Prototype de GéoVisualisation
Description, Méthodes, Usages

1. Organisation du projet (dossier « TouristsInParis »)

Le prototype est rangé dans le dossier nommé TouristsInParis téléchargeable depuis Github, il est écrit en java et appelle un grand nombre de primitives Processing pour, notamment, l'affichage du flux de mobilité.
L'application est actuellement composée de 18 classes java ayant chacune un rôle bien précis :

- **Classes d'encadrement :**
  - *Session.java* : contient et initialise toutes les variables nécessaires au fonctionnement du prototype. On y trouve aussi l'ensemble des Getter() et Setter()
  - *Application.java* : instancie une nouvelle Session de travail à chaque démarrage du prototype

- **Classes de fonctionnement :**
  - *TouristFlow.java* : classe centrale, elle capte les actions de l'utilisateur et appelle les autres classes en fonction
  - *Gephi.java* : décrit le comportement d'un objet « Gephi » (un objet graphe) issu d'un Gexf donné.
  - *Temps.java* : gère la navigation temporelle au sein du graphe de flux
  - *BoutonMenu.java* : décrit le comportement d'un objet « BoutonMenu »
  - *Stick.java* : décrit le comportement d'un objet « Curseur »
  - *Bibliotheque.java* : regroupe l'ensemble des fonctions de calcule utilisées dans le reste du prototype
  - *Affichage.java* : gère l'affichage des légendes et titres des visualisations, l'affichage des distributions et de l'échelle
  - *Edge.java* : gère l'affichage des arcs du graphe de flux
  - *Node.java* : gère l'affichage des nœuds du graphe de flux
  - *Arrow.java* : décrit le comportement d'un objet graphique « flèche »
  - *Smooth.java* : génère les cartes lissée et les champs de flèches
  - *HeatMap.java* : permet de visualiser le graphe de flux sous la forme d'une HeatMap
  - *KMeans.java* : contient l'algorithme de Kmeans utilisé pour l'agrégation des oursins
  - *Redistribution.java* : modifie la répartition originale des BTS et édite un nouveau graphe normalisé au format CSV

Dans la mesure du possible, il faut ranger chaque nouvelle variable globale dans *Session.java* et penser à implémenter les accesseurs get() et set() correspondants.

Dans un souci de clarté toute les sources de données d'entrée sont rangées dans le dossier :

TouristsInParis→Ressources

De même, les scripts pythons utiles pour transformer les données en entrée sont rangés dans :

TouristsInParis→Scripts python

2. Précisions sur la classe *TouristFlow.java*

Classe centrale du prototype, il est nécessaire d'expliquer son fonctionnement (basé sur celui d'un sketch processing) pour comprendre la structure du projet.
La classe peut être vue comme telle :

```
class TouristFlow {

        void setup( ){
                . . .
        }

        void draw( ){
                . . .
        }

        void keyReleased( ){
                . . .
        }
        void mousePressed( ){
                . . .
        }
        . . .

}
```

Block setup() lancé une fois, utilisé pour les initialisations

Block draw() tourne en boucle, appelle les classes d'affichages et de visualisations

Block key/mouse() capte les actions de l'utilisateur

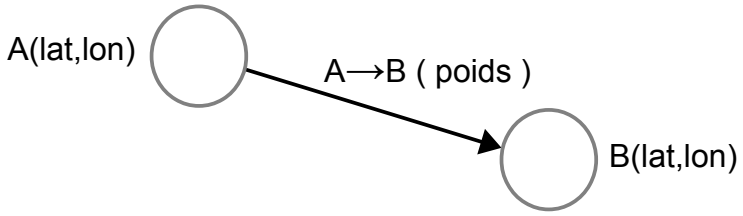
Au lancement du prototype la fonction setup() initialise les variables, les tableaux, les matrices. Si l'utilisateur fait une action signifiante, celle ci est captée par une fonctions key/mouse() qui change le statut d'une variable de type booléen (rangée dans *Session.java*). En fonction du statut de cette variable, la fonction draw() appelle la classe correspondante, transformant ainsi l'action de l'utilisateur en affichage à l'écran.

3. Charger un graphe de flux

3.1 Comment structurer les données en entrée ?

Le prototype permet de visualiser un flux de mobilité évoluant au cours du temps au sein d'un territoire géographique donné.
Tout d'abord, l'utilisateur doit établir une période d'étude (une journée, une semaine …) discrète (divisée en heure, en journée ...). Ainsi le flux sera représenté par un ensemble de **graphes datés, successifs et régulièrement espacés**.
Chaque graphe est défini comme **un ensemble de nœuds géoréférencés et d'arcs valués orientés**.

Dans le cadre de notre étude un nœud symbolise une antenne de téléphonie mobile unique (couple latitude/longitude) et un arc symbolise le déplacement d'un groupe d'individus (poids) d'une antenne A vers une antenne B :



Le prototype travaille sur des graphes au format Gexf.
Concrètement, l'utilisateur peut dans un premier temps créer un fichier CSV décrivant les nœuds comme tel :

"lat";"lon";"id"
48.1437;2.46986;1
48.1487;2.69595;2
48.1685;2.65105;3
48.1711;2.52148;4
(…)

Puis il crée autant de fichiers CSV décrivant les arcs qu'il a défini d'intervalles sur sa période d'étude :

"source";"target";"poids "
342;438;1.0
890;891;5.0
739;738;1.0
739;738;1.0
(…)

Enfin en utilisant le script python :

TouristsInParis→Scripts python→ CSV_to_GEXF_multi_files.py

On obtient les Gexf correspondants :

```
<node id="538" label="538">
        <attvalues>
                <attvalue for="0" value="48.78221"/>
                <attvalue for="1" value="2.229156"/>
                <attvalue for="2" value="0.0"/>
        </attvalues>
 </node>

(…)

<edge source="741" target="706" id="1487" weight="4.0">
        <attvalues/>
</edge>

(...)
```

note : Il est possible de rajouter des attributs supplémentaires aux nœuds et arcs, le prototype devra être adapté en conséquence.

3.2 Où charger les Gexf dans le prototype ?

Les chemins vers les Gexf doivent être indiqués dans :

Bibliotheque.java→loadGraphe( )

Pour chaque Gexf il faut répéter :

```
Application.session.setTableauGephi(nb, new Gephi());
Application.session.getTableauGephi()[nb].loadG("Chemin vers Gexf");
```

Où « nb » correspond à l'ordre du graphe (0 pour le premier de la période d'étude, 1 pour le suivant et ainsi de suite).

3.3 Mécanisme de parsing des Gexf

Voici sous forme algorithmique le processus de transformation des Gexf en données utilisable par le prototype :

Initialisation :

setup( ) → Bibliotheque.loadGraph( )

Fonction Bibliotheque.loadGraph() :

Un tableau d'Objet « Gephi » a été crée lors de l'initialisation (classe *Session.java*)

setTableauGephi(nb, new Gephi())
getTableauGephi()[nb].loadG('....') } ***Créer et ranger de nouveaux objets Gephi dans un tableau***

Objet Gephi :

Un objet Gephi est une structure représentant via ses attributs (Tableaux, matrices) un graphe Gexf alors utilisable dans un programme.

btsLat[ ]
btsLon[ ]
btsDegree[ ]
btsAttribut1[ ]
Edge[ ][5]

} ***Objet Gephi***

Un objet Gephi possède une méthode loadG('chemin') qui en utilisant le Gephi Toolkit parse le graphe pointé par le chemin et range les données dans des tableaux.

Fin :

Si l'utilisateur a correctement renseigné les chemins vers les Gexf, le tableau TableauGephi[ ] contient désormais tous les graphes nécessaires à l'affichage du flux complet.

