Projet informatique

L'ENERGIE SOLAIRE

Table des matières

Introduction

L'énergie solaire est une énergie renouvelable, mais malheureusement souvent peu rentable car basée sur un grand nombre de paramètres liés à la fois aux conditions temporelles (date, heure) et météorologiques (temps, humidité, altitude...) ainsi que géographiques (latitude, longitude, inclinaison, orientation).
De plus, le panneau photovoltaïque lui-même possède un rendement faible (moins de 20% en général). Il est donc difficile d'obtenir de bonnes conditions de production d'électricité à partir d'énergie solaire.

Cependant, des calculs peuvent être effectués en fonction de ces différents paramètres pour obtenir une production optimale, notamment à l'aide de l'outil informatique. Le but de ce projet est justement de concevoir un programme capable de prévoir à partir de différents paramètres d'entrée le rayonnement solaire reçu par un panneau. Pour cela, on se placera dans le cas d'un temps ensoleillé.

Découpage fonctionnel du projet

La prise en compte d'un grand nombre de paramètres pour cette détermination implique un nombre important de saisie d'informations de l'utilisateur, puis de calculs utilisant les valeurs entrées. Il faudra éventuellement convertir certaines de ces informations, soit parce qu'elles sont plus parlantes sous une certaine forme pour l'utilisateur et qu'elles ne sont pas nécessairement sous la même forme pour les calculs, soit parce que les valeurs utilisées pour les calculs sont inconnues de l'utilisateur. Par exemple, l'utilisateur est capable d'entrer l'heure légale au moment de la saisie, mais probablement pas l'heure solaire vraie qui lui correspond.

Il sera également nécessaire de communiquer à l'utilisateur la fonction du programme, ainsi que le résultat (le rayonnement solaire global) de manière parlante, et éventuellement en récapitulant l'ensemble de ses saisies.

Etude des fonctions de saisie

L'étude de la méthode de calcul fournie par le document technique a abouti à une liste de saisies par l'utilisateur nécessaires, qui sont les suivantes (de la manière la plus parlante possible pour l'utilisateur) :

- La date (sous la forme jj / mm/ aaaa)
- L'heure légale (sous la forme hh : mm : ss)
- La latitude Nord et la longitude Est sous forme sexagésimale (degrés, minutes secondes → dd : mm : ss)
- L'inclinaison et l'orientation du panneau (en degrés)
- L'altitude du lieu considéré (en mètres)
- La température (en °C)
- Le milieu auquel appartient le lieu considéré (montagnard, rural, urbain ou industriel)
- Le taux d'humidité en %

Pour cela, il était utile de déclarer de nouveaux types (types composés) permettant de stocker les informations plus facilement

D'abord, un type `date` composé de 3 entiers (jour, mois et année) qui permettra de stocker la date entrée par l'utilisateur

```
struct date
{
	int j;
	int m;
	int a;
};
```

Un type `heure` composé de 3 entiers (heures, minutes, secondes) qui permettra de stocker l'heure légale entrée par l'utilisateur

```
struct heure
{
	int h;
	int m;
	int s;
};
```

Un type `coordonnee` composé de 3 entiers (degrés, minutes, secondes), qui permettra de stocker les valeurs sexagésimales de latitude EST et de longitude NORD saisies par l'utilisateur

```
struct coordonnee
{
	int d;
	int m;
	int s;
};
```

Un type `situation` composé de 2 réels, qui contiendra la latitude et la longitude décimales une fois converties

```
struct situation
{
	float la;
	float lo;
};
```

Deux groupes de travail se sont chargés de ces fonctions de saisie, c'est-à-dire à la fois assurer le respect des conditions de saisie et parfois modifier la forme des informations reçues.

Travail du 1er groupe

Membres du groupe :

- Clément Fayolle
- Antoine Peyrard
- Romain Velu
- Charles Gadeceau

Responsable : Corentin Faure

1ère fonction : saisie de la date
`Saisie_date` : la fonction (vide) → 1 date

R : Demande à l'utilisateur de saisir une date valide (dans le cas contraire, le programme doit redemander la date jusqu'à ce qu'elle soit valide, en tenant compte des années bissextiles)
E : vide
S : 1 date (type à déclarer, composé de 3 champs : 3 entiers (jour, mois et année sous la forme jj mm aaaa))
Le type composé permet dans ce cas de retourner les trois informations relatives à la date en une seule fonction.

On utilise une boucle `do while` qui redemandera de nouvelles valeurs pour le jour le mois et l'année tant que celle-ci seront incorrectes (qu'elles ne répondront pas aux conditions énoncé précédemment).

Pour cette fonction les tests suivants ont été réalisés : on compare si le jour le mois et l'année ne sont pas négatifs ou nuls, ou supérieurs à leur valeur habituelle, on vérifie ensuite que le jour entré existe en prenant compte des mois à 30 ou 31 jours, et des années bissextile concernant février (pour savoir si l'on a 28 ou 29 jours)

```
do {
	cout << "Saisir un jour sous la forme jj :     ";
	cin >> j1;
	cout << "Saisir un mois sous la forme mm :     ";
	cin >> m1;
	cout << "Saisir une annee sour la forme aaaa : ";
	cin >> a1;
```

On effectuera cette demande ainsi que toutes les vérifications suivantes tant que la saisie est mauvaise. C'est-à-dire :

Premièrement, on vérifie si le mois entré est inférieur ou égal à 12 :

```
	if (m1 <= 12)


	{
```

Dans ce cas, on initalise le booléen `erreur` à `false`. Il passera à `true` dès que la saisie sera incorrecte :

```
		erreur = false;
```

Si le jour, mois ou année est négatif

```
		if (m1 <= 0 || j1 <= 0)
		{
			erreur = true;
			cout << "Date entree incorrecte" << endl << endl;
		}
```

On élimine tous les cas où la saisie est correcte sauf pour le mois de février

```
		else
		{
				if (!(m1 == 01 && j1 <= 31 || m1 == 03 && j1 <= 31 || m1 == 05 &&
				j1 <= 31 || m1 == 07 && j1 <= 31 || m1 == 8 && j1 <= 31 || m1 ==
				10 && j1 <= 31 || m1 == 12 && j1 <= 31 || m1 == 04 && j1 <= 30 ||
				m1 == 06 && j1 <= 30 || m1 == 9 && j1 <= 30 || m1 == 11 && j1 <=
				30))
```

Ensuite, on regarde si l'année n'est pas bissextile. Dans ce cas, le jour associé à février ne doit pas être supérieur à 28. (Pour savoir si l'année est bissextile, on utilise une fonction créée par le groupe 3 que l'on détaillera ensuite)

```
			{
				if (!(est_bissextile(a1)))
				{
					if (j1 > 28)
					{
						erreur = true;

						cout << "Date entree incorrecte"
						<< endl<< endl;
					}
				}
```

Dans le cas contraire (l'année est bissextile), le jour ne doit pas être supérieur à 29.

```
				else
				{
```

```
                                        if (j1 > 29
                                        {

                                                erreur = true;
                                                cout << "Date entree incorrecte" << endl << endl;
                                        }
                                }
                        }
                }
                }
```

Enfin, on précise que lorsque la condition de départ (mois inférieur à 12) n'est pas vérifiée, le booléen passe évidemment à `true`.

```
        else
        {
                erreur = true;
                cout << "Date entree incorrecte" << endl << endl;
        }
```

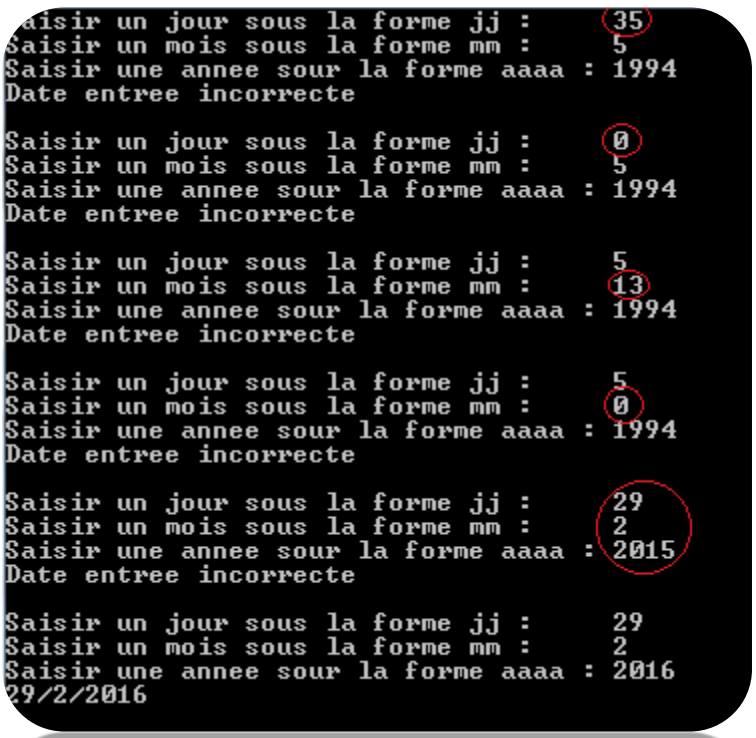
On effectuera donc toutes les saisies et vérifications tant que les 3 données entrées par l'utilisateur seront incorrectes :

```
        } while (erreur == true);
```

Nb : il n'y a pas de contrainte de saisie pour l'année (peut être positive, négative ou nulle)
Enfin, on affecte les différentes saisies à la variable de type date que l'on retourne ensuite.

```
        dateprojet.j = j1;
        dateprojet.m = m1;
        dateprojet.a = a1;

        return dateprojet;
}
```



En rouge : les valeurs fausses (pour les 2 dernières saisies, le 29/2/2015 n'existe pas car 2015 n'est pas une année bissextile, et le 29/2/2016 existe car 2016 est une année bissextile).

2ème fonction : saisie de l'heure légale

`Saisie_heure` *: la fonction (vide)* → *1 heure*

R : Demande à l'utilisateur de saisir une heure légale valide (dans le cas contraire, le programme doit redemander l'heure jusqu'à ce qu'elle soit valide)
E : vide
S : 1 heure (type à déclarer, composé de 3 champs : 3 entiers (heures, minutes, secondes))

Pour cette fonction on vérifie que les couples heures/minutes/secondes saisie sont bien composés d'heures positives et strictement inférieur à 24, pour les minutes nous avons vérifié qu'elles soient bien positives et inférieures strictement à 60 (ce qui représenterait une heure), et pour finir on réédite la même opération pour les secondes (de la même façon que pour les minutes, car 60 secondes représenteraient une minute).

```
while (time.h < 0 || time.h > 23 || time.m < 0 || time.m > 59 || time.s < 0 || time.s > 59) {
            cout << "Les heures, minutes ou secondes que vous avez entrees ne sont pas
            valides, ressaisissez les : " << endl;
            cin >> time.h;
            cin >> time.m;
            cin >> time.s;
      }

      return time;
}
```

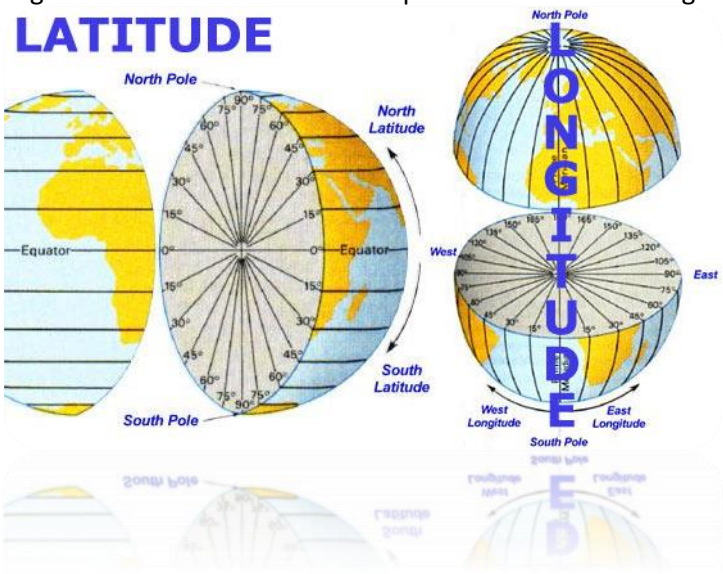
On utilise une boucle while qui va afficher un message d'erreur et saisir une nouvelle valeur sous la forme heure minutes secondes tant que les conditions ne seront pas respectées.
On utilise le type composé heure pour pouvoir retourner les trois saisies heures, minutes et secondes dans la même fonction.

