Projet de Visualisation scientifique

Visualisation de données météorologiques et environnementales dispersées

Pierre-Antoine Augereau & Kévin Polisano

9 mars 2013

1 Introduction

Objectif. Le projet a pour but de visualiser, par des méthodes présentées dans le cours, les données mises à disposition publiquement par Météo-France et par l'observatoire de l'air de la région Rhône-Alpes, au sein du logiciel Google Earth.

Description. Météo-France donne l'accès à des mesures de température, de pression, de force et de direction du vent. L'observatoire de l'air de la région Rhône-Alpes met à disposition publiquement des données relatives à la qualité de l'air, issues d'une centaine de stations de mesures. Les principaux polluants, comme les particules PM10 (particules d'un diamètre inférieure à 10 micromètres) le benzène ou encore l'ozone, sont notamment mesurés toutes les heures.

Méthodologie. Le travail à réaliser comporte quatre étapes principales :

1. Importation et lecture des données localisées

2. Interpolation des données pour une évaluation en tout point, par la méthode de Shepard et la méthode des multiquadriques de Hardy

3. Implémentation de trois algorithmes, le premier calculant une image par carte de couleur, le second calculant des courbes iso-valeurs et le troisième calculant des lignes de courant.

4. Exportation des sorties des algorithmes dans des fichiers au format KML (Keyhole Markup Language), pour leur visualisation dans le logiciel Google Earth.

Langage utilisé. L'ensemble de ce projet a été réalisé avec `Processing` qui est une surcouche du langage Java qui se prête bien à la visualisation, simulation, animation, etc. L'installation, très simple, est décrite à cette adresse http://processing.org/learning/gettingstarted/.

2 Implémentation

2.1 Vision haut niveau de la classe Display

Nous travaillons sur l'espace géographique correspondant à la carte de France ou de la région Rhône Alpes. Cet espace va être confiné dans un certain périmètre rectangulaire défini par ses deux sommets opposés (ox, oy) et (px, py). A certains emplacements définis par la longitude/-latitude se trouvent des stations météorologiques effectuant certaines mesures. Celles-ci sont représentées en bleues sur la Figure 1, ce sont les données Data. En ces points nous disposons donc de relevés de température, pression, etc que nous souhaitons interpoler sur toute la région étudiée. Pour cela nous discrétisons l'espace géographique suivant une grille de $g_x \times g_y$ noeuds que nous interpolons suivant les méthodes de Shépard ou Hardy. Enfin pour visualiser le résultat obtenu nous avons le choix entre appliquer une color map, afficher les lignes de courant ou encore les courbes isovaleurs en effectuant un marching square sur les pixels de la grille.

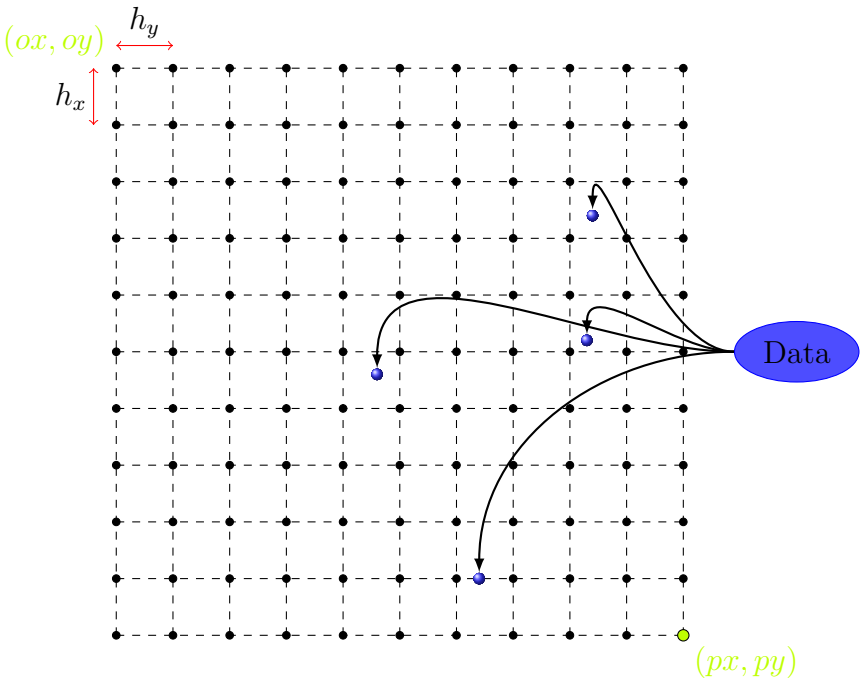

Figure 1 – Définition de la grille de Display

Tout ceci est regroupé dans les attributs et méthodes de la classe principale `Display`, voir le diagramme de classe UML de la Figure 2 (seuls les attributs et méthodes principaux et pertinents pour la compréhension sont mentionnés). Un objet Display permet donc l'affichage sur une grille de données interpolées, d'une grandeur physique décrite par l'indice v_i (température, pression, pollution, etc). Il contient également les dimensions de la grille et ses constituants : des pixels, eux-mêmes composés d'arêtes et de noeuds. On a donc une relation de dépendance entre la classe `Display` et les classes `Pixel`, `Edge` et `Node`, que nous allons décrire plus en détails ci-après. La classe `Display` est donc la classe mère permettant de représenter un jeu de données à une date fixée, son interface d'utilisation est somme toute très simple, un constructeur initialisant les structures de données des classes filles, une méthode `loadData` chargeant les données Data (fournies par la classe `MTO`), les 3 méthodes d'interpolations, les 3 méthodes de calculs et d'affichage des lignes de courant `dispTraj`, de la color map `dispData` et des courbes isovaleurs `dispIso`.

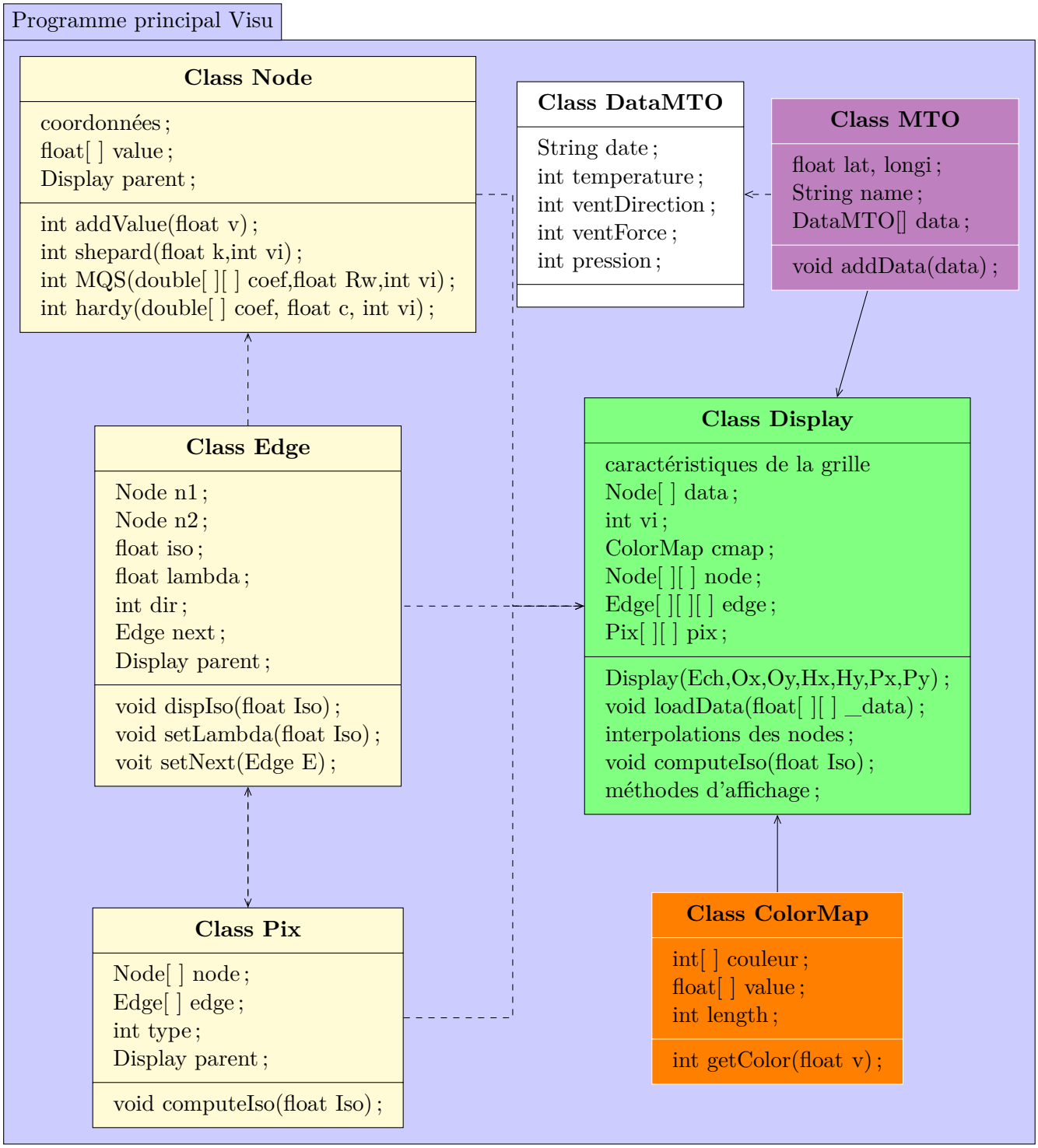

Figure 2 – Diagramme UML représentant l'architecture du programme

2.2 Détails d'implémentation de la classe Display

Class Node

Un objet noeud est un point de notre grille, ses attributs sont donc ses coordonnées réelles et celles dans le maillage, un tableau de valeurs contenant les différentes mesures en ce point (température,pression,etc selon l'indice v_i du tableau), une méthode d'ajout de valeur dans celui-ci, et les 2 méthodes d'interpolations `shepard` et `hardy` permettant de calculer la valeur effective en ce noeud par interpolation avec les données Data de la classe `Display` (d'où le besoin d'un attribut parent Display). Nous soulignons que nous avons implémenté une 3ème méthode : la version améliorée de shepard dite `MQS`. Cette dernière, tout comme Hardy, requiert un tableau de coefficients que nous remplissons au préalable dans la classe parent pour gagner en efficacité.

