Devoir maison d'informatique
Mini-Projet Calculatrice NPI - Prépa des INP de la Réunion

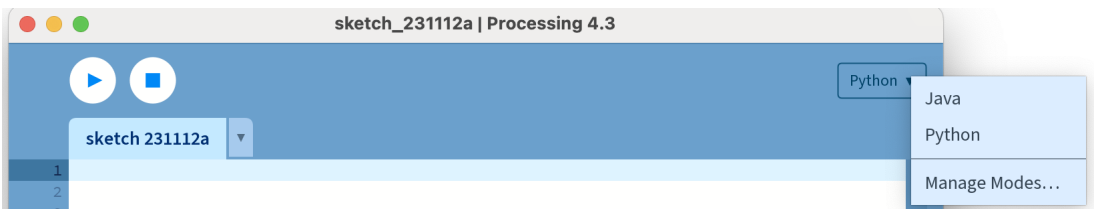
Gilles Hoareau

CPP2A - A rendre avant le samedi 16 décembre 2023 à 19h00

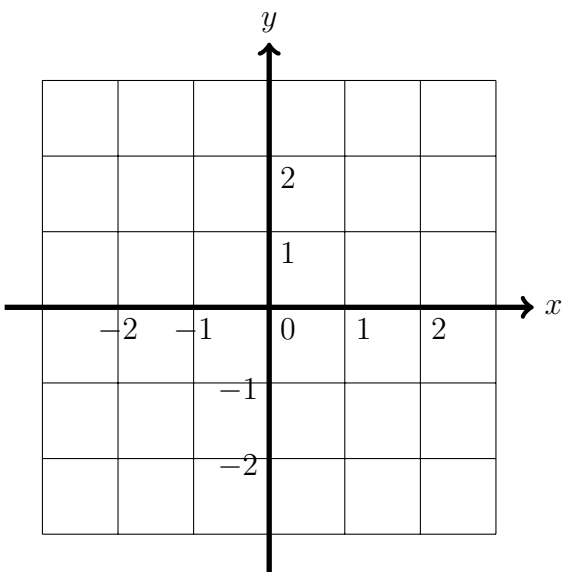
Tout retard sera sanctionné.

Processing

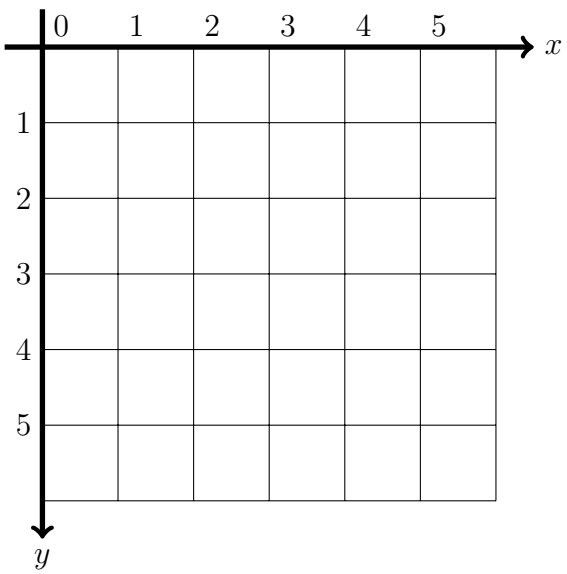
Installer le logiciel **Processing** sur votre système d'exploitation. Une fois le logiciel installé et ouvert, installer le mode Python.



Parlons maintenant du logiciel **Processing**. Ce logiciel est tout particulièrement adapté à la création graphique interactive. Il fonctionne sur Macintosh, Windows, Linux et Android. Il est basé sur la plate-forme Java. Mais il est également possible de programmer avec le langage Python. Pour ce cours, nous utiliserons le mode Python. La difficulté, lorsque l'on travaille avec un environnement graphique, est de bien comprendre le mode de repérage utilisé. Vous connaissez sans doute très bien la notion de repère cartésien du plan, avec les notions d'abscisse et d'ordonnée. Vous avez également l'habitude de travailler avec une certaine orientation de ce plan.



