# PRÁCTICA 0: INTRODUCCIÓN A PYTHON/NUMPY/SCIPY/MATPLOTLIB

Micaela Kortsarz
*Redes Neuronales y Aprendizaje Profundo para Visión Artificial - Instituto Balseiro*
*(Fecha: 22 de agosto de 2020)*

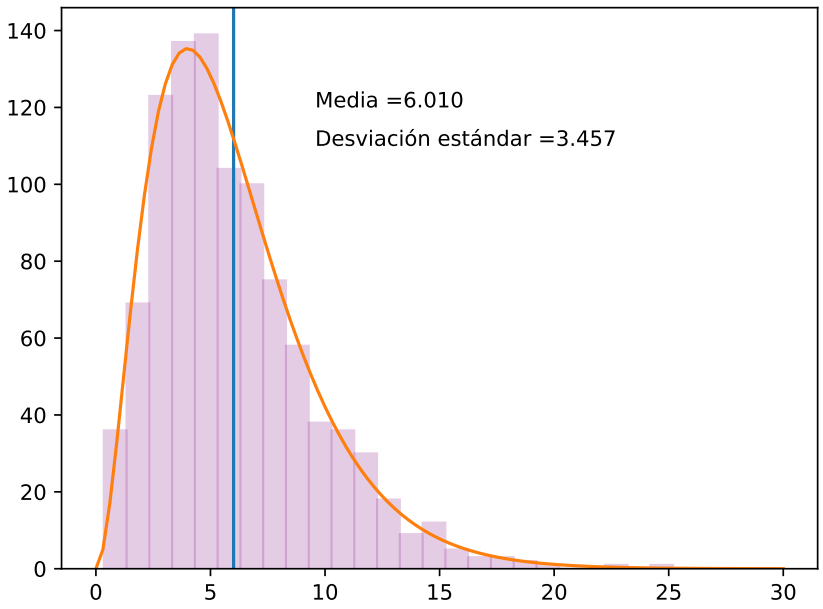
## I. EJERCICIO 1

Para resolver el sistema de ecuaciones requerido por la consigna utilicé la función linsolve de la librería sympy. Para ello, primero definí los símbolos asociados a (x,y,z) utilizando la función symbols de la misma librería.

El resultado para el sistema de ecuaciones que obtuve fue (x=1, y=-2, z=-3), coincidente con el resultado que obtuve al resolverlo yo misma manualmente.

## II. EJERCICIO 2

Para este ejercicio utilicé las librerías numpy (para calcular el histograma de la distribución, la media $\mu$ y la desviación estándar $\sigma$) y matplotlib (para graficar). En el gráfico se observa un histograma que sigue una distribución gamma con $\mu = 6{,}107$ y $\sigma = 3{,}561$. Los resultados pueden variar levemente según la semilla utilizada en la función pseudoaleatoria.