Class Edge

Une arête est formée de 2 noeuds n_1 et n_2 ayant pour valeurs respectives v_1 et v_2. Nous avons vu que le marching square s'appuie sur les arêtes pour tracer les courbes isovaleurs, la valeur Iso passée en attribut permet d'en déterminer un autre, la valeur barycentrique $\lambda = (Iso - v_1)/(v_2 - v_1)$ précisant la position du point d'application de l'isoligne sur l'arête, voir schéma (a) de la Figure 3 ci-dessous. Enfin un objet Edge pointe sur l'arête « suivante » au sens du marching square, association symbolisée par le vecteur $\overrightarrow{p_2p_1}$ sur le schéma (c), où les indices p_1 et p_2 (avec $p_1 < p_2$) correspondent au numéro de l'arête dans un pixel (b). Ce chainage permet de parcourir les différentes courbes une par une, afin d'améliorer leur affichage dans la sortie KML. Voyons plus en détail comment est réalisé le marching square sur les pixels.

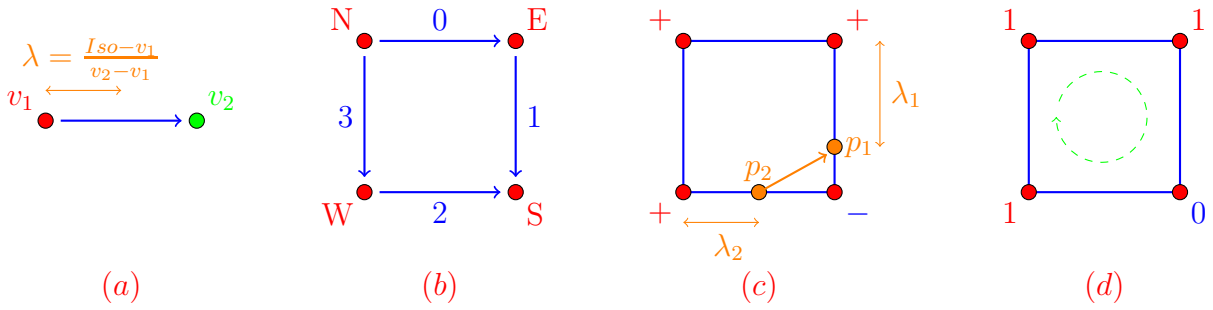

Figure 3 – (a) Class Edge, calcul de λ (b) Class Pixel, représentation d'un pixel (c) représentation de l'isoligne $\overrightarrow{p_2p_1}$ avec $p_1 = 1$ et $p_2 = 2$ (d) codage sur 4 bits du type de pixel ($1101 = 13$)

Class Pix

Comme illustré sur la Figure 3 un pixel est composé d'un tableau de 4 noeuds et d'un tableau de 4 arêtes indicés de 0 à 3. Les noeuds sont également étiquettés par 0 ou 1 (plus communément − ou +) selon que leur valeur est inférieure ou supérieure à la valeur Iso. Il existe donc 16 cas possibles, que l'on code sur 4 bits dans le sens horaire, par exemple sur le schéma (d) nous avons représenté le cas $1101 = 13$. La seule méthode agissant sur un objet Pix est celle du marching square, à savoir `computeIso` qui à partir des valeurs aux noeuds et de la valeur Iso détermine dans lequel des 16 cas l'on se trouve. Ces différents cas sont représentés sur la Figure 4. En réalité on observe une symétrie des cas, ce qui permet de traiter les cas avec seulement 3 types et une variable précisant le sens de l'association $\overrightarrow{p_1p_2}$ (de manière à ce que les valeurs + soient toujours situées à gauche du vecteur). Cette représentation permet d'implémenter le marching square de manière très concise.

```
void computeIso(float Iso)//Marching Square
 {
   float N=node[0].value[parent.v],
         E=node[1].value[parent.v],
         S=node[2].value[parent.v],
         W=node[3].value[parent.v];
 int n,e,s,w;
   n=(N<Iso)?0:1;
   e=(E<Iso)?0:1;
   s=(S<Iso)?0:1;
   w=(W<Iso)?0:1;

   int cas=(n<<3) | (e<<2) | (s<<1) | w;
   int p1=-1,p2=-1,sens=0;
   switch(cas)
   {
     case 0:case 15:type=0;break;
     case 1:sens=1;case 14:type=1;p1=2;p2=3;break;
     case 2:sens=1;case 13:type=1;p1=1;p2=2;break;
     case 3:sens=1;case 12:type=2;p1=1;p2=3;break;
     case 4:sens=1;case 11:type=1;p1=0;p2=1;break;
     case 5:sens=1;case 10:type=3;break;
     case 6:sens=1;case 9:type=2;p1=0;p2=2;break;
     case 7:sens=1;case 8:type=1;p1=0;p2=3;break;
   }

   if(type==3)
   {
   int p3,p4;
   int middle=((N+E+S+W)/4<Iso)?0:1;
   if((cas==5 && middle==1) || (cas==10 && middle==0))
     { p1=0;p2=3;p3=2;p4=1;}
   else
     { p1=0;p2=1;p3=2;p4=3;}

     edge[p3].setLambda(Iso);
     edge[p4].setLambda(Iso);
     if(sens==1)
       edge[p3].setNext(edge[p4]);
     else
       edge[p4].setNext(edge[p3]);
   }
   if(type!=0)
     {
         edge[p1].setLambda(Iso);
         edge[p2].setLambda(Iso);
         if(sens==1)
           edge[p1].setNext(edge[p2]);
         else
           edge[p2].setNext(edge[p1]);
    }
 }
```

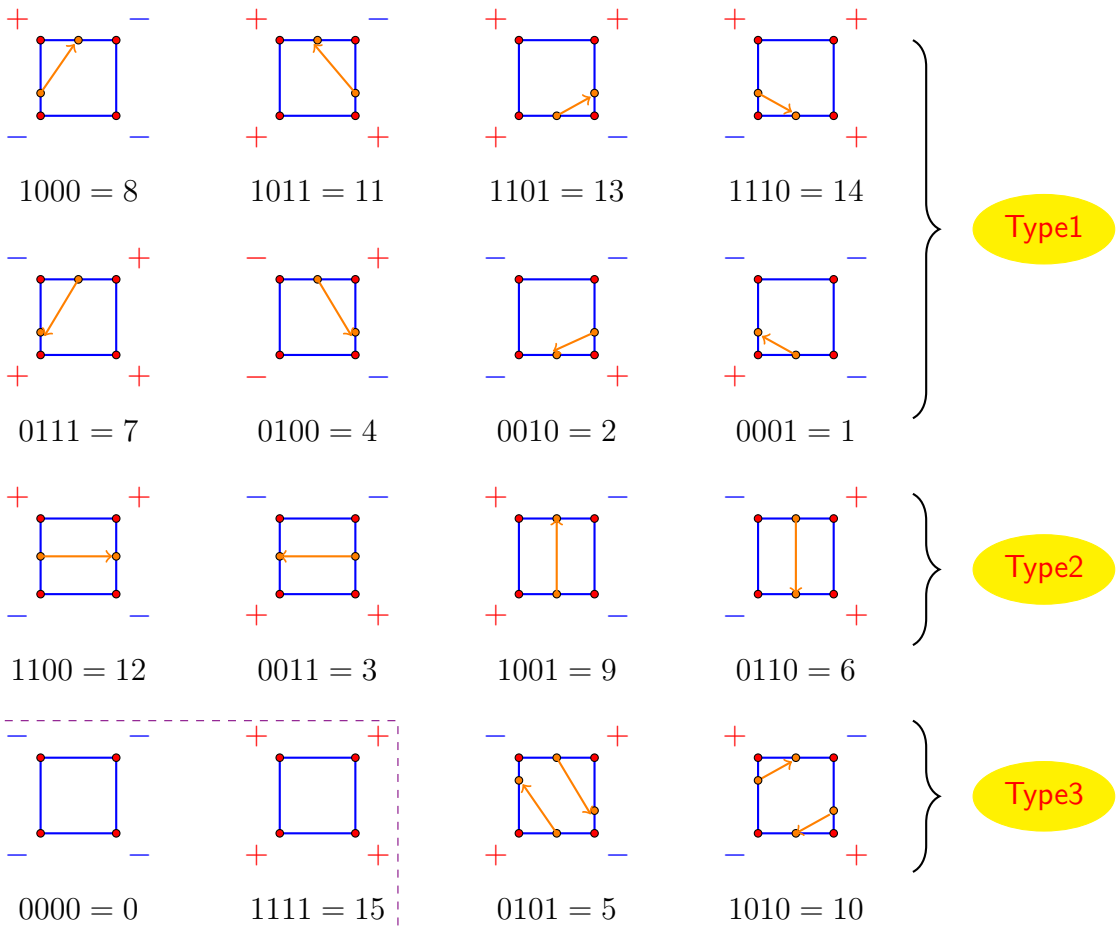

Figure 4 – Distinction des 16 cas possibles du marching square

Class Display

Nous avons déjà expliquer le fonctionnement global de cette classe principale. En réalité la grille est formée d'un tableau de noeuds à 2 dimensions et d'un tableau d'arêtes à 2 dimensions suivant les 2 directions (donc 3 dimensions au total, voir la Figure 5).

