Analyse des séries temporelles: Application au procédé de polissage chimico-mécanique

THEO GACHET KAWTAR EL MAMOUN

ECOLE DES MINES DE SAINT ETIENNE

6 DECEMBRE 2022

Choix de la structure de données

Listes chaînées

Définition d'une structure pour chaque série temporelle

```c
// chaque Element d'une telle liste chaînée aura la structure suivante :
typedef struct Element Element;
struct Element
{
    double data;
    Element *suivant;
};

// cette structure Liste contient un pointeur vers le premier élément de la liste
typedef struct Liste Liste;
struct Liste
{
    Element *premier;
};
```

Manipulation des listes chaînées

```
300     // insertion() permet d'insérer un élément en paramètre au début d'une liste chaînée
301   ⊞void insertion(Liste *liste, double new_data){ ... }
312
313     // supprimer_premier_element() permet de supprimer le premier élément d'une liste chaînée
314   ⊞void supprimer_premier_element(Liste *liste){ ... }
326
327     // supprimer_element() permet de supprimer la première occurence d'une donnée dans une liste chaînée
328   ⊞int supprimer_element(Liste *liste, double value){ ... }
359
360     // afficherListe() permet de visualiser sur le terminal les éléments d'une liste
361   ⊞void afficherListe(Liste *liste){ ... }
377
378     // init() permet d'initialiser une liste chaînée par un double
379   ⊞Liste *init(double debut){ ... }
393
394     // tronquer() permet de tronquer une liste chaînée de l'élément d'indice a à celui d'indice b (inclus et a<b)
395   ⊞Liste *tronquer(Liste *liste, int a, int b){ ... }
```

Initialisation des listes de données

```
 32    ⊞Liste *init_USAGE_OF_BACKING_FILM() { ... }
 46    ⊞Liste *init_USAGE_OF_DRESSER() { ... }
 60    ⊞Liste *init_USAGE_OF_POLISHING_TABLE() { ... }
 74    ⊞Liste *init_USAGE_OF_DRESSER_TABLE() { ... }
 88    ⊞Liste *init_PRESSURIZED_CHAMBER_PRESSURE() { ... }
102    ⊞Liste *init_MAIN_OUTER_AIR_BAG_PRESSURE() { ... }
116    ⊞Liste *init_CENTER_AIR_BAG_PRESSURE() { ... }
130    ⊞Liste *init_RETAINER_RING_PRESSURE() { ... }
144    ⊞Liste *init_RIPPLE_AIR_BAG_PRESSURE() { ... }
158    ⊞Liste *init_USAGE_OF_MEMBRANE() { ... }
172    ⊞Liste *init_USAGE_OF_PRESSURIZED_SHEET() { ... }
186    ⊞Liste *init_SLURRY_FLOW_LINE_A() { ... }
200    ⊞Liste *init_SLURRY_FLOW_LINE_B() { ... }
214    ⊞Liste *init_SLURRY_FLOW_LINE_C() { ... }
228    ⊞Liste *init_WAFER_ROTATION() { ... }
242    ⊞Liste *init_STAGE_ROTATION() { ... }
256    ⊞Liste *init_HEAD_ROTATION() { ... }
270    ⊞Liste *init_DRESSING_WATER_STATUS() { ... }
284    ⊞Liste *init_EDGE_AIR_BAG_PRESSURE() { ... }
```

Initialisation des listes de données

```
47  Liste *init_USAGE_OF_DRESSER()
48  {
49      Liste *liste = (Liste *)malloc(sizeof(Liste));
50      Element *element = (Element *)malloc(sizeof(Element));
51
52      if (liste == NULL)
53          exit(EXIT_FAILURE);
54
55      element->data = 534.074074074;
56      element->suivant = NULL;
57      liste->premier = element;
58
59      return liste;
60  }
```