Note : aucun filtrage sur les données n'est effectué à ce niveau, il ne s'agit que d'une transformation d'un format vers un autre. Néanmoins, lors de cette étape, le script a calculé dans la fonction loadG() toutes les informations de type degré Max/Min, poids Max/Min, distances … Ces informations sont stockées dans des variables (classe *Session.java*) utilisées plus tard pour divers calculs .

4. Gérer le temps

Dans la partie précédente nous avons pointé du doigt la nécessité de définir une durée d'étude découpée en intervalles réguliers. Nous obtenons ainsi deux informations importantes : la **période** et le **pas**.

4.1 Que doit on renseigner dans le code ?

Il existe encore dans le prototype quelques informations « codées en dur » que l'on se doit de compléter à la main. Concernant le temps dans la classe *Temps.java* l'utilisateur doit modifier :

la période :

```
public static String firstHourStamp = "2009033100";
public static String lastHourStamp = "2009033124";
```

firstHourStamp contient la date de début de la période (année – mois – jour – heure) et lastHourStamp la date de fin.

le pas :

le pas doit être toujours défini en heure, ainsi on renseigne le nombre de millisecondes dans un intervalle

```
public static final long MILLIS_PER_4HOURS = pas * 60 * 60 * 1000;
```

4.2 Comment passer d'un intervalle à un autre ?

Lors de l'initialisation du prototype (setup( ) → Temps.setupDates( )) la fonction setupDates() parse firstHourStamp et lastHourStamp et déduit ainsi le nombre d'intervalles en fonction du pas :

```
hourCount = (lastHourMillis - firstHourMillis) / MILLIS_PER_4HOURS;
```

setupDates() place alors le compteur hourIndex à 0, ainsi à l'ouverture du prototype celui ci est calé sur le premier intervalle. Quand l'utilisateur clique sur la timeline pour avancer ou reculer il modifie la valeur de hourIndex. Si hourIndex est < 0 alors hourIndex = hourCount – 1, si hourIndex est > hourCount – 1 alors hourIndex = 0. La fonction drawDateSelector() transforme ensuite hourIndex en année – mois – jour – heure pour l'affichage écran, ainsi l'utilisateur à l'impression de parcourir un cadrant.

4.3 La variable globale Index

Nous venons de le voir, le compteur local hourIndex appartient à l'intervalle [0 ; hourCount – 1], à chaque changement de ce dernier sa valeur est copiée dans la variable global Index (classe *Session.java*) :

```
Application.session.setIndex(hourIndex);
```

Le lien est ainsi fait entre la classe *Temps.java* et l'ensemble du prototype. Dans la fonction draw() lorsque l'index change :

```
if (Application.session.getIndexBis() != Application.session.getIndex()){
        ( . . . )
}
```

On appelle la fonction Bibliotheque.remplirTableauImage(index) qui fonctionne comme telle :

- initialisation de deux Matrices (une pour les nœuds : MatNode, une pour les arcs : MatEdge)
- récupérer l'objet Gephi correspondant à l'index dans TableauGephi
- stocker les données des nœuds dans MatNode

– stocker les données des Arcs dans MatEdge

Désormais, **on accède au graphe courant via ces deux Matrices**, évitant ainsi un accès trop couteux à l'objet Gephi. Si l'index change encore, on appelle à nouveau remplirTableauImage(index) pour mettre à jour les matrices de travail.

Lors d'un changement d'Index il n'y a pas que cette fonction qui est appelée, d'autres routines de mises à jour sont invoquées comme nous le verrons par la suite.

5. Afficher un fond de carte

5.1 Librairie Unfolding

L'évolution de notre graphe de flux est à la fois temporelle et géographique, nous cherchons donc à encrer celui ci dans un territoire précis. Pour ce faire, le prototype utilise la librairie Unfolding qui lui permet de générer des fonds de cartes.
La librairie est librement utilisable (licence BSD) et régulièrement mise à jour.

Intégration au prototype :

définir les variables map et location (Session.java)
de.fhpotsdam.unfolding.Map map;
de.fhpotsdam.unfolding.geo.Location locationCourante;

initialisation (setup())
Application.session.setMap(new de.fhpotsdam.unfolding.Map(this));
Application.session.map.zoomAndPanTo(new Location(lat1, lon1), zoom);
MapUtils.createDefaultEventDispatcher(this, Application.session.getMap());

il faut renseigner le couple (lat1, lon1) pour la position départ de la carte et la variable zoom pour le niveau de zoom initial de la carte (10 semble être une bonne hauteur d'ensemble)

affichage (draw())
Application.session.getMap().draw();

note : par défaut le fond de carte est tiré du projet Open Street Map, mais il est possible d'obtenir des cartes venant d'autres sources (Microsoft, Google Maps, CloudeMade) en modifiant comme tel :

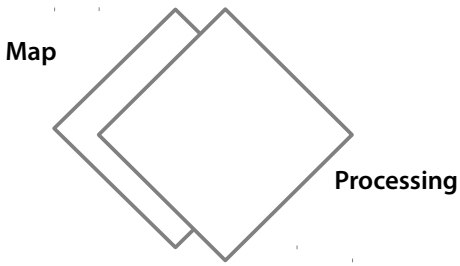
setMap(new de.fhpotsdam.unfolding.Map(this, new Microsoft.AerialProvider()));

5.2 Coordonnées

En vue d'une visualisation du graphe, il faut maintenant faire la différence entre deux types de coordonnées :

- les coordonnées réelles de la carte (lat, lon)
- les coordonnées graphiques de l'écran de visualisation processing (x, y)

Qui ne sont pas situées sur le même plan :



Ainsi il faut effectuer des conversions lorsque l'on veut représenter par un point (x, y) à l'écran un nœud (lat, lon) du graphe (et inversement).

Créer une localisation (x, y) depuis un couple (lat, lon) :

```
Location l1 = new Location(lat, lon);
float xy[] = Application.session.getMap().getScreenPositionFromLocation(l1);
```

les coordonnées écran seront (xy[0], xy[1])

Obtenir un couple (lat, lon) depuis une localisation (x, y) :

```
Location l2 = Application.session.getMap().getLocationFromScreenPosition(x, y);
```

les coordonnées carte seront (l2.getLat(), l2.getLon())

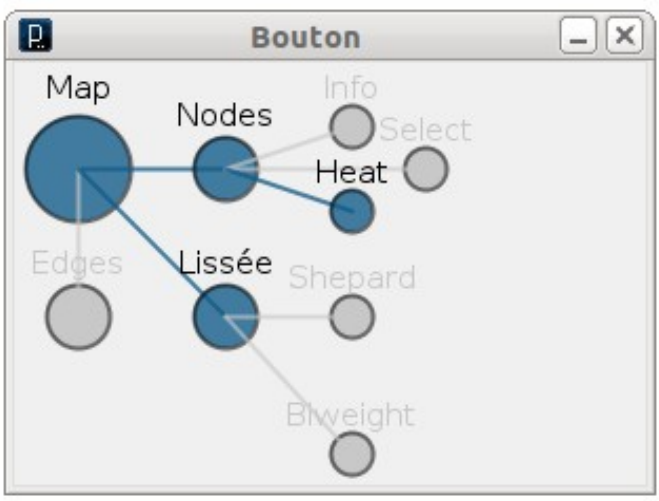
5.3 Interactions

La carte Unfolding autorise la navigation dans l'espace géographique (flèches directionnelles, souris dragged) et le zoom (souris rolled, + et -).
Pour améliorer l'immersion le prototype propose une échelle en mètres (Affichage.afficheEchelle()) affichant la distance entre deux points fixes (lat, lon) de la carte.
Pour ce faire la fonction Bibliotheque.distFrom(lat1,lon1,lat2,lon2) utilise la méthode du Great Circle Distance pour trouver le plus court chemin entre deux points d'une sphère.