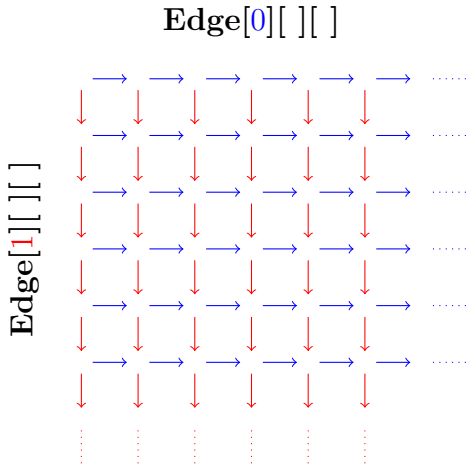

Figure 5 – Stockage des Edge

```
int getColor(float v)
{
  int c=0;
  if(v<value[0])
  { c=couleur[0];}
  else if(v>value[length-1])
  { c=couleur[length-1];}
  else
  {
    for(int i = 1;i<length;i++)
    {
      if(v<=value[i])
      {
        float alpha=(v-value[i-1])/(value[i]-value[i-1]);
        for(int j=0;j<25;j+=8)
         {
            c|= (int((couleur[i-1]>>j & 0xFF)+((couleur[i]>>j & 0xFF
               )-(couleur[i-1]>>j & 0xFF))*alpha))<<j;
         }
         break;
      }
    }
  }
  return c;
}
```

```
void dispTraj(float x,float y,float h,int vx,int vy, int iter)
{
  int i=floor(x)/hx,j=floor(y)/hy;
   while(iter>0 && i>=0 && j>=0 && i<gx && j<gy)
   {
   print("("+i+","+j+")\n");
   Pix cas=pix[i][j];
   float dx=lin( lin(cas.node[0].value[vx],cas.node[1].value[vx],  (
      float)(floor(x)%hx)/hx )    ,lin(cas.node[3].value[vx],cas.
      node[2].value[vx], (float)(floor(x)%hx)/ hx),     (float)(
      floor(y)%hy)/ hy);
   float dy=lin( lin(cas.node[0].value[vy],cas.node[1].value[vy],  (
      float)(floor(x)%hx)/hx )    ,lin(cas.node[3].value[vy],cas.
      node[2].value[vy], (float)(floor(x)%hx)/ hx),     (float)(
      floor(y)%hy)/ hy);
   float ndxy=sqrt(pow(dx,2)+pow(dy,2));
   dx*=h/ndxy;
   dy*=h/ndxy;
   line(x,y,x+dx,y+dy);
   x+=dx;
   y+=dy;
   iter--;
   i=floor(x)/hx;
   j=floor(y)/hy;
  }
}
```

Les 3 méthodes d'interpolations calculent les tableaux de coefficients nécessaires une bonne fois pour toute, puis se contentent de parcourir et de s'appliquer à tous les noeuds, le code est disponible en annexe. Une fois le maillage rempli (chacun des noeuds est doté d'une valeur) on peut visualiser le résultat grâce à la color map (et plus précisément la méthode `getColor` de la classe ColorMap ci-dessus), ou appliquer `computeIso` qui va de nouveau calculer les isolignes sur tous les pixels via la méthode du même nom de la classe Pix. L'affichage est réalisé par `dispIso` facilité par le chainage effectué sur les arêtes. Enfin les lignes de courant peuvent être afficher par la méthode `dispTraj` qui effectue un schéma d'Euler.

2.3 Schéma synoptique

Afin de clarifier encore davantage notre architecture nous récapitulons ici les étapes franchies et le rôle joué par chacune des classes. Rappelons les objectifs :

1. Importation et lecture des données localisées
2. Interpolation des données pour une évaluation en tout point, par la méthode de Shepard et la méthode des multiquadriques de Hardy ✓
3. Implémentation de 3 algorithmes, le premier calculant une image par carte de couleur, le second calculant des courbes iso-valeurs et le troisième calculant des lignes de courant ✓
4. Exportation des sorties des algorithmes dans des fichiers au format KML (Keyhole Markup Language), pour leur visualisation dans le logiciel Google Earth

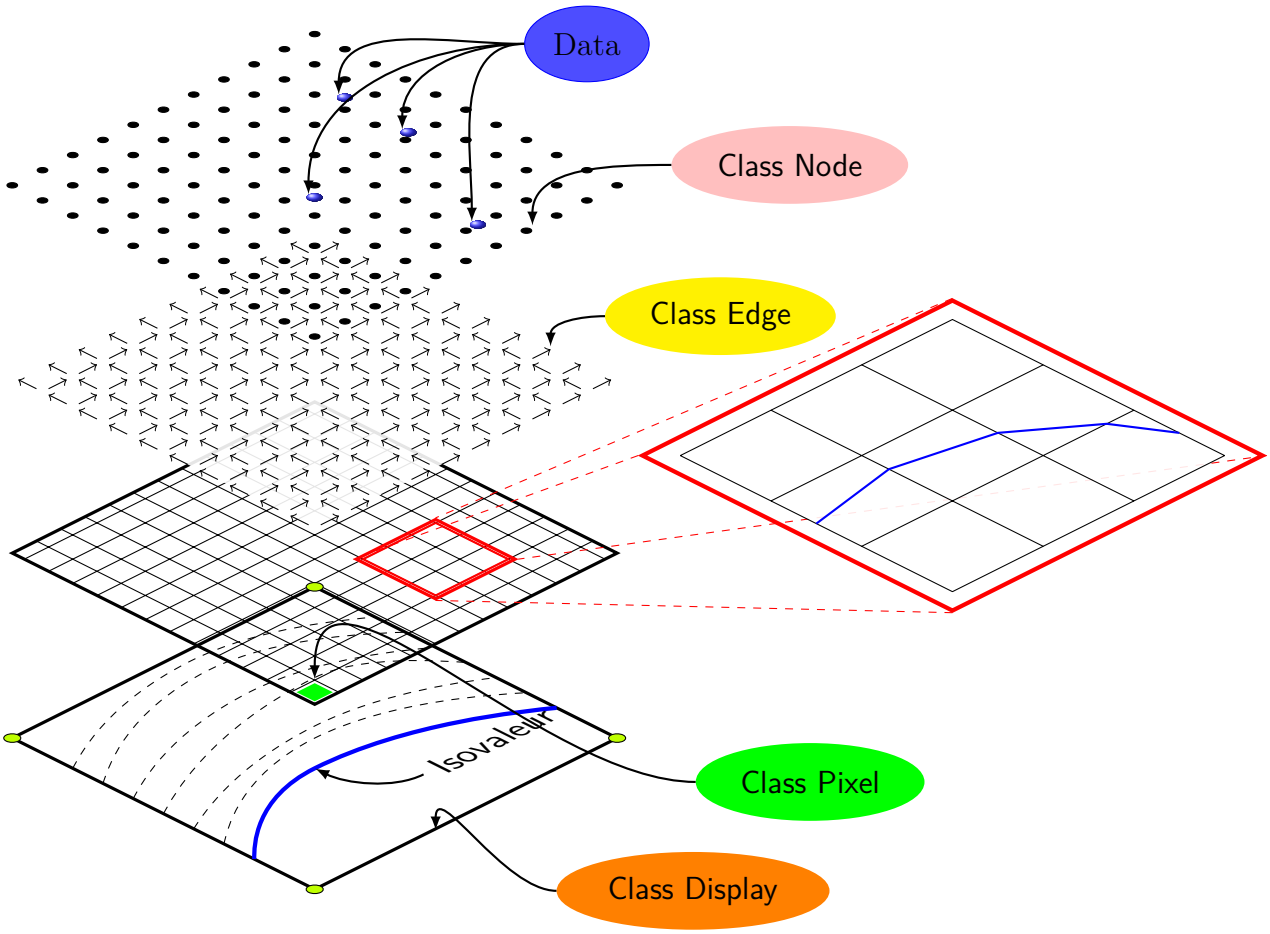

Figure 6 – Schéma synoptique : représentation en couches du programme

2.4 Importation/Exportation

En mode boite noire le programme les entrées et sorties sont représentées par le schéma suivant :