Extraction des données

```
516   int stock_USAGE_OF_BACKING_FILM(char *training, Liste *liste) { ... }
584   int stock_USAGE_OF_DRESSER(char *training, Liste *liste) { ... }
650   int stock_USAGE_OF_POLISHING_TABLE(char *training, Liste *liste) { ... }
716   int stock_USAGE_OF_DRESSER_TABLE(char *training, Liste *liste) { ... }
782   int stock_PRESSURIZED_CHAMBER_PRESSURE(char *training, Liste *liste) { ... }
848   int stock_MAIN_OUTER_AIR_BAG_PRESSURE(char *training, Liste *liste) { ... }
914   int stock_CENTER_AIR_BAG_PRESSURE(char *training, Liste *liste) { ... }
980   int stock_RETAINER_RING_PRESSURE(char *training, Liste *liste) { ... }
1046  int stock_RIPPLE_AIR_BAG_PRESSURE(char *training, Liste *liste) { ... }
1112  int stock_USAGE_OF_MEMBRANE(char *training, Liste *liste) { ... }
1178  int stock_USAGE_OF_PRESSURIZED_SHEET(char *training, Liste *liste) { ... }
1244  int stock_SLURRY_FLOW_LINE_A(char *training, Liste *liste) { ... }
1310  int stock_SLURRY_FLOW_LINE_B(char *training, Liste *liste) { ... }
1376  int stock_SLURRY_FLOW_LINE_C(char *training, Liste *liste) { ... }
1442  int stock_WAFER_ROTATION(char *training, Liste *liste) { ... }
1508  int stock_STAGE_ROTATION(char *training, Liste *liste) { ... }
1574  int stock_HEAD_ROTATION(char *training, Liste *liste) { ... }
1640  int stock_DRESSING_WATER_STATUS(char *training, Liste *liste) { ... }
1706  int stock_EDGE_AIR_BAG_PRESSURE(char *training, Liste *liste) { ... }
```

Extraction des données

```
546     double new_DRESSING_WATER_STATUS;
547     double new_EDGE_AIR_BAG_PRESSURE;
548
549     int parcours = 0; // parcours représente le nombre d'éléments lus, donc il est nul initialement
550
551     ligne = (char *)malloc(linesize * sizeof(char)); // on alloue dynamiquement de la mémoire à la ligne que l'on va lire
552     fichier = fopen(training, "r");                  // on ouvre le fichier contenant nos données
553
554     if (fichier == NULL)
555         printf("Erreur d'ouverture du fichier");
556
557     if (fichier != NULL)
558     {
559         while (fgets(ligne, linesize, fichier) != NULL && !feof(fichier)) // on éxécute le code ci-dessous tant qu'il reste des l
560         {
561             parcours = fscanf(fichier, "%d  %d  %lf %lf %s  %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %
562                               &new_MACHINE_ID, &new_MACHINE_DATA, &new_TIMESTAMP, &new_WAFER_ID, &new_STAGE,
563                               &new_CHAMBER, &new_USAGE_OF_BACKING_FILM, &new_USAGE_OF_DRESSER,
564                               &new_USAGE_OF_POLISHING_TABLE, &new_USAGE_OF_DRESSER_TABLE, &new_PRESSURIZED_CHAMBER_PRESSURE,
565                               &new_MAIN_OUTER_AIR_BAG_PRESSURE, &new_CENTER_AIR_BAG_PRESSURE, &new_RETAINER_RING_PRESSURE,
566                               &new_RIPPLE_AIR_BAG_PRESSURE, &new_USAGE_OF_MEMBRANE, &new_USAGE_OF_PRESSURIZED_SHEET,
567                               &new_SLURRY_FLOW_LINE_A, &new_SLURRY_FLOW_LINE_B, &new_SLURRY_FLOW_LINE_C, &new_WAFER_ROTATION,
568                               &new_STAGE_ROTATION, &new_HEAD_ROTATION, &new_DRESSING_WATER_STATUS, &new_EDGE_AIR_BAG_PRESSURE);
569
570             // chaque ligne est composée de 25 séries temporelles, il faut donc obligatoirement toutes les lire
571
572             if (parcours > 0)
573             {
574                 insertion(liste, new_USAGE_OF_BACKING_FILM); // mais on ne va insérer à notre liste que la donnée utile (USAGE_OF
575                 i++;
576             }
577         }
578     }
579     free(ligne);     // on libère la mémoire allouée dynamiquement
580     fclose(fichier); // on ferme le fichier
581     return i;        // on renvoie le nombre de lignes lues
582 }
```