6. Menu et arborescence des visualisations

6.1 Principe

Le prototype offre à l'utilisateur la possibilité de voir et d'explorer le graphe de flux. Il lui permet de choisir entre différents modes de visualisations , entres différentes approches d'explorations, de les combiner, de les confronter. Pour le guider nous avons mis en place un menu arborescent, exploratoire, reliant thématiquement les visualisations :



A chaque niveau, le menu révèle les visualisations suivantes encourageant ainsi l'exploration, tout en gardant une trace du cheminement intellectuel, du parcours des calques.

6.2 Mise en place pratique

Le menu est constitué d'objets BoutonMenu (classe *BoutonMenu.java*). Hormis le bouton principal « Map » chaque bouton est relié à un parent et peu posséder des enfants. Lorsque qu'un bouton est cliqué son statut change ('normal' → 'selected' ou 'selected' → 'normal') et modifie celui de ses descendants en conséquence.
Activer ou désactiver un bouton du menu revient à activer ou non une variable booléenne associée (classe *Session.java*), suivant l'état de ces variables la fonction draw() invoquera ou non la classe correspondant au mode de visualisation sélectionné par l'utilisateur.

Ajouter un nouveau bouton :

les BoutonMenu sont rangés dans une ArrayList Boutons (*classe Session.java*), il faut donc ajouter le nouveau BoutonMenu à la liste dans setup() comme tel :

- s'il n'a pas d'enfant :

```
Boutons().add(new BoutonMenu(taille, x, y, "Nom Bouton"));
```

- s'il a des enfants, il fait ajouter une liste avec le nom de ceux ci :

```
String [ ] listeEnfant = new String [ ] {"Nom Enfant1", "Nom Enfant2", …};
Boutons().add(new BoutonMenu(taille, x, y, "Nom Bouton", listeEnfant));
```

Il faut ensuite renseigner l'activation ou non des booléens associés au BoutonMenu dans les méthodes constantesOn() et constantesOff(). Si deux modes de visualisations ne peuvent être utilisés en même temps il faut indiquer ces incompatibilités dans la méthode incompatib().

7. Affichage des arcs

arborescence menu : **carte→edge**

7.1 Méthode

Pour chaque arc du graphe de flux (classe *Edge.java*) représenté par une ligne, on détermine 2 paramètres :

- l'épaisseur
- la transparence

L'épaisseur est déterminée en fonction du poids de l'arc via la primitive map() de Processing :

epaisseur = map(poids, poidsMin, poidsMax, 1, 15);

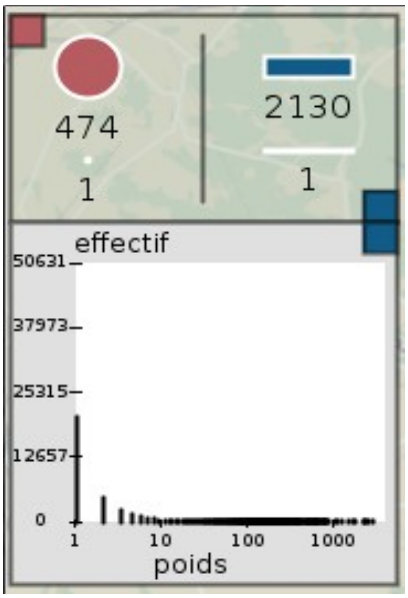
La transparence est déterminée de la même manière :

transparence = map(poids, poidsMin, poidsMax, 15, 255);

note : On applique ici deux filtres successifs, le premier n'affiche pas les arcs dont le poids vaut 0 ou 1, le second n'affiche que les arcs dont l'une au moins des extrémités se trouve dans la zone visible à l'écran.

7.2 Légende et Distribution

Une légende (fonction Affichage.afficheLegendeNodeEdge()) indique les valeurs min et max des arcs. En cliquant sur le carré bleu on affiche la distribution des arcs en fonction du poids :



Lorsque nous appliquerons des transformations au poids des arcs ou des nœuds, les min et max seront recalculés en fonction

L'axe du poids en abscisse est présenté suivant une échelle logarithmique

Pour masquer la distribution il suffit de cliquer à nouveau sur le carré bleu. De manière générale tous les affichages de distributions fonctionnent sur le même principe :

Initialisation :

dans setup() la fonction Bibliotheque.effectif() détermine l'effectif de la classe de poids la plus importante pour les nœuds et les arcs en renseignant les variables nodeEffMax et edgeEffMax (classe *Session.java*)

Appel :

Affichage.java→afficheLegende( ) →afficheDistribution( )

Fonction afficheDistribution() :

On édite successivement les graduations de l'axe des abscisses (échelle logarithmique allant de 0 à edge/nodeMax) et de l'axe des ordonnées (échelle allant de 0 à node/edgeEffMax)
Puis on dessine une ligne correspondant à la hauteur de l'effectif de chaque classe de poids via la fonction drawStick()

note : Toutes les fenêtres de distributions et de légendes sont encrées (coin supérieure gauche de la fenêtre) par rapport aux paramètres de taille de la fenêtre générale Processing (width et height). Ainsi si cette fenêtre est agrandie par l'utilisateur (fonction size(width, height)) les proportions seront conservées. En revanche la taille d'une fenêtre de distribution ou de légende ne change pas, il convient donc de ne pas réduire à outrance la taille de la fenêtre Processing.

7bis . Affichage des nœuds

arborescence menu : carte→node

Concrètement il n'y a pas de grande différence entre le mécanisme d'affichage des nœuds et des arcs. Comme nous voulons représenter les nœuds par des cercles, nous ne déterminons qu'un paramètre : le rayon
Il est fonction du degré du nœud et comme la transparence et l'épaisseur il se calcule via la primitive map() :

```
radius = map(degree, NodeMin, NodeMax, 1, 15);
```

Les fonctionnements de la légende et des répartitions sont similaires à ceux des arcs.

8. Transformation logarithmique et BoxCox

8.1 Logarithme

arborescence menu : carte→edge→log

Le prototype permet la transformation des données en vue de normaliser la répartition du poids des arcs. Pour ce faire il suffit de modifier dans l'affichage des arcs le calcul des paramètres d'épaisseur et de transparence, par exemple :

epaisseur = map(log(poids), log(poidsMin), log(poidsMax), 1, 15);

Grâce au jeu des booléens (classe *Session.java*) la transformation logarithmique est répercutée sur la légende, les distributions.

8.1 BoxCox

arborescence menu : carte→edge→box cox

carte→node→box cox node

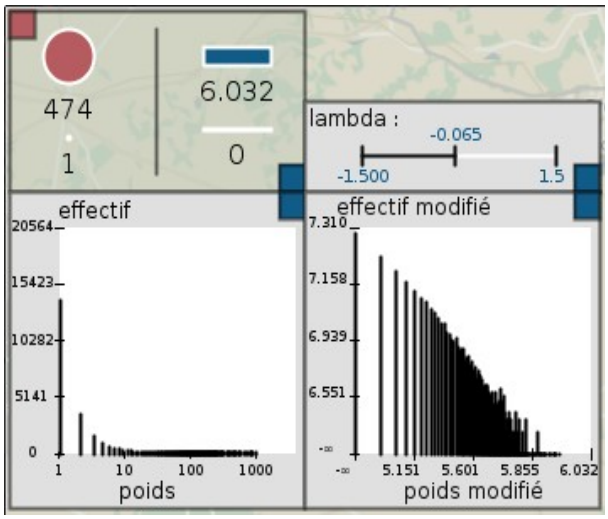
Cette transformation BoxCox applique à une valeur x la fonction suivante :

$$f(x) = \log(x) \quad \text{si } \lambda = 0$$

$$f(x) = \frac{(x^{\lambda} - 1)}{\lambda} \quad \text{sinon}$$

Où λ est fixé par l'utilisateur

Comme pour le logarithme on vient directement modifier le calcul de l'épaisseur et de la transparence en appelant la fonction Bibliotheque.CoxBox(). Là encore la transformation est répercutée sur la légende et le prototype permet de confronter la distribution avant et après la transformation :



L'utilisateur fixe à la volée, via le curseur, la valeur de λ

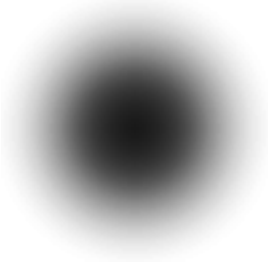
Et effectue des comparaisons avant et après normalisation

note : Cette transformation est disponible pour les nœuds également, elle modifie la répartition du degré de ceux ci.