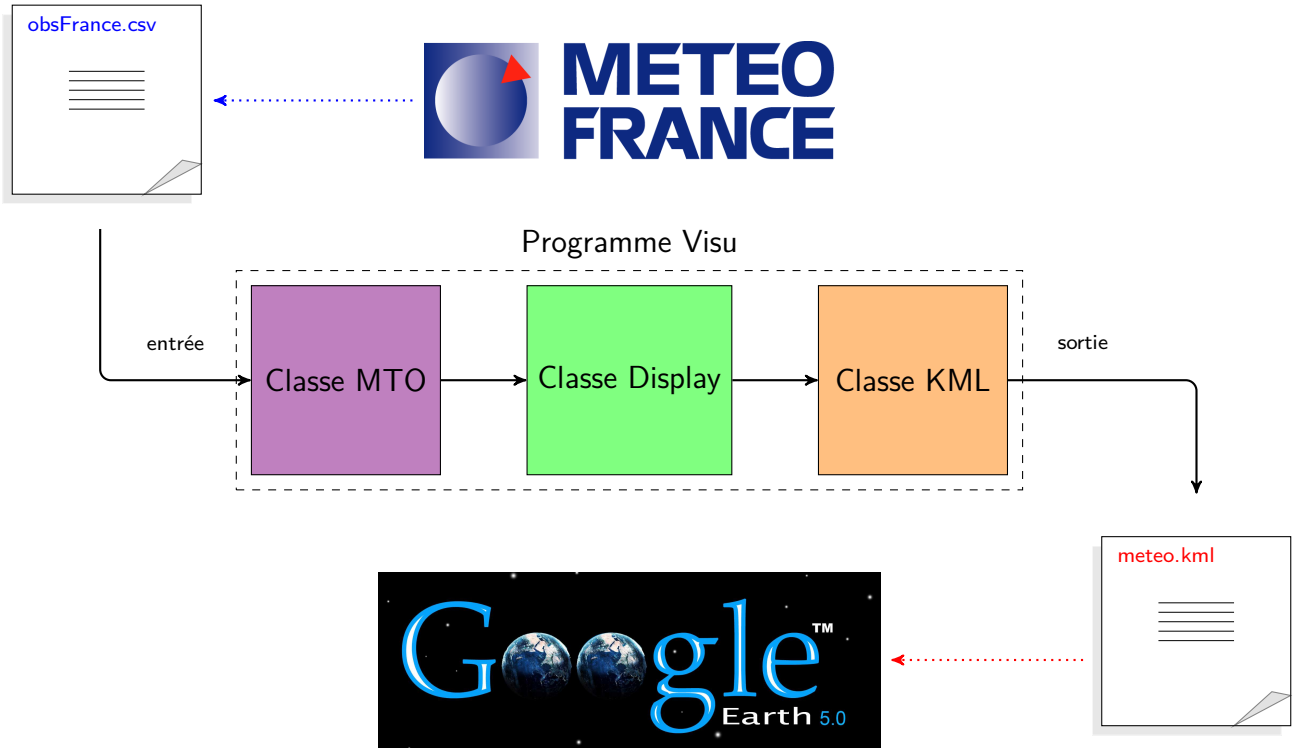

Figure 7 – Schématisation en boite noire

Classe MTO - Importation des données

Sur le site de météo France nous récupérons des observations (températures, pressions, etc) sur une journée (dans notre cas le 2/02/2013) toutes les 6h sur chacune des stations météorologiques de France. Toutes ces informations figurent dans le fichier `obsFrance.csv` dont la classe MTO se charge d'en extraire l'information pertinente. La méthode principale est `setupMTO()` qui joue le rôle de *parser*. Elle parcourt les lignes du fichier `obsFrance` en cherchant à mettre en correspondance l'ID avec les noms de stations répertoriés dans le fichier `stationMTO.csv`. Si elle en a trouvé une alors elle parse les informations d'heure, température, pression, direction et force du vent via l'emploi d'expressions régulières. Ces informations sont stockées dans les attributs d'un objet DataMTO. Pour chaque nouvelle station rencontrée, on créé un objet MTO contenant son nom, sa localisation géographique et la date des observations. On ajoute alors les informations récoltées à l'attribut `data` qui est un tableau d'objets DataMTO. On a ainsi créé pour chaque station météorologique un objet MTO contenant les différentes observations à intervalles de temps réguliers. Ces structures sont ensuite récupérées par la classe principale `Display` via la méthode `loadData`.

Classe KML - Exportation des données

Après traitement de l'information tel que décrit dans la partie précédente un objet `Display` est créé et contient tous les ingrédients nécessaires à l'affichage, soit d'une color map, de lignes de courant ou encore de courbes isovaleurs. Bien que notre programme permette d'exporter des images, nous souhaitons pouvoir visualiser le résultat par le biais du logiciel Google Earth. Celui-ci prend en entrée des fichiers au format KML, c'est pourquoi nous avons du traduire nos

informations dans ce format, c'est le rôle joué par la classe KML de notre programme.
La classe KML est composée de méthodes `print` qui encapsulent les informations dans différents objets Display à l'intérieur de balises du langage KML, pour ensuite les écrire dans le fichier `meteo.kml`. Ainsi les méthodes suivantes sont employées :

- `void printTemp(Display[] disp,String imgName,String[] date1,String[] date2)` : prend en entrée un tableau d'objets Display,dont chaque objet décrit l'état des grandeurs physiques observées entre les dates date1 et date2. Elle appelle sur chaque objet Display la méthode interne `dispKML` contenant les balises KML permettant d'indiquer au logiciel la date et l'emplacement de la grille $(ox, oy) - (px, py)$, et d'y superposer par transparence (gérée par la color Map) sur la carte de France l'image color map générée par notre programme.

- `void printPression(Display[] disp,String[] date1,String[] date2,int v)` : prend les même entrées que la méthode précédente. En revanche le code permettant la traduction en KML est plus conséquent car contrairement à la température où l'on se contente de calquer l'image sur la carte de Google Earth, ici on écrit directement en KML les segments composant les courbes isovaleurs. De plus l'implémentation par chainage du marching square permet d'afficher les courbes d'un seul coup.

- `void printMTO(MTO[] mto)` : fait également appel sur chaque station MTO à la méthode interne `dispKML` écrivant en KML les informations de localisation géographique des stations météorologiques ainsi que la température, pression, etc sur ce site à la date voulue. Cela nous permet d'afficher des info bulles décrivant l'état du système lorsque l'on clique sur une station.

A ce stade nous avons remplis tous les objectifs :

1. Importation et lecture des données localisées ✓
2. Interpolation des données pour une évaluation en tout point, par la méthode de Shepard et la méthode des multiquadriques de Hardy ✓
3. Implémentation de 3 algorithmes, le premier calculant une image par carte de couleur, le second calculant des courbes iso-valeurs et le troisième calculant des lignes de courant ✓
4. Exportation des sorties des algorithmes dans des fichiers au format KML (Keyhole Markup Language), pour leur visualisation dans le logiciel Google Earth ✓

3 Résultats

Nous présentons ici les résultats obtenus par le biais de plusieurs captures d'écrans. Pour vérifier le bon fonctionnement de notre programme, exécutez `Visu` qui vous génèrera plusieurs images ainsi qu'un fichier KML depuis lequel vous pouvez alors lancer Google Earth.

3.1 A partir des images générées

Travaillons sur les données recueillies le 2/03/2013. A titre de comparaison nous avons téléchargé les cartes de températures et de pressions mises à disposition sur le site de meteociel.fr (à gauche sur les figures ci-après). Quant à nos images à droite, nous avons utilisé pour la température une échelle de quantification des couleurs pour y voir plus clair, c'est-à-dire qu'on se fixe une palette de n niveaux de couleurs et que selon la valeur interpolée on affecte la couleur la plus « proche ».

