Préparation à réaliser avant le début du TP

Soit une suite de points p_0,p_1, ... p_n du plan (n>=3).

On cherche à approximer l'ensemble de ces points par une droite.

- ➔ Écrivez le système d'équations (sur contraint) que l'on chercherait à satisfaire (dans le cas où les points seraient tous alignés).
- ➔ Exprimez ce système sous forme matricielle.
- ➔ Rappelez la solution au sens des moindres carrés de ce système exprimé de manière matricielle.
- ➔ Répondez au même questions dans le cas où l'on cherche à approximer ces n points par une parabole.

Introduction à la librairie Eigen

Installation d'Eigen.

➔ Téléchargez Eigen (v3) à partir de ce lien:
http://eigen.tuxfamily.org

➔ Décompressez cette archive dans un répertoire à part et placez y une ligne de commande.

La compilation d'Eigen se réalise à partir de CMake.
Créez un répertoire build:
```
$ mkdir build
$ cd build/
```
Lancez la configuration interactive de CMake
```
$ cmake-gui ..
```

Dans l'onglet CMAKE, cherchez le champs CMAKE_INSTALL_PREFIX
Modifiez le chemin vers un répertoire de votre compte (la librairie s'installera dans ce répertoire).
Lancez "*Configure*" puis "*Generate*".
En ligne de commande, assurez vous qu'un Makefile soit présent dans le répertoire, et taper:
```
$ make install
```

Un répertoire `include/` doit apparaître dans votre répertoire cible.

Utilisation d'Eigen.

➔ Créez un autre répertoire où vous allez désormais coder votre programme C++ qui utilisera la librairie Eigen.
➔ Créez un fichier source (dans ce répertoire) et tapez les lignes suivantes:

```
#include "[CHEMIN_LOCALE_VERS_LIBRAIRIE_EIGEN]/Eigen/Dense"
int main()
{
    return 0;
}
```

➔ Assurez-vous que ce programme compile et que les fichiers d'en tête d'Eigen soient bien reconnus.

➔ Assurez-vous que le code suivant fonctionne (on modifiera potentiellement les en-têtes d'inclusions par rapport aux chemins relatifs).

```
#include "../../eigen/include/eigen3/Eigen/Dense"
#include "../../eigen/include/eigen3/Eigen/Eigenvalues"

int main()
{
    Eigen::Matrix2d a;
    a<<1,2,
       3,4;

    Eigen::Matrix2d b=a.inverse();
    std::cout<<a<<std::endl;
    std::cout<<b<<std::endl;

    Eigen::EigenSolver<Eigen::Matrix2d> c(a);
    std::cout<<c.eigenvalues()<<std::endl;
    std::cout<<c.eigenvectors()<<std::endl;

    return 0;
}
```

Application.

Soit les points $p_0(1,1.1)$, $p_1(2,1.9)$, $p_2(3,3.1)$

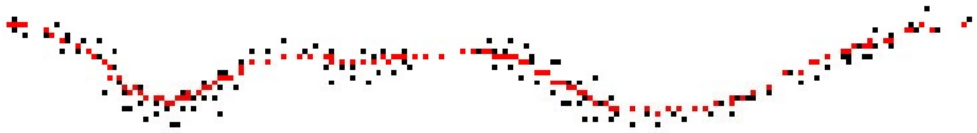
➔ Déterminez la droite qui approxime le mieux au sens des moindres carrés ces trois points.

On pourra se référer à cette page pour la solution du système linéaire sur contraint:
http://eigen.tuxfamily.org/dox-devel/group__LeastSquares.html

Régularisation de courbe

On considère une image contenant un courbe bruitée. Celle-ci ce présente sous la forme d'un ensemble de points dispersés tel que le présente la figure suivante. Notez que ces points ne sont pas ordonnés.

On cherche à modifier la position des points tel que ceux-ci viennent suivre la "courbe moyenne".
La figure suivante montre, en rouge, une solution possible à ce problème



Pour obtenir ce résultat, nous allons considérer l'approche des moindres carrés glissants (Moving Least Squares, ou MLS) qui viens approximer une courbe au sens des moindres carré de manière locale.

Considérons un point quelconque du plan p. On note $p_i=(x_i,y_i)$ une position de l'ensemble des points initiaux.
On cherche une approximation d'une courbe au voisinage de p.
On va réaliser une approximation linéaire, c'est à dire une courbe de la forme `y=ax+b`
Contrairement au cas précédent, on ne va pas considérer un fitting global, mais on va venir pondérer localement les données. C'est à dire que les positions p_i proches de p auront une importance plus grande que ceux éloignés.