9. HeatMap

arborescence menu : carte→node→HeatMap

La méthode HeatMap permet au prototype de fixer un paramètre (le degré des nœuds) de le transformer en variable 2D et d'en estimer la valeur via un dégradé de couleurs (blanc au rouge).
Concrètement, chaque nœud sera remplacé par l'image d'un point au contour lissé :



la fonction HeatMap.drawHeatMap( ) modifiera la couleur de chacun des pixels de l'image (dégradé radial) en fonction de l'intensité du degré associé.

Initialisation :

dans setup( ) il faut charger l'image comme telle
Application.session.setMyPoints(loadImage("./Ressources/ppp.png"));

Appel :

Node.afficheNode() → pour chaque nœud → HeatMap.drawHeatMap(rayon)

Fonction drawHeatMap(rayon) :

déterminer un pourcentage associé à la valeur du degré du nœuds
percent = norm(poids, NodeMin, NodeMax);

transformer ce pourcentage en couleur
seuil = lerpColor(fromWhite, toRed, percent);

attribuer une taille à l'image proportionnelle au rayon du nœud
(passé en paramètre à la fonction)

l'image

pour chaque pixel calculer la distance entre lui et le centre de l'image
distance = dist (xCentre, yCentre, xPix, yPix);

lui attribuer une couleur du dégradé proportionnelle à cette distance
percent2 = norm(distance, 0, Application.session.getMyPoints().width/2);
couleur = lerpColor(seuil, from, percent2);

les pixels

La superposition finale de toutes les images de points flous à la place des nœuds fera ressortir des zones de fortes intensités, des zones de chaleurs. Comme nous travaillons sur le degrés des nœuds la HeatMap révèlera les zones de **forte centralité** au sein du graphe de flux.

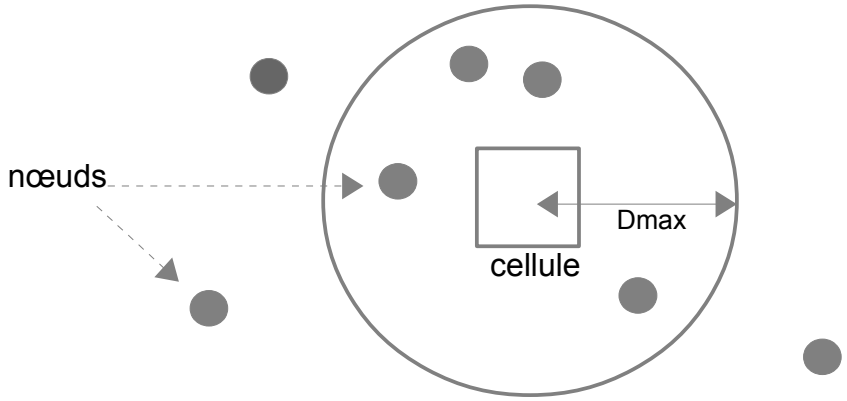
10. Carte Lissée

arborescence menu : carte→lissée→Biweight

→Shepard

10.1 Principe

Comme une HeatMap, une carte lissée fixe un paramètre et le transforme en variable 2D dans le but de créer des zones de classes. Hors cette fois ci, nous ne partons plus des nœuds mais du territoire.
L'écran Processing est désormais considéré comme une grille régulière où chaque cellule se voit attribuer un score. Ce score est calculé via une moyenne pondérée du poids des nœuds situés à une certaine distance Dmax (classe *Session.java*) de la cellule. Le résultat est enfin traduit en couleur.



10.2 Structure de données en entrée

La carte lissée travaille sur le poids, le degré ou tout autre attribut des nœuds d'un graphe. Ce graphe peut être :

- le graphe déjà chargé dans le prototype :

  dans Smooth.miseAjourCarteLissee() remplacer tous les NBRoamBTSMoy par des MatNode

- un graphe sous format CSV extérieur (cas actuel cette étape n'est pas à refaire) :

  dans setup() parser le CSV avec la fonction Bibliotheque.readData() à qui l'on doit indiquer le chemin vers le CSV. La fonction range les données dans la matrice NBRoamBTSMoy[ ][ ] (classe *Session.java*).

note : Dans Smooth.miseAJourCarteLissee() si l'intervalle de temps du csv est inférieur à l'intervalle de temps du prototype (par exemple le csv découpé en tranche d'1h et le prototype en 4h) alors l'instruction

« **int nbInterval = 24/Temps.getHourCount();** » + « **Application.session.getIndex()*nbInterval + 3** »

utilisera le premier intervalle csv contenu dans l'intervalle du prototype.

10.3 Méthode

Appel :

draw( ) → **Smooth.initBuff1( )**
→ **Affichage.afficheLegendeLissee( )**

Fonction initBuff1( ) :

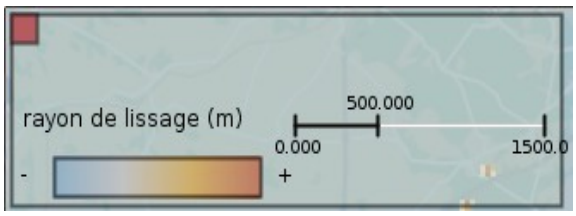
calculer et afficher une carte lissée est très couteux, les fonctions de calculs que nous verrons par la suite ne doivent être appelées que si les événements suivants sont détectés :

- premier affichage
- zoom
- déplacement de la carte
- changement d'intervalle
- changement du rayon de lissage (Biweight) et ou du facteur de puissance (Shepard)

Après la détection de l'un ou l'autre de ces événements la fonction initBuff1() est appelée et remet à zéro toutes les matrices utilisées par le moteur de lissage ainsi que la variable booléenne **init** (classe *Smooth.java*). Si cette variable est false alors le moteur (fonction Smooth.lissage()) relance le calcul de lissage.

Fonction afficheLegendeLissee( ) :

dans un premier temps on affiche la légende correspondant à la visualisation :



L'utilisateur définit le rayon de lissage grâce au curseur

une fois le rayon de lissage Dmax définit il faut le convertir. La distance choisie par l'utilisateur est en mètre, or notre grille appartient au plan de la fenêtre Processing, les calculs de distances doivent donc être en pixel. La conversion est effectuée via la fonction Bibliotheque.meter2Pixel().

Il faut limiter au maximum le temps de calcul, pour ce faire nous choisissons de ne pas prendre en compte dans les calculs suivants les nœuds non visibles à l'écran. La fonction Smooth.miseAJourMatriceLissee() définit une zone 1,5 fois plus grande que l'écran Processing. Tous les nœuds se trouvant dans cette zone sont stockés dans une matrice NodePourLissage[3][ ] (classe Session.java) qui conservera pour chaque nœud les 3 informations suivantes :

- latitude
- longitude
- valeur du paramètre à lisser

enfin nous appelons la fonction de lissage Smooth.lissage()

Fonction lissage( ) :

Cette fonction effectue un choix. Si la variable booléenne init est fausse alors elle appelle la fonction de calcul des scores Smooth.calculScore1() puis passe à la colorisation des cellules, sinon (il n'y a pas lieu de

recalculer) on passe directement à la colorisation.