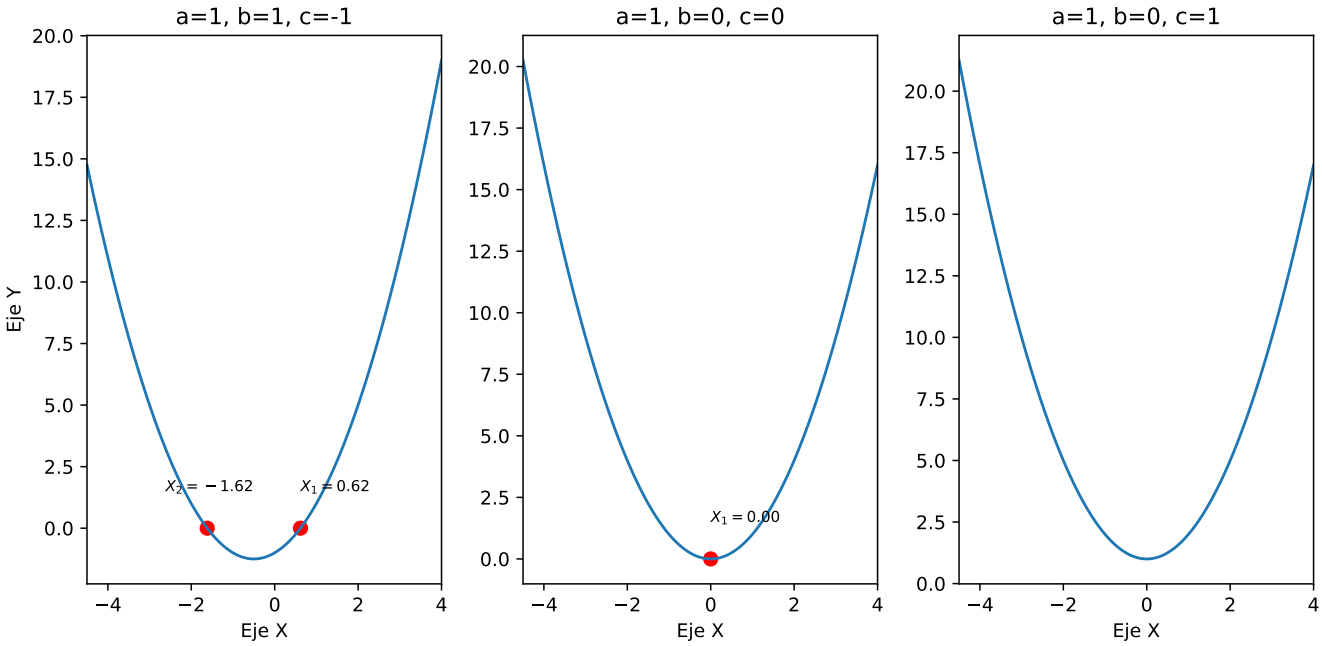

Para generar la distribución gamma de color naranja del gráfico utlicé scipy.stats.gamma(3,0,2).

La media y la desviación estándar de la distribución teórica son 6 y 3.46, prácticamente coincidentes con los valores que obtuve.

## III. EJERCICIOS 3 Y 4

Para el ejercicio 3, siguiendo la fórmula de la consigna, calculé las raíces de la función cuadrática $ax^2 + bx + c = 0$. Luego, para graficar identifiqué si existen o no soluciones reales. En caso de que existiesen, en el ejercicio 4, las grafico

con la función scatter de matplotlib y coloco sus etiquetas utilizando annotate de la misma librería y en formato latex. En el gráfico se muestran ejemplos de los 3 tipos de soluciones posibles (2 raíces reales, raíz real en cero y 2 raíces imaginarias).




## IV. EJERCICIOS 5 Y 6

Para definir la clase Lineal utilicé los métodos *__init__* y *__call__*. Con *__init__* se crea el objeto con los atributos a y b. Utilizando *__call__*, directamente al llamar al objeto resuelvo la ecuación $ax + b$, para un x dado. La clase Exponencial se generó heredando el constructor de la clase Lineal y definiendo el método *__call__* a modo que resuelva la ecuación $ax^b$.

## V. EJERCICIOS 7, 8 Y 9

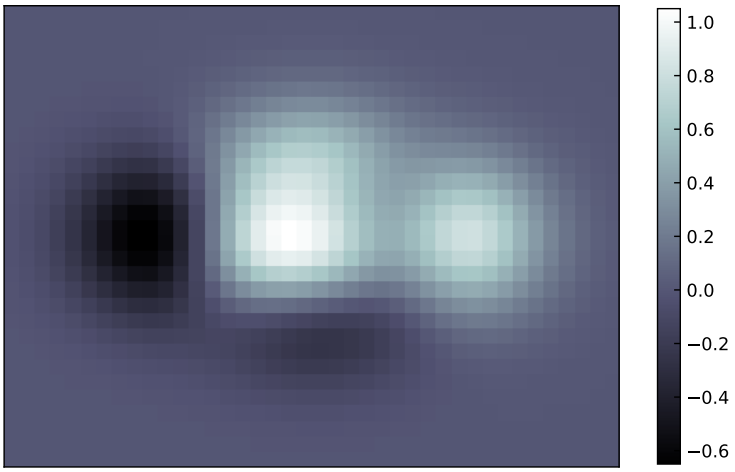
Para el punto 7 generé un módulo llamado circunferencia.py, donde se define la constante PI y la función area(r) que calcula el área para una circunferencia de radio r. Importé el módulo con el alias circle y. también utilizando la sintaxis "from circunferencia import PI, area". Luego verifiqué, tomando r=1 para el cálculo del área, que el código funcione al importar de las dos formas diferentes. Al comparar, obtuve que las áreas calculadas con las funciones importadas de las dos maneras distintas son distintos objetos, mientras que la constante PI y la función area sin evaluar son los mismos objetos.