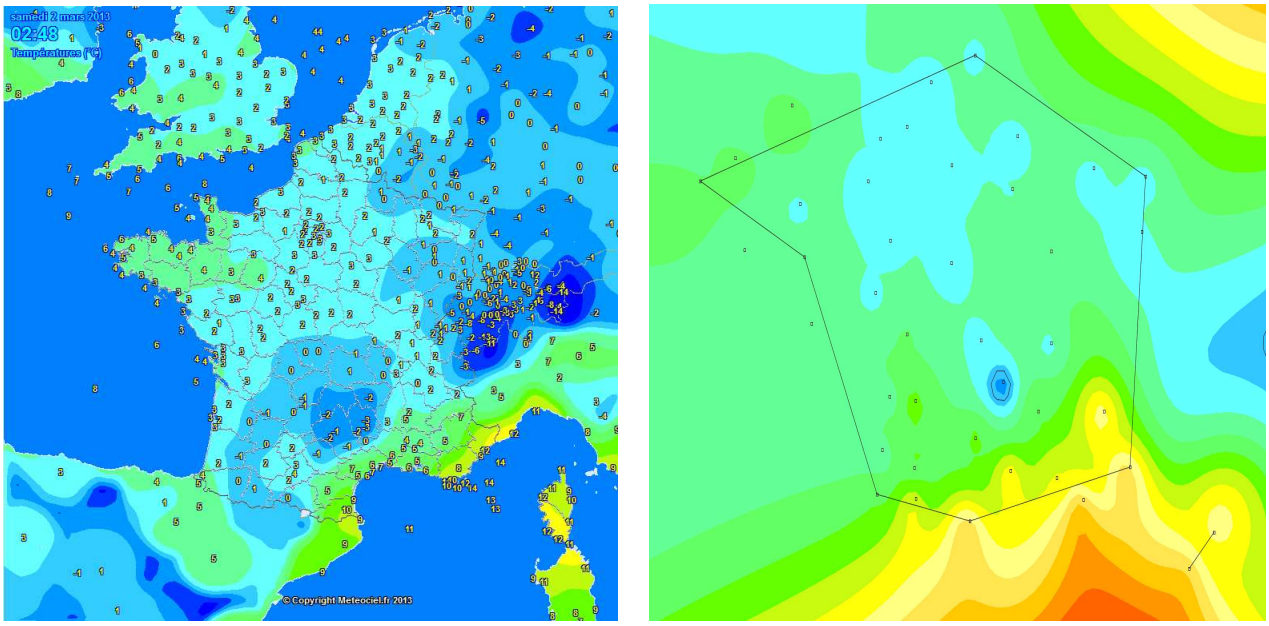

Figure 8 – Carte des températures par MQS

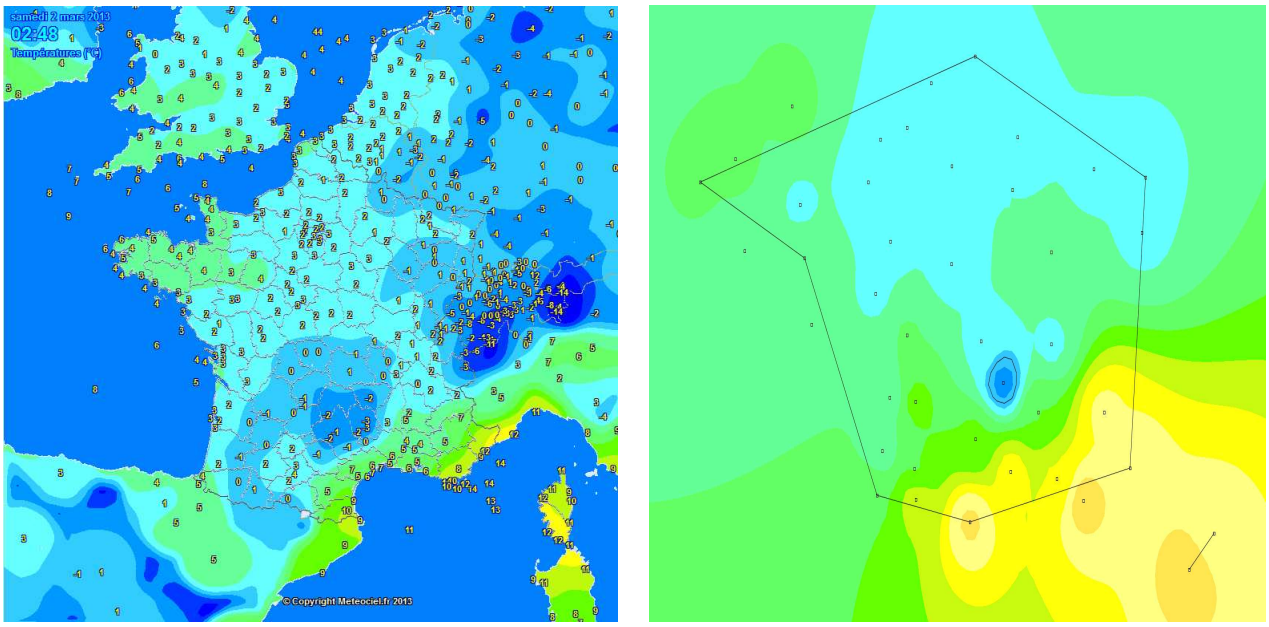

Figure 9 – Carte des températures par Shepard

On constate dans les deux cas que nos résultats sont globalement fidèles aux cartes de températures réelles, on retrouve bien les zones froides et les zones chaudes aux mêmes endroits. Nous avons une légère préférence pour la méthode de Shépard (coefficient $k = 3$), dans la mesure où la méthode MQS a tendance à adopter un comportement linéaire dans les zones où nous ne disposons pas de données, typiquement ici au nord-est en dehors de la France où on observe un dégradé indésirable.

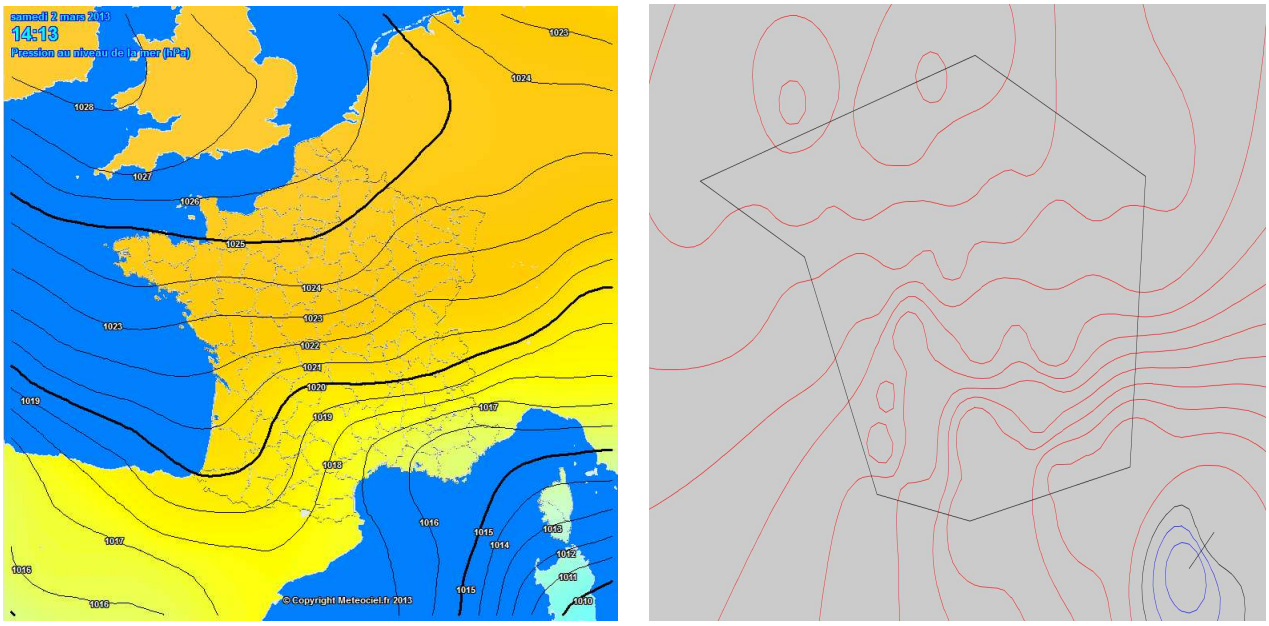

Figure 10 – Carte des pressions

Les courbes isovaleurs en rouge correspondent aux zones anticycloniques ($> 1014hPa$) et en bleu aux zones dépressionaires ($< 1014hPa$), qui sont utiles entre autres pour déterminer la direction du vent. Nos différentes courbes balayent l'intervalle de pression $[980hPa, 1040hPa]$ avec un pas de $1hPa$ ce qui est déjà une plage conséquente, $980hPa$ étant la pression au centre d'un petit ouragan et $1040hPa$ est une pression remarquable dans un anticyclone, puisque proche du record en France de $1050hPa$ qui date de 1821 ! A noter que les valeurs minimales et maximales jamais enregistrées sont respectivement $870hPa$ et $1086hPa$ en Sibérie. On constate là aussi que nos courbes sont cohérentes avec la réalité, au sein du territoire français elles sont alignées dans le même sens et suivent le même comportement que celles de la carte de météociel. On observe également les même anticyclones au niveau de la Corse et de la Grande-Bretagne.

3.2 Sur Google Earth

En ouvrant le fichier KML avec Google Earth, on peut visualiser par exemple la carte des températures calquée sur la carte de France. Toujours à titre de comparaison, on affiche à gauche la carte de météociel. Les observations sont celles du 11/02/2013. On constate alors que les deux cartes sont encore une fois relativement similaires.

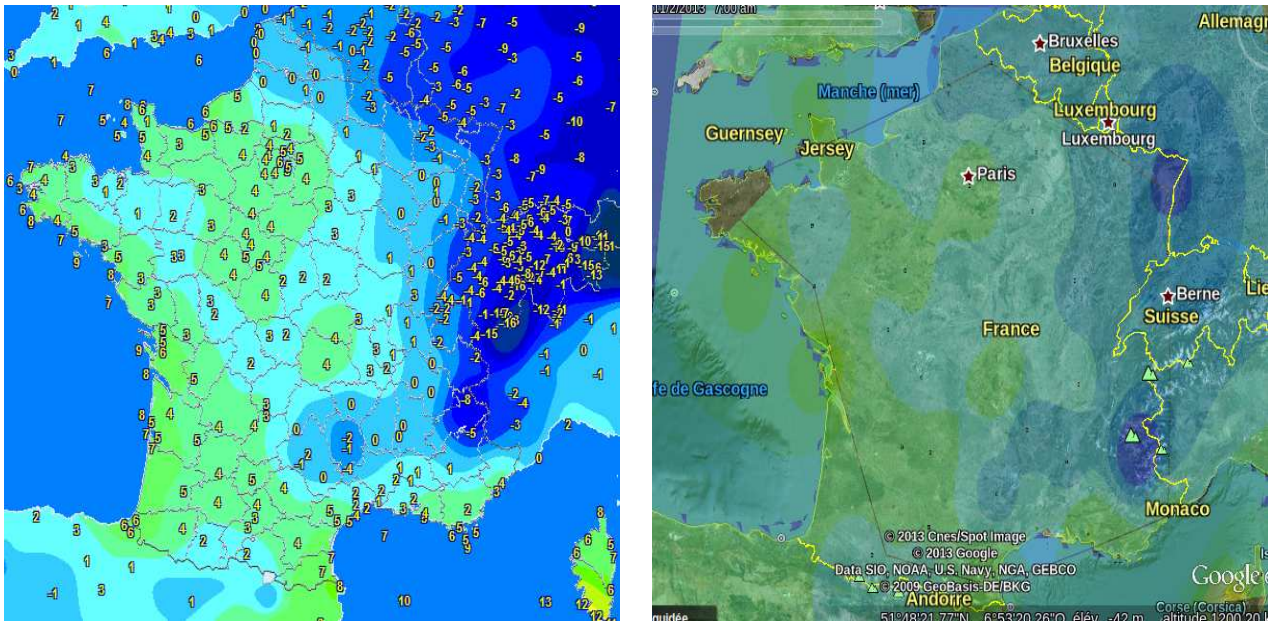

Figure 11 – Carte des températures à 7h sur météociel (à gauche) et sur Google Earth (à droite)

Nous avons opté pour une color map qui imite celle de meteociel.fr, de manière à pouvoir comparer correctement les deux cartes. On constate que l'on retrouve bien les même distributions de températures, notamment au centre de la France avec la zone circulaire en vert entourée d'une zone plus froide en bleu turquoise, puis encore en dessous d'une zone encore plus froide en bleu. On retrouve également à l'est et au niveau de la suisse et de l'allemagne des températures plus froides représentées en bleu foncé.