3ème fonction : saisie de la latitude NORD

`Saisie_latitude` : la fonction (vide) → 1 coordonnée

R : Demande à l'utilisateur de saisir une latitude Nord en degrés sexagésimaux valide (dans le cas contraire, le programme doit redemander la latitude jusqu'à ce qu'elle soit valide)
E : vide
S : 1 coordonnée (type composé des degrés, des minutes et des secondes (3 entiers))

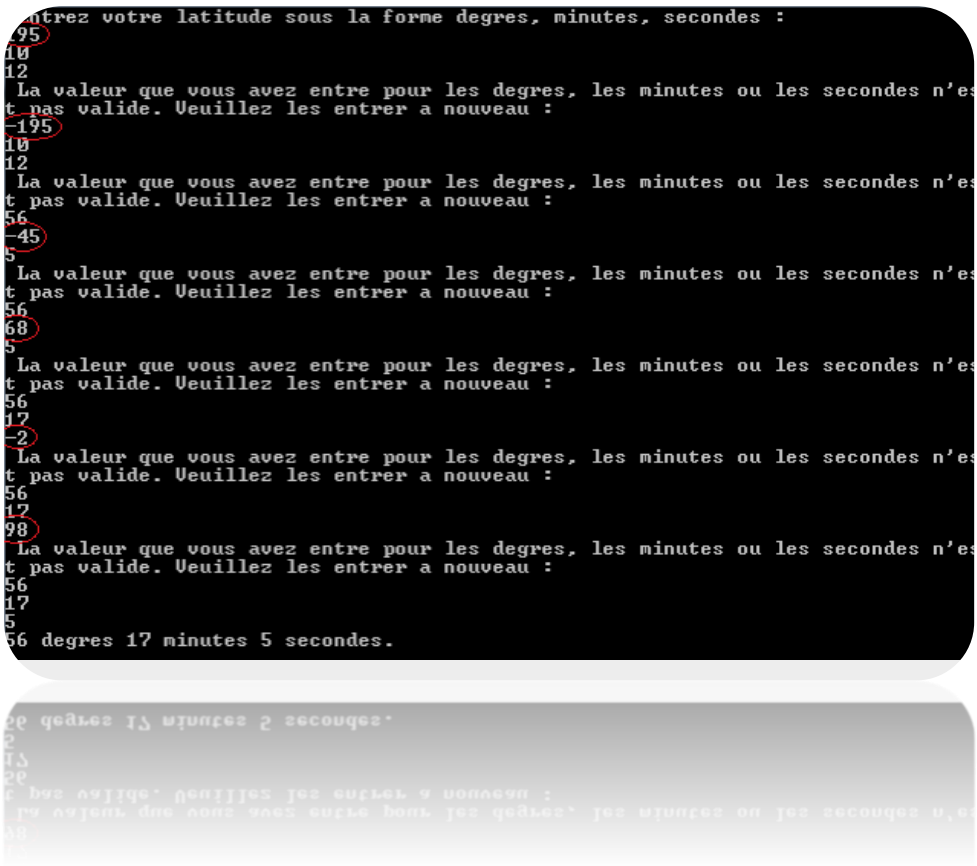
On effectue des tests pour vérifier que les valeurs saisies par l'utilisateur sont correctes. On vérifie donc que la partie degrés de la latitude est bien comprise entre -90 et 90 degrés qui sont extremums atteignables. (voir figure)



```
        while (lati.d > 90 || lati.d < -90 || lati.m < 0 || lati.m > 59 || lati.s < 0 ||
        lati.s > 59 || ((lati.d==90 || lati.d==-90) && (lati.m!=0 || lati.s!=0))
        {
                cout << " La valeur que vous avez entre pour les degres, les minutes ou les
                secondes n'est pas valide. Veuillez les entrer a nouveau : " << endl;
                cin >> lati.d;
                cin >> lati.m;
                cin >> lati.s;
        }
        return lati;
}
```

On vérifie que les degrés sont bien compris entre -90 et +90, les minutes et secondes entre 0 et 59. Enfin, du fait de la forme sexagésimale, lorsqu'on entre -90 ou +90, il faut que les champs minutes et secondes soient nuls, sinon on dépasse -90 ou 90 degrés.

Le type composé coordonnee permet de retourner les trois informations sexagésimales relatives à la latitude (degrés, minutes, secondes) en une seule fonction.



4ème fonction : saisie de la longitude EST

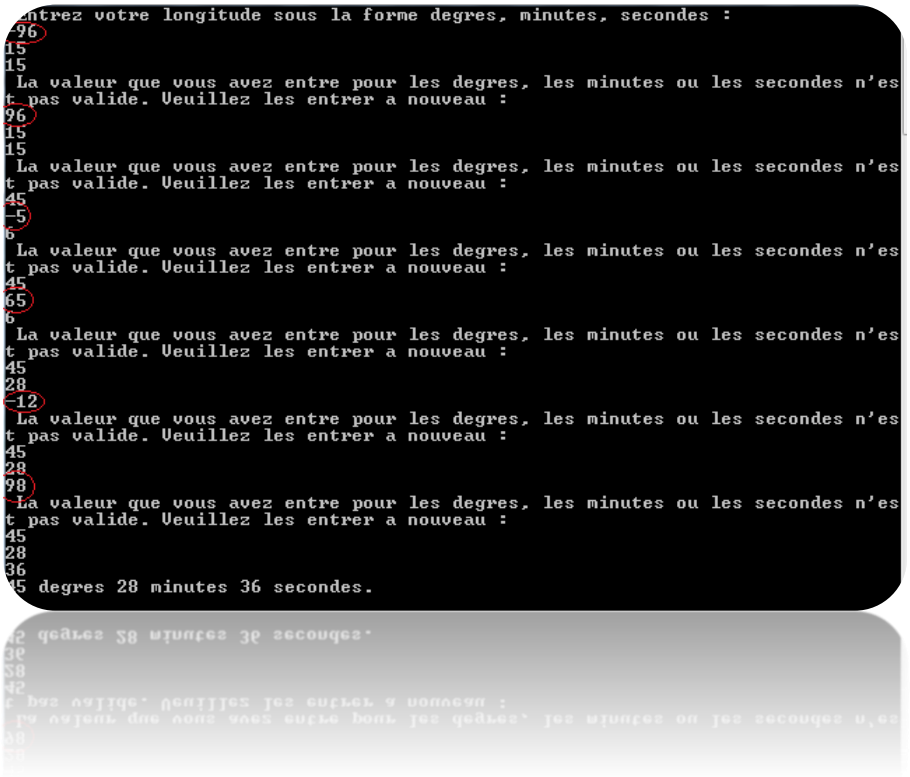
`Saisie_longitude` : la fonction (vide) → 1 coordonnée

R : Demande à l'utilisateur de saisir une longitude Est en degrés sexagésimaux valide (dans le cas contraire, le programme doit redemander la longitude jusqu'à ce qu'elle soit valide)
E : vide
S : 1 coordonnée (type composé des degrés, des minutes et des secondes (3 entiers))

Les tests effectués pour la fonction sont semblables à ceux de celle de la latitude, à l'exception près que la longitude est, elle, comprise entre -180 et 180 degrés.

```
while (lng.d < -180 || lng.d > 180 || lng.m < 0 || lng.m > 59 || lng.s < 0 || lng.s >
59 || ((lng.d==180 || lng.d==-180) && (lng.m!=0 || lng.s!=0)) )
{
	cout << " La valeur que vous avez entre pour les degres, les minutes ou les
	secondes n'est pas valide. Veuillez les entrer a nouveau : " << endl;
	cin >> lng.d;
	cin >> lng.m;
	cin >> lng.s;
}
return lng;
```

Le type composé coordonnee permet encore une fois de stocker des données sexagésimales et de les retourner en une seule fonction (ici, pour les données de longitude).



5ème fonction : saisie de l'inclinaison du panneau

Saisie_inclinaison : la fonction (vide) → 1 réel

R : Demande à l'utilisateur de saisir une inclinaison en degrés décimaux du panneau valide (dans le cas contraire, le programme doit redemander l'inclinaison jusqu'à ce qu'elle soit valide)
E : vide
S : 1 réel (l'inclinaison du panneau en degrés)

```
while (incli > 90 || incli < 0)
	{
	cout << "Votre inclinaison n'est pas comprise entre 0 et +90 degres, ressaisissez la : " <<
	endl;
	cin >> incli;
	}
	return incli;
```

L'inclinaison du panneau doit être comprise entre 0 et 90 degrés, d'où le test effectué précédemment dans la boucle while.

Travail du 2ème groupe

Membres du groupe :

- Hugo Morais-Costa
- Nicolas Galley
- Amine Khalis (démission)

1ère fonction : saisie de l'orientation du panneau

`Saisie_orientation` : la fonction (vide) → 1 réel

R : Demande à l'utilisateur de saisir une orientation en degrés décimaux du panneau valide (dans le cas contraire, le programme doit redemander l'orientation jusqu'à ce qu'elle soit valide)
E : videS : 1 réel (l'orientation en degrés)

La fonction `Saisie_orientation` permet de demander à un utilisateur de saisir une orientation valable et retourne la valeur de cette dernière à l'algorithme principal si elle est valable. Afin de vérifier la fiabilité de la valeur saisie, une boucle « Tant que » redemande à l'utilisateur de saisir une orientation valable tant qu'elle n'est pas comprise entre -180 et 180 degrés décimaux.

```
while (orient > 180 || orient <= -180)
                    {
                    cout<<"Votre orientation n'est pas comprise entre -180 et +180 degres,
                    ressaisissez la : " << endl;
                    cin>>orient;
                }
```

```
Entrez une orientation en degres decimaux entre -180 (exclu) et +180 degres :
-566
Votre orientation n'est pas comprise entre -180 (exclu) et +180 degres, ressaisi
ssez la :
26
Appuyez sur une touche pour continuer...
```

2ème fonction : saisie de l'altitude du lieu

`Saisie_alt` : la fonction (vide) → 1 réel

R : Demande à l'utilisateur de saisir une altitude en mètres valide (dans le cas contraire, le programme doit redemander l'altitude jusqu'à ce qu'elle soit valide)
E : vide
S : 1 réel (l'altitude)

La fonction `Saisie_alt` sert à demander à l'utilisateur de rentrer l'altitude voulue et vérifie si elle est valable par l'intermédiaire d'une boucle « Tant que ». Si c'est le cas elle l'envoie à l'algorithme principal. Sinon elle signale l'erreur à l'utilisateur et demande une nouvelle valeur à l'utilisateur qui est vérifiée de la même manière. Nous avons pris comme valeurs extrêmes -422 mètres (Niveau de la mer morte, soit le plus bas) et 8848 mètres (Niveau de l'Everest, soit le plus haut)

```
while (alt>8848 || alt<-422)
                {
                cout<< "Erreur: "<<alt<<" n'est pas une altitude valable, saisir une altitude
                valable."<<endl;
                cin>>alt;
                }
```

```
Saisir une altitude valable (en metres)
9000
Erreur: 9000 n'est pas une altitude valable, saisir une altitude valable.
-403
Appuyez sur une touche pour continuer...
```

3ème fonction : saisie de la température

`Saisie_temperature` : la fonction (vide) → 1 réel

R : Demande à l'utilisateur de saisir une température en °C rationnelle (si elle ne l'est pas, le programme doit redemander la température jusqu'à ce qu'elle le soit)
E : vide
S : 1 réel (la température)

La fonction `Saisie_temperature` a pour but de permettre à l'utilisateur d'entrer la température choisie, nécessaire à l'exécution des calculs ultérieurs. L'algorithme de la fonction précise à l'utilisateur d'entrer une température en degrés Celsius, et rationnelle (On considère qu'une température est rationnelle si elle est comprise entre -70 et 70°C). L'algorithme de la fonction comporte une boucle « Tant que » qui vérifie cette condition, afin d'éviter une erreur de saisie menant à une température invraisemblable.

```
while (temp<-70 || temp>70
	{
	cout<<"Erreur,la temperature doit etre en degre celsius et rationnelle. Veuillez
	recommencer"<<endl;
	cin>>temp;
	}
```

```
Veuillez saisir une temperature (en degre celsius) rationnelle.
71
Erreur,la temperature doit etre en degre celsius et rationnelle. Veuillez recomm
encer
50
Appuyez sur une touche pour continuer...
```

4ème fonction : saisie du milieu

`Saisie_coef` : la fonction (vide) → 1 réel

R : Demande à l'utilisateur de saisir un milieu (1 : rural, 2 : montagnard, 3 : urbain ou 4 : industriel) et redemande de saisir ce milieu tant qu'il n'a pas saisi un de ces 4, puis selon le milieu saisi retourne le bon coefficient de trouble atmosphérique
E : vide
S : 1 chaîne de caractères (le milieu)

La fonction `Saisie_milieu` permet à l'utilisateur de saisir un milieu parmi les suivant:
-Montagnard : dans ce cas on retournera 0.01
-Rural : dans ce cas on retournera 0.05
-Urbain : dans ce cas on retournera 0.1
-Industriel : dans ce cas on retournera 0.2

La valeur correspond au coefficient de trouble atmosphérique (lié à la pollution), qui servira pour des calculs ultérieurs.