Para el punto 8 generé un paquete llamado geometria, el cual contiene al módulo circunferencia creado para el punto 7 y, otro análogo que se llama rectangulo, que contiene también una función que calcula el área para un rectángulo. Para generar el paquete geometria generé una carpeta en donde coloqué los archivos circunferencia.py y rectangulo.py. Luego de importar dicho paquete, verifiqué el correcto funcionamiento de las funciones áreas. Para ello consideré un círculo de r=1 y un rectángulo de lados a=1 y b=2.

Para el punto 9 generé el paquete p0_lib, creando una carpeta que contiene los módulos circunferencia.py y rectangulo.py de los puntos 7 y 8. Además, generé e incluí en dicha carpeta un nuevo módulo llamado elipse.py, que contiene la constante PI y calcula el área de una elipse. Importé los módulos y paquetes según lo pedido en la consigna y verifiqué el correcto funcionamiento de todas las funciones del paquete p0_lib. La verificación la hice para un círculo con r=1, un rectángulo de lados a=1 y b=2 y, además, una elipse de radio mayor 2 y radio menor 1.

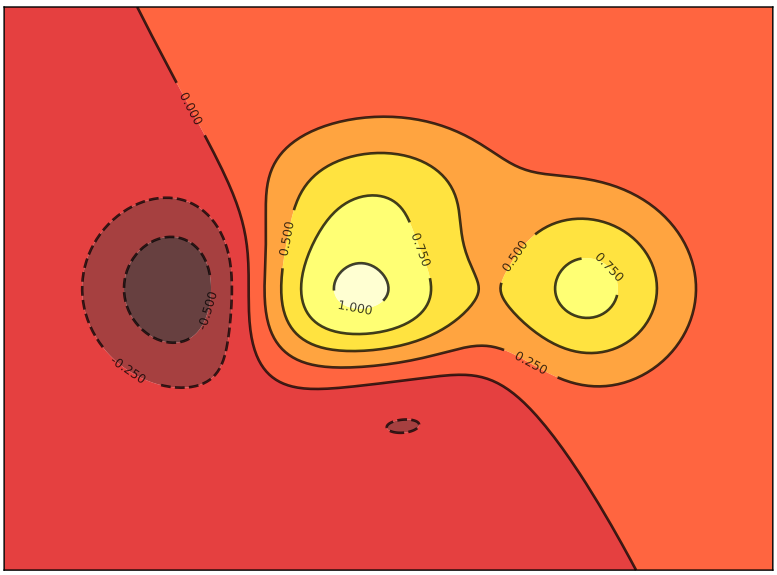
## VI. EJERCICIO 10

A partir de la definición de la función y el código de la consigna, generé el gráfico pedido. Para generar X e Y utilicé la función de numpy meshgrid, usando x e y dados en la consigna. A lo obtenido, le apliqué la función requerida por la consigna y lo grafiqué con imshow de matplotlib. Para reproducir el gráfico de la manera más parecida posible utilicé: mapa de colores "bone", sin interpolación, moví el origen utilizando la opción "lower" y, determiné valores máximos y mínimos de la escala de colores. Con la función colorbar de matplotlib generé la barra de colores.