En déplaçant le curseur en haut à gauche on peut visualiser la carte de température à d'autres moments de la journée, voir la Figure 12. On visualise bien le refroidissement global de la France de 20h à 2h du matin (passage du vert au bleu). De nouveau toutes les cartes sont cohérentes avec celles fournies par meteociel.fr.

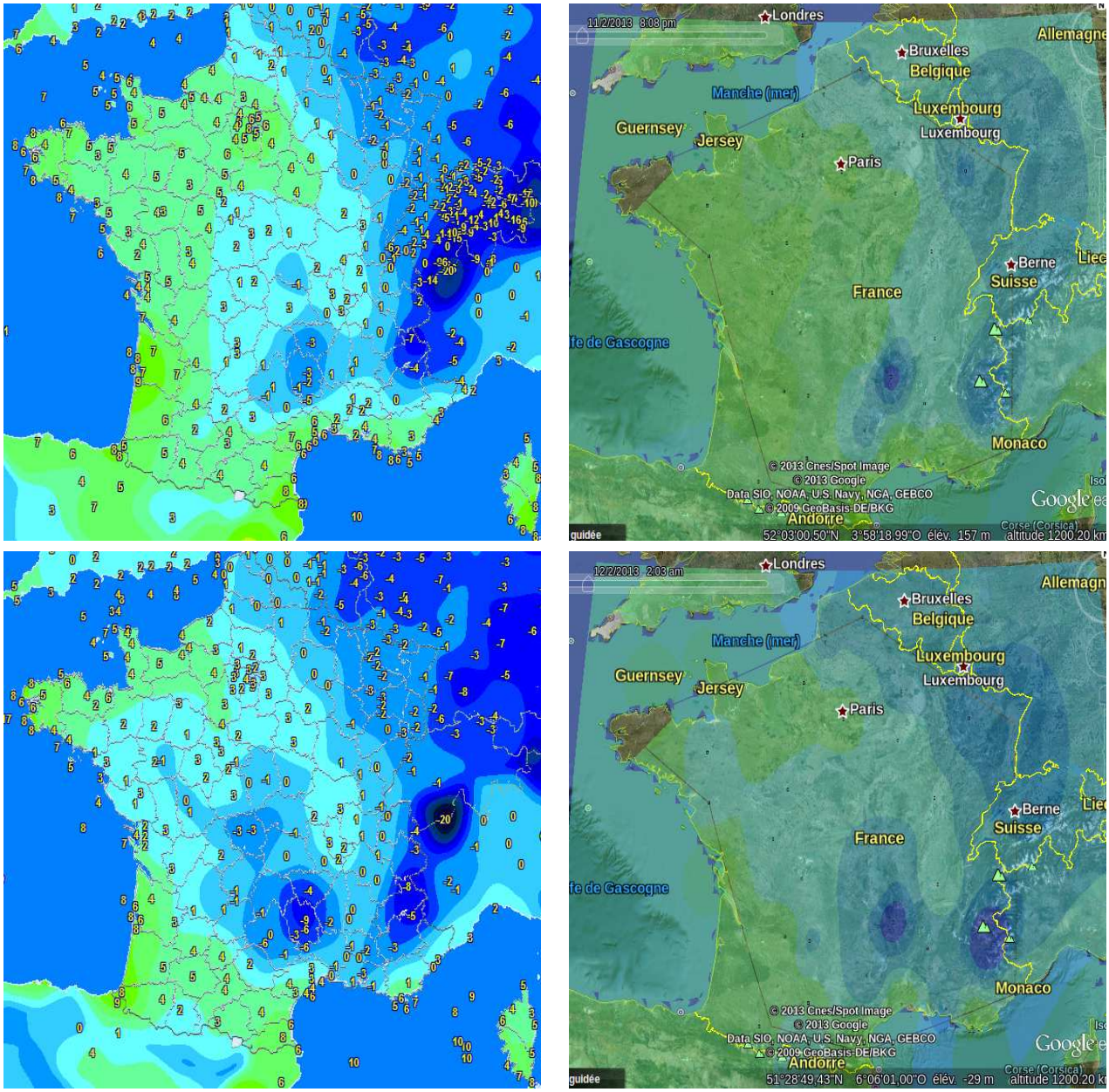

Figure 12 – Carte des températures à 20h puis au lendemain 2h

En cochant StationMTO on peut épingler les différentes stations météorologiques sur la carte de France, et si on clique sur l'une d'elle une infobulle s'affiche pour nous renseigner sur sa position géographique et ainsi que les observations mesurées.

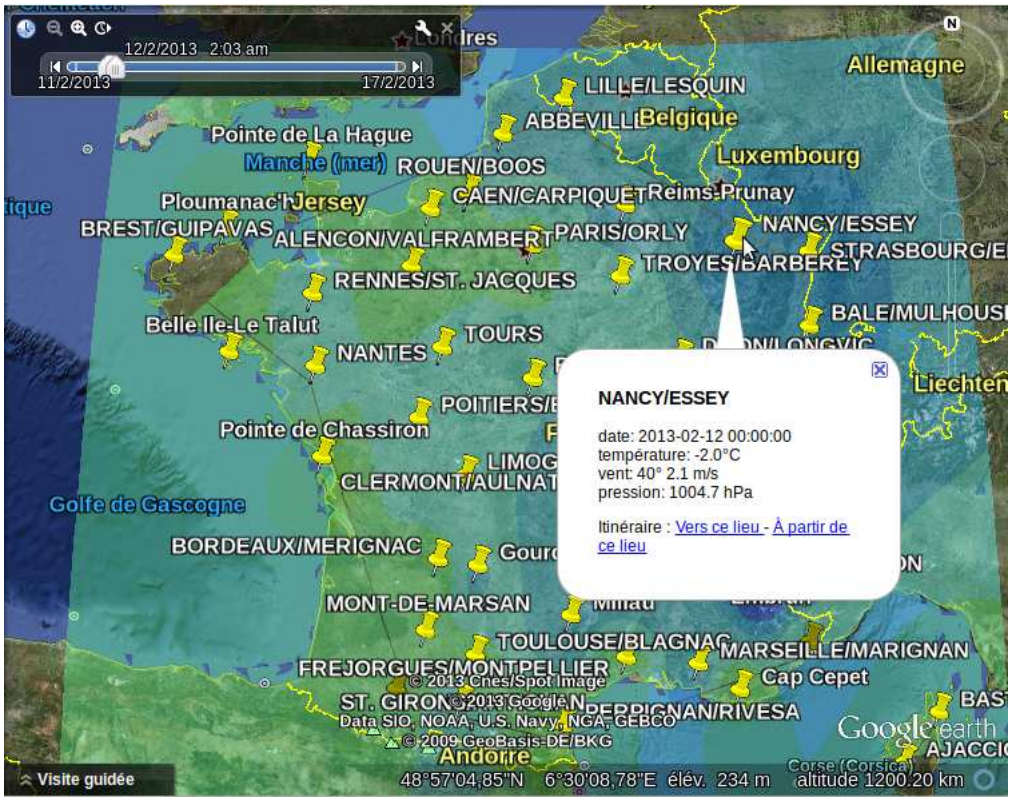

Figure 13 – Affichage des stations météo et des infobulles

On peut aussi faire figurer les courbes d'isopression :