On cherche ainsi à trouver les coefficients (a,b) tels qu'ils minimisent

$$\sum_i \theta(\|p - p_i\|) \ \ (ax_i + b - y_i)^2$$

Avec theta étant une fonction de pondération décroissante vers 0 lorsque la distance augmente.
On pourra considérer par exemple:
theta(x)=exp(-s x^2), avec a un paramètre de localité.

On peut montrer que la solution à ce problème revient à trouver la solution au système sur déterminé suivant:

$$\mathrm{W}\,\mathrm{A}\,\mathbf{x} = \mathrm{W}\,\mathbf{b}$$

Avec W la matrice diagonale suivante

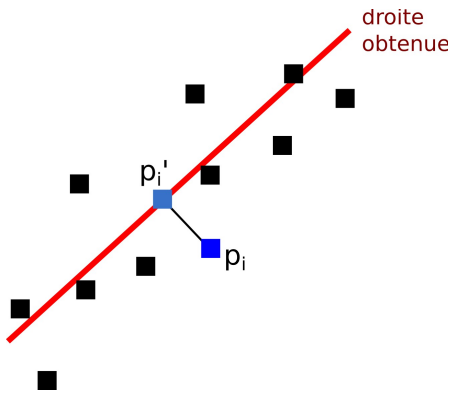
$$\mathrm{W} = \mathrm{diag}\left(\theta(\|p - p_0\|), \theta(\|p - p_1\|), \dots\right)$$

Et A, et b étant les vecteurs dépendant des coordonnées que l'on retrouverait dans un moindre carré standard.
La solution au sens des moindres carrés peut être obtenue par la relation matricielle suivante.

$$\mathbf{x} = \left(\mathrm{A}^t\mathrm{W}\,\mathrm{A}\right)^{-1}\mathrm{A}^t\mathrm{W}\,\mathbf{b}$$

Notez qu'on l'applique ici pour un fitting de droite avec deux paramètres, mais pourrait se généraliser à tout type de fitting local (ayant des paramètres linéaires).

Principe de la régularisation: L'idée consiste à calculer la meilleur approximation de droite locale pour chaque point p_i des données.
On associe alors au point p_i le point p_i' qui est le projeté orthogonal de p_i sur la droite ainsi trouvée.
En appliquant ce procédé pour chaque point p_i, on obtient ainsi une courbe "moyenne" passant au milieu du nuage de points.



La méthode de projection consiste à trouver la droite approximant le nuage de point localement (droite rouge), puis à y projeter orthogonalement le point pi.

➔ Implémentez ce fitting grâce à la librairie Eigen. Observez l'influence de "s" sur la forme de la courbe finale.

Aide:
- Le code fourni vous propose une classe de gestion d'image, une fonction de chargement et d'écriture au format ppm, et une fonction permettant d'extraire les coordonnées des points noirs, ainsi que d'ajouter d'autres points de couleurs à partir d'un vecteur de coordonnées.
- La matrice W possède potentiellement de nombreux zéros suivant le choix des pondérations locales. On peut rendre la résolution du système plus efficace en limitant la matrice aux entrées non nulles et ainsi réduire considérablement la taille des matrices.
- Afin d'éviter les divergences associé aux calculs matriciels, on normalise les coordonnées des sommets entre 0 et 1.

Analyse d'un dessin

➔ Compilez et lancez le programme qui vous est fourni.
Celui-ci propose une interface avec Qt.

Pour pouvez dessiner un ensemble de points dans la zone blanche (*Drawing Mode*) ou bien récupérer l'information d'un clic souris (*Click Mode*).
Pour l'instant, seul le mode de dessin apporte une information visuelle.

Le code du projet se décompose de la manière suivante:
- une classe `window` viens prendre en compte les événements (clic de boutons)
- La classe `window` contient la classe `render_area`. Celle-ci gère la partie de dessin et l'interaction entre la souris et cette zone.

Cette classe interagit également avec la classe point_set qui stocke et traite les données.
En particulier, lorsque la souris se déplace (`mouseMoveEvent`), les positions des clics sont récupérés et stockés dans la structure `data_src` de `point_set`.

Lors de chaque affichage (fonction `paintEvent` de `render_area`) l'ensemble de ces points sont parcourus et affichés.

Le but du programme est de calculer et d'afficher différentes informations par rapport à cet ensemble de points.
Notez que l'architecture du code est déjà mis en place, seul la classe `point_set` doit être complétée.

- Premièrement, on va chercher à afficher de manière dynamique (au fur et à mesure du tracé) le **centre de gravité** et les **axes principaux d'inerties** de l'ensemble de points.
- Deuxièmement, on va également proposer de connaître la **droite approximant** l'ensemble de points proche du dernier clic souris.

Axes d'inerties.