Fonction calculScore1( ) :

L'écran Processing est divisé en cellules de 3*3 pixels
pour chaque cellule on détermine le score de lissage suivant la méthode choisie par l'utilisateur (Biweight ou Shepard) puis le score est les coordonnées de la cellule sont stockés dans un buffeur :

coordonnées : buff1[2][...]
score : buff2Score[…] } **Buffeur de lissage**

fin calculscore1()

Une couleur est alors attribuée à chaque cellule du buffeur de la sorte :

percent = norm(buff1Score[…], 1, valeur paramètre lissage max)
si l'on travaille sur le poids le % se fera entre 1 et le poids Max

si percent > 0.16 alors classe couleur 4
si percent > 0.12 alors classe couleur 3
si percent > 0.07 alors classe couleur 2
sinon classe couleur 1
les seuils sont arbitraires et doivent être définis ad hoc par l'utilisateur

10.4 Moyenne Pondérée, Biweight() ou Shepard()

Une moyenne pondérée est définie comme telle :

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$ où w est un facteur de pondération

Le facteur de pondération dans notre cas doit être fonction de la distance entre la cellule et les nœuds : la rayon de lissage Dmax. Plus le nœud est éloigné, moins il affecte la cellule.
Il existe nombre de fonctions de pondération, le prototype en implémente 2 aux choix depuis le menu :

- Methode de Biweight (fonction Smooth.Biweight())
- Methode Shepard (fonction Smooth.Shepard())

Ces fonctions sont appelées depuis la fonction calculScore1().

Methode de Biweight :

pour une cellule A donnée, pour tout nœud Ni situé à une distance d < Dmax de la cellule et ayant un poids Pi, on définit le facteur de pondération Wi comme tel :

$$w_i = \left(1 - \left(\frac{d}{Dmax}\right)^2\right)^2$$

Methode de Shepard :

pour une cellule A donnée, pour tout nœud Ni situé à une distance d < Dmax de la cellule et ayant un poids Pi, on définit le facteur de pondération Wi comme tel :

$w_i = \frac{1}{d^p}$ où P le facteur de puissance est définit par l'utilisateur via un curseur dans la légende

11. Oursins

11.1 De la sélection aux oursins

arborescence menu : carte→node→select

Le prototype exploite le caractère orienté du graphe (les arcs ont une source et une cible) et permet d'isoler un nœud et les arcs qui y arrivent ou en partent. L'utilisateur sélectionne un nœud en cliquant dessus et visualise l'ensemble complexe [nœud ; arcs entrant ; arcs sortant] tel que :

- entrant : ligne bleue
- sortant : ligne rouge

Appel :

Node.afficheNode() → si nœud cliqué → Affichage.selection(x, y, rayon)

Fonction Affichage.selection( ) :

On interroge chaque arcs auquel on confronte les coordonnées du nœud, s'il y a correspondance avec la source ou la cible de l'arc alors on l'affiche en choisissant la couleur adaptée

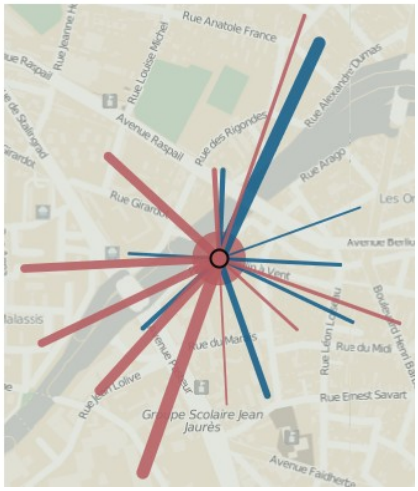
11.2 Simplifier l'ensemble [nœud ; arcs entrant ; arcs sortant]

arborescence menu : carte→node→select→oursin

La forme graphique visualisée à l'étape précédente fait sens une fois mise en relation avec le territoire. Il serait intéressant la confronter avec celles d'autres nœuds, pour ce faire le prototype va simplifier cette forme pour obtenir un objet normalisé suivant le protocole ci dessous :

- sélectionner un ensemble [nœud ; arcs entrant ; arc sortant]
- définir un cadrant autour du nœud sélectionné
- dans chaque zone du cadrant déterminer un arc entrant/sortant moyen de tous les arcs entrant/sortant du cadrant, de longueur moyenne et de poids somme

L'objet obtenu est appelé « oursin »



11.2 Mise en place pratique

Appel :

draw() → si nœud selectionné → Affichage.selectionOursins(xNoeud, yNoeud)

Fonction Affichage.selectionOursins( ) :

connaissant les coordonnées du nœud à transformer, on confronte celles ci à l'ensemble des arcs, s'il y a concordance avec la source/cible d'un arc alors toutes les informations de ce dernier sont stocker dans la matrice (classe *Session.java*) correspondante :

- **Entrant[5][...]**
- **Sortant[5][...]**

} **L'ensemble des arcs du nœud**

note : on associe à chaque arc un angle par rapport à un axe vertical passant par le centre du nœud en utilisant la primitive Processing trigonométrique atan2()

on définit 2 nouveaux tableaux :

- **PointsCardinauxEntrant[32]**
- **PointsCardinauxSortant[32]**

} **Les futurs arcs moyens de l'oursin**

on appelle 2 fois de suite (côté entrant puis côté sortant) la fonction de division de l'espace en cadrant Bibliotheque.remplissagePointsCardinaux()

Fonction remplissagePointsCardinaux( ) :

L'espace autour du nœud est vu comme un cadrant divisé arbitrairement en 16 zones d'angles PI/8
Grâce à leur angle les arcs de la matrice Entrant/Sortant[ ][ ] sont répartis dans 16 tableaux :

- Est[ ]
- nordEstEst[ ]
- (…)

Un tableau peut très bien ne pas contenir un seul arc.

dans l'ordre Est →Sud→Ouest→Nord on applique à chacun de ces tableaux la fonction de calcul Bibliotheque.traitementPointsCardinaux()

Fonction traitementPointsCardinaux( ) :

On calcule la moyenne des longueurs et la somme des poids de chacun des arcs présent dans le tableau. La moyenne et la somme sont passées au logarithme (normalisation) et dans une primitive processing map() pour obtenir des valeurs directement utilisable par l'affichage telles que :

```
moy = map(log(moy), 0, log(DistMax), 0, 1500);
sum = map(log(sum), 0, log(EdgeMax), 0.5, 15);
```

ces valeurs sont placées dans PointsCardinauxEntrantSortant[ ] :

```
pointsCardinauxE/S[i] = sum;
pointsCardinauxE/S[i + 1] = moy;
```

Dès lors, pointsCardinauxEntrant[0] correspond au poids de l'arc moyen Entrant de la division Est et

pointsCardinauxEntrant[1] représentera sa longueur, ainsi de suite dans l'ordre Est →Sud→Ouest→Nord jusqu'à pointsCardinauxEntrant[31].