Selon le milieu choisi, la fonction retourne le coefficient de trouble atmosphérique relatif à ce dernier, nécessaire à des calculs ultérieurs. Une boucle « Tant que »permet de redemander à l'utilisateur de choisir un des 4 milieux tant qu'il ne le fait pas correctement, c'est à dire tant qu'il ne rentre pas un numéro correspondant à un de ces milieux.

```
while (choix!=1 && choix!=2 && choix!=3 && choix!=4)
		{
		cout<<"Erreur dans la saisie du milieu choisi, veuillez saisir un chiffre
		adequat"<<endl;
		cin>>(choix);
		}
	switch (choix)
			{case 1: coeff=0.02;
				break;
```

```
                case 2: coeff=0.05;
                        break;
                case 3: coeff=0.1;
                        break;
                case 4: coeff=0.2;
                        break;
            }
```

```
Veuillez saisir le chiffre correspondant au milieu choisi: 1) Montagnard 2) Rura
l 3) Urbain 4) Industriel
5
Erreur dans la saisir du milieu choisi, veuillez saisir un chiffre adequat
3
Le coefficient du milieu choisi est 0.1
Appuyez sur une touche pour continuer...
```

5ème fonction : saisie du taux d'humidité

`Saisie_hum` : la fonction (vide)→ 1 réel

R : Demande à l'utilisateur de saisir un taux d'humidité relative en % valide (dans le cas contraire, le programme doit redemander le taux jusqu'à ce qu'il soit valide), le convertit en taux pour 1 (entre 0 et 1) et le retourne
E : vide
S : 1 réel (le taux d'humidité relative (ex : 0,3))

La fonction `Saisie_hum` demande à l'utilisateur de rentrer la valeur du taux d'humidité (en %) présent dans le milieu qu'il a choisi. Si ce dernier ne rentre pas une valeur correcte, c'est à dire non comprise entre 0% et 100% alors une boucle « Tant que » permet de lui demander de saisir de nouveau cette valeur. Lorsque la valeur est correcte, la fonction la convertit sous forme décimale et la retourne à l'algorithme global.

```
    while (h<0 || h>100)
                {
                cout<<"Erreur, veuillez saisir un taux en pourcentage correct"<<endl;
                cin>>h;
                }
```

```
Saisir un taux d'humidite en %.
125
Erreur, veuillez saisir un taux en pourcentage correct
95
Appuyez sur une touche pour continuer...
```

Etude des fonctions de conversion

Après avoir garanti une saisie optimale des différentes informations par l'utilisateur, il est nécessaire de convertir certaines de ces saisies dans le but de traiter ces informations dans des calculs.
Tout d'abord, les heures telles qu'elles ont été saisies ne sont pas directement interprétables car sous forme composée (heures, minutes, secondes) : il faut donc les convertir en heures décimales.
C'est aussi le cas de la saisie de la date : il faudra associer par exemple à la date 15/02/2014 le rang du jour dans l'année (dans ce cas, 31+15=46). Pour cela, il sera important de savoir si l'année est bissextile.
On fait de même pour les coordonnées sexagésimales de la latitude et la longitude : on doit passer en coordonnées décimales pour les calculs.
Il faudra aussi pour des raisons pratiques, qui seront évoquées pour la partie concernant les calculs, pouvoir convertir un angle en degrés en radians, et inversement.
Pour éviter des calculs superflus par la suite, on déterminera également si l'heure solaire d'étude est comprise dans l'intervalle entre l'heure de lever et l'heure de coucher du soleil.
Ces différentes tâches ont été confiées aux groupes de travail n°3 et 4.

Travail du 3ème groupe

Membres du groupe :
- Alexis Chaudier
- Bastien Chabal
- Mohamed Oudouni

Responsable : Louis Poy-Tardieu

1ère fonction : conversion de l'heure composée en heure décimale

`Heures_decimales` : la fonction (h : 1 heure) → 1 réel

R : Calcule le temps en heures correspondant à une heure sous la forme heures, minutes, secondes
E : 1 heure (type composé des heures, des minutes et des secondes (3 entiers))
S : 1 réel

Pour pouvoir calculer l'heure sous forme décimale il faut tout d'abord diviser le champ représentant les secondes par 3600, puis on divise champ représentant les minutes par 60.
Enfin on additionne le type composé représentant les heures au résultat des deux premières opérations et on obtient une heure décimale.

```
h1=(1.0/60)*h.m;
        h2=(1.0/3600)*h.s;
        htot=h.h+h1+h2;
        return htot;
```

```
Entrez une date sous la forme hh, mm, ss
10
53
49
La valeur dúcimale est :10.8969
```

2ème fonction : conversion de la latitude et de la longitude sexagésimales en deux valeurs décimales

`Situation_geo` : la fonction (la : 1 latitude, lo : 1 longitude) → 1 situation

R : Calcule à partir de la latitude Nord et de la longitude Est sexagésimales leur équivalent respectif en degrés décimaux, les stocke sous une variable de type situation et la retourne
E : 2 coordonnées (type composé des degrés, minutes, secondes, pour la latitude Nord et la longitude Est ()
S : 1 situation (type composé de la latitude (1réel) et de la longitude (1réel) du lieu en degrés décimaux

La conversion des degrés sexagésimaux en degrés décimaux consiste à : additionner la partie minute de la division réelle des minutes par 60 et des secondes par 3600.

Pour effectuer les conversions il a fallu créer 2 variables locales.
- `situ.la`, de type réel permettant d'accéder à la conversion en degrés sexagésimaux de la latitude Nord.
- `situ.lo`, elle aussi de type réel permettant d'accéder à la conversion en degrés sexagésimaux de la longitude Est.

Ces deux conversions constituent le type composé de la situation géographique.

```
situ.la = latin.d + (latin.m / 60.0) + (latin.s / 3600.0);
situ.lo = longie.d + (longie.m / 60.0) + (longie.s / 3600.0);
```

```
return situ;
```

```
Saisir la coordonnúe en degrús, minutes et secondes de la latitude NORD:
33
49
13
Saisir la coordonnúe en degrús, minutes et secondes de la longitude EST:
44
58
22
La coordonnúe de la latitude NORD en degrús dúcimaux est: 33.8203La coordonnúe d
e la longitude EST en degrús dúcimaux est: 44.9728
```

3ème et 4ème fonction : conversion d'un angle en degrés en radians et inversement

Deg2Rad : la fonction (aD : 1 réel) → 1 réel

R : Convertit un angle en degrés en radians
E : 1 réel (angle en degrés)
S : 1 réel (angle en radians)

Rad2Deg : la fonction (aR : 1 réel) → 1 réel

R : Convertit un angle en degrés en radians
E : 1 réel (angle en degrés)
S : 1 réel (angle en radians)

Ces fonctions ne présentent pas de difficultés, elles consistent simplement en une déclaration d'une constante de conversion et d'un calcul intermédiaire avant de renvoyer la valeur en radians ou en degrés.

```
rad=(deg*PI)/180;
return rad;

deg=rad*(180/PI);
return deg;
```

Résultat du calcul pour la conversion de degrés vers radians :

```
Entrer la valeur des degres :
89
89 degres font : 0.680678 radians
```

5ème fonction : détermination des heures de lever et de coucher du soleil, et vérification par rapport à l'heure solaire considérée

Lever_coucher : la fonction (la : 1 réel, dec : 1 réel, h : 1 réel) → 1 booléen

R : Calcule à partir de la latitude Nord et de la déclinaison les heures de lever et de coucher de soleil, renvoie VRAI si l'heure saisie par l'utilisateur est dans cet intervalle et FAUX sinon
E : 3 réels (la latitude et la déclinaison en degrés décimaux et l'heure en heures décimales)
S : 1 booléen

Pour effectuer les calculs de cette fonction, il fallait incorporer « math.h » dans le .cpp. En effet, nous devions avoir accès aux fonctions tangentes et arccosinus pour calculer les heures de lever et de coucher du soleil. Cela constituait le seul vrai problème de la fonction. Le reste ne consistait qu'en l'utilisation de Si et de variable booléenne. On retourne finalement la variable « res » correspondant à l'état de l'heure solaire vraie entrée par l'utilisateur : Soit à l'intérieur de l'intervalle, soit à l'extérieur. Du fait que les fonctions trigonométriques travaillent en radians dans le module « math.h », il fallait prendre en compte le fait que la fonction tangente prend des radians en entrée et arcosinus donne des radians en sortie, d'où l'utilité des deux fonctions Deg2Rad et Rad2Deg. La déclinaison est une quantité dont la méthode de calcul sera détaillée dans la partie « calcul ».

```
h_lever = 12 - (Rad2deg (acos (-tan (Deg2rad (la)) * tan (Deg2rad (dec)))))/15;
h_coucher = 12 + (Rad2deg (acos (-tan (Deg2rad (la)) *tan (Deg2rad(dec)))))/15;

if (h_lever < h && h_coucher > h)
{
	res = true;
}
else
{
	res = false;
}

return res;
```

Ci-contre, vous pouvez observer le déroulement de la fonction.
On entre dans ce cas des valeurs telles qu'on sait que l'heure sera comprise dans l'intervalle et donc le booléen sera VRAI. (remarquons que lorsqu'on souhaite afficher la valeur d'un booléen en C++, la fonction cout affiche 0 ou 1 selon si sa valeur est `false` ou `true`)

6ème fonction : détermination du caractère bissextile d'une année

`est_bissextile` : la fonction (a : 1 entier)→ 1 booléen

R : Retourne VRAI si l'année passée en paramètre est bissextile et FAUX sinon
E : l'année considérée (1 entier)
S : VRAI ou FAUX selon l'année

Cette fonction a déjà été vue auparavant dans le chapitre sur les structures conditionnelles du cours d'info1. Cet arbre d'imbrication des « si » repose sur le fait qu'une année bissextile doit être divisible par 4. Si celle-ci est séculaire, on doit également vérifier si elle est divisible par 400 et pas par 4000.

```
if(r4!=0)
{
	res=false;
}
else
{
	if(r100!=0)
	{
		res=true;
	}
	else
	{
		if(r400==0 && r4000!=0)
		{
			res=true;
		}
		else
		{
			res=false;
		}
	}
}
return res;
```

7ème fonction : calcul du rang du jour associé à une date

La fonction pour rôle de renvoyer le nombre entier correspondant à la date passée en paramètres, en tenant compte des années bissextiles.

