Radial Basis Functions

Ce qu'on doit retenir du cours

En gros un fonction radial ça va servir à quoi ? On va dire par exemple, **"Tiens, au point (1, 1) je veux qu'il y ait une montagne ! Et qu'elle soit haute de 10 !"**
Et la fonction radiale va permettre à ce que les cubes autour forment une montagne, et qu'il n'y ait pas juste un pic quoi.

Mais du coup comment on fait ça ? Parce que bon...

Alors, easy easy, on va tout décrypter ! Chaque point de la map va être calculé à partir des points appelés **de contrôle** pour que tout soit harmonieux et joli ! Pour cela, on a une fonction dans le cours qui est appelée $g(x)$ et qui se note :

$$g(x) = \sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$$

Ouhlala ça parait compliqué... Alors il faut comprendre que cette formule, c'est le cours, on va pas essayer de comprendre pourquoi la somme, pourquoi les omega tout ça. On va juste retenir qu'ici on a des ω, qu'on ne semble pas connaitre, une fonction qui s'appelle Φ et qui prend en paramètre le x qu'on cherche et les points de contrôle (les x_i).
Nous ce qui nous intéresse c'est **Comment coder tout ça ?**
En gros il faut retenir qu'on va avoir :

1. Une liste de points de contrôle
2. Une liste de valeurs associées une à une à ces points
3. Des ω
4. Des Φ

Comment on code tout ça ?

Avec Eigen ! Eigen, pour rappel, ça prend des Matrices et des vecteurs, donc on va essayer de voir ici en quoi c'est pertinent.
Déjà, si tu lis ça j'espère que tu as le cours du prof ouvert à côté sinon : Voilà
En vrai on l'utilise pas tout de suite, mais après c'est utile. Déjà on va commencer doucement.

1. Qu'est-ce qu'on va utiliser dans Eigen ? Qu'est-ce qui est sous quelle forme ?

En fait ce qu'on veut c'est la hauteur de chaque endroit de notre map. Donc on peut interpréter ça comme si on avait une grille qui correspond à tout nos points et que dans chaque case ya écrit la hauteur du point à cette coordonnée. Une grille ? Donc une matrice quoi !
On va donc avoir une fonction qui retournera notre map qui est du type `Eigen::MatrixXd` (notez qu'on prend Xd parce qu'on va lui mettre une grande taille, genre 50 cubes par 50 cubes, et qu'elle va contenir des int, d'où le d).
Ensuite on a nos points de contrôle x_i. On a donc à chaque fois 2 coordonnées (x et z comme on cherche la hauteur y), et on a N points de contrôles. On a donc une matrice qui va être sous la forme

$$\begin{pmatrix} x_1 & z_1 \\ x_2 & z_2 \\ x_3 & z_3 \end{pmatrix}$$

si on a 3 points de contrôle par exemple.
Enfin, on va avoir nos valeurs associées à ces points de contrôle que Mr Nozick a noté u_i. Là il s'agit donc d'un simple vecteur :

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

On a donc notre premier point qui a comme valeur de contrôle u_1, etc...
OK ! Ca faisait beaucoup alors résumé !

Mais avant mini rappel pour Eigen, pour déclarer une matrice, on écrit Eigen::MatrixXd maMatrice(hauteur, largeur) et Eigen::VectorXd monVecteur(taille).

DONC
On a au début 3 variables :

- `Eigen::MatrixXd` **map**(taille, taille) : contient les hauteurs de tous les points de notre carte
- `Eigen::MatrixXf` **ptsDeControle**(nbPoints, 2) : nos points de contrôle
- `Eigen::VectorXf` **valeurs**(nbPoints) : nos valeurs associées aux points

2. Notre première fonction

On va l'appeler `getMap`. Elle retourne notre map, donc notre matrice avec les hauteurs, et elle prend en paramètres les contraintes, donc les **ptsDeControle** et les **valeurs**. On a donc

In [ ]:
```
Eigen::MatrixXd getMap(const Eigen::MatrixXd ptsDeControle, const Eigen
::VectorXd valeurs)
```

Maintenant qu'est-ce qu'on doit faire ? Alors on regarde le cours déjà. On regarde particulièrement bien le **4.**. Il est écrit que pour trouver la hauteur d'un point donc, on doit utiliser des ω et des Φ de bidule. On y va un par un donc :

a. Trouver les ω

Page 3, on explique un truc très simple en fait. Ces ω, c'est des valeurs qui vont être trouvées grâce à nos contraintes ! On remarque d'ailleurs qu'ils sont sous la forme d'un `Eigen::VectorXf`. Il est écrit qu'une matrice composée de $\Phi(|x_i - x_j|)$ multiplié par les ω est égal à nos valeurs u_i.
On va donc créer une fonction que j'ai personnellement appelé `findOmega` qui va donc nous renvoyer les omegas, c'est-à-dire un `Eigen::VectorXf` et qui prend en paramètre nos points de contrôle, et nos valeurs. On a donc :