Sur un écran d'ordinateur ou dans une fenêtre graphique, on se repère par rapport aux pixels disponibles en abscisse et en ordonnées, qui dépendent de la définition choisie. Prenons l'exemple d'un écran défini de 36 pixels (6×6) : l'origine est placée en haut à gauche de la fenêtre graphique, l'axe des abscisses est orienté vers la droite, et l'axe des ordonnées vers le bas. Vous devrez vous habituer à cette nouvelle orientation du plan.



Fenêtre graphique

Lorsque vous écrivez une ligne de code, et appuyer sur le bouton **play**, une fenêtre s'ouvre.

Afin de régler la taille de cette fenêtre, vous pouvez utiliser la fonction `size`.

```
size(largeur,hauteur)
```

où **largeur** et **hauteur** désignent des grandeurs exprimées en pixels.

Gestion des couleurs

1. La fonction `background` : cette fonction va nous permettre de donner une couleur à notre fenêtre, à l'arrière-plan. Elle s'utilise de la manière suivante :

   ```
   background(R,G,B)
   ```

   Les paramètres utilisés dans cette fonction sont la composition de la couleur en RGB : on précise trois valeurs entières chacune comprise entre 0 et 255. Par exemple, `background`(0,0,0) ou `background`(0) désignera un fond **noir**, tandis que `background`(255,255,255) ou `background`(255) désignera un fond **blanc**.

2. La fonction **stroke** permet de donner une couleur à un trait, ou de donner une couleur autour d'une forme.

3. La fonction **fill** permet de remplir une forme avec une couleur.

Gestion des formes

Voici les fonctions permettant de créer des formes :

1. La fonction **point** permet de créer un point. Ainsi, si l'on souhaite créer le point $P(x, y)$, on écrira l'instruction :

```
point(x,y)
```

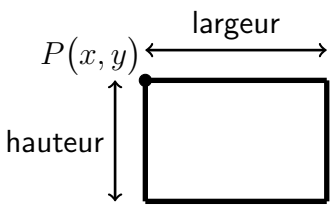
2. La fonction **strokeWeight** permet de préciser l'épaisseur du trait de construction pour les objets que l'on souhaite dessiner.

3. La fonction **line** permet de relier deux points à l'aide d'un segment. Ainsi, si l'on souhaite créer le segment d'extrémités $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$, on écrira l'instruction :

```
line(x1,y1,x2,y2)
```

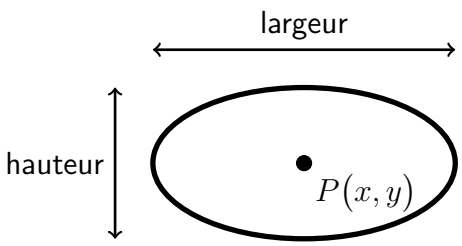
4. La fonction **rect** permet de dessine run rectangle. Ainsi, si l'on souhaite créer le rectangle de sommet $P(x, y)$, de dimensions **largeur** et **hauteur**, on écrira l'instruction :

```
rect(x,y,largeur,hauteur)
```



5. La fonction **ellipse** permet de dessiner une ellipse. Ainsi, si l'on souhaite créer l'ellipse centrée au sommet $P(x, y)$, de dimensions **largeur** et **hauteur**, on écrira l'instruction :

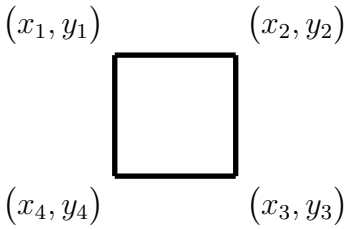
```
ellipse(x,y,largeur,hauteur)
```



6. La fonction `quad` permet de dessiner un quadrilatère, connaissant les coordonnées de ses quatre sommets. On écrira l'instruction :

```
quad(x1,y1,x2,y2,x3,y3,x4,y4)
```

   Le schéma ci-dessous indique dans quel ordre on passe les points en paramètres de la fonction.



7. Enfin, lorsque vous dessinez toutes ces figures, le trait n'est pas net, c'est dû à la pixellisation. Pour y remédier, la fonction `smooth` est là.