Statistiques descriptives centrales et de dispersion

```
426     // moyenne() calcule la moyenne des valeurs prises par les éléments d'une liste composée de structures à un élément
427   ⊞double moyenne(Liste *liste){ ... }
446
447     // minimum() renvoie le plus petit élément d'une liste chaînée de double
448   ⊞double minimum(Liste *liste){ ... }
466
467     // maximum() renvoie le plus grand élément d'une liste chaînée de double
468   ⊞double maximum(Liste *liste){ ... }
486
487     // ecart_type() renvoie l'écart-type d'un échantillon représenté par une liste chaînée
488   ⊞double ecart_type(Liste *liste){ ... }
```

Statistiques descriptives centrales et de dispersion

```c
double moyenne(Liste *liste)
{
    if (liste == NULL)
        exit(EXIT_FAILURE);

    Element *actuel = liste->premier;

    double moy = 0.0; // on initialise la moyenne à 0 en respectant les types
    int nb_ligne = 0; // on sauvegarde le nombre de ligne car il sera utile plus loin dans le calcul

    while (actuel != NULL) // on parcourt la liste entièrement
    {
        moy += actuel->data; // on augmente la moyenne à chaque nouvel élément rencontré
        actuel = actuel->suivant;
        nb_ligne++; // et on itère le nombre de valeurs lues
    }

    return (moy / nb_ligne); // on retourne enfin la moyenne de toutes les valeurs rencontrées
}
```

Statistiques descriptives centrales et de dispersion

```c
// minimum() renvoie le plus petit élément d'une liste chaînée de double
double minimum(Liste *liste)
{
    if (liste == NULL)
        exit(EXIT_FAILURE);

    Element *actuel = liste->premier;

    double min = actuel->data; // on prend par défaut le premier élément de la liste comme minimum

    while (actuel != NULL)
    {
        if (actuel->data < min) // puis on le change dès que l'on trouve plus petit que lui
            min = actuel->data;
        actuel = actuel->suivant;
    }

    return (min); // on retourne enfin le minimum de tous les éléments de la liste
}
```

Statistiques descriptives centrales et de dispersion

```c
double maximum(Liste *liste)
{
    if (liste == NULL)
        exit(EXIT_FAILURE);

    Element *actuel = liste->premier;

    double max = actuel->data; // on prend par défaut le premier élément de la liste comme maximum

    while (actuel != NULL)
    {
        if (actuel->data > max) // puis on le change dès que l'on trouve plus grand que lui
            max = actuel->data;
        actuel = actuel->suivant;
    }

    return (max); // on retourne enfin le maximum de tous les éléments de la liste
}
```

Statistiques descriptives centrales et de dispersion

```c
double ecart_type(Liste *liste)
{
    if (liste == NULL)
        exit(EXIT_FAILURE);

    Element *actuel = liste->premier;

    double moy = moyenne(liste); // on récupère d'abord la moyenne de l'échantillon

    actuel = liste->premier;
    double ecart_type = 0.0; // on initialise l'écart-type à 0

    int nb_ligne = 0; // on garde en mémoire le nombre d'éléments lus

    while (actuel != NULL)
    {
        ecart_type += pow(actuel->data - moy, 2); // on applique la formule de l'écart-type
        actuel = actuel->suivant;
        nb_ligne++;
    }

    ecart_type = sqrt(ecart_type / nb_ligne); // on normalise la valeur obtenue
    return (ecart_type);                      // et on retourne l'écart-type de l'échantillon considéré
}
```