Supposons que $g=(x_g,y_g)$ soit le centre de gravité de l'ensemble des points.
L'inertie d'un ensemble de points $p_i=(x_i,y_i)$ est quantifiée par la matrice d'inertie I telle que

$$\mathrm{I} = \sum_i (\mathbf{p}_i - \mathbf{g})^t \, (\mathbf{p}_i - \mathbf{g}) = \begin{pmatrix} \sum_i (x_i - x_g)^2 & \sum_i (x_i - x_g)(y_i - y_g) \\ \sum_i (x_i - x_g)(y_i - y_g) & \sum_i (y_i - y_g)^2 \end{pmatrix}.$$

Les axes d'inerties correspondent au vecteurs propres de cette matrice I.
Notons que cela revient à étudier l'ensemble de points par l'approche dite de PCA (principal component analysis).

Notons λ_0,λ_1 les deux valeurs propres, et $\mathbf{e}_0,\mathbf{e}_1$ les deux vecteurs propres.
Alors les axes $\mathbf{a}_0$ et $\mathbf{a}_1$ tels que

$$\mathbf{a}_0 = \sqrt{\lambda_0}\,\mathbf{e}_0$$
$$\mathbf{a}_1 = \sqrt{\lambda_1}\,\mathbf{e}_1$$

correspondent aux axes principaux du nuage de points.

Le plus grand axe indique la direction suivant laquelle les points sont préférentiellement alignés. En analyse par PCA, il correspondrait à l'axe porteur de l'information principal.
L'axe le plus petit est quant à lui orthogonal à l'autre axe et vient indiquer l'axe suivant lequel les points sont le moins alignés.
Quelques exemples de nuages de points et de leurs axes principaux sont illustrés à la figure suivante

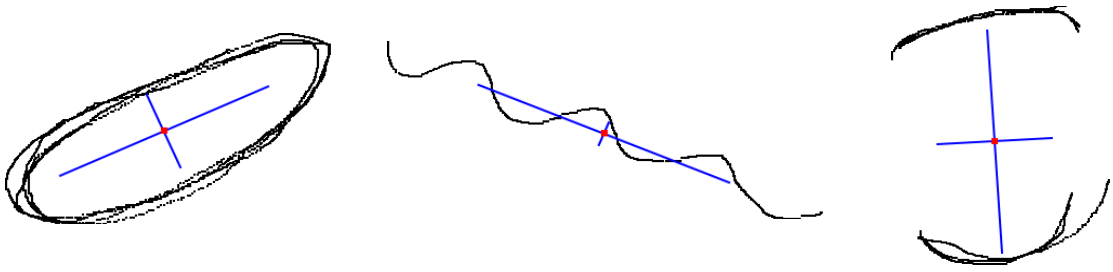


Fig. Les axes d'inerties en bleus indiquent les orientations principales de l'ensemble des points noirs.

Il est également courant de tracer l'ellipse dont les axes principaux sont a0 et a1. Celle-ci donne une vision simplifiée de la forme du nuage de points.
Elle permet de quantifier finement un nuage de point qui se répartirait dans l'espace suivant une distribution aléatoire Gaussienne.

Droite locale approximante.

En utilisant la méthode des moindres carrés glissants, il est possible d'estimer, en tout point du plan, la droite qui approxime localement le mieux le nuage de point.
En affichant cette droite (point sélectionné projeté sur la droite + vecteur directeur), on peut obtenir l'image de la droite "tangente" à un nuage de point par rapport à une position quelconque du plan.
Cette droite est illustrée sur les figures suivante

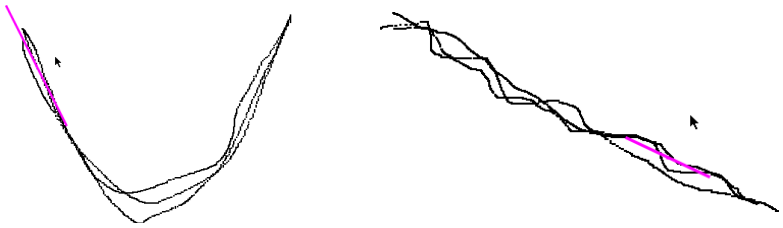


Fig. La droite magenta indique la droite approximant localement le nuage de point par rapport à la position de la souris.

Application:

Note: Lorsqu'il ne restera plus qu'une heure, passez à la partie sur Python.

- ➔ Calculez le centre de gravité dans la fonction `point_set::compute_center`.
  Les sommets à analyser sont stockés dans `data_src`. Les coordonnées du centre de gravité doivent être mises à jour dans la variable center.
  Vérifiez que ce centre soit bien correctement mise à jour de manière visuelle lorsque vous dessinez de nouveaux points.