Un premier exemple : dessiner une balle

On va écrire un programme sous **Processing** capable de dessiner une fenêtre graphique de 1200 pixels de largeur, par 800 pixels de hauteur, dont le fond sera noir. On y dessinera ensuite une balle blanche : disque blanc dont le centre se situe au point de coordonnées $(100, 250)$ et de rayon 50 pixels.

```
size(1200,800)
smooth()
background(0)
fill(255)
stroke(255)
ellipse(100,250,50,50)
```

Créer une balle animée : les fonctions setup et draw

Processing propose deux fonctions que le programmeur devra compléter :

✓ la fonction setup qui sera appelée une seule fois dès le début de l'exécution du programme,

✓ la fonction draw qui sera appelée à chaque image.

Ces deux fonctions ne prennent aucun paramètre et ne retournent aucune valeur.

Ce sont des fonctions pré-implémentées et prêtes à l'emploi. L'intérêt premier, est d'éviter les répétitions inutiles (sorte de "factorisation" du code).

Dans la fonction setup, nous déclarerons toutes les instructions qui seront valables tout au long de notre programme. Les instructions que nous y mettons utilisent en général les fonctions background et size. Vous pouvez bien sûr y mettre d'autres fonctions de votre choix.

La fonction draw exécute de manière continue l'ensemble des instructions s'y trouvant, à la manière d'une boucle infinie. Il est conseillé de préciser le taux de rafraîchissement souhaité, grâce à la fonction frameRate, que l'on peut appeler par exemple dans la fonction setup. Voici ci-dessous un code assez simple à mettre en place, en utilisant par exemple la position de la souris :

```
def setup():
    size(1200,800)
    smooth()
    background(0)
    fill(255)
    stroke(255)
def draw():
    background(0)
    ellipse(mouseX,mouseY,100,100)
```

Les variables mouseX et mouseY contiennent à tout moment respectivement l'abscisse et l'ordonnée de la position de la souris sur la fenêtre ouverte.

Création d'une classe de vecteurs et de points

On souhaite créer une classe de vecteurs et de points géométriques du plan.

Un point du plan sera représenté par une instance de la classe Point, qui possèdera deux attributs : les coordonnées du point dans le plan.

De même, un vecteur du plan sera représenté par une instance de la classe Vecteur, qui possèdera deux attributs : les coordonnées du vecteur dans le plan.

Exercice 1

Créer une classe Vecteur, qui possède deux attributs, qui représentent dans l'ordre l'abscisse x et

l'ordonnée y du vecteur $\overrightarrow{v}$ de coordonnées (x, y).

Outre la méthode constructeur, cette classe possèdera également les méthodes suivantes :

1. la méthode **Scal**, qui appliquée à une instance $\overrightarrow{v_1}$ de la classe **Vecteur** et prenant en argument une instance $\overrightarrow{v_2}$ de la classe **Vecteur**, renvoie en sortie la valeur du produit scalaire euclidien $\overrightarrow{v_1} \,.\, \overrightarrow{v_2}$.

2. la méthode **Norme**, qui appliquée à une instance $\overrightarrow{v}$ de la classe **Vecteur**, renvoie en sortie la norme euclidienne de ce vecteur, à savoir $\left\|\overrightarrow{v}\right\|_2$.

3. la méthode **Soust**, qui appliquée à une instance $\overrightarrow{v_1}$ de la classe **Vecteur** et prenant en argument une instance $\overrightarrow{v_2}$ de la classe **Vecteur**, renvoie en sortie une instance de la classe **Vecteur** qui représente le vecteur $\overrightarrow{v_1} - \overrightarrow{v_2}$.

4. la méthode **Mult**, qui appliquée à une instance $\overrightarrow{v}$ de la classe **Vecteur** et prenant en argument un scalaire λ (du type **float**), renvoie en sortie une instance de la classe **Vecteur** qui représente le vecteur $\lambda\,\overrightarrow{v}$.

Exercice 2

Créer une classe **Point**, qui possède deux attributs, qui représentent dans l'ordre l'abscisse x et l'ordonnée y du point de coordonnées (x, y).

Outre la méthode constructeur, cette classe possèdera également les méthodes suivantes :

1. la méthode **Distance**, qui prend en argument deux instances P_1 et P_2 de la classe **Point** et qui renvoie en sortie la distance euclidienne P_1P_2 séparant ces deux points.