Tout d'abord, on vérifie si l'année est bissextile ou non. (Fonction déjà vue précédemment)
Si l'année est bissextile, un booléen passe à VRAI. Sinon, il est FAUX.
On observe ensuite deux conditions :

- Si le booléen est vrai, alors février comptera 29 jours.
- Si il est faux, alors février comptera 28 jours.

```
res = est_bissextile(d.a);
```

Après avoir traité l'année, le calcul de l'entier à renvoyer est différent selon le mois. On utilise donc la fonction SELON.
Si le mois est égal à 1, janvier, alors l'entier correspond au nombre de jours, car janvier et le premier mois de l'année.
Si le mois est égal à 2, février, alors l'entier correspond au nombre de jours, plus le nombre de jours que contient janvier. (soit 31+j)

```
switch(d.m)
{
case 1: jour=d.j;
        break;
case 2: jour=31+d.j;
        break;
case 3: jour=31+28+d.j;
        break;
case 4: jour=31+28+31+d.j;
        break;
case 5: jour=31+28+31+30+d.j;
        break;
case 6: jour=31+28+31+30+31+d.j;
        break;
case 7: jour=31+28+31+30+31+30+d.j;
        break;
case 8: jour=31+28+31+30+31+30+31+d.j;
        break;
case 9: jour=31+28+31+30+31+30+31+31+d.j;
        break;
case 10: jour=31+28+31+30+31+30+31+31+30+d.j;
        break;
case 11: jour=31+28+31+30+31+30+31+31+30+31+d.j;
        break;
case 12: jour=31+28+31+30+31+30+31+31+30+31+30+d.j;
        break;
default: cout<<"Erreur de saisie"<<endl;
}
```

Le booléen intervient à partir du mois 3 (mars), car il change le nombre de jours que contient février. On ajoutera alors 28 ou 29. Le système est le même pour tous les mois suivants, on prend le nombre de jours passé en paramètre (dans le type composé de la date), et on y ajoute le nombre de jours que contiennent les mois précédents.

```
if(res==true)
{
        if(d.m>=3)
        {
                jour=jour+1;
```

```
        }
    }
```

La fonction retourne finalement l'entier correspondant au nombre de jours.

Travail du 4ème groupe

Membres :

- Charles Silvestre
- Damien Ciprelli

Responsable : Théo Basty

Afin de calculer l'ensoleillement, il est nécessaire de calculer l'heure solaire correspondante à l'heure légale choisie par l'utilisateur.
Pour cela il a fallu créer la fonction heure_solaire_vraie :

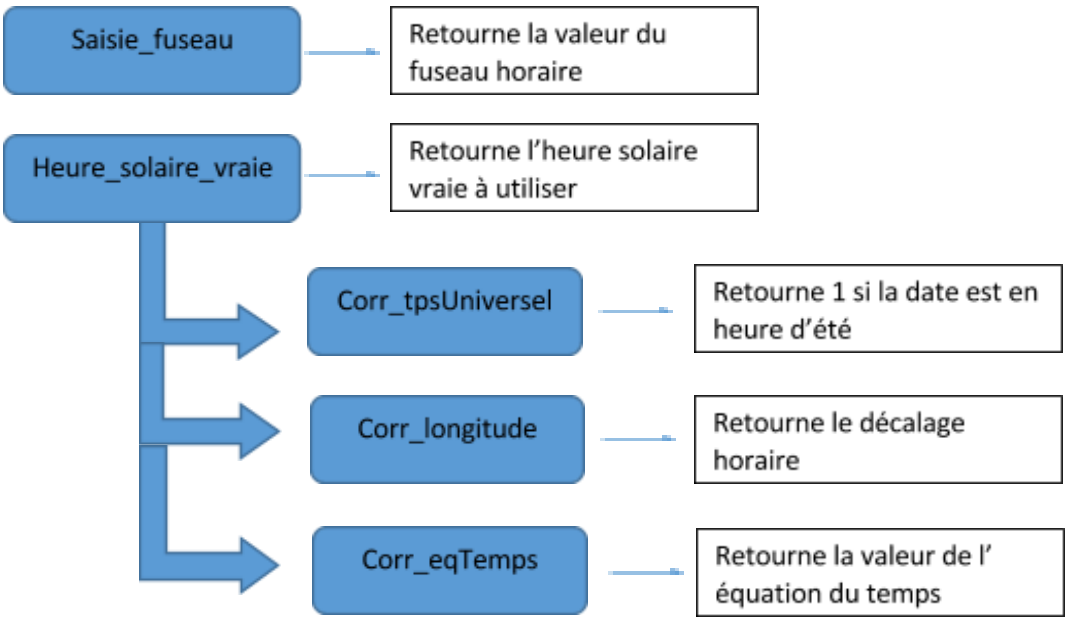
`Heure_solaire_vraie :` la fonction (lon : 1 réel, heure_dec : 1 réel, date : 1 date)→ 1 réel

R : Calcule à partir de la longitude en degrés décimaux, de l'heure légale en heures décimales (ex : 1,23 h et pas 1h 20min 5s…) et de la date (pour connaître si c'est l'heure d'hiver ou l'heure d'été, sachant que les dates de passage heure d'hiver/été et été/hiver varient selon l'année considérée), l'heure solaire vraie (en heures décimales également)
Ps : on considère qu'on utilise le système heure d'été / heure d'hiver français
E : 2 réels (longitude en degrés décimaux et heure légale en heures décimales), 1 date (type composé de 3 entiers : jour (jj), mois (mm) et année (aaaa))
S : 1 réel (heure solaire vraie)

L'heure solaire se calcule en soustrayant différents éléments à l'heure légale, calculés chacun à l'aide d'une sous-fonction :

- Le fuseau horaire qui peut être soustrait directement à l'heure légale
  `Saisie_fuseau :` la fonction (vide)→ 1 entier
- L'heure d'été qui rajoute une heure au fuseau horaire :
  `Corr_tpsUniversel :` la fonction (today : 1 date)→ 1 entier
  - Cependant, en France, la période de l'heure d'été commence le dernier dimanche du mois de mars et se termine le dernier dimanche du mois d'octobre. Il a donc fallu créer une autre sous fonction pour terminer le rang du dernier dimanche dans le mois :
    `Dernier_dimanche` : la fonction (d : 1 date)→ 1 entier
- Le décalage dû à l'inclinaison par rapport au méridien de référence
  `Corr_longitude :` la fonction (lon : 1 réel)→ 1 réel
- L'équation du temps qui permet de corriger les irrégularités du mouvement de la terre en fonction du jour :
  `Corr_eqTemps` : la fonction (date_in : 1 date)→ 1 réel
  - Dans le document fournis, l'équation du temps est découpée en deux sous fonctions, l'ellipticité, et l'oblicité. Afin d'être plus clair, on a préféré créer deux fonctions qui calculent ces deux résultats indépendamment :
    `Ellipticite :` la fonction (j : un entier non signé)→ 1 réel
    `Oblicite :` la fonction (j : un entier non signé)→ 1 réel

L'appel de ces fonctions s'effectue de la manière suivante :



Le principal problème rencontré a été de redécouper la fonction heure_solaire_vraie en d'autres sous fonctions plus simple.
Pour Saisie_fuseau, il fallait tester si le fuseau entré était correct. Pour cela on utilise une boucle FAIRE TANT QUE, avec un booléen error que l'on fait passer à true lorsque la saisie est mauvaise et à false lorsque la valeur est juste en utilisant un SI

```
do
{
	cout << "Saisissez le fuseau horaire correspondant à l'heure saisie (Ex.:
	1 ou -1) : ";
	cin >> fuseau;
	if (fuseau < -12 || fuseau > 12)
	{
		error = true;
		cout << "la valeur saisie est incorrecte !" << endl << endl;
	}
	else
	{
		error = false;
	}
} while (error);
```

Pour Dernier_dimanche, on utilise une équation permettant de déterminer quel est le dernier jour du mois (0 pour le dimanche, 1 pour le lundi …) puis on soustrait cette valeur au nombre de jour dans le mois (31 pour mars et octobre). L'équation demande cependant de calculer un modulo d'un réel ce qui est impossible en C, on affecte donc la valeur dans une variable entière intermédiaire.

```
intermed = ceil(31 + d.a + (d.a / 4.0) - (d.a / 100.0) + (d.a / 400.0) + 31 * (d.m / 12.0));
j = 31 - (intermed % 7);
```

Pour Corr_tpsUniversel, On effectue différents tests sur le mois pour déterminer si c'est un mois bascule (mars ou octobre), un mois d'été ou un mois d'hiver. Si c'est un mois bascule, on vérifie si le jour se situe avant ou après le dernier dimanche du mois puis on affecte la valeur à retourner en fonction de la période.

```
if (today.m > 3 && today.m < 10) {
	corr_periode = 1;
} else if (today.m == 3) {
	if (today.j < dernier_dimanche(today)) {
		corr_periode = 0;
	} else {
```

```
                corr_periode = 1;
        }
    } else if (today.m == 10) {
        if (today.j >= dernier_dimanche(today)) {
                corr_periode = 0;
        } else {
                corr_periode = 1;
        }
    } else {
        corr_periode = 0;
    }
```

Pour `Corr_longitude`, on applique simplement l'équation t = -4*λ qui donne le décalage en minutes puis on le convertit en heures en divisant par 60.
En effet, la longitude EST est comptée négativement dans ce calcul, contrairement à la façon habituelle de signer la longitude (où elle est positive et la longitude OUEST négative)

```
    delta=-4*lon;
    t=delta/60.0;
```

Pour `Corr_eqTemps`, on commence par convertir la date en nombre de jours grâce à la fonction `Nb_jours` puis on appelle les deux fonctions `Oblicite` et `Ellipticite` retournant chacune une composante de l'équation de Temps dont on fait la somme, puis que l'on divise par 60 pour l'avoir en heures.

`Ellipticite`

```
    const float A = 0.98560028;
    const float B = 357.5261;

    //algo local
    C = (1.9148*sin(Deg2rad(B + A + j)) + 0.02*sin(Deg2rad(2 * (B + A*j))) + 0.0003*sin(Deg2rad(3 * (B
+ A*j)))) * 4;
```

`Oblicite`

```
    const float T = 0.98564736;
    const float R = 280.4665;
    float O;



    //algo local
    O = (-2.468*sin(Deg2rad(2 * (R + T*j))) + 0.053*sin(Deg2rad(4 * (R + T*j))) - 0.0014*sin(Deg2rad(6
    * (R + T*j)))) * 4;
```

`Corr_eqTemps`

```
    j = Nb_jours(date_in);
    c = Ellipticite(j);
    o = Oblicite(j);
    Eq = (c + o)/60;
```

Enfin, `Heure_Solaire_Vraie` appelle les 3 fonctions `Corr_tpsUniversel`, `Corr_longitude` et `Corr_eqTemps` puis convertit l'heure légale en heure solaire vraie en soustrayant les 3 valeurs obtenues grâce aux fonctions appelées et le fuseau horaire à l'heure légale.

```
    eqTemps = Corr_eqTemps(d);
    longi = corr_longitude(longitude);
    ete_hiver = Corr_tpsUniversel(d);

    heure_solaire_vraie = heure - fuseau - ete_hiver - longi - eqTemps;
```

Etude des fonctions de calcul

Une fois toutes les différentes valeurs saisies et converties de manière optimale, il faut procéder aux calculs nécessaires à la détermination du rayonnement solaire global. Cela nécessitait la détermination des quantités suivantes :

- La déclinaison
- L'angle horaire
- La hauteur solaire
- L'azimut
- Le coefficient d'incidence
- L'énergie solaire
- La masse d'air optique relative
- L'épaisseur optique de Rayleigh
- Le coefficient de trouble de Linke
- Le rayonnement solaire direct
- Le rayonnement solaire diffus

On peut distinguer deux parties dans le raisonnement bien qu'elles soient liées :

La première concerne des calculs plutôt angulaires. En effet, la déclinaison, l'angle horaire, la hauteur solaire et l'azimut constituent des angles liés à la position du soleil en fonction de l'heure de la journée et de la situation géographique. Le coefficient d'incidence est lié aux angles d'inclinaison et d'orientation du panneau solaire.
Tout d'abord, la déclinaison se calcule par rapport au jour de l'année.
On calcule l'angle horaire en fonction de l'heure solaire vraie associée au cas donné.
Grâce à ces deux quantités ainsi qu'à la latitude du lieu considéré, on en déduit la hauteur solaire correspondante.
Enfin, à partir de ces trois valeurs calculées (déclinaison, angle horaire et hauteur solaire), on en déduit l'azimut du soleil.
Le coefficient d'incidence du panneau est ensuite calculé à l'aide de ce même azimut ainsi que de la hauteur solaire. Entrent également en considération l'inclinaison et l'orientation du panneau photovoltaïque.
Cela a constitué l'ensemble du travail du 5ème groupe.