In [ ]:
```
Eigen::VectorXf findOmega(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs)
```

Maintenant, un tout petit peu de maths : si on note la matrice des Φ mettons, A. On sait que

$$A * \omega_i = u_i \iff \omega_i = A^{-1} * u_i$$

Or on a les u_i, ce sont nos **valeurs**, et les x_i utilisés dans les fonctions Φ sont nos **ptsDeControle**, donc on les a aussi ! Du coup écrivons cette fonction !

On va donc créer une variable A, qui sera une matrice carrée vide. Elle est aussi grande que le nombre de points de contrôle, c'est à dire le nombre de ligne de la matrice (puisque les points de contrôle sont on le rappelle sous la forme :

$$\begin{aligned} point1 &-> (x_1 \quad z_1) \\ point2 &-> (x_2 \quad z_2) \\ point3 &-> (x_3 \quad z_3) \end{aligned}$$

donc une ligne = un point.
On va donc déclarer A comme ceci :

In [ ]:
```
Eigen::MatrixXf A = Eigen::MatrixXf::Zero(ptsDeControle.rows(), ptsDeCo
ntrole.rows());
```

On veut ensuite la remplir avec les $\Phi(|x_i - x_j|)$, où i et j nous permettent en fait de parcourir les index des colonnes et lignes de la matrice ! (voir page 3 du cours toujours) On va donc simplement faire deux boucles **for** et remplir la case avec $\Phi(|x_i - x_j|)$ (pour l'instant on l'écrit juste comme ça, on verra après comment calculer). Ce qui nous donne :

In [ ]:
```
for(int i=0; i<A.rows(); i++){
        for(int j=0; j<A.cols(); j++){
            A(i, j) = phi(valeurAbsolue(xi - xj);
        }
    }
```

Mais en fait c'est quoi `xi` et `xj` ? Bah c'est dans nos points de contrôle ! Donc c'est notre points de contrôle à l'index i et celui à l'index j. Comme on l'a vu, ça veut donc dire à la **ligne** i, et la **ligne** j. On le notera donc `ptsDeControle.row(i)` et `ptsDeControle.row(j)`.

Ensuite qu'est-ce que la valeur absolue de la différence entre deux points ? Bah en fait c'est tout bêtement la norme, et ça Eigen sait le faire ! (On va appeler cette technique

)
Enfin on renvoie donc notre matrice inversée multipliée par nos valeurs.
Donc ouhla c'était compliqué mais on obtient enfin notre première fonction complète qui nous permet d'obtenir les ω !

In [ ]:
```
Eigen::VectorXf findOmega(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs){
    Eigen::MatrixXf A = Eigen::MatrixXf::Zero(ptsDeControle.rows(), pts
DeControle.rows());

    for(int i=0; i<A.rows(); i++){
        for(int j=0; j<A.cols(); j++){
            A(i, j) = phi(
                (ptsDeControle.row(i) - ptsDeControle.row(j)).norm()
                );
        }
    }

    return A.inverse() * valeurs;
}
```

Mais attendez... Il y a toujours cette fonction Φ là ? O.o
Bah c'est notre fonction radiale ! Alors moi j'ai pris pour l'instant, une gaussienne, qui se note $\Phi(a) = e^{-0.2a^2}$.
Donc petite fonction facile à coder, qui prend en paramètre un float et qui renvoie un float.

In [ ]:
```
float phi(const float a){
    return exp(-0.2*a*a);
}
```

b. On revient à notre fonction générale `getMap`

Ca y est ! On a nos ω ! On peut donc écrire dans notre fonction déjà :

In [ ]:
```
Eigen::VectorXf w = findOmega(ptsDeControle, valeurs);
```

On sait que chaque hauteur dans notre map va être définie par

$$g(x) = \sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$$

Donc on va devoir parcourir la map, et pour chaque point, appliquer cette formule. Déjà

Parcourir la map

Easy ! On va créer une matrice (celle qu'on va renvoyer), de la taille de notre map, et on la parcourt avec une double boucle **for** :

In [ ]:
```
Eigen::MatrixXd values = Eigen::MatrixXd::Zero(WORLD_TAILLE, WORLD_TAIL
LE);

    for(int i=0; i<values.rows(); i++){
        for(int j=0; j<values.cols(); j++){
           //Trouver la valeur
        }
    }
```