Statistiques descriptives centrales et de dispersion

```c
// STATISTIQUES POUR CHAQUE SERIE TEMPORELLE APRES LA REGLE DU K*SIGMA (peut être décliné pour chaque série temporelle)

Liste *data_USAGE_OF_BACKING_FILM = init_USAGE_OF_BACKING_FILM();
int count_USAGE_OF_BACKING_FILM = stock_USAGE_OF_BACKING_FILM("resultat.csv", data_USAGE_OF_BACKING_FILM);
data_USAGE_OF_BACKING_FILM = is_valeur_atypique_solo(data_USAGE_OF_BACKING_FILM);
printf("\n\nStatistiques descriptives pour USAGE_OF_BACKING_FILM");
printf("\nEffectif   : 190 638"); // obtenu grace à decompte_liste()
printf("\nMoyenne    : %lf", moyenne(data_USAGE_OF_BACKING_FILM));
printf("\nMinimum    : %lf", minimum(data_USAGE_OF_BACKING_FILM));
printf("\nMaximum    : %lf", maximum(data_USAGE_OF_BACKING_FILM));
printf("\nEcart-type : %lf", ecart_type(data_USAGE_OF_BACKING_FILM));
Liste *data_USAGE_OF_BACKING_FILM_tri = tri(data_USAGE_OF_BACKING_FILM);
printf("\nQuartile 1 : %lf", (get_index(data_USAGE_OF_BACKING_FILM_tri, 47659) + get_index(data_USAGE_OF_BACKING_FILM_tri, 47660)) / 2);
printf("\nMédiane    : %lf", (get_index(data_USAGE_OF_BACKING_FILM_tri, 95319) + get_index(data_USAGE_OF_BACKING_FILM_tri, 95320)) / 2);
printf("\nQuartile 3 : %lf", (get_index(data_USAGE_OF_BACKING_FILM_tri, 142978) + get_index(data_USAGE_OF_BACKING_FILM_tri, 151564)) / 2);
printf("\n");
```

Statistiques descriptives centrales et de dispersion

```
Statistiques : USAGE_OF_BACKING_FILM
Effectif    : 202084
Moyenne     : 4796.244355
Minimum     : 29.166667
Maximum     : 10299.166667
Ecart_type  : 3175.889822
Quartile 1  : 1917.500000
Médiane     : 4865.416667
Quartile 3  : 7480.833333


Statistiques : USAGE_OF_DRESSER
Effectif    : 202084
Moyenne     : 399.445810
Minimum     : 5.185185
Maximum     : 768.888889
Ecart_type  : 234.189513
Quartile 1  : 166.666667
Médiane     : 425.925926
Quartile 3  : 604.814815
```

```
Statistiques : USAGE_OF_DRESSER_TABLE
Effectif    : 202084
Moyenne     : 2887.778387
Minimum     : 2664.750000
Maximum     : 3205.750000
Ecart_type  : 146.771640
Quartile 1  : 2753.500000
Médiane     : 2890.500000
Quartile 3  : 2992.000000


Statistiques : PRESSURIZED_CHAMBER_PRESSURE
Effectif    : 202084
Moyenne     : 53.301020
Minimum     : 0.000000
Maximum     : 188.571429
Ecart_type  : 38.756810
Quartile 1  : 0.000000
Médiane     : 73.809524
Quartile 3  : 78.571429
```

Détection des observations atypiques

```c
int index_of_data(Liste *liste, double value)
{
    Element *actuel = liste->premier;
    int index = 1;

    while (actuel != NULL && actuel->data != value)
    {
        index++; // on incrémente l'indice lorsque l'on avance dans la liste
        actuel = actuel->suivant;
    }

    if (index == 1 && liste->premier->data != value)
        return -1;

    return index + 1;
}
```