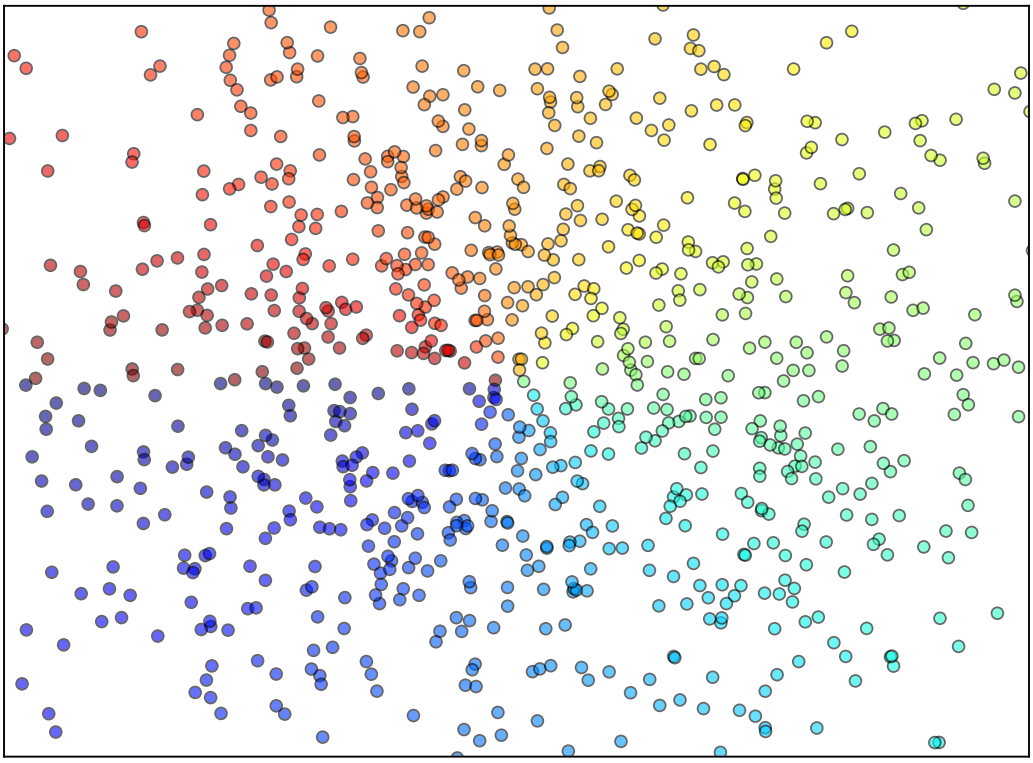

## VII. EJERCICIO 11

A partir de la definición de la función y del código del punto 10, generé la figura que muestro en esta sección. Para generar las distintas superficies de colores utilicé la función contourf con mapa de colores "hot" y transparencia 0.75. Para hacer las líneas que delimitan los niveles utilicé la función contour. Para agregar las etiquetas asociadas a cada nivel utilicé la función clabel. Estas 3 funciones pertenecen a la librería matplotlib.



## VIII. EJERCICIO 12

A partir del código dado en la consigna generé un gráfico utilizando scatter de matplotlib. Para ello tuve en cuenta el tamaño de los puntos (s=25) y la transparencia de los mismos (alpha=0.6). Por otro lado, el color de cada punto lo asigné según su ángulo en coordenadas polares usando np.arctan2(Y, X) de numpy.



Al asignar los colores según el ángulo polar, es posible distinguir con mayor facilidad los 4 cuadrantes del gráfico. Además, debido a la transparencia de los puntos, podemos observar que hay ciertas regiones donde los colores son más intensos, asociado a una acumulación de más puntos. De este modo, lo que podemos determinar de este gráfico es una distribución de probabilidad asociada a los valores generados por la función pseudoaleatoria np.random.normal(). Así, las regiones donde los colores son más intensos pueden asociarse a las regiones en donde hay una mayor probabilidad de hallar un punto generado por la función np.random.normal().

## IX. EJERCICIO 13

Para poder simular el problema en 2D armé una clase llamada R2. Los atributos de dicha clase son las coordenadas x e y. Los métodos de dicha clase definen la forma en que se imprimen en pantalla los objetos de R2 ($\_\_str\_\_$) y como se realizarán las operaciones (suma, resta, multiplicación por un escalar, división por un escalar y módulo de un vector). Para definir las operaciones básicas recurrí a los métodos especiales $\_\_add\_\_$, $\_\_sub\_\_$, $\_\_mul\_\_y\_\_truediv\_\_$.

A continuación, definí la clase Pez. Los objetos de dicha clase tienen dos atributos: posición y velocidad. Ambos atributos son objetos de la clase R2.

La última clase que definí fue la clase Cardumen. Los atributos de los objetos de dicha clase son: el tamaño del espacio en que se moverán los peces "size", el número de peces en el cardumen "N_peces", un array de "N_peces" objetos de la clase Pez, la velocidad máxima y la distancia máxima a considerar para calcular $dv_2$. Estos dos últimos atributos se definieron como privados utilizando doble guión bajo.