On a ensuite une deuxième partie de calculs liés plutôt à des conditions climatiques et météorologiques mais qui utilisent aussi les calculs précédents pour arriver au résultat final du rayonnement solaire global.
On calcule d'abord l'énergie solaire à partir du jour de l'année considérée.
Ensuite, à partir de l'altitude du lieu considéré, on calcule la masse d'air optique relative puis directement à partir de celle-ci on détermine l'épaisseur optique de Rayleigh.
Parallèlement, on utilise, la température, le coefficient de trouble atmosphérique ainsi que le taux d'humidité relative saisis par l'utilisateur pour calculer le facteur de trouble de Linke.
A partir de cette même quantité ainsi que de la masse d'air optique relative, l'épaisseur optique de Rayleigh, l'énergie solaire et le coefficient d'incidence, on peut ensuite calculer le rayonnement solaire direct.
On utilise aussi l'inclinaison ainsi que la hauteur solaire pour déterminer le rayonnement solaire diffus, puis la somme de ces deux résultats constitue le rayonnement solaire global reçu par le panneau.
Cela a constitué l'ensemble du travail du 6ème groupe.

Ce raisonnement est résumé via le schéma en annexe.

Ces calculs utilisent les fonctions trigonométriques de la librairie « math.h ». On doit donc encore une fois utiliser les fonctions Deg2Rad (pour les fonctions travaillant avec des paramètres en radians comme cos, sin, tan) et Rad2Deg (pour les fonctions renvoyant un résultat en radians comme arcos, arcsin, arctan).
Par la suite, les variables avec le suffixe `rad` seront des valeurs en radians (préalablement converties de degrés vers radians grâce à la fonction Deg2Rad) et les valeurs avec le suffixe `deg` seront en degrés. C'est un choix des deux groupes de travail pour bien distinguer valeurs en degrés et valeurs en radians.

Travail du 5ème groupe

Membres :

- Alix Jeannerot
- Paul Bouchillou
- Sébastien Cizeron

Responsable : Fabien Bouteyre

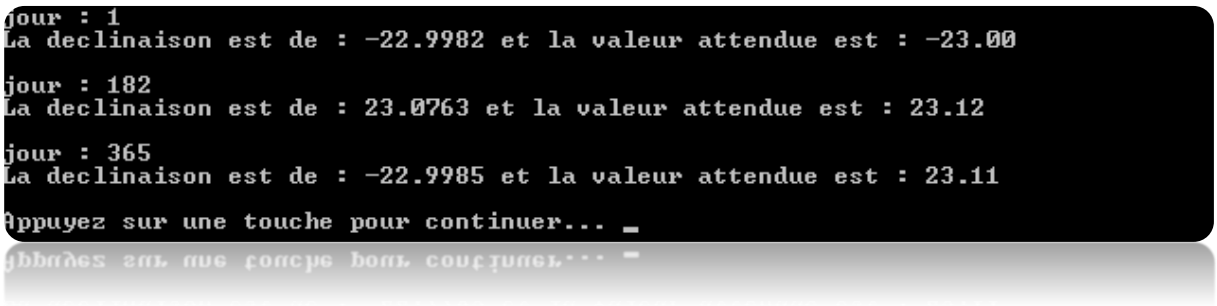
1ère fonction : calcul de la déclinaison

`Declinaison` : la fonction (jour : 1 entier) → 1 réel

R : Calcule à partir du jour de l'année la déclinaison correspondante
E : 1 entier (le jour)
S : 1 réel (la déclinaison en degrés décimaux)

Cette fonction utilise arcsinus (asin en C), donc Rad2deg est encore présente. La déclinaison se calcule de la manière suivante :

```
decrad=asin(0.398*sin(Deg2rad(0.989*jour-80)));
decdeg=Rad2deg(decrad);
return decdeg;
```



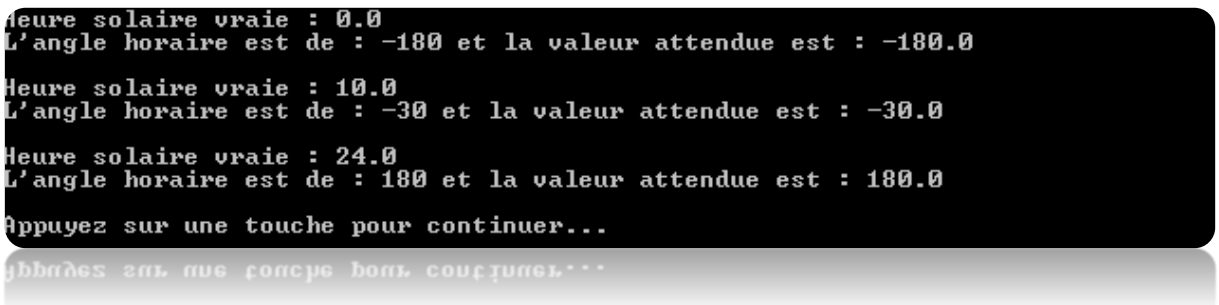
2ème fonction : calcul de l'angle horaire

`Angle_horaire` : la fonction (hsv : 1 réel) → 1 réel

R : Calcule à partir de l'heure solaire vraie l'angle horaire correspondant
E : 1 réel (l'heure solaire vraie en heures décimales)
S : 1 réel (l'angle horaire en degrés décimaux)

L'angle horaire se calcule de la manière suivante :

```
ang=180*(heuresolairev/12.0-1);
return ang;
```



3ème fonction : calcul de la hauteur solaire

`Hauteur_solaire` : la fonction (lat : 1 réel, dec : 1 réel, ah : 1 réel) → 1 réel

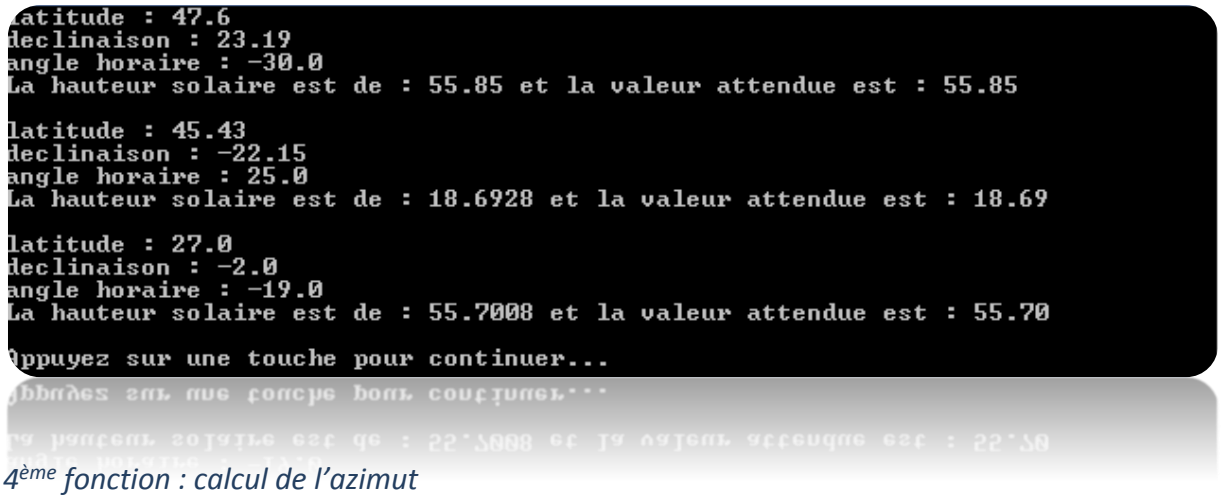
R : Calcule à partir de la latitude, de la déclinaison et de l'angle horaire la hauteur solaire
E : 3 réels (latitude, déclinaison et angle horaire en degrés décimaux)

S : 1 réel (hauteur solaire)

Cette fonction utilise la fonction arcsinus, et donc nécessairement Rad2deg pour retourner un résultat final en degrés. La hauteur solaire se calcule de la manière suivante :

```
hauteurrad=asin(sin(latituderad)*sin(declinaisonrad)+cos(latituderad)*cos(declinaisonrad)*cos
(anglerad)) ;


hauteurdeg=Rad2deg(hauteurrad) ;
return hauteurdeg;
```



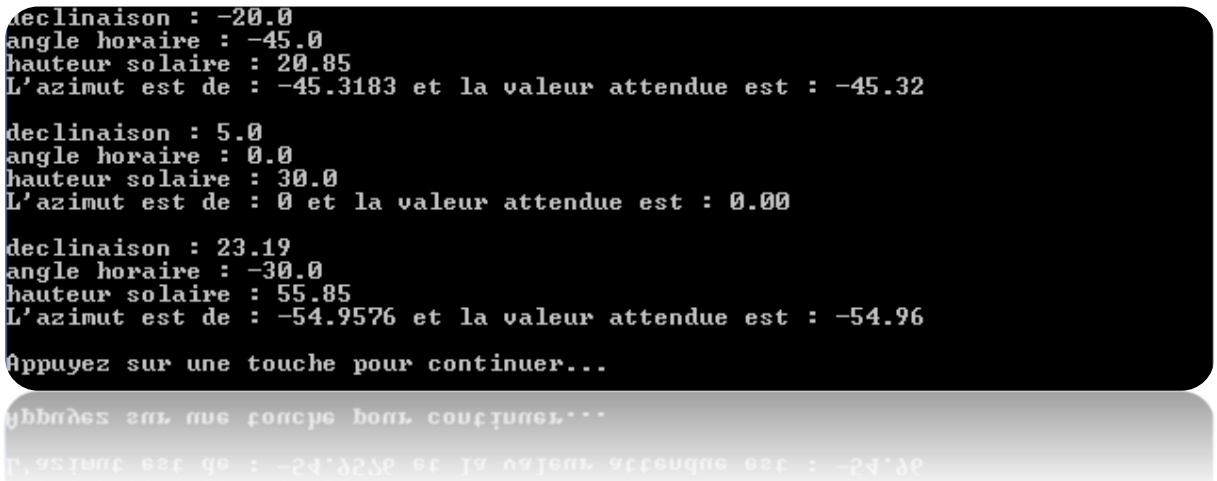
4ème fonction : calcul de l'azimut

`Azimut` : la fonction (dec : 1 réel, ah : 1 réel, hs : 1 réel) → 1 réel

R : Calcule à partir de la déclinaison, de l'angle horaire et de la hauteur solaire l'azimut correspondant
E : 3 réels (déclinaison, angle horaire, hauteur solaire en degrés décimaux)
S : 1 réel (l'azimut en degrés décimaux)

La nécessité d'un calcul d'arcsinus demande encore une fois la présence de la fonction Rad2deg avant de retourner l'azimut.

```
arad=asin((cos(decrad)*sin(ahrad))/(cos(hsrad)));
adeg=Rad2deg(arad);
return adeg;
```