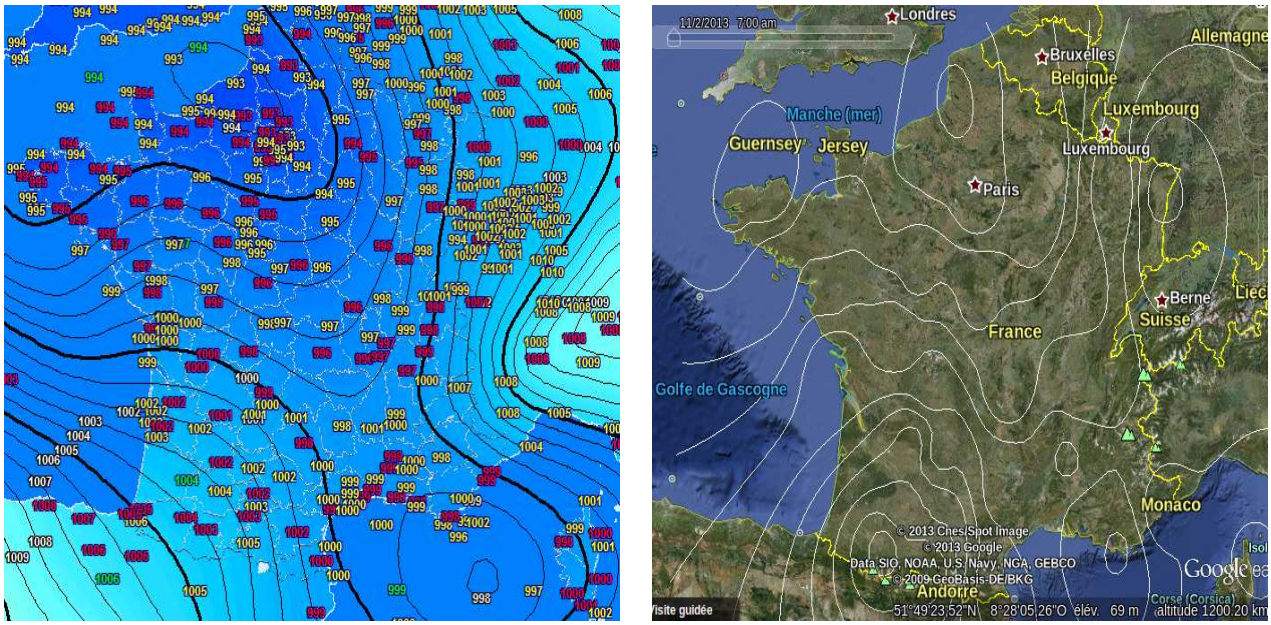

Figure 14 – Carte des pressions à 7h sur météociel (à gauche) et sur Google Earth (à droite)

Toujours dans l'optique d'améliorer la visualisation, nous avons (comme dans les cartes météo classiques) fait apparaître certaines lignes isobares en gras. Par ailleurs nous avons placé des échelles de températures et de vitesse du vent de façon à pouvoir interpréter directement la carte d'un coup d'oeil.

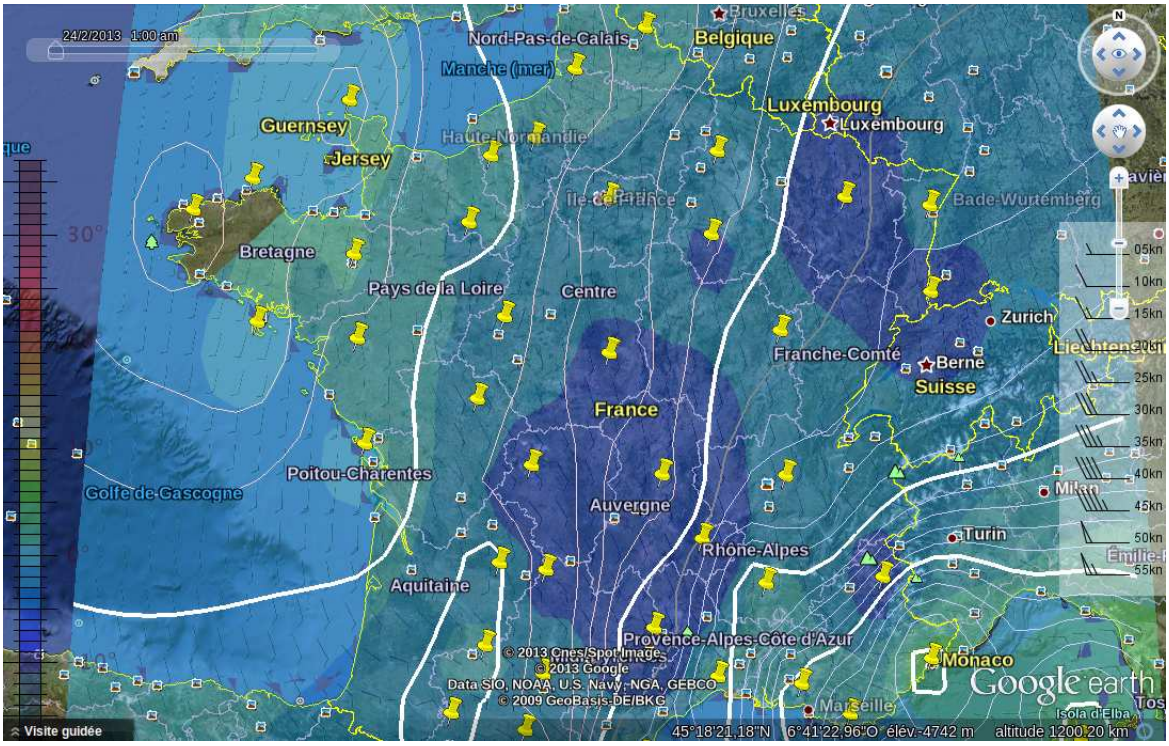

Figure 15 – Echelles de températures et de vitesse du vent

Si on zoome davantage voici la façon dont nous avons modélisé le champ de vecteurs vitesse du vent. L'inclinaison du segment donne l'orientation, le sens est dirigé de la queue (dont le nombre de barres spécifie la force, cf. échelle des vitesses) vers la tête.

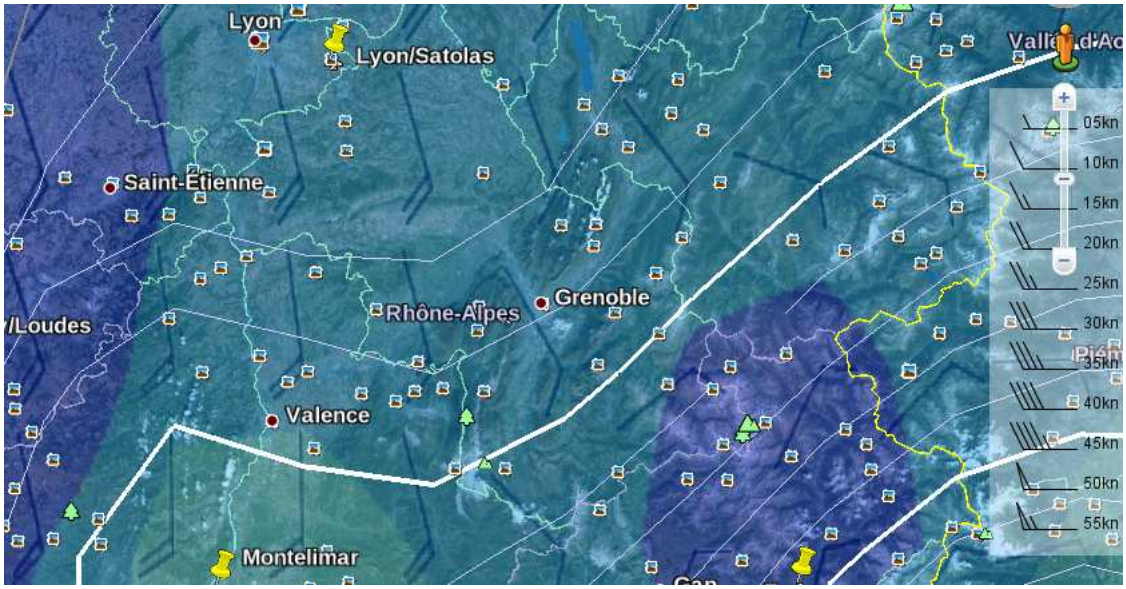

Figure 16 – Représentation du champ de vecteurs vitesse du vent

Enfin on peut rajouter le tracé des lignes de courant et on obtient une carte météorologique complète :

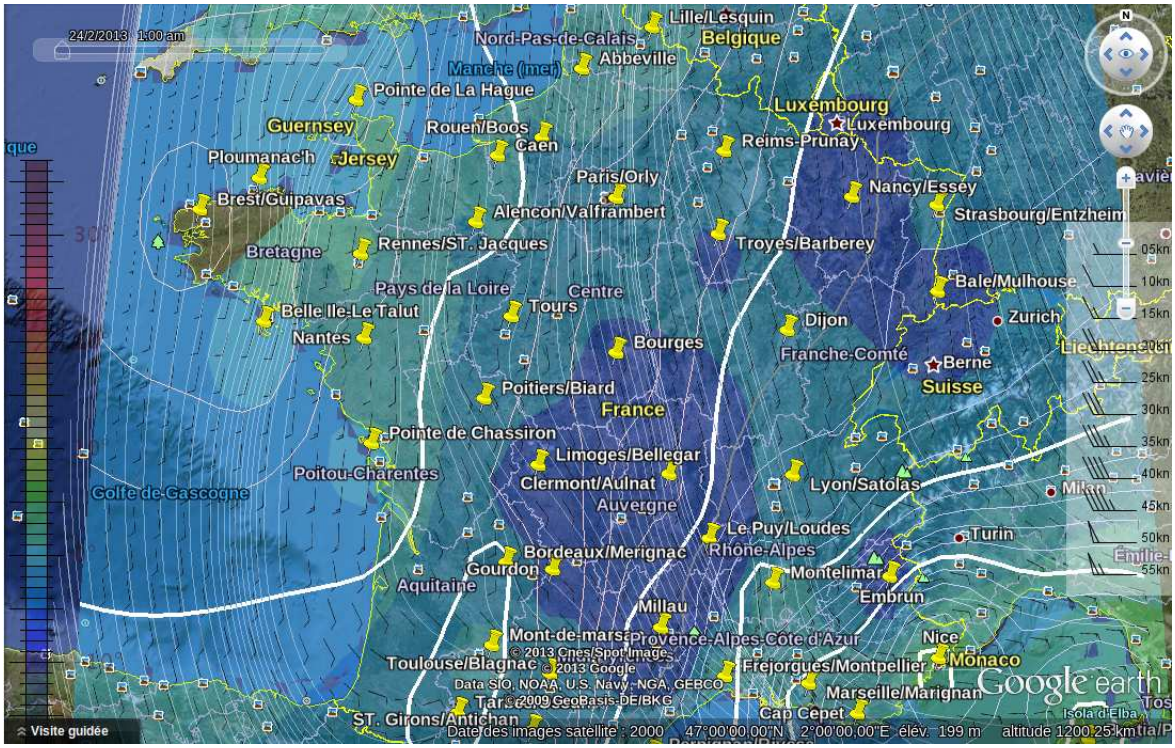

Figure 17 – Ajout des lignes de courants

Pour finir, nous avons généré un deuxième fichier `pollution.kml`, à partir cette fois-ci non pas d'observations météorologiques, mais de relevés de pollution délivrés par l'observatoire de l'air de la région Rhône-Alpes. Ces observations étaient regroupées dans un fichier que nous avons du de nouveau « parser », à ceci près que l'encodage utilisé ne permet pas de traiter les noms ou types de stations comportant des accents. Nous avons par ailleurs adapté l'échelle de couleurs à l'étude de la concentration en particules PM10 (https://fr.wikipedia.org/wiki/PM10#France). Le résultat obtenu est le suivant :