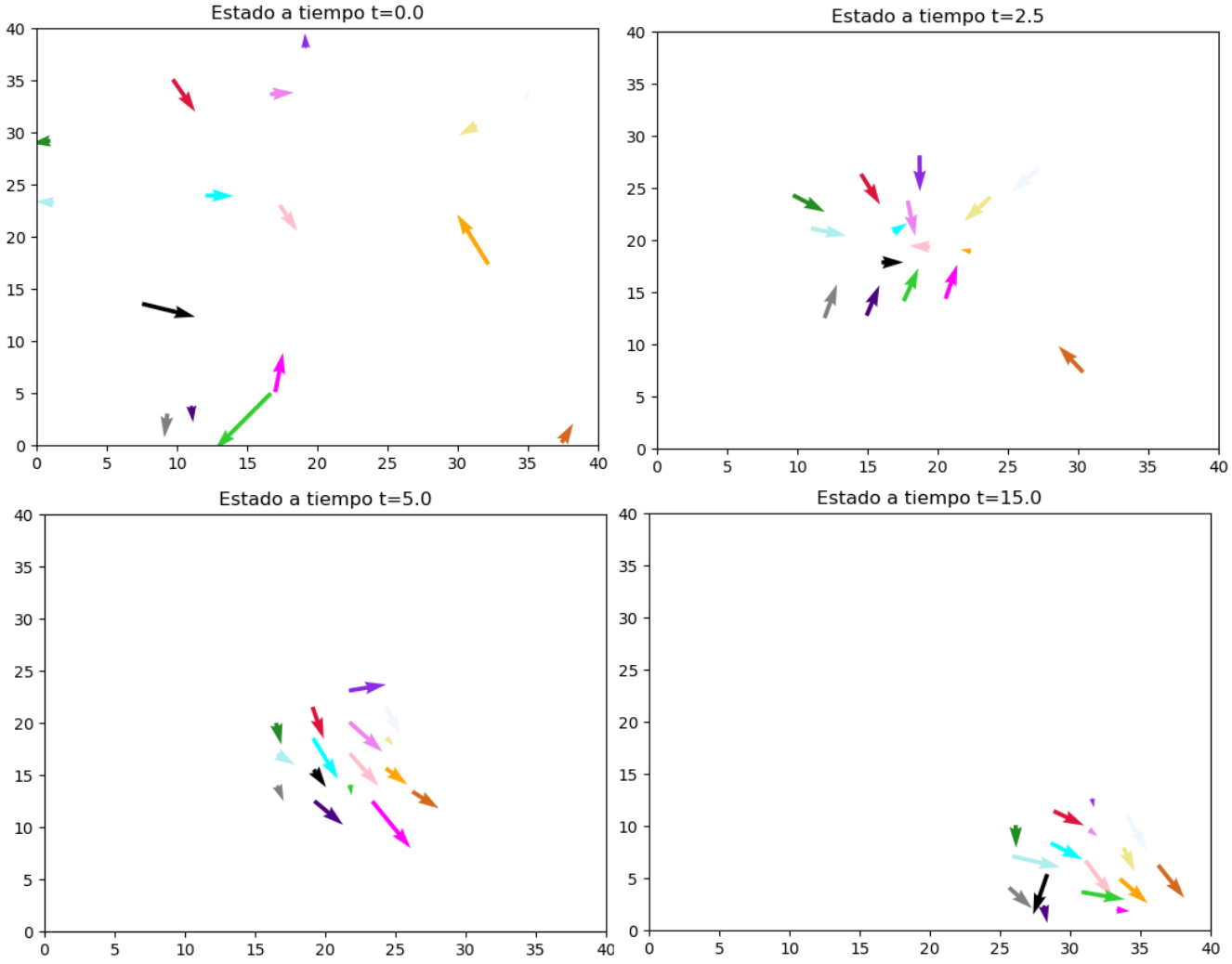
Para inicializar la posición de los peces, le asigné a cada pez una posición pseudoaleatoria entre 0 y size usando np.random.uniform(0,size). La velocidad también es inicializada de una manera similar pero teniendo en cuenta que su módulo no puede superar la velocidad máxima indicada.