5ème fonction : calcul du coefficient d'incidence lié à la position du panneau

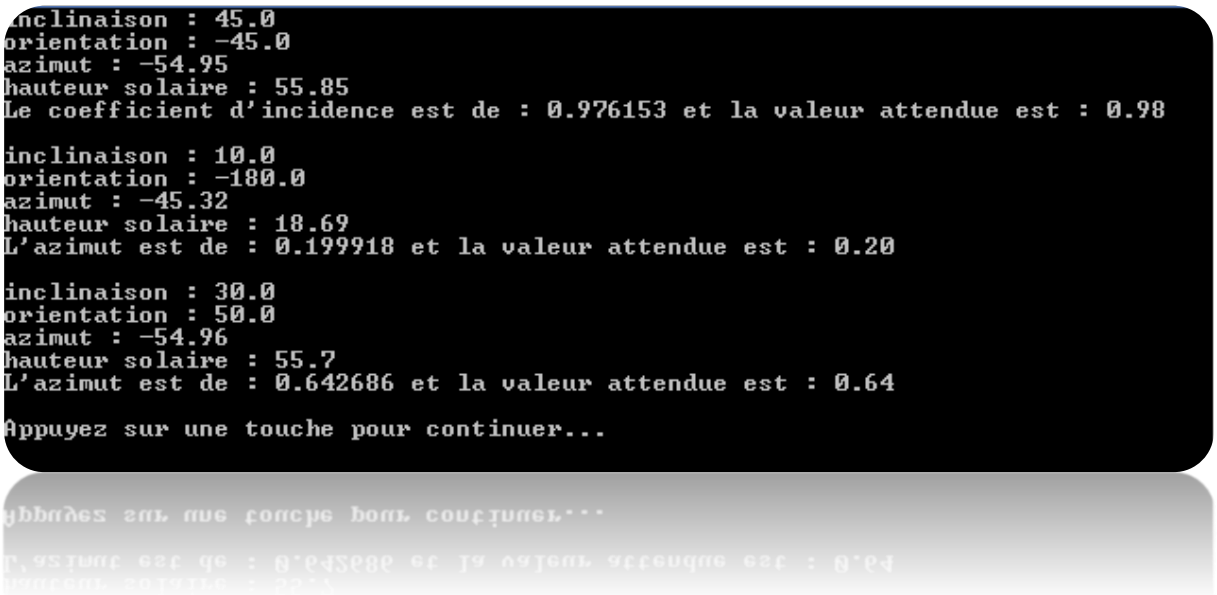
`Coef_incidence` : la fonction (i : 1 réel, o : 1 réel, a : 1 réel, hs : 1 réel) → 1 réel

R : Calcule à partir de l'inclinaison, de l'orientation, de l'azimut et de la hauteur solaire le coefficient d'incidence correspondant
E : 4 réels (inclinaison, orientation, azimut et hauteur solaire en degrés décimaux)

S : 1 réel (coefficient d'incidence)

Le coefficient d'incidence se calcule de la manière suivante :

```
coeff_incid=sin(inclinaisonrad)*cos(hautsolrad)*cos(orientationrad-
azimutrad)+cos(inclinaisonrad)*sin(hautsolrad);
return coeff_incid;
```



Les valeurs attendues ont été soit calculées par ce groupe de travail avec excel, soit prises dans le fascicule. Excel fonctionne aussi en radians, il devait donc y avoir le même raisonnement que lors de la programmation (conversions degrés / radians pour cosinus, sinus et radians / degrés pour arcsinus)

Travail du 6ème groupe

Membres :

- Grégoire Bouland
- Charles Royer
- Geoffrey Dumon
- Nour Hassoun

Responsable : Manon Portay

1ère fonction : calcul de l'énergie solaire

`Energie_solaire` : la fonction (j : 1 entier) → 1 réel

R : Calcule à partir du nombre de jours correspondant à la date saisie précédemment l'énergie solaire correspondante
E : 1 entier (nième jour de l'année)
S : 1 réel (l'énergie solaire en W/m²)

On utilise la fonction de conversion Deg2rad afin de convertir les résultats en degrés en radians.
Le calcul s'effectue de la manière suivante :

```
deg = 360 * (j - 2.7206) / 365.25;
      rad = Deg2rad(deg);
      Esol = 1367 * (1 + 0.0334*cos(rad));
```

```
181
Le nombre de jours est : 181
  et l'0nergie solaire correspondante est : 1321.47
```

2ème fonction : calcul de la masse d'air optique relative

`Masse_air` : la fonction (z : 1 réel, hs : 1 réel) → 1 réel

R : Calcule à partir de l'altitude et de la hauteur solaire la masse d'air optique relative correspondante
E : 2 réels (altitude en m et hauteur solaire en degrés décimaux)
S : 1 réel (masse d'air optique relative)

On utilise donc la fonction de conversion Deg2rad afin de convertir les résultats en degrés en radians.
Cette fonction nécessite d'utiliser la fonction exponentielle pour coder e^x, le code est : « exp(x) » disponible dans le répertoire math.h.
Pour calculer la masse d'air d'optique, on a besoin de la pression atmosphérique calculée à partir de l'altitude, et de la hauteur solaire en degrés décimaux.

```
patm = 101325 * pow((1 - 2.26*0.00001*z), 5.26);
radh = Deg2rad(hs);
m = patm / (101325 * sin(radh) + 15198.75*pow((3.885 + hs), -1.253));
return m;
```

```
Fonction : Masse_air
260
55.85
 avec l'altitude   = 260 , la hauteur solaire   = 55.85
                                                          la masse d'air o
ptique est :1.17021
```

3ème fonction : calcul de l'épaisseur optique de Rayleigh

`Epaisseur_optique` : la fonction (m : 1 réel) → 1 réel
R : Calcule à partir de la masse d'air optique relative l'épaisseur optique de Rayleigh correspondante
E : 1 réel (masse d'air optique relative)
S : 1 réel (épaisseur optique de Rayleigh)

Le calcul de l'épaisseur optique s'effectue de la manière suivante :

```
er = 1 / (0.9*m + 9.4);
      return er;
```

```
Fonction : Epaisseur_optique
1.17
 avec la masse d'air optique relative   = 1.17
                                    l'Úpaisseur de Rayleigh est :0.0956663
```

4ème fonction : calcul du coefficient de trouble de Linke

`Trouble_Linke` : la fonction (t : 1 réel, b : 1 réel, hu : 1 réel) → 1 réel
R : Calcule à partir de la température, du coefficient de trouble atmosphérique et du taux d'humidité relative le facteur de trouble de Linke correspondant

E : 2 réels (température en °C , coefficient de trouble atmosphérique et taux d'humidité entre 0 et 1)
S : 1 réel (facteur de trouble de Linke)

Le calcul du coefficient de trouble de Linke à partir de la température, du coefficient de trouble atmosphérique et du taux d'humidité relative a été divisé en 3 calculs.
Premièrement, la pression de vapeur saturante est calculée à partir de la température. Ensuite, la pression partielle de vapeur d'eau est calculée à partir de la pression de vapeur saturante et du taux d'humidité. Finalement, le trouble Linke est calculé grâce à la pression partielle de vapeur d'eau et du coefficient de trouble atmosphérique.
Cette fonction nécessite d'utiliser la fonction puissance : pour coder x^y, le code est : « pow(x,y) » disponible dans le répertoire math.h.
Cette fonction nécessite également d'utiliser la fonction logarithme népérien : pour coder ln(x), le code est « log(x) » disponible dans le répertoire math.h.

```
pvs = 2.165*pow(1.089 + (temp / 100), 8.02);
pv = pvs*hu;
tl = 2.4 + 14.6*b + 0.4*(1 + 2 * b)*log(pv);
return tl;
```

```
Fonction : trouble_linke
20
0.1
0.5
 avec la temperature =20le coefficient de trouble atmospherique =0.1 et le taux
d'humidite =0.5
                          le trouble linke est :4.87534
```

5ème fonction : calcul du rayonnement solaire direct

`Solaire_direct` : la fonction (Esol : 1 réel, eR : 1 réel, m : 1 réel, tL : 1 réel, ci : 1 réel) → 1 réel

R : Calcule à partir de l'énergie solaire, de l'épaisseur optique de Rayleigh, de la masse d'air optique relative, du facteur de trouble de Linke et du coefficient d'incidence le rayonnement solaire direct (S) sur un plan récepteur (o, i)*
E : 1 entier (nombre de jours) et 4 réels (épaisseur optique de Rayleigh, masse d'air optique relative, facteur de trouble de Linke et coefficient d'incidence)
S : 1 réel (rayonnement solaire direct S en W/m²)*

Pour le calcul du rayonnement solaire direct, on calcule d'abord le rayonnement solaire sur un plan normal à ce rayonnement à partir de l'énergie solaire, de l'épaisseur optique de Rayleigh, de la masse d'air optique et facteur de trouble de Linke.
Le rayonnement solaire direct est obtenu grâce au rayonnement solaire sur un plan normal à ce rayonnement et au coefficient d'incidence.
Cette fonction nécessite d'utiliser la fonction exponentielle, pour coder e^x, le code est : « exp(x) » disponible dans le répertoire math.h.

```
i = Esol * exp(-eR *m*tL);
       Sd = i*ci;
       return Sd;
```

```
Fonction : solaire_direct
1321.47
0.095
1.17
4.9
0.976
 avec l'energie solaire  = 1321.47 , l'epaisseur optique de Rayleigh  = 0.095 ,
la masse d'air optique relative = 1.17 , le facteur de trouble de Linke = 4.9 ,
le coefficient d'incidence = 0.976
                                        le rayonnement solaire direct est :
748.127
```

6ème fonction : calcul du rayonnement solaire global

`Solaire_global` : la fonction (sdirect : 1 réel, h : 1 réel, i : un réel) → 1 réel

R : Calcule à partir du rayonnement solaire direct S, de la hauteur solaire et de l'inclinaison du panneau le rayonnement solaire global G* correspondant*
E : 3 réels (rayonnement solaire direct en W/m², hauteur solaire et inclinaison du panneau en degrés décimaux)
S : 1 réel (rayonnement solaire global en W/m²)

On utilise encore une fois la fonction de conversion Deg2rad afin de convertir les résultats de degrés à radians pour la fonction cos(x).
Afin de calculer le rayonnement solaire global on calcule d'abord le rayonnement solaire diffus à partir de la hauteur solaire et de l'inclinaison du panneau puis le rayonnement solaire global à partir de ce rayonnement solaire diffusé et du rayonnement solaire direct.

```
	hrad = Deg2rad(h);
	irad = Deg2rad(i);   snh = sin(hrad);
	D = 125 * (pow(snh, 0.4))*((1 + (cos(irad))) / 2.0) + 211.86*(pow(snh, 1.22))*((1 -
cos(irad)) / 2.0);
	G = D + sdirect;

	return G;
```

```
Fonction : Solaire_global
748.12
55.85
45
 avec le rayonnement solaire direct   = 748.12 , la hauteur solaire   = 55.85 et l
'inclinaison du panneau = 45
                                      le rayonnement solaire global est :871.66
4
```

Fonctions de présentation du programme et d'affichage des résultats

Une fois le raisonnement achevé et le programme fonctionnel, il est nécessaire de présenter le programme à l'utilisateur et ensuite lui communiquer le résultat obtenu. Les chefs de projet se sont chargés de créer deux fonctions dans ce but.