Détection des observations atypiques

```c
void supprimer_index(Liste *liste, int index)
{
    Element *actuel = liste->premier;
    int parcours = 0;

    while (parcours < index)
    {
        actuel = actuel->suivant;
        parcours++; // "parcours" représente l'indice de l'élement que l'on examine
    }
    if (parcours == index) // si on a trouvé l'élément à supprimer, on s'en débarasse via supprimer_element()
        supprimer_element(liste, actuel->data);
}
```

Détection des observations atypiques

++ Détection et élimination des valeurs atypiques liste par liste

- - Segmentation fault en cas de manipulation sur l'ensemble du fichier

```c
Liste *is_valeur_atypique_solo(Liste *liste)
{
    int k = 3;
    double sigma = ecart_type(liste);
    int index = 1;

    Element *actuel;

    for (actuel = liste->premier; actuel != NULL; actuel = actuel->suivant)
    {
        if (fabs(actuel->data) > (k * sigma)) // condition de la loi statistique du k*sigma
        {
            index = index_of_data(liste, actuel->data);
            actuel = actuel->suivant;
            supprimer_element(liste, actuel->data);
        }
    }
    return liste;
}
```

Distributions empiriques

```c
int frequence(Liste *liste, double min, double max)
{
    int occ = 0;

    if (liste == NULL)
        exit(EXIT_FAILURE);

    Element *actuel = liste->premier;

    while (actuel != NULL)
    {
        if ((actuel->data <= max) && (actuel->data >= min)) // notons que les intervalles sont fermés dans le cas des histogrammes
            occ++;                                          // on compte le nombre d'éléments dans l'intervalle
        actuel = actuel->suivant;
    }
    return (occ);
}
```

Distributions empiriques

```c
void histogramme(Liste *liste, int bins)
{
    printf("\nMaximum = %lf", maximum(liste));
    printf("\nMinimum = %lf", minimum(liste));
    double intervalle = maximum(liste) - minimum(liste); // l'étendue se calcule comme max - min
    double pas = intervalle / bins;                      // on divise l'étendue en "bins" intervalles de même taille "pas"
    printf("\nPas     = %lf", pas);

    printf("\n\nOn obtient les bins suivants :");
    double parcours = minimum(liste);
    while (parcours < maximum(liste))
    {
        printf("\n[%lf;%lf] de fréquence %d / %d", parcours, parcours + pas, frequence(liste, parcours, parcours + pas), 202084);
        parcours += pas;
    }
}
```

Distributions empiriques

```c
// HISTOGRAMMES AVANT LA REGLE DU K*SIGMA (complet et fonctionne pour chaque série temporelle)

Liste *data_USAGE_OF_BACKING_FILM = init_USAGE_OF_BACKING_FILM();
int count_USAGE_OF_BACKING_FILM = stock_USAGE_OF_BACKING_FILM("resultat.csv", data_USAGE_OF_BACKING_FILM);
printf("\n\n--- Génération de l'histogramme : %s ---\n", "USAGE_OF_BACKING_FILM");
histogramme(data_USAGE_OF_BACKING_FILM, 5); // on prend 5 arbitrairement

Liste *data_USAGE_OF_DRESSER = init_USAGE_OF_DRESSER();
int count_USAGE_OF_DRESSER = stock_USAGE_OF_DRESSER("resultat.csv", data_USAGE_OF_DRESSER);
printf("\n\n--- Génération de l'histogramme : %s ---\n", "USAGE_OF_DRESSER");
histogramme(data_USAGE_OF_DRESSER, 5); // on prend 5 arbitrairement

Liste *data_USAGE_OF_POLISHING_TABLE = init_USAGE_OF_POLISHING_TABLE();
int count_USAGE_OF_POLISHING_TABLE = stock_USAGE_OF_POLISHING_TABLE("resultat.csv", data_USAGE_OF_POLISHING_TABLE);
printf("\n\n--- Génération de l'histogramme : %s ---\n", "USAGE_OF_POLISHING_TABLE");
histogramme(data_USAGE_OF_POLISHING_TABLE, 5); // on prend 5 arbitrairement

Liste *data_USAGE_OF_DRESSER_TABLE = init_USAGE_OF_DRESSER_TABLE();
int count_USAGE_OF_DRESSER_TABLE = stock_USAGE_OF_DRESSER_TABLE("resultat.csv", data_USAGE_OF_DRESSER_TABLE);
printf("\n\n--- Génération de l'histogramme : %s ---\n", "USAGE_OF_DRESSER_TABLE");
histogramme(data_USAGE_OF_DRESSER_TABLE, 5); // on prend 5 arbitrairement
```