Los métodos de la clase cardumen son:

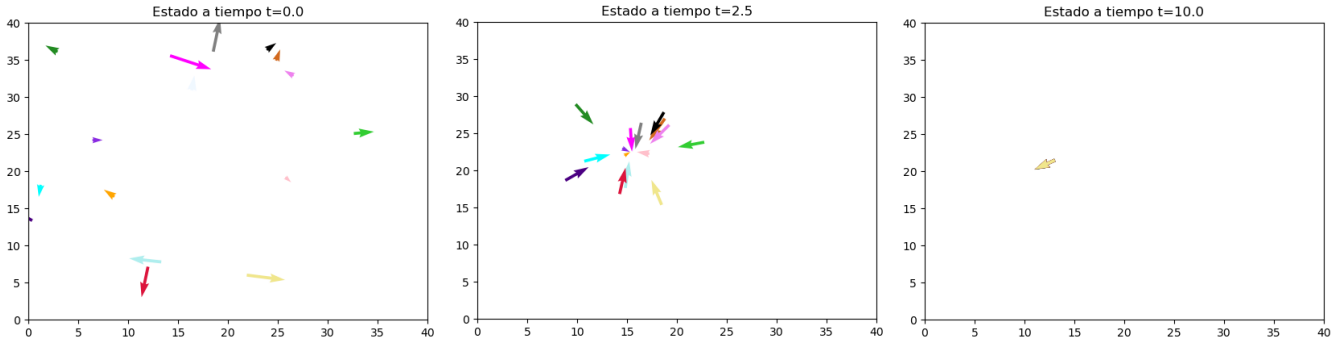
- meanR: Calcula el valor medio de la posición de los peces.

- meanV: Calcula el valor medio de la velocidad de los peces.
-v_actualizar: Calcula dv según las reglas requeridas por la consigna. Además, verifica que al actualizar, la velocidad no supere la velocidad máxima.
-rebote: Consideré condiciones de contorno rígidas para el problema. Este método permite que los peces "reboten" al alcanzar uno de los extremos del cuadrado de lado size.
-doStep: Actualiza las posiciones y velocidades. Aquí se llama al método rebote. El valor de dt está definido al inicio del código como una variable global.
-print_gif: Imprime un video de la dinámica de los peces. Utiliza la función quiver de matplotlib.
-print: Genera gráficos del estado de los peces cada n_it iteraciones. También utiliza la función quiver.

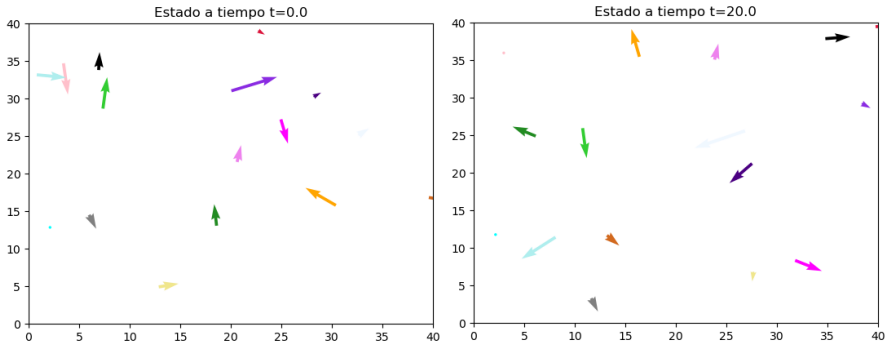
Para inicializar el cardumen utilicé: size=40, maxDist=3, maxV=4 y N_peces=16. Además, utilicé dt=0.25 y número de iteraciones niter=100. En el siguiente gráfico muestro los resultados que obtuve para distintos instantes de tiempo. Vemos que para t=0, los peces empiezan con posiciones y velocidades arbitrarias. A medida que avanzan las iteraciones, vemos que los peces tienden a agruparse y a moverse en el espacio como un conjunto.