2. la méthode **Vect**, qui prend en argument deux instances P_1 et P_2 de la classe **Point** et qui renvoie en sortie une instance de la classe **Vecteur** qui représente le vecteur $\overrightarrow{P_1P_2}$.

3. la méthode **Translater**, qui appliquée à une instance P de la classe **Point**, et ayant en argument une instance $\overrightarrow{V}$ de la classe **Vecteur**, renvoie en sortie une instance de la classe **Point** représentant l'image du point P par la translation de vecteur $\overrightarrow{V}$.

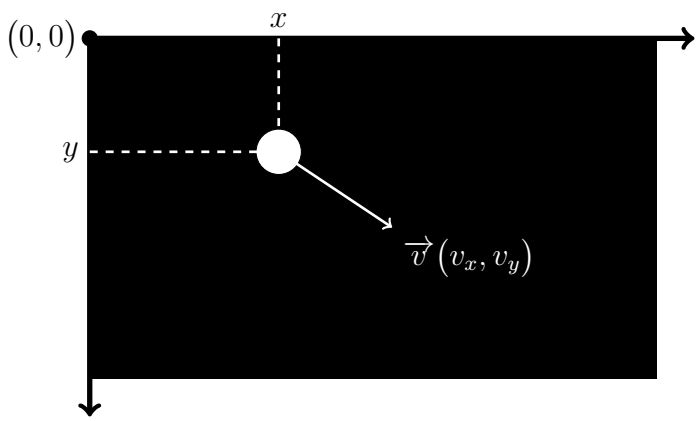
Création d'une classe de balle animée

On considère dans cette partie une fenêtre graphique de largeur 1200 pixels, et de hauteur 800 pixels de fond noir. On souhaite y animer une petite balle de couleur blanche (symbolisée par un cercle de diamètre 20 pixels).

A tout moment, il sera très utile de considérer les paramètres suivants de la balle, qui seront susceptibles d'être modifiés (en cas de mouvement) :

✓ ses coordonnées x et y,

✓ sa vitesse instantanée, qui est un vecteur ayant pour composantes la valeur v_x en abscisse et la valeur v_y en ordonnée.



Exercice 3

Écrire une classe **Balle** qui possède deux attributs :

1. le premier attribut est une instance de la classe **Point** et représentera le centre de la balle,
2. le deuxième attribut est une instance de la classe **Vecteur** et représentera le vecteur vitesse instantanée de la balle.

Exercice 4

Créer la méthode **Dessiner** de la classe **Balle** qui permet de dessiner, pour chaque instance de classe, un cercle blanc de diamètre 20 pixels à l'écran.

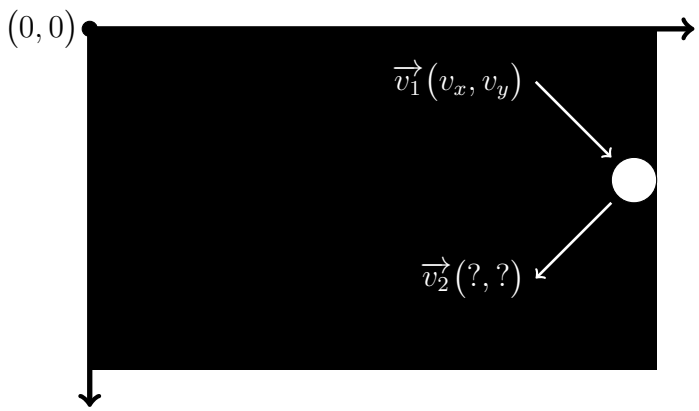
Créer la méthode **Deplacer** de la classe **Balle** qui permet, lorsqu'on l'applique à une instance de la classe **Balle**, de mettre à jour les coordonnées de son centre selon son vecteur vitesse instantanée, sachant que le mouvement de la balle doit être rectiligne uniforme.

Gestion des rebonds de la balle

Il est possible de modéliser le rebond de la balle sur un bord de l'écran. Supposons par exemple que la balle doive rebondir lorsqu'elle rencontre le bord droit de l'écran.