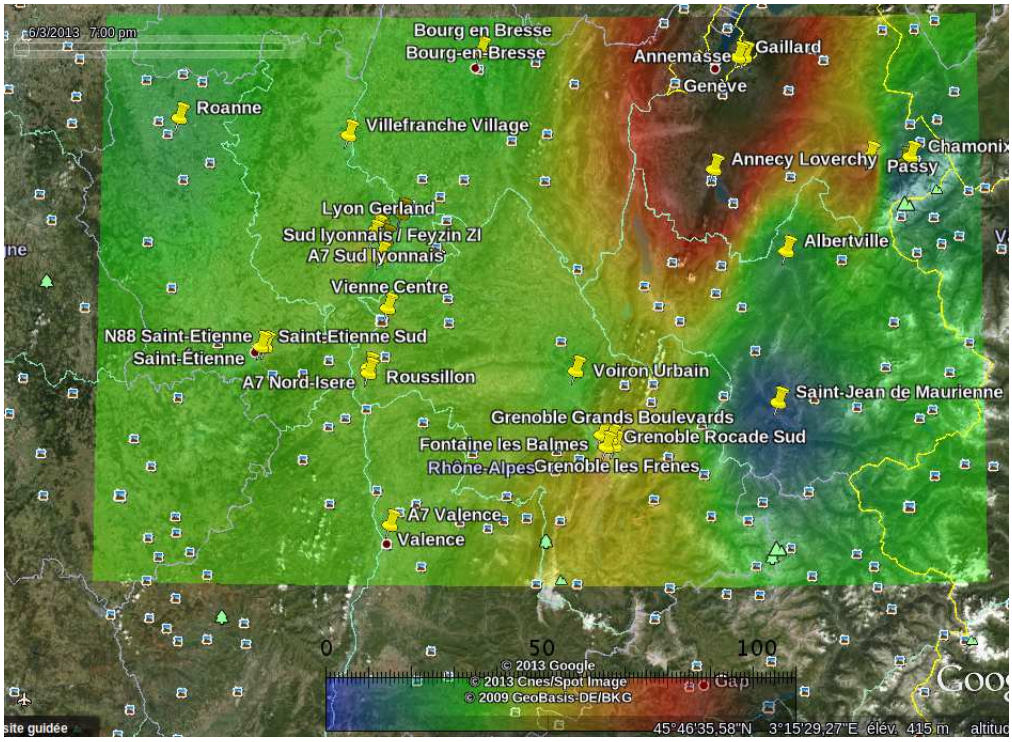

Figure 18 – Concentration en particules PM10 sur la région Rhône-Alpes

4 Annexes

4.1 Shépard

```
 int shepard(float k,int vi)
{
  int ret=0;
 for(int i=0;i<node.length;i++)
 for(int j=0;j<node[i].length;j++)
 {ret=node[i][j].shepard(k,vi);}

  return ret;
}
```

```
 int shepard(float k,int vi)
{
  float shep=0,wj=0,wi=0;
  for(int i=0;i<parent.data.length;i++)
  {
    // si le noeud tombe exactement sur un des points donnes pour l'
       interpolation
    if(rex==parent.data[i].rex && rey==parent.data[i].rey)
    { return addValue(parent.data[i].value[vi]);} // le noeud prend la
        valeur du point
    else
    {
      wi=pow(dist2(parent.data[i]),k); // distance du noeud aux
         donnees
      shep+=(parent.data[i].value[vi]/wi);
      wj+=(1./wi);
    }
  }
  return addValue(shep/wj);
}
```

4.2 MQS

```
 double[][] coefMQS(float Rq,int vi)
{
  float[] Phi= new float[data.length];
  Node[] B=new Node[data.length];
  float[] Bk=new float[5];
  double[][] Mat=new double[5][5];
  double[] A= new double[5];
  double[][] result = new double[data.length][5];

  for(int i = 0;i<data.length;i++)//pour chaque point
  {
       for(int j=0;j<data.length;j++)// on calcul le pi(pj) et les Bk
       {
          if(i!=j)
          {
```

```
                Phi[j]=phi(data[i],data[j],Rq);
                B[j]=data[j].subN(data[i]);
            }
        }

        for(int h=0;h<5;h++)
        {for(int j=0;j<5;j++)//MAJ MATRICE
             Mat[h][j]=0;
        A[h]=0;
        }


        for(int k=0;k<data.length;k++)//on rempli la matrice
            triangulaire sup ( symetrie)
        {
            if(i!=k)
            {
                 Bk[0]=B[k].rex;
                 Bk[1]=B[k].rey;
                 Bk[2]=B[k].rex*B[k].rex;
                 Bk[3]=B[k].rey*B[k].rey;
                 Bk[4]=B[k].rey*B[k].rex;

                for(int h=0;h<5;h++)
                {  for(int j=h;j<5;j++)
                       Mat[h][j]+=Bk[h]*Bk[j]*Phi[k];
                       A[h]+=B[k].value[vi]*Phi[k];
                }
            }
        }

        for(int h=1;h<5;h++)
        for(int j=0;j<h;j++)
          Mat[h][j]=Mat[j][h];

        Matrix M=new Matrix(Mat);
        Matrix AA=new Matrix(A,1);
        M=M.inverse();
        AA=M.times(AA.transpose());
        arrayCopy(AA.getColumnPackedCopy(),result[i]);
    }

    return result;
}
```

```
 int MQS(float Rw,float Rq,int vi)
{

 double[][] Coef=coefMQS(Rq,vi);
 int ret=0;
 for(int i=0;i<node.length;i++)
 for(int j=0;j<node[i].length;j++)
 {ret=node[i][j].MQS(Coef,Rw,vi);}
```

```
  return ret;
}
```

```
 int MQS(double[][] coef,float Rw,int vi)
{

  double shep=0,wj=0,wi=0,qi=0;
  Node K=new Node(x,y,rex,rey,parent);
  K.setValue(parent.data[0].value);
  K.value[vi]=0;
  for(int i=0;i<parent.data.length;i++)
  {
    if(rex==parent.data[i].rex && rey==parent.data[i].rey)
    {  return addValue(parent.data[i].value[vi]);}
    else
    {
      wi=parent.phi(parent.data[i],K,Rw);
      Node Ki=K.subN(parent.data[i]);
      qi=-Ki.value[vi]+(coef[i][0]+coef[i][2]*Ki.rex+coef[i][4]*Ki.rey
         )*Ki.rex+(coef[i][1]+coef[i][3]*Ki.rey)*Ki.rey;
      shep+=(wi*qi);
      wj+=wi;
    }
  }
  return addValue((float)(shep/wj));
}
```

4.3 Hardy

```
 double[] coefHardy(float c,int vi)
{
  int m = data.length;
  double[][] tab = new double[m][m];
  // on remplit la matrice A
  for (int i=0;i<m;i++)
  {
    for (int j=0;j<m;j++) {
      // distance entre point Pi et Pj
     float d = data[i].dist2(data[j]);
     tab[i][j] = sqrt(d*d+c*c);
    }
  }
  Matrix A = new Matrix(tab);
  //Matrix invA = A.inverse();
  //invA.print(m,1);
  double[] bb = new double[m];
  for (int i=0;i<m;i++)
  {
    bb[i] = data[i].value[vi];
  }
  Matrix b = new Matrix(bb,m);
```

```
  //b.print(m,1);
  Matrix l = A.solve(b);
  //l.print(m,1);
  double [] lambda = new double[m];
   for (int i=0;i<m;i++)
  {
    lambda[i] = l.get(i,0);
  }
  return lambda;
}
```

```
 int hardy(float c, int vi)
{
 double[] Coef=coefHardy(c,vi);
 int ret=0;
 for(int i=0;i<node.length;i++)
 for(int j=0;j<node[i].length;j++)
   ret=node[i][j].hardy(Coef,c,vi);
 return ret;

}
```

```
 int hardy(double[] coef, float c, int vi)
{
  float s = 0;
  for (int i=0;i<parent.data.length;i++) {
    s += coef[i]*sqrt(pow(dist2(parent.data[i]),2)+c*c);
  }
  return addValue(s);
}
```