- ➔ Complétez la fonction `compute_inertia`.
  Vous calculerez tous d'abord la matrice d'inertie, puis ses valeurs et vecteurs propres.
  Placez la valeur $\mathbf{a}_0$ et $\mathbf{a}_1$ (voir définition précédente) dans les variables `eigen_0` et `eigen_1`.
  Observez cette fois que les axes d'inerties sont bien mis à jour de manière visuelle.

- ➔ Complétez la fonction `point_set::compute_line_approximation`.
  Cette fonction reçoit les coordonnées (x,y) du point d'évaluation (coordonnées de la souris). Elle doit calculer la droite qui approxime le mieux l'ensemble des points voisins de (x,y). Enfin, on placera dans les variables `projected_vertex` et `direction_line` respectivement la position du projeté orthogonal de (x,y) sur la droite, et le vecteur directeur de cette droite.

Introduction à Python

➔ Ouvrez un fichier nommé `[NOM].py` à l'aide d'un éditeur de texte.
➔ Copiez le code suivant dans votre fichier.

```
T=[1,4,7,6,1]

print(T)

T.append(-4)

for valeur in T:
    print(valeur)
```

➔ Lancez l'interpréteur Python sur ce fichier à partir de la ligne de commande.
`$ python [NOM].py`

Python permet l'affichage de manière aisé, pour cela, on utilise la bibliothèque `matplotlib`, ainsi que la librairie de calcul numérique `numpy`.

➔ Recopiez ce code, et observez le résultat à l'écran.

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

N=100
t=np.linspace(0,2*np.pi,N)

plt.plot(np.cos(t),np.sin(t))
plt.plot(np.cos(t),np.sin(t),'r.')
plt.axis('equal')
plt.show()
```

Application:

➔ Recopiez le code suivant. Observez que les calculs sont réalisés de manière vectoriel (sur l'espace des nombres complexes).

➔ Vers quelle figure converge cette image lorsque l'on fait croître le nombre d'itération de la boucle.

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

N=200
u=np.linspace(-2,2,N)

x,y=np.meshgrid(u,u)
z=x+1j*y

c=z

for k in range(3):
    z=z**2+c

norme=z.real**2+z.imag**2

im=plt.imshow(norme)
im.set_clim(0,4)
plt.show()
```

Exercice:

On considère le polynôme (de la variable complexe)
$p(z)=z^3-1$
Ce polynôme possède 3 racines: 1, -1/2+i sqrt(3)/2 et -1/2-i sqrt(3)/2.

Étant donné un point de départ z_0, on construit la suite $(z_1,z_2,...,z_n)$ obtenue par application de la formule de recherche de racines de Newton.
On rappelle que cette formule permet de converger vers les racines d'une fonction lorsque le point de départ est suffisamment proche de l'une des racine. On utilise l'itération:
$z_{n+1}=z_n-F(z_n)/F'(z_n)$

➔ Quelle est l'expression de z_{n+1} en fonction de z_n?

➔ Similairement au cas précédent, calculez ces itérations pour l'ensemble du domaine complexe entre [-2,2]. Faites en sorte d'assigner une couleur différente suivant que la valeur soit proche d'une racine ou d'une autre.
Observez la figure obtenue, interprétez.

Aide.

Pour l'assignation d'une couleur différente en fonction de la proximité avec une racine ou un autre, on pourra par exemple utiliser le code suivant:

```
z0,z1,z2=1,-1/2+1j*np.sqrt(3)/2,-1/2-1j*np.sqrt(3)/2
id_racine[abs(z-z0)<0.01,abs(z-z1)<0.01,abs(z-z2)<0.01]
norme=1.0*id_racine[0]+2.0*id_racine[1]+3.0*id_racine[2];
```

Exercice complémentaire.

Soit un polynôme quelconque dont les coefficients sont des réels (potentiellement aléatoire).

Construisez une fonction Python qui, étant donné les coefficient d'un polynôme renvoie l'ensemble des coefficients des polynômes dérivés (jusqu'au polynôme nulle).

Construisez une fonction Python qui, étant donné un polynôme renvoie sa matrice compagnon.

Soit p un polynôme tel que:

$$p(x) = c_0 + c_1\,x + c_2\,x^2 + \cdots + c_{n-1}\,x^{n-1} + x^n$$

On rappelle que la matrice compagnon A du polynôme p est

$$\mathrm{A} = \begin{pmatrix} 0 & 0 & \dots & 0 & -c_0 \\ 1 & 0 & \dots & 0 & -c_1 \\ 0 & 1 & \dots & 0 & -c_2 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -c_{n-1} \end{pmatrix}$$

Les valeurs propres de A sont les racines du polynôme p.

- ➔ Affichez la position des racines (complexes) du polynôme p.
- ➔ Affichez la position des racines de tous les polynômes dérivés de p (affichez les racines des dérivées k$^{\text{ième}}$ d'une couleur différente).