Ce rebond est supposé **élastique**, c'est-à-dire que l'énergie cinétique est globalement conservée pendant le rebond. Dans ces conditions, lorsque la balle rencontre le bord droit de l'écran, son vecteur vitesse est modifié de la manière illustrée ci-dessous, de sorte que sa norme est conservée, mais pas sa direction, ni son sens.

Ainsi, si $\vec{v_1}$ désigne son vecteur vitesse avant rebond, et $\vec{v_2}$ celui après rebond, alors on a la représentation suivante :



Exercice 5

Créer la méthode **Rebondir** de la classe **Balle** qui permet, lorsqu'on l'applique à une instance de la classe **Balle**, de mettre à jour les coordonnées de son vecteur vitesse, afin de faire rebondir la balle sur tous les bords de l'écran.

Bien entendu, en dehors des rebonds, la balle suit un mouvement rectiligne uniforme.

Création d'une classe de liste de balles

On souhaite créer une classe permettant de gérer l'animation complète d'un nombre $N \in \mathbb{N}^*$ de balles : le déplacement rectiligne uniforme, le rebond sur les bords de l'écran et la collision entre les balles.

Exercice 6

Écrire une classe **ListeBalles** qui possède quatre attributs :

1. le premier attribut est un entier N non nul, qui désigne le nombre de balles que l'on souhaite créer,
2. le deuxième attribut v_{min} est un flottant et désigne la valeur minimale autorisée pour les

coordonnées du vecteur vitesse instantanée des différentes instances de balles que l'on va créer,

3. le deuxième attribut v_{max} est un flottant et désigne la valeur maximale autorisée pour les coordonnées du vecteur vitesse instantanée des différentes instances de balles que l'on va créer,

4. le quatrième attribut est une liste, qui va contenir les N instances de balles que l'on souhaite créer. La position du centre des balles, ainsi que les coordonnées du vecteur vitesse instantanée de ces balles seront choisies aléatoirement.

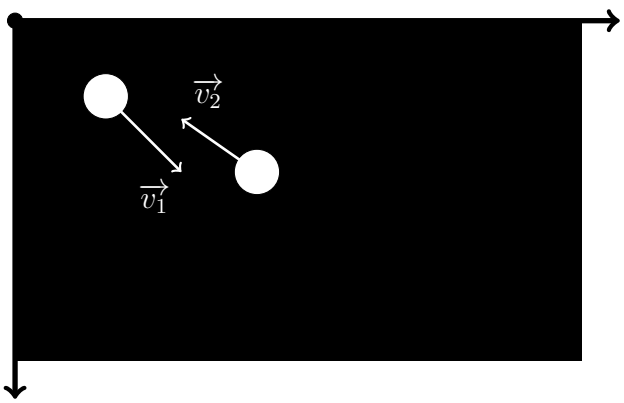
Exercice 7

Créer la méthode **Animer**, qui appliquée à une instance de la classe **ListeBalles**, parcourt les différentes balles de la liste, et les dessine, les déplace et les fait rebondir sur les bords de l'écran.

Gestion des collisions entre balles

On souhaite dorénavant gérer les collisions entre deux instances de la classe **Balle**. Les collisions seront supposées **élastiques**.

Afin de vérifier si deux balles entrent en collision, nous allons tout simplement calculer la distance euclidienne reliant les centres de chacune de ces balles. Si cette distance est inférieure au diamètre des balles, alors on dira qu'il y a collision.