Distributions empiriques

```
--- Génération Histogramme : USAGE_OF_BACKING_FILM ---

Maximum = 10299.166667
Minimum = 29.166667
Pas     = 2054.000000

On obtient les bins suivants :
[29.166667 ; 2083.166667] de fréquence 57755 / 202084
[2083.166667 ; 4137.166667] de fréquence 33868 / 202084
[4137.166667 ; 6191.166667] de fréquence 38884 / 202084
[6191.166667 ; 8245.166667] de fréquence 32606 / 202084
[8245.166667 ; 10299.166667] de fréquence 38971 / 202084
```

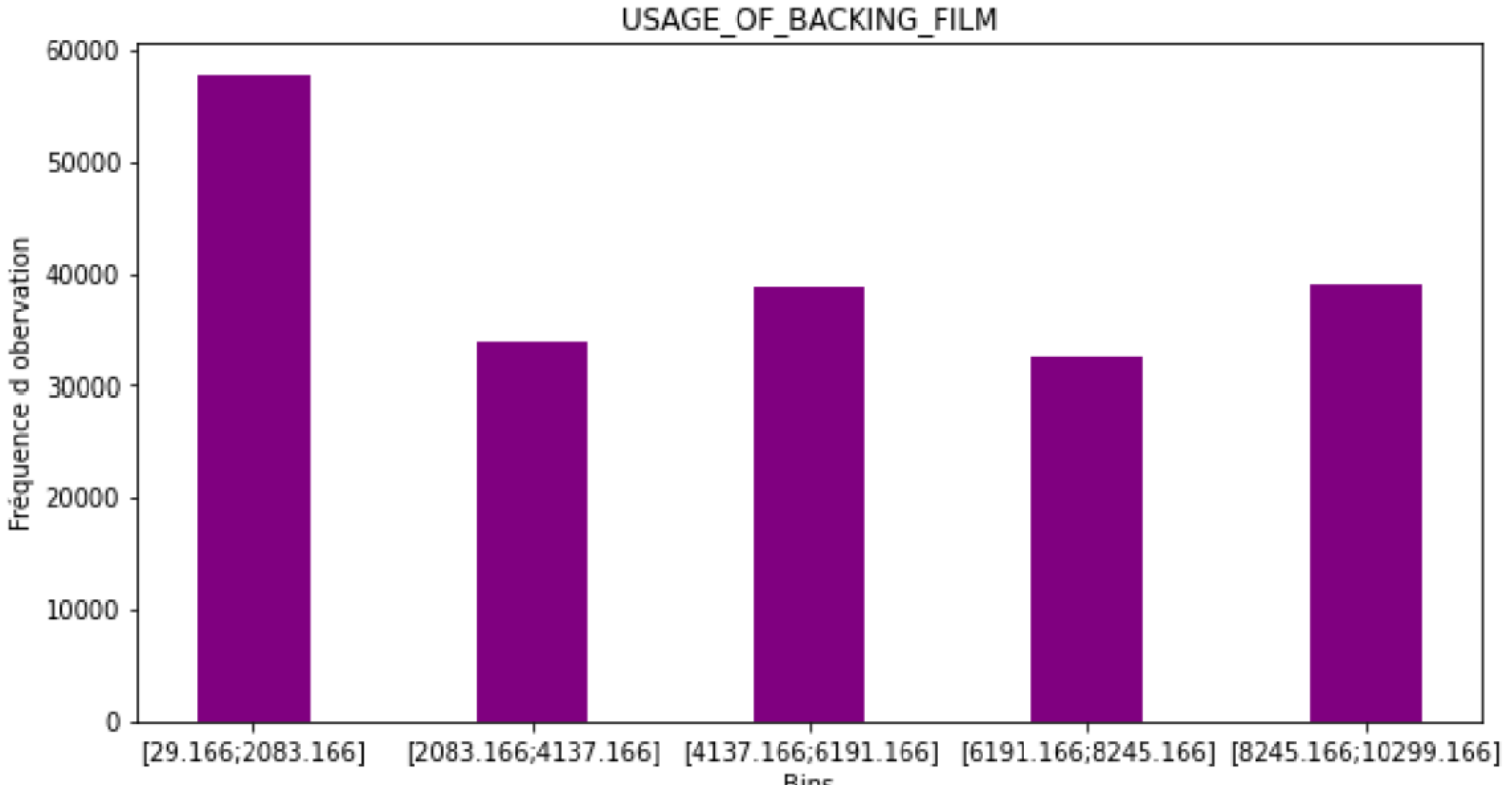
Distributions empiriques

```python
import numpy as np
import matplotlib.pyplot as plt

# programme d'affichage des histogrammes
# des différentes séries temporelles
# dans cet exemple démonstratif,
# nous visualisons USAGE_OF_BACKING_FILM

data = {"[29.166;2083.166]":57755,
        "[2083.166;4137.166]":33868,
        "[4137.166;6191.166]":38884,
        "[6191.166;8245.166]":32606,
        "[8245.166;10299.166]":38971}
donnees = list(data.keys())
valeurs = list(data.values())
fig = plt.figure(figsize = (10, 5))
plt.bar(donnees, valeurs, color ='purple',width = 0.4)
plt.xlabel("Bins")
plt.ylabel("Fréquence d obervation")
plt.title("USAGE_OF_BACKING_FILM")
plt.show()
```