Retour à Affichage.selectionOursins( ) :

pointsCardinauxE/S[ ] est désormais renseigné on peut donc créer un objet Oursin (classe *Oursin.java*) que l'on rangera dans un ArrayList Oursins (classe *Session.java*) comme tel :

Oursins.add(new Oursin(pointsCardinauxEntrant, pointsCardinauxSortant, x, y, lat, lon));

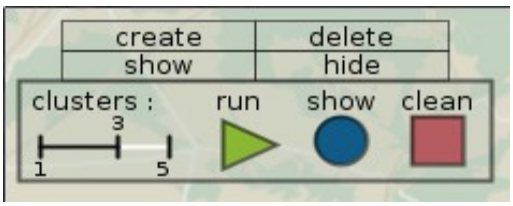
Objet Oursin :

Un objet Oursin est une structure contenant l'ensemble des informations nécessaires à l'affichage et au questionnement d'un oursin unique (couple lat, lon). Un oursin est, autrement dit, un vecteur de 2 fois 32 paramètres :

(lat, lon)
branchesEntrantes[32]
branchesSortantes[32]
statut
SommeEntrant
SommeSortant

} **Objet oursin unique défini à un instant donné**

La méthode pressed() définit le statut (visible ou non) de l'Oursin. Les méthodes draw() et drawArc() se chargent de l'affichage des branches de l'oursin.
Note : le poids de chaque branche est sommé dans SommeE/S, si SommeE > SommeS alors un cercle bleu est dessiné au centre de l'oursin (et inversement). Ainsi l'on peut savoir si l'oursin est plus Entrant ou plus Sortant.



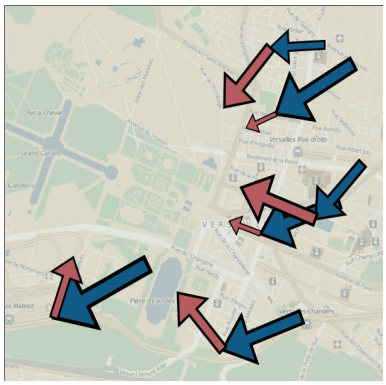
Côté utilisateur il suffit de cliquer sur un nœud pour le transformer en oursin, individu par individu.
Il peut aussi créer un ensemble d'oursin d'un seul coup en cliquant sur « create » dans la légende tel que tout nœud visible dans la zone de l'écran Processing et dont le degré est >30 sera transformé en oursin (fonction Bibliotheque.creerOursinsVue()).
En cliquant sur « delete » on supprime tous les oursins de la ArrayList Oursins(fonction Bibliotheque.effacerOursins()). A chaque changement d'intervalle, l'ensemble des oursins déjà crées (contenus dans la ArrayList Oursins) sont recalculés (fonction Bibliotheque.miseAJourOursins()).

12. Flèches

arborescence menu : carte→lissée→Arrow

12.1 Principe

Riche, de par les informations qu'il contient, un oursin reste néanmoins relativement peu interprétable visuellement, immédiatement. Le Prototype propose une forme encore plus simple : les flèches.
Concrètement une flèche est le vecteur moyen des branches d'un oursin, sa taille représentant le poids total des flux orientés. Ainsi après transformation un oursin se verra remplacé par deux flèches : une entrante et une sortante.



Utiliser le vecteur moyen révèle la tendance plus ou moins anisotrope de l'oursin dont sont issues les flèches. Un oursin central, distribuant ses arcs dans toutes les directions de manière équilibré sera représenté par deux flèches de même taille. Au contraire, un oursin orienté, témoin d'une tendance de fort déplacement se verra représenter par une nette différence de taille des deux flèches.

12.2 Processus préalable de création

Calculer et afficher les flèches à la volée est couteux et diminuerait les performance du prototype. Nous allons donc créer une fois pour toute l'ensemble des flèches correspondant à notre Graphe de mobilité sur toute la période d'étude suivant ce process :

Retour à l'Objet Oursin :

Soit un repère $(O, \vec{x}, \vec{y})$ où O, l'origine, n'est autre que le centre de l'oursin.
Toute branche i de l'oursin peut être vu comme un vecteur $\vec{OA}_i$ où $O(0,0)$ et $A_i(x_i, y_i)$
Une flèche est le vecteur moyen $\vec{OB}$ des n branches tel que $O(0,0)$ et

$$B\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right)$$

Ainsi lors de l'affichage de chaque branche (méthode drawArc()) on détermine les sommes :

```
xMoyEntrant = xMoyEntrant + Bibliotheque.meter2Pixel(rayon) * cos(angle);
yMoyEntrant = yMoyEntrant + Bibliotheque.meter2Pixel(rayon) * sin(angle);
```

Puis dans la méthode Oursin.draw()

```
if (Application.session.isArrow()){
. . .
drawVectMoy(xy[0], xy[1], xMoyEntrant / nEntrant + xy[0], yMoyEntrant / nEntrant + xy[1], true);
. . .
drawVectMoy(xy[0], xy[1], xMoySortant / nSortant + xy[0], yMoySortant / nSortant + xy[1], false);
}
```

pour appeler la méthode de création de flèches drawVectMoy() qui créera un objet Arrow (classe Arrow.java) tel que :

```
arrow = new Arrow(lat_Origine, lon_Origine, angle, lat_extrem, lon_extrem, sens);
Application.session.arrowsIN.add(arrow);
```

Désormais pour tout oursin crée, un objet flèche entrante et un objet flèche sortante sont ajoutés aux ArrayList ArrowsIN/OUT (classe Session.java).

Créer l'ensemble des flèches :

Pour un intervalle donné :

- dans Bibliotheque.java décommenter la première version de creerArrow() et commenter la 2nd
- dézoomer pour englober toute la zone d'étude
- créer l'ensemble des oursins de la zone (bouton « create »)

toutes les flèches sont automatiquement créées.

Sauvegarder les flèches dans un CSV :

Avant de passer à l'intervalle suivant il faut sauvegarder les flèches contenues dans ArrowsIN/OUT
Pour cela associer la fonction Bibliotheque.writeArrowData( ) à une touche du clavier dans TouristFlow.keyRelease() comme tel :

```
if ( key == 'a' ) {
        Bibliotheque.writeArrowData( ) ;
}
```

Cette fonction génèrera un csv (contenant toutes les informations nécessaires à la création des flèches).

Répéter la création des nœuds et l'appel de writeArrowData() pour chaque intervalle de temps.
Fermer le prototype et penser à commenter à nouveau la première version de creerArrow() et décommenter la suivante. Si l'on change de graphe en entrée il convient de recalculer les flèches.

12.3 Manipulation des flèches

Nous venons de créer et de sauvegarder les flèches dans des csv, il ne sera maintenant plus besoin de repasser par les étapes Oursin → Calcul des flèches → affichage pour les voir, nous passerons directement à l'affichage.

Initialisation :

dans setup () il faut renseigner le chemin vers les CSV crées plus tôt :

```
Application.session.setReferencesArrows(new String[6]);
Application.session.setReferencesArrows(0, "./Ressources/Anisotropie_31 March 2009 00h.csv");
Application.session.setReferencesArrows(1, "./Ressources/Anisotropie_31 March 2009 04h.csv");
(. . .)
```

Appel :

draw() → Bibliotheque.creerArrow()

Fonction creerArrow() :

la fonction récupère le chemin du CSV correspondant à l'Index courant (classe Session.java)
elle le parse et crée les objets Arrow :

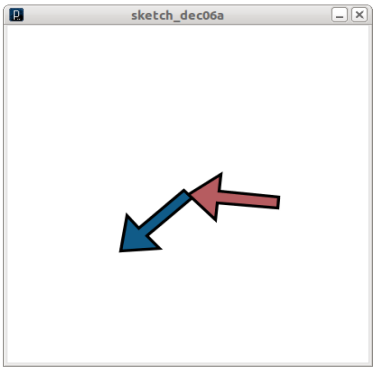
```
Arrow a = new Arrow(Float.parseFloat(mots[0]), Float.parseFloat(mots[1]) . . . );

( . . . )

 if (Boolean.parseBoolean(mots[7])) {
Application.session.arrowsIN.add(a);
 } else {
Application.session.arrowsOUT.add(a);
}
```

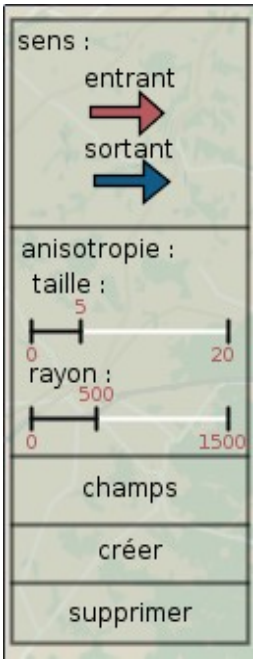
Objet Arrow :

Un objet Arrow (classe *Arrow.java*) est une structure représentant une flèche.
Il s'agit avant tout de créer un objet graphique via la méthode Arrow.update().