1ère fonction : présentation du programme

`Presentation` : la fonction (vide) → vide

R : Présenter au mieux à quoi sert le programme (informations nécessaires, utilité, etc.)
E : vide
S : vide

Dans cette fonction, on communique tout simplement à l'utilisateur le rôle du programme ainsi que les informations qu'il devra entrer :

```
void Presentation(void)
{
	cout << "Ce programme vous permet de calculer le rayonnement solaire global potentiel sur
	votre panneau. Pour cela, vous devrez renseigner differentes informations :" << endl << endl;
	cout << "- La date (jour, mois, annee)" << endl;
	cout << "- L'heure (heures, minutes, secondes" << endl;
	cout << "- La latitude Nord et la longitude Est du lieu sous forme sexagesimale (degres,
	minutes, secondes)" << endl;
	cout << "- L'inclinaison et l'orientation de votre panneau en degres decimaux" << endl;
```

```
        cout << "- L'altitude du lieu en m" << endl;
        cout << "- La temperature en °C" << endl;
        cout << "- Le milieu dans lequel vous vous trouvez (montagnard, rural, urbain, industriel)"
        << endl;
        cout << "- Le taux d'humidite du lieu en %" << endl << endl;
}
```

2ème fonction : affichage du résultat

`Affiche_res` : la fonction (sglobal : 1 réel, d : 1 date, h : 1 heure, lat : 1 coordonnee, lon : 1 coordonnee, i : 1 réel, o : 1 réel, z : 1 réel, temp : 1 réel, coef : 1 réel, hu : 1 réel) → vide

R : Affiche de manière parlante le rayonnement global et récapitule les saisies passées en paramètre (unité, etc.)
E : 1 réel (le rayonnement global)
S : vide

On récapitule les saisies effectuées par l'utilisateur, puis on affiche le résultat (rayonnement solaire globlal) associé à ses saisies.

```
void Affiche_res(float res, date d, heure h, coordonnee lat, coordonnee lon, float i, float o, float
z, float temp, float coef, float hu)
{
        cout<<endl<<"Vous avez saisi les valeurs suivantes :"<<endl;
        cout<<"Date : "<<d.j<<"/"<<d.m<<"/"<<d.a<<endl;
        cout<<"Heure : "<<h.h<<":"<<h.m<<":"<<h.s<<endl;
        cout<<"Latitude du lieu : "<<lat.d<<" degres "<<lat.m<<" minutes "<<lat.s<<" secondes
        EST"<<endl;
        cout<<"Longitude du lieu : "<<lon.d<<" degres "<<lon.m<<" minutes "<<lon.s<<" secondes
        NORD"<<endl;
        cout<<"Inclinaison du panneau : "<<i<<" degres"<<endl;
        cout<<"Orientation du panneau : "<<o<<" degres"<<endl;
        cout<<"Altitude du lieu : "<<z<<" m"<<endl;
        cout<<"Température : "<<temp<<" degres Celsius"<<endl;
        cout<<"Coefficient de trouble atmospherique associe au milieu : "<<coef<<endl;
        cout<<"Taux d'humidite : "<<hu<<endl<<endl;
        cout << "Le rayonnement solaire global sur votre panneau obtenu (en cas de beau temps) est de
" << res << "W/m^2, (watt par metre^2)." << endl;
}
```

Etude du programme principal

Après avoir établi l'ensemble des fonctions, il faut évidemment créer un programme principal qui les exécute selon leur utilité dans le calcul du rayonnement solaire global.
Il était d'abord nécessaire de déclarer dans le lexique les différentes variables liées à la saisie (le suffixe comp correspond à une variable sous forme de type composé) :

```
        date d_comp;
        heure h_comp;
        coordonnee lat_comp,lon_comp;
        float inclinaison, orientation;
        float altitude;
        float temperature;
        float coef_trouble;
        float humidite;
```

Ensuite, les variables liées aux conversions :

```
        float h_dec;
        int nombre_j;
        situation s_geo;
        float hs_vraie;
```

Le booléen indiquant directement si l'on a un rayonnement nul :

```
bool rayon_nul;
```

Enfin, les différentes quantités calculées jusqu'au rayonnement solaire global :

```
float decl;
float a_horaire;
float h_solaire;
float az;
float c_incidence;
float ener_sol;
float m_air;
float e_optique;
float t_linke;
float s_direct;
float s_global;
```

Arrive l'algorithme principal. On commence par effectuer les différentes saisies après avoir présenté le programme :

```
Presentation();
d_comp=Saisie_date();
h_comp=Saisie_heure();
lat_comp=Saisie_latitude();
lon_comp=Saisie_longitude();
inclinaison=Saisie_inclinaison();
orientation=Saisie_orientation();
altitude=saisie_alt();
temperature=Saisie_temperature();
coef_trouble=saisie_milieu();
humidite=Saisie_hum();
```

On effectue ensuite les différentes conversions nécessaires :

```
h_dec=Heures_decimales(h_comp);
nombre_j=Nb_jours(d_comp);
s_geo=Situation_geo(lat_comp, lon_comp);
hs_vraie=Heure_solaire_vraie(s_geo.lo, h_dec, d_comp);
```

Ensuite, on effectue d'abord les calculs utiles pour reconnaître directement si le rayonnement correspondant à la saisie sera nul (on connaît par les conversions le paramètre l'heure solaire vraie, il reste à calculer la déclinaison pour pouvoir faire fonctionner `lever_coucher`) :

```
decl=Declinaison(nombre_j);
rayon_nul=lever_coucher(s_geo.la, decl, hs_vraie);
```

Si le booléen retourné a pour valeur `false`, cela signifie que l'heure saisie est durant la nuit donc on indique directement à l'utilisateur que le rayonnement sera nul :

```
if (rayon_nul==false)
	{
		cout<<"L'heure que vous avez saisie correspond a une heure d'ensoleillement
		nul. Vous n'aurez aucun rayonnement sur votre panneau."<<endl;
	}
```

Sinon, on poursuit les calculs jusqu'au rayonnement solaire global puis on affiche le résultat :

```
else
	{
		a_horaire=Angle_horaire(hs_vraie);
		h_solaire=Hauteur_solaire(s_geo.la, decl, a_horaire);
		az = Azimut(decl, a_horaire, h_solaire);
		c_incidence=Coef_incidence(inclinaison, orientation, az, h_solaire);
		ener_sol=Energie_solaire(nombre_j);
```

```
            m_air=Masse_air(altitude, h_solaire);
            e_optique=Epaisseur_optique(m_air);
            t_linke=trouble_linke(temperature, coef_trouble, humidite);
            s_direct=solaire_direct(ener_sol, e_optique, m_air, t_linke, c_incidence);
            s_global=Solaire_global(s_direct, h_solaire, inclinaison);
            Affiche_res(s_global, d_comp, h_comp, lat_comp, lon_comp, inclinaison,
            orientation, altitude, temperature, coef_trouble, humidite);
        }
```

On peut ensuite tester le programme avec les valeurs du fascicule (ville de Mulhouse). Pour cela, il faut convertir les 10h heure solaire vraie utilisées en heure légale (car dans le fascicule les calculs ont été faits directement à partir d'une heure solaire vraie). On obtient grâce à cet algorithme disponible sur le site :
http://michel.lalos.free.fr/cadrans_solaires/outils/Tables-hs-hl.php

Calculs pour : **Mulhouse**	
Latitude : 47.60000000 = **+47° 36' 00"**	Longitude : -7.34000000 = **- 7° 20' 23"**
Date :	01/07/2013
Correction longitude : *C1* =	- 00 h 29 min 21 s
Equation du temps : *C2* =	00 h 03 min 48 s
Période "heure d'été ou d'hiver" : *C3* =	02 h 00 min 00 s
Correction totale : *C* = *C1* + *C2* + *C3* =	01 h 34 min 27 s

Heure solaire	Heure légale
10 h 00 min 00 s	11 h 34 min 27 s

Le calcul du nombre de jours associé au 1er juillet a été fait dans le cas d'une année non bissextile (il est compté 28 jours à février) donc 2013 a été saisi pour respecter cette condition.
Nb : il y a une erreur dans le fascicule au niveau du nombre de jours correspondant au 1er juillet (il est de 181 dans celui-ci au lieu de 182 en réalité, car 30 jours ont été comptés à janvier)

Le résultat avec l'ensemble des valeurs donne le résultat suivant :

```
Vous avez saisi les valeurs suivantes :
Date : 1/7/2013
Heure : 11:34:27
Latitude du lieu : 47 degres 36 minutes 0 secondes EST
Longitude du lieu : 7 degres 20 minutes 23 secondes NORD
Inclinaison du panneau : 45 degres
Orientation du panneau : -45 degres
Altitude du lieu : 260 m
Tempúrature : 20 degres Celsius
Coefficient de trouble atmospherique associe au milieu : 0.1
Taux d'humidite : 0.5

Le rayonnement solaire global sur votre panneau obtenu (en cas de beau temps) es
t de 871.034W/m^2, (watt par metre^2).
```

En faisant en sorte d'avoir 181 jours (comme dans le fascicule), le résultat obtenu est 871.288 W/m² pour 871.66 dans le fascicule.
On obtient donc un résultat d'une précision convenable.

Exportation des résultats sous forme de tableaux de données

(proposé par Théo Basty)

Nous souhaitions au final que le programme soit capable de générer un fichier contenant diverses données et pouvant être ouvert à l'aide d'un tableur.
Pour cela, il a fallu revoir un peu le découpage fonctionnel du programme :

- Une fonction regroupant tous les appels à des fonctions de calculs situées précédemment dans la fonction main a été créée pour pouvoir facilement insérer une boucle pour calculer l'ensoleillement pour plusieurs valeurs consécutive de manière rapide.

```
double Ensoleillement(date d_comp, heure h_comp, coordonnee lat_comp, coordonnee
   lon_comp, float inclinaison, float orientation, float altitude, float temperature,
   float coef_trouble, float humidite, int fuseau)
```

- Une autre fonction a été créée, contenant un menu pour demander à l'utilisateur s'il désire générer un rapport.

```
bool menu_rapport()
```

- Une dernière fonction a enfin été créée pour générer ce fichier de rapport.

```
void rapport_build(date d_comp, heure h_comp, coordonnee lat_comp, coordonnee
    lon_comp, float inclinaison, float orientation, float altitude, float temperature,
    float coef_trouble, float humidite, int fuseau);
```

Conception

La fonction Ensoleillement reprend donc les appels consécutifs de toutes les fonctions de calcul et de traitement. Les paramètres sont toutes les valeurs saisies par l'utilisateur pour décrire les conditions d'installation de son panneau solaire.
La petite particularité est le rayon nul. Durant la nuit, l'ensoleillement d'un panneau solaire est très faible. Dans ce cas, on ne calcule pas l'ensoleillement en retournant simplement un 0.

```
bool rayon_nul; //Booléen renvoyant TRUE si rayonnement non nul et FALSE sinon

if (rayon_nul == false)
{
       s_global = 0;
}
```

On se sert d'ailleurs de ce 0 dans le main afin d'afficher un message spécifique

```
ens_global = Ensoleillement(d_comp, h_comp, lat_comp, lon_comp, inclinaison, orientation, altitude,
    temperature, coef_trouble, humidite, fuseau);

if (ens_global != 0)
{
    Affiche_res(ens_global, d_comp, h_comp, lat_comp, lon_comp, inclinaison, orientation, altitude,
    temperature, coef_trouble, humidite);
}
```

La fonction menu_rapport est une simple fonction de saisie en menu. Deux choix s'offrent à l'utilisateur et elle renvoie un booléen signifiant oui ou non en fonction du choix de l'utilisateur pour répondre à la question.

```
       do{
              cout << endl << "Voulez vous generer un tableau d'ensoleillement au format csv ?" <<
              endl
                     << "1 : oui" << endl
                     << "0 : non" << endl
                     << "Votre choix (0 ou 1):";
```

```
		cin >> entree;
	} while (entree != 0 && entree != 1); //Attente d'une saisie juste

	//correspondance de la sortie avec le choix
	if (entree == 0){
		choix = false;
	}else{
		choix = true;
	}
```

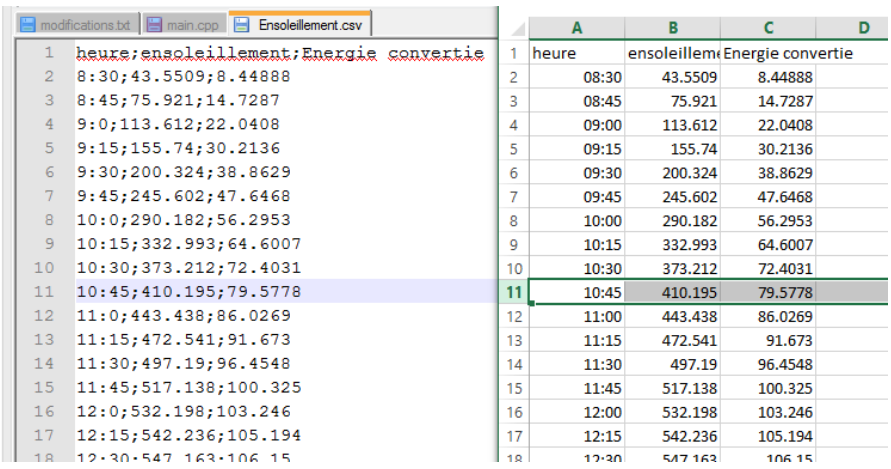
La fonction build_rapport est plus compliquée. En effet, elle écrit des données dans un fichier. On utilise donc une bibliothèque spéciale : `#include<fstream>`
On commence par ouvrir le fichier, puis on vérifie si il a bien été ouvert (si par exemple le fichier est déjà ouvert, le programme ne peut pas l'ouvrir en écriture). Sinon on affiche un message d'erreur et on propose à l'utilisateur de réessayer ou d'abandonner car parfois la raison est plus complexe.

```
	do
	{
		ofstream myFile("Ensoleillement.csv");
		if (myFile)
		{
			//Génération du fichier
		}
		else
		{

			cout << "ERREUR: Impossible d'ouvrir le fichier." << endl
				<< "Verifiez qu'il ne soit pas ouvert par un autre programme" << endl << endl;
			do
			{
				cout << "Voulez vous reessayer de l'ouvrir ? [o/n] ";
				cin >> retry_in;
				switch (retry_in)
				{
				case 'o':
					retry = true;
					break;

				case 'n':
					retry = false;
					cout << "Abandon de l'ecriture du rapport..." << endl;
				}
			} while ((retry_in != 'o') && (retry_in != 'n'));
		}
	} while (retry);
```

Ensuite, il existe un format de fichier nommé CSV (comma separated value) qui permet de représenter des tables de données et qui peut être ouvert par Excel. Il s'agît en fait d'un fichier texte basique contenant des valeurs séparées par un caractère définit dans les paramètres de régions de l'ordinateur (pour séparer les colonnes) sur plusieurs lignes (pour séparer les lignes du tableau). Ils sont donc faciles à générer.