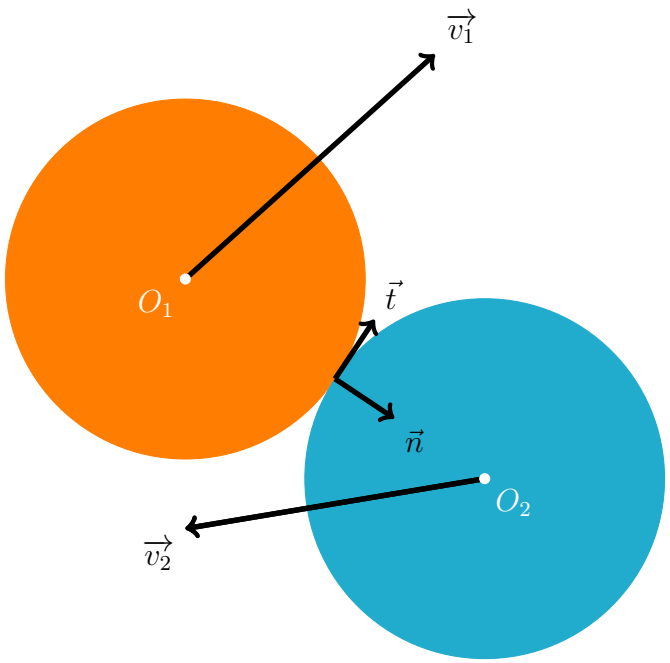


Afin de modéliser un choc **élastique** entre deux balles de même masse m, nous allons distinguer ce qui se passe juste avant la collision, et ce se passe juste après. En particulier, il est intéressant d'étudier le changement brusque des vecteurs vitesses de chacune des deux balles.

Pour chacune des balles B_i, pour $i \in \Big\{1, 2\Big\}$, définissons les grandeurs physiques suivantes :

1. m désigne la masse commune des deux balles,
2. $\overrightarrow{v_i}$ désigne le vecteur vitesse de la balle B_i avant la collision,
3. $\overrightarrow{w_i}$ désigne le vecteur vitesse de la balle B_i après la collision,
4. $\overrightarrow{p_i}$ désigne la quantité de mouvement de la balle B_i avant la collision,
5. $\overrightarrow{q_i}$ désigne la quantité de mouvement de la balle B_i après la collision,
6. $\overrightarrow{x_i}$ désigne le vecteur position de la balle B_i,
7. O_i désigne le centre de la balle B_i.

Au niveau du point de contact des deux balles (voir figure ci-dessous), on notera $\vec{n}$ le vecteur unitaire normal à la surface des deux balles et $\vec{t}$ le vecteur tangent unitaire.



On remarquera que :

$$\vec{n} = \frac{\overrightarrow{O_1O_2}}{\left\|\overrightarrow{O_1O_2}\right\|} = \frac{\overrightarrow{x_2} - \overrightarrow{x_1}}{\left\|\overrightarrow{x_2} - \overrightarrow{x_1}\right\|}$$

Le principe de conservation de la quantité de mouvement donne :

$$m\,\overrightarrow{w_1} + m\,\overrightarrow{w_2} = m\,\overrightarrow{v_1} + m\,\overrightarrow{v_2}$$

Le principe de conservation de l'énergie cinétique donne :

$$m\left\|\overrightarrow{w_1}\right\|^2 + m\left\|\overrightarrow{w_2}\right\|^2 = m\left\|\overrightarrow{v_1}\right\|^2 + m\left\|\overrightarrow{v_2}\right\|^2$$

Puisqu'aucune force tangentielle ne s'exerce sur les deux boules pendant la collision (sans frottement), la composante tangentielle du vecteur vitesse est conservée après la collision :

$$\begin{cases} \overrightarrow{w_1}.\overrightarrow{t} &= \overrightarrow{v_1}.\overrightarrow{t} \\ \overrightarrow{w_2}.\overrightarrow{t} &= \overrightarrow{v_2}.\overrightarrow{t} \end{cases}$$

Exercice 8

Grâce à la conservation de la composante tangentielle, on peut dorénavant réécrire les lois concernant la conservation de la quantité de mouvement et de l'énergie cinétique uniquement en tenant compte de la composante normale. Cela donne un système que l'on va résoudre.

Montrer alors que :

$$\begin{cases} \overrightarrow{w_1} &= \overrightarrow{v_1} - \left((\overrightarrow{v_1} - \overrightarrow{v_2}).\overrightarrow{n}\right)\overrightarrow{n} \\ \overrightarrow{w_2} &= \overrightarrow{v_2} - \left((\overrightarrow{v_2} - \overrightarrow{v_1}).\overrightarrow{n}\right)\overrightarrow{n} \end{cases}$$

Exercice 9

Créer la méthode **EntrerCollision** de la classe **ListeBalles** qui permet de gérer les collisions entre toutes les instances de balles de la liste.

Bien entendu, en dehors des collisions, les balles poursuivent leurs déplacement rectiligne uniforme, et continuent de rebondir sur les bords de l'écran.