La taille de la flèche est fonction du poids des flux allant dans un même sens et est définie par la méthode Arrow.calculeSize() :

```
size = dist(xy1[0], xy1[1], xy[0], xy[1]);
size = map(size, 0, 5, 0, 5);
```



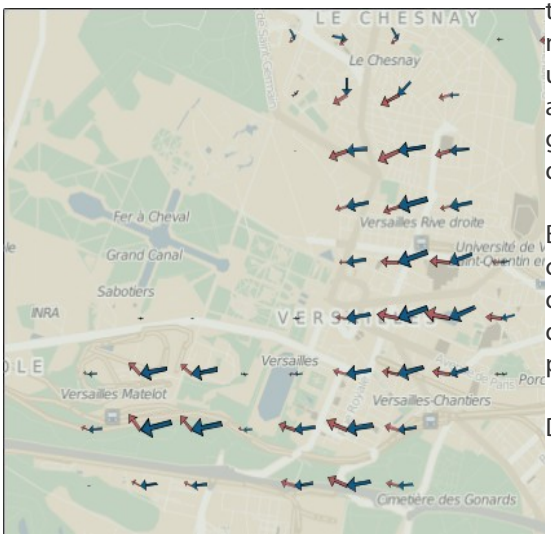
Note : une fois le chemins vers les csv renseigné il suffit à l'utilisateur dans la légende Arrow de cliquer sur « créer » pour créer l'ensemble des flèches de la carte (fonction Bibliothèque.creerArrow( )). En cliquant sur « supprimer » on efface toutes les flèches des ArrayList ArrowsIN/OUT. A chaque changement d'intervalle on invoque à nouveau créerArrow().

13. Champ de flèches

arborescence menu : carte→lissée→Arrow

13.1 Principe

Cette visualisation est le prolongement de la précédente. Comme pour la carte lissée, le prototype permet de transformer une information ponctuelle (l'anisotropie d'un nœud) en information de surface, un champ de flèche basé sur un découpage du territoire en grille. Ainsi les tendances anisotropes (fortement orientées) de certaines parties du graphe seront vues sur le territoire comme des zones de fortes directions du flux.



Basé sur le même moteur que la carte lissée, on cherchera pour chaque cellule de la grille à déterminer une moyenne pondérée des coordonnées et de la taille de toute flèche située à une distance inférieure au rayon de lissage Dmax. La moyenne sera pondérée par la Méthode de Biweight.

Dans chaque cellule, une nouvelle flèche sera redessinée.

13.2 Mise en place

Déjà vue lors du chapitre sur le lissage nous n'expliquerons que l'enchainement des fonctions

initialisation :

Smooth.initBuff1()

Appel :

Draw() → **Affichage.afficheArrow() :** afficher la légende, invoquer le lissage

→ **Smooth.setGrille(Bibliotheque.getGrille()) :** créer la grille sur le territoire
→ **Smooth.lissageArrow( )**

→ **si (!init) :** s'il y a eu un événement particulier
→ **Smooth.calculscore2() :** pour chaque cellule
→ **Smooth.calculLissaegArrow() :** calculer le score de lissage depuis les flèches crées plus tôt

→ dans chaque cellule visible à l'écran dessiner une flèche lissée

```
Arrow a = new Arrow(xOrigine, yOrigine, angle, rayon, sens);
a.updateLight();
```

note : les objets Arrow ici crées sont légèrement différents de ceux du chapitre précédent, ayant une durée de vie réduite, ils ne sont définis que sur le plan Processing (coordonnées x, y de l'origine en pixels et non en lat, lon) et ont pour attribut un angle et une taille.

13.3 Précisions sur la grille

En vue d'une visualisation ultérieure nous avons fait évoluer le système de grille.

Fonction Bibliotheque.getGrille(pixel) :

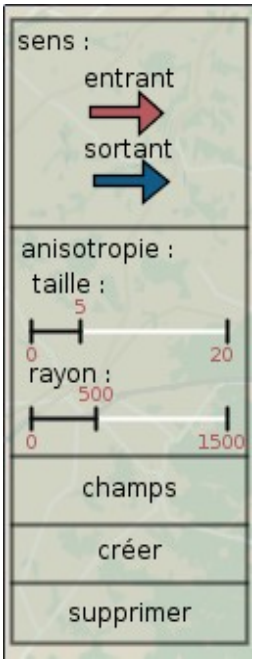
La taille de la cellule en pixel est passée en paramètre. La fonction traduit cette taille en écart de latitude et longitude pas_X et pas_Y, ainsi, partant de la latitudeMin et de la longitudeMin (des nœuds) on peut définir la grille comme une matrice :

```
for (float i = latitudeMax; i > latitudeMin; i =  i - pas_X) {
        for (float j = longitudeMax; j > longitudeMin; j = j - pas_Y) {
                mat[0][cpt] = i;
                mat[1][cpt] = j;
                cpt ++ ;
        }
}
```

Dans Affichage.afficheArrow() à chaque changement d'altitude (niveau de zoom) Bibliotheque.getGrille() est invoquée. La matrice résultante est stockée localement dans la classe Smooth.java.

```
Smooth.setGrille(Bibliotheque.getGrille(40));
```

L'écartement en pixel restant fixe, plus l'on zoomera plus la grille sera précise.



Note : Il faut d'abord créer les flèches (« créer ») puis les lisser et générer le champs (« champs »). Pour revenir aux flèches simples il faut cliquer à nouveau sur « champs ».
Le lissage est modifiable via le curseur tout comme la taille globale des flèches du champs. Concrètement, plus nous avons une vue d'ensemble, moins la grille est précise, plus il faut augmenter le rayon de lissage pour obtenir des tendances générales et inversement pour une vue en détaillée. De même pour des vues larges il convient d'augmenter la taille globale des flèches (on conserve le rapport de grandeur) en agissant dans la fonction smooth.updateLight() sur le paramètre ArrowsMax :

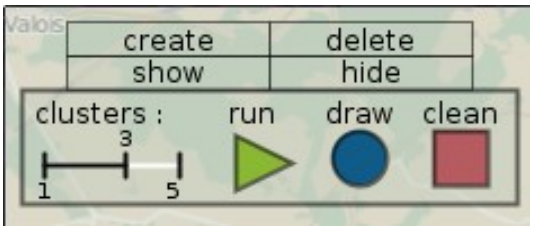
```
taille = map(poids, 0, 200, 0, 4);
```

14. Clustering

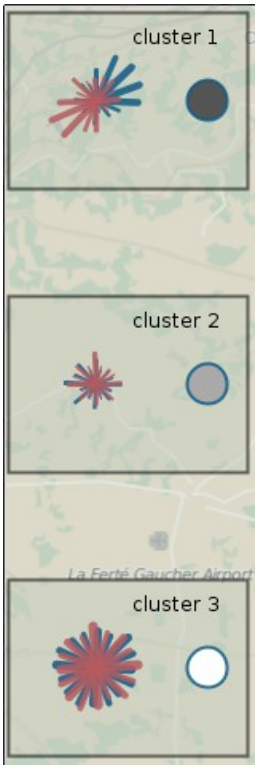
arborescence menu : carte→node→select→oursin

14.1 Principe

Créer un oursin seul et observer sa forme par rapport au territoire est intéressant mais apporte des connaissances limitées. Il serait utile de le comparer avec d'autres individus et, de par leurs différences ou leurs ressemblances, en tirer des connaissances plus riches.
Le prototype implémente un système de clustering rangeant les oursins dans des classes. Il est alors plus facile de comparer chaque classe entre elles, la forme de leur oursin type, son rôle, son emplacement sur le territoire à un instant donné.