De plus, l'écriture dans un fichier fonctionne de la même manière que la fonction cout. La génération d'un fichier CSV est donc très facile. Il suffit juste d'appliquer une boucle sur la fonction de calcul Ensoleillement() puis d'enregistrer les valeurs dans le fichier ouvert. On crée une constante SEPARATOR de type caractère afin de stocker le séparateur de colonnes.
Nous avons créé plusieurs boucles afin de pouvoir tracer plusieurs graphiques intéressants en fonction de différents paramètres à l'aide d'un tableur.
Pour les tableaux en fonctions de l'humidité, de la température, de l'orientation et de l'inclinaison, les boucles sont quasiment identiques à part la plage parcourue par la boucle. On utilise une boucle for car elle permet facilement de déclarer une variable indice, de tester une condition et d'incrémenter l'indice. On appel juste Ensoleillement avec la valeur de l'indice pour obtenir l'ensoleillement global avant de l'écrire dans le fichier après l'indice.

```
                    //Tableau orientation
                    myFile << "orientation" << SEPARATOR << "ensoleillement" << SEPARATOR <<
"Energie convertie"; //Écriture des entêtes de colonnes
                    for (int ori = -170; ori <= 180; ori = ori + 10)
                    {
                           ens_loop = Ensoleillement(d_comp, h_comp, lat_comp, lon_comp,
inclinaison, ori, altitude, temperature, coef_trouble, humidite, fuseau);
                           if (ens_loop != 0)
                           {
                                  myFile << endl << ori << SEPARATOR << ens_loop << SEPARATOR <<
ens_loop * 0.2 * 0.97;
                           }
                    }
```

Pour le tableau en fonction du mois, le principe est le même que précédemment. On intègre juste l'indice de la boucle dans un type complexe date avant de l'envoyer à Ensoleillement.

```
                    d_loop.a = d_comp.a;
                    d_loop.j = d_comp.j;
                    for (int m = 1; m <= 12; m++)
                    {
                           d_loop.m = m;
```

Pour le jour, on reste dans le même fonctionnement que le mois. Mais le nombre de jours étant différents dans chaque mois, il est nécessaire de déterminer la borne maximale avant la boucle.

```
                    if (d_comp.m == 1 || d_comp.m == 3 || d_comp.m == 5 || d_comp.m == 7 ||
d_comp.m == 8 || d_comp.m == 10 || d_comp.m == 12){
                           d_max = 31;
                    }else{
                           if (d_comp.m == 4 || d_comp.m == 6 || d_comp.m == 9 || d_comp.m == 11){
                                  d_max = 30;
                           }else{
                                  if (est_bissextile(d_comp.a)){
                                         d_max = 29;
                                  }else{
                                         d_max = 28;
                                  }
                           }
                    }
                    d_loop.a = d_comp.a;
                    d_loop.m = d_comp.m;
                    for (int j = 1; j <= d_max; j++)
                    {
                           d_loop.j = j;
```

Pour l'heure, afin d'obtenir un nombre suffisant de valeurs, il est nécessaire de calculer un ensoleillement tous les quarts d'heures. On utilise deux boucles for emboîtées pour générer 'indice de l'heure et l'indice du quart d'heure. On multiplie l'indice du quart d'heure par 15 pour obtenir le nombre de minutes à affecter au type composé passé en paramètre à la fonction Ensoleillement.

```
                    h_loop.s = 0;
                    for (int h = 0; h < 24; h++){
                           h_loop.h = h;
                           for (int m = 0; m < 4; m++){
                                  h_loop.m = m * 15;
                                  ens_loop = Ensoleillement(d_comp, h_loop, lat_comp, lon_comp,
inclinaison, orientation, altitude, temperature, coef_trouble, humidite, fuseau);
                                  if (ens_loop != 0){
                                         myFile << endl << h_loop.h << ":" << h_loop.m << SEPARATOR
<< ens_loop << SEPARATOR << ens_loop * 0.2 * 0.97;
                                  }
                           }
                    }
```

On sépare enfin les différents tableaux par une ligne vide :

```
myFile << endl << endl;
```

Conclusion du projet

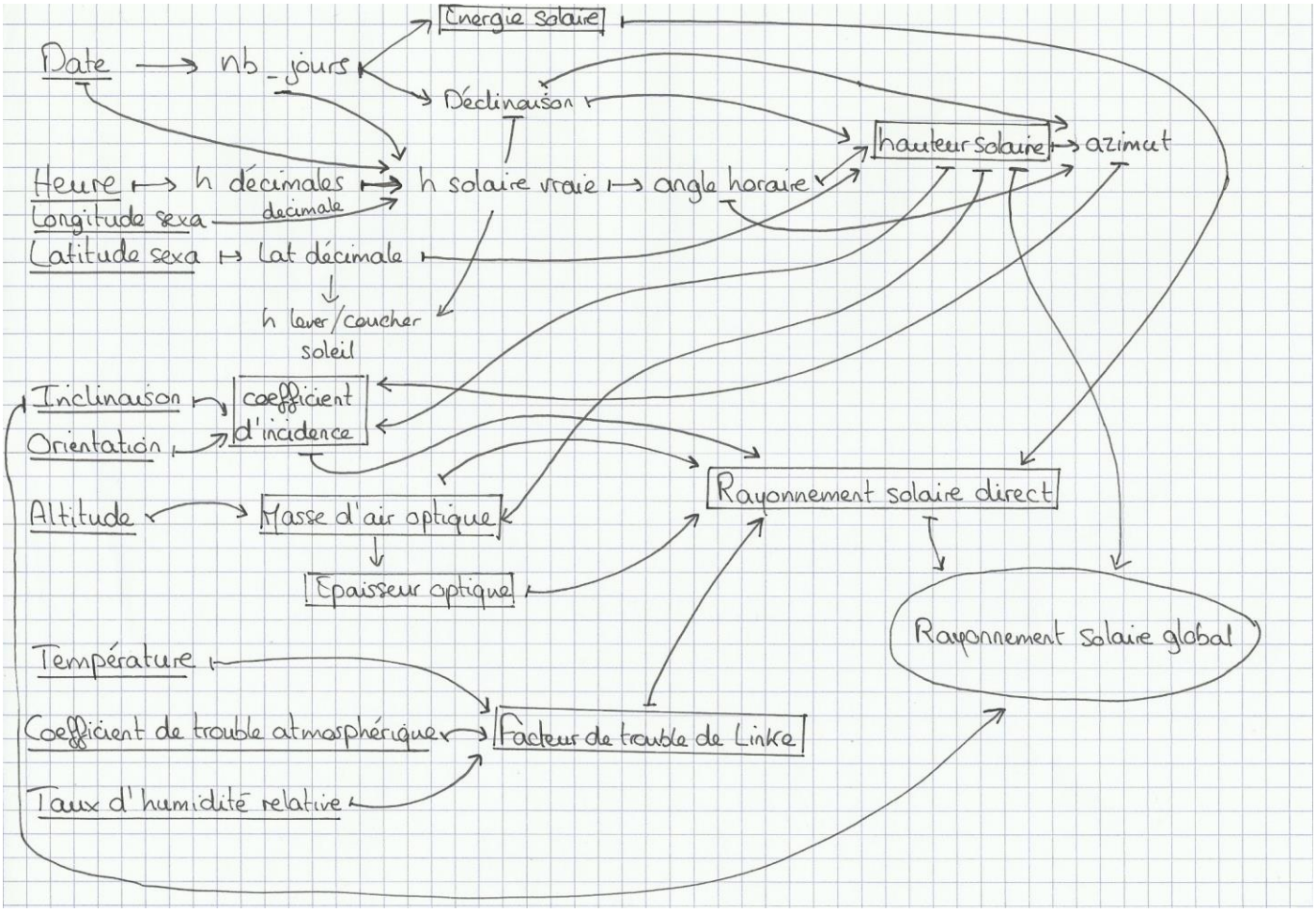
Tout d'abord, le bilan de réalisation du projet est plutôt positif : en effet, nous sommes arrivés au but principal, celui d'exprimer le rayonnement solaire global mais nous avons également pu approfondir la problématique.
Nous avons ensuite pu utiliser le programme dans le but de générer un fichier Excel (.csv) afin d'écrire des données dans un tableur. Cela a permis d'exprimer le rayonnement solaire global pour un lieu donné en fonction de l'un de ses paramètres, notamment le jour de l'année considéré, l'heure dans la journée ou encore l'orientation du panneau photovoltaïque. Le programme possède donc une réelle utilité par rapport à la problématique de base : prévoir un fonctionnement optimal pour un panneau solaire.

De plus, ce projet nous a permis de maîtriser la plupart des concepts algorithmiques de base :

- Il nécessitait d'abord l'utilisation de structures conditionnelles et itératives. Les structures conditionnelles étaient importantes pour distinguer les différents cas possibles sur une saisie par exemple, avec les années bissextiles et non-bissextiles. Les structures itératives ont été indispensables au niveau de la saisie. Elles ont permis de mettre en place un système de saisie de manière à éviter que des erreurs de saisie soient commises par l'utilisateur.
- Les types composés ont aussi joué un rôle important. Ils ont permis de faciliter les fonctions de saisie pour les dates par exemple, composées de plusieurs informations, dont on ne peut retourner qu'une information si l'on n'utilise pas de type composé. Cela possède aussi un autre avantage : par exemple, la latitude et la longitude sont souvent exprimées de manière sexagésimale, ainsi un type composé des degrés, minutes et secondes a permis à l'utilisateur d'entrer ses informations de manière plus parlante, et dans ce cas précis qu'il n'ait pas à chercher la correspondance décimale de ses coordonnées
- L'utilisation de booléens était judicieuse pour la résolution du problème. En effet, ils permettent de distinguer deux cas lorsqu'il n'y a que deux issues possibles, notamment « saisie correcte » ou « saisie incorrecte ».
- Le raisonnement du découpage fonctionnel a été essentiel au bon déroulement du projet. Il s'agissait d'étudier de manière précise le besoin et le raisonnement de résolution du problème pour établir une liste de fonctions permettant sa résolution en C++.

Enfin, nous avons pu approfondir les possibilités offertes par le langage C en générant un fichier Excel avec les données du programme, ainsi qu'en utilisant une librairie complémentaire « math.h » contenant les principales fonctions usuelles de mathématiques pour les calculs.

Annexe


Energie solaire
Date
nb_jours
Déclinaison
hauteur solaire
azimut
Heure
h décimales
h solaire vraie
angle horaire
Longitude sexa
décimale
Latitude sexa
Lat décimale
h lever/coucher soleil
Inclinaison
Orientation
coefficient d'incidence
Rayonnement solaire direct
Altitude
Masse d'air optique
Epaisseur optique
Rayonnement solaire global
Température
Coefficient de trouble atmosphérique
Facteur de trouble de Linke
Taux d'humidité relative