Distributions empiriques



Extraction des caractéristiques agrégées par wafer

```c
void regrouper_wafer1(char *filename, char *file2)
{
    FILE *fichier = fopen(filename, "r");
    FILE *fichier2 = fopen(file2, "w");

    double wafer = 0.0;
    char *ligne;
    int linesize = 250;
    int i = 0;

    ligne = (char *) malloc(linesize * sizeof(char));


    if (fichier != NULL)
    {
        while (fgets(ligne, linesize, fichier))
        {

            fscanf(fichier, "%*d    %*d %*lf    %lf %*s %*lf    %*
                           &wafer);

            fprintf(fichier2, "%lf \n", wafer);

            i++;

        }
    }
    free (ligne);
    fclose(fichier);
}
```

Extraction des caractéristiques agrégées par wafer

```c
void regrouper_wafer2(char *filename, char *file2, int tab[426])
{
    FILE *fichier = fopen(filename, "r");
    FILE *fichier2 = fopen(file2, "w");

    double wafer = 0.0;
    char *ligne;
    int linesize = 25;
    double controle = 371447024.000000; //initialisation du premier
    int i = 0;
    int cpt = 0;
    int j = 0;


    ligne = (char *) malloc(linesize * sizeof(char));


    if (fichier != NULL)
    {
        while (fgets(ligne, linesize, fichier))
        {
            fscanf(fichier, "%lf", &wafer);

            if (wafer == controle)                    //on contrôl
            {                                         // si c'est
                                                      // données u

                cpt++;
                i++;
                continue;
            }

            fprintf(fichier2, "%lf \n", controle);

            if (wafer != controle)                    //sinon , on
            {
                controle = wafer;
                tab[j] = cpt;
                j++;
                cpt = 0;
            }

            i++;
        }
```