Tout d'abord, il faut sélectionner sa zone de travaille via l'écran Processing, puis créer les oursins (« create »). L'utilisateur choisit le nombre de classes (curseur « cluster ») et lance le clustering « run ».
Pour afficher les résultats il faut cliquer sur « draw ». Dans un souci de clarté on peut cacher les oursins via « hide » et inversement via « show ». Si l'on veut relancer le clustering il faut nettoyer impérativement les buffeurs en cliquant sur « clean ».



La méthode de clustering choisie est un algorithme de type K-means qui a l'avantage d'être rapidement implémenté et permet de prédéterminer le nombre final de clusters. En revanche, il a l'inconvénient de partir d'une initialisation random. On ne peut pas prédire la forme des clusters à l'avance. Leur composition est toujours légèrement variable. Néanmoins une répétition de l'algorithme permettrait de le faire converger vers une réponse stable.
Les classes résultats sont présentées comme telles, avec d'un côté l'oursin type de la classe (le centroïde final) et de l'autre le point de couleur superposé sur chaque oursin de la classe.

En cliquant sur la représentation de la classe on cache momentanément à l'écran tous les individus de celle ci, permettant d'étudier une ou plusieurs classes en particulier

14.2 Algorithme de K-means

Initialisation :

- déterminer le nombre de clusteurs n
- choisir n oursins au hasard parmis la population d'étude : n centroïdes

Création des clusteurs :

- pour chaque centroïde créer une table
- pour chaque oursin de la population calculer la mesure de distance entre lui et chacun des centroïdes
    - **note :** la mesure de distance est la question à laquelle on veut répondre, dans notre cas nous calculerons une distance euclidienne sur l'ensemble des paramètres de l'oursin, pour déterminer une ressemblance de forme générale. Un oursin x étant un vecteur de 64 paramètres $(x_{0,}x_1 \ldots, x_{64})$ on peut dire que :

$$\forall (x, y), d(x, y) = \sqrt{\sum_{i=i}^{n} |x_i - y_i|^2}$$

    D'autres questions plus spécifiques peuvent être ici posées (sur le poids, la longueur, l'angle...).
- Ranger l'oursin dans la table correspondant au centroïde avec lequel il est le plus proche (distance minimum)

Détermination des nouveaux centroïdes :

- pour chaque table déterminer son oursin moyen (moyenne de chaque paramètre)
- l'oursin moyen est le nouveau centroide de la table

Boucle et Condition d'arrêt :

- Entrer dans la boucle
    - répéter les étapes « Création des clusteurs » et « Détermination des nouveaux centroïdes »
    - comparer les n tables de l'itération précédente avec les n tables nouvellement crées, sont elles identiques ?
        - Si non, on boucle
        - Si oui, les clusters sont stabilisés, sortir de la boucle
- Fin de l'algorithme, chaque table représente une classe

14.3 Implémentation

Par défaut le nombre de clusters n vaut 3.

draw() → si Run et Oursins non vide :

→ **Kmeans.kmeansInit() :**

soit les arrayList Centroïdes, OldCentroïdes, Clusters, OldClusters

OldCentroides
Clusters
OldClusters
} **Initialisés comme {}**

Centroides ← **Kmeans.matriceInit() :** n objets oursins pris au hasard dans Oursins

Bloc 1

pour tout oursin de Oursins (classe session.java)
pour tout centroide de Centroides
distance ← **Kmeans.distance() :** calcul de la distance euclidienne entre 2 vecteurs de 64 paramètres
fin pour

index ← **minDistance() :** numéro de cluster de la plus petite distance
Clusters.add(index, oursin)
un oursin est ici représenté par son couple (lat, lon) unique, un clusters est donc un tableau de couples (lat, lon)
fin pour

Boucle

do
→ **Kmeans.newCentroides() :** mémoriser Centroides dans OldCentroides
effacer Centroides
calculer le nouveau centroide de chaque Clusters
le stocker dans Centroides

→ **Kmeans.clustering() :** mémoriser Clusters dans OldClusters
effacer Clusters
répéter Bloc 1

while non Kmeans.conditionArret() : boolean egalite = true
pour chaque cluster de Clusters
while (egalite = true ou cluster non parcouru)
egalite = member2(oldCluster, couple)
retourner egalite

Ainsi dès qu'il y a égalité entre chaque cluster et son oldCluster (cluster à l'état précédent) on sort de la boucle et on termine KmeansInit()

note : Kmeans.drawCluster() placera un cercle de couleur sur la position de chaque couple (lat, lon) des n clusters. Kmeans.KmeansClean() videra tous les buffeurs et toutes les Arraylists.

15. Normaliser la distribution des nœuds sur le territoire

15.1 Principe

En entré, nous ne maitrisons pas toujours la répartition et la densité géographique des capteurs de flux (les nœuds du graphe). Cela est dans notre cas fonction du réseau de téléphonie. S'appuyer à ce point sur la structure du réseau sans la remettre en cause peut biaiser certains de nos résultats et de nos interprétation. Il est donc plus rigoureux de normaliser spatialement la répartition des nœuds, sur (dans un premier temps) une grille régulière, opérer des agrégations par cellules, des modifications de positions pour n'avoir qu'un seul capteur par cellule. Nous obtiendront ainsi de nouveaux Gexf.

15.2 Mise en place

Soit le process suivant :

Fonction Distribution.getGrilleAgreger(mètre) :

Basée sur la fonction Bibliotheque.getGrile(pixel), ici l'on passera en paramètre une taille de cellule en mètre telle que la fonction nous retournera une grille sous forme de matrice.

mat[cpt] = new Location(i, j);

La grille est pour la première fois posée sur le plan de la carte et non de l'écran processing, elle ne change donc pas en fonction du niveau de zoom.

Fonction Distribution.Agreger() :

pour chaque Cellule i :

if (isBTS(mat[i]))
Y a t'il au moins un BTS à l'intérieur ? Si Oui :

//id
Node[0][cpt] = cpt + 1;
//lat
Node[1][cpt] = mat[i].getLat() + X / 2;
créer un nœud en son centre , définir l'id, la latitude et la longitude de ce nœud

Pour chaque Arc :
si (source et cible) dans Cellule i
Cet arc disparaitra, son poids sera transféré sur le poids du nœud qui témoignera d'un mouvement interne à la cellule
Node[3][cpt] = Node[3][cpt] + Edge[4][j];
si (source) dans Cellule i
source ← id nœud
si (cible) dans Cellule i
cible ← id nœud
fin pour
fin pour

retourner les nouvelles matrices Node[ ][ ] et Edge[ ][ ] et les sauvegarder dans des CSV

Fonction writeCSV() :

On va créer un fichier pour les nœuds (id, lat, lon, poids, interne) et un fichier pour les arcs (id source, id cible, poids). Il faudra faire appel aux scripts python CSV_to_GEXF_Aggregation.py ou CSV_to_GEXF_Aggregation_multi_files.py pour générer de nouveaux GEXF.

A chaque intervalle de temps, il faut appeler manuellement la fonction getGrilleAgreger(mètre) puis Agreger() en les assignant à des touches du clavier dans keyReleased() par exemple.