Maintenant comment on trouver cette valeur ? On regarde le cours ! Ca dit que ça va être $\sum_{i=1}^{k} \omega_i * \Phi\left(|x - x_i|\right)$. Donc on a notre fonction Φ de codée, il nous faut une fonction qu'on va appeler `findValue` qui va nous calculer la valeur à écrire dans la map. Elle prend en paramètre la position dans la map, donc le i et le j, et enfin les points de contrôle (dans la formule, ils sont notés x_i) et nos ω

c. La fonction `findValue`

On sait qu'elle va retourner un int, la hauteur à cet endroit de la carte. On sait aussi qu'elle a en paramètres le i et le j, et enfin les points de contrôle et nos ω. On obtient donc:

In [ ]:
```
int findValue(const int i, const int j, const Eigen::MatrixXf ptsDeCont
role, const Eigen::VectorXf w)
```

Dans la formule, chaque valeur est une somme de plein de trucs. Donc déjà on sait qu'on va avoir une boucle, qui va tourner autant de fois qu'il y a de points de contrôle, et qu'on va tout sommer.
On commence donc par écrire :

In [ ]:
```
float value = 0;

for(int index=0; index < ptsDeControle.rows(); index ++){
    //Faire quelque chose
}
```

On sait aussi que i et j désigne en fait le point qu'on va utiliser dans notre fonction Φ. On va donc convertir ces 2 int en un VectorXf qui prend 2 float en paramètres.

In [ ]:
```
Eigen::VectorXf point(2);
point << i, j;
```

Enfin, qu'est-ce qu'on va faire dans notre boucle ? Eh bien on somme nos ω avec nos Φ. Vous vous rappelez de la méthode

? Bah on la réutilise ici ! Sauf que cette fois c'est notre point par rapport à nos points de contrôle.
ATTENTION ! Eigen met les vecteurs sous la forme

$$\begin{pmatrix} x_1 & z_1 \end{pmatrix}$$

au lieu de

$$\begin{pmatrix} x_1 \\ z_2 \end{pmatrix}$$

donc on doit penser à transposer avant de soustraire ! On obtient donc notre fonction en entier :

In [ ]:
```
int findValue(const int i, const int j, const Eigen::MatrixXf ptsDeCont
role, const Eigen::VectorXf w){
    float value = 0;
    Eigen::VectorXf point(2);
    point << i, j;

    for(int index=0; index < ptsDeControle.rows(); index ++){
        value += w(index) * phi(
            (point.transpose() - ptsDeControle.row(index)).norm()
        );
    }

    return round(value);
}
```

Vous remarquerez qu'à la fin je renvoie `round(value)` et pas juste value parce que je veux que mes cubes restent à des hauteurs entières.

d. Du coup c'est quoi notre fonction générale ?

Bah du coup là vous devriez tout comprendre :

In [ ]:
```
Eigen::MatrixXd getValues(const Eigen::MatrixXf ptsDeControle, const Ei
gen::VectorXf valeurs){
    Eigen::VectorXf w = findOmega(ptsDeControle, valeurs);
    Eigen::MatrixXd map = Eigen::MatrixXd::Zero(WORLD_TAILLE, WORLD_TAI
LLE);

    for(int i=0; i<map.rows(); i++){
        for(int j=0; j<map.cols(); j++){
            map(i, j) = findValue(i, j, ptsDeControle, w);
        }
    }
```

```
    return map;
}
```

3. Mdr je mets ça où dans mon code général ?

Du coup moi j'ai appelé ce code dans une fonction `loadMonde()` dans ma classe Cube. Je déclare une map, des points de contrôle des valeurs.

In [ ]:
```
void Cube::loadMonde(){
    Eigen::MatrixXd map(WORLD_TAILLE, WORLD_TAILLE);

    const int nbPoints = 3;

    Eigen::MatrixXf ptsDeControle(nbPoints, 2);
    ptsDeControle << 10, 10,
    2, 3,
    30, 40;
    Eigen::VectorXf valeurs(nbPoints);
    valeurs << 10, -2, -20;

    map = getValues(ptsDeControle, valeurs);
```

Puis je remplis mon tableau de cube avec un glm::vec3 grâce à une triple boucle **for** où x et z sont incrémentés bêtement.
Pour y, je le fait commencer du "fond" de mon monde, pour remplir jusqu'à la hauteur souhaitée, c'est-à-dire `map(x, z)`. J'obtiens donc :

In [ ]:
```
for(int x=0; x<WORLD_TAILLE; x++){
    for(int z = 0; z<WORLD_TAILLE; z++){
        for(int y= WORLD_DEPTH; y <= map(x, z); y++){
            monTableauDeCubes.push_back(glm::vec3(x, y, z));
        }
    }
}
```

Voilàààà