* »). Il s'agit d'un fichier séquentiel ouvert en mémoire implémentant un mécanisme de type FIFO avec une lecture destructive (une donnée lue à « l'extrémité » du « tube » est enlevée du tube). Comme ceux-ci ne se voient pas attribuer de nom dans ce contexte d'usage particulier, on parle de « tube anonyme » (« *anonymous pipe* »). Les « tubes » sont des mécanismes de communication très utilisés dont nous reparlerons lorsque nous aborderons les IPC (« *Inter Processus Communication* »).

Le schéma ci-dessous résume l'idée maîtresse :

Principe d'un PIPELINE



Lorsque le processus 2 réalise un `scanf()`, il lit en fait les données écrites par le processus 1 *via* un `printf()`. De même pour le processus 3 par rapport au processus 2. La seule réelle limite à la longueur d'un pipeline est imposée par les ressources dont dispose le système utilisé.

Pour réaliser un pipeline, on « chaîne » les processus concernés par l'intermédiaire de l'opérateur `|` (normalement, sur un clavier AZERTY, c'est la combinaison des touches `[AltGr]` et `[6]`. Ce symbole s'appelle justement le symbole « pipe ». Certains systèmes d'exploitation à la fois antiques et exotiques (OS-9 et OS-9000 par exemple) utilisent le symbole `' ! '` à la place).

Par exemple, si on souhaite réaliser un pipeline entre les processus exécutant respectivement les programmes P1, P2 et P3, on entrera :

\$ P1 | P2 | P3


Nous en savons assez pour appliquer le « *pipeline* » à notre problème de filtre passe-bas du cinquième ordre :


☞ En remarquant que la fonction de transfert $H(p)$ correspond à la mise en série des fonctions de transfert du premier ordre $\frac{2}{1+0,1 \cdot p}$, $\frac{1}{1+0,15 \cdot p}$, $\frac{1}{1+0,2 \cdot p}$, $\frac{1}{1+0,25 \cdot p}$, $\frac{1}{1+0,3 \cdot p}$ et enfin $\frac{1}{1+0,35 \cdot p}$, écrire le pipeline équivalent au filtre du cinquième ordre à réaliser en utilisant `filtre`. Pour les tests, on utilisera une période d'échantillonnage de 0,01 s (entrer quelques valeurs pour vérifier le fonctionnement correct).

Le système ainsi construit présente un temps de réponse dont l'ordre de grandeur est de quelques secondes. Avec la période d'échantillonnage imposée, cela nécessite de saisir plusieurs centaines de valeurs 1 consécutives pour simuler la réponse indicielle.

☞ Indiquer comment l'usage de la fonction `cat` et d'un fichier texte doté d'un contenu adapté permet d'éviter une manipulation aussi fastidieuse.

☞ Ecrire un pipeline permettant en une seule ligne de commande de calculer la réponse indicielle de notre système du cinquième et de stocker les valeurs de sortie dans un fichier nommé `sortie-filtre.txt` SANS MODIFIER `filtre`.

 A l'aide de votre tableur préféré (Libre Office, Open Office, EXCEL, etc.) importer le fichier texte (tous les tableurs dignes de ce nom le permettent) et tracer la réponse indicielle.

 A l'aide de ce qui a été vu, tenter de résumer en 5 lignes maximum les opérations réalisées par le shell pour créer le pipeline. Parmi celles-ci, quelle est celle qui paraît encore la plus « mystérieuse » ?