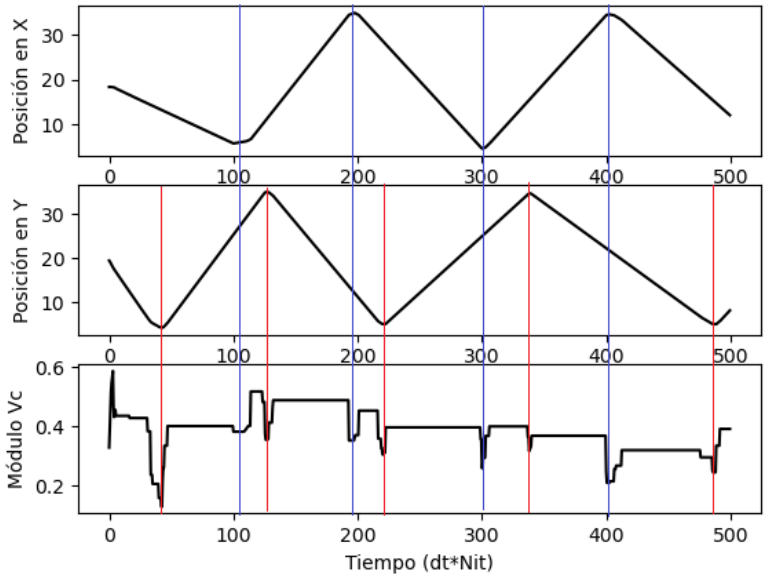

A modo de analizar en más detalle los efectos de cada una de las tres reglas para dv, analicé dos casos. En el primer caso, solo incluí las variaciones de velocidad $dv_1$ y $d_3$. El resultado de esto, como se observa en el siguiente gráfico, es que la posición de los peces converge al centro de masa y todas sus velocidades serán tenderán a la velocidad media.




A continuación, solo tuve en cuenta la variación de velocidad $dv_2$. Como es de esperarse, esta componente de velocidad produce que los peces tiendan a alejarse entre sí. Esto se ve claramente en el siguiente gráfico.




Otro factor que me pareció interesante de analizar, fue las variaciones en la velocidad media en función del tiempo causadas por la condición de paredes rígidas. Para ello, aumenté la cantidad de iteraciones a 1000 y dt=0.5. En el siguiente gráfico podemos ver, en función del tiempo, la velocidad media junto con la variación de las posiciones medias en x y en y. Con líneas verticales azules indico los momentos donde los peces alcanzan las paredes en la dirección x. Análogamente, con líneas verticales rojas indico los instantes de tiempo cuando los peces "chocan" con paredes en la dirección y. Podemos ver claras correlaciones entre disminuciones en la velocidad media del cardumen y el acercamiento de los peces a los límites rígidos.



## X. EJERCICIO 14

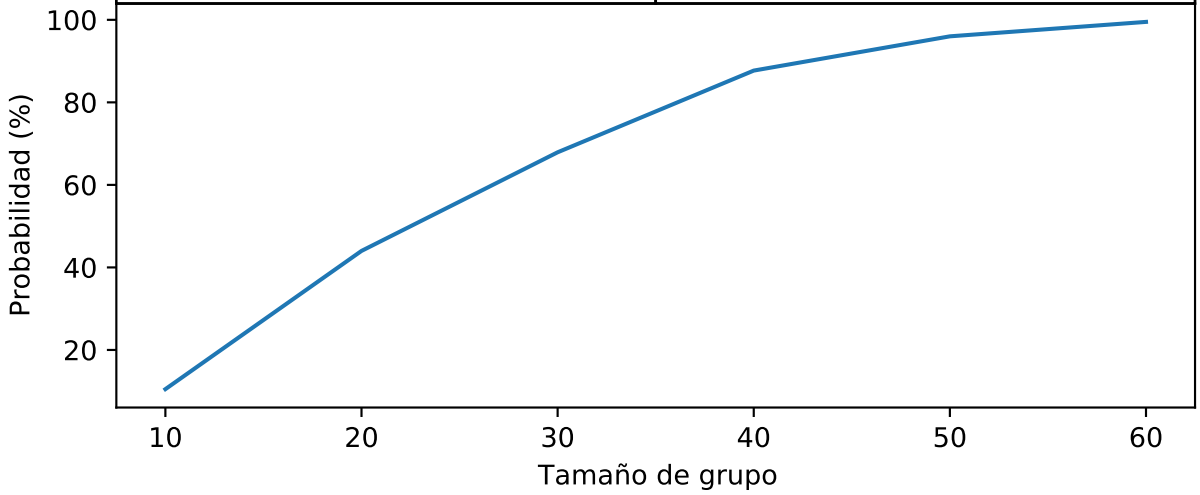
Para resolver el punto 14, lo primero que hice fue crear una clase llamada Grupo. Dicha clase tiene un único atributo llamado gente, que corresponde a un vector con cantidad de elementos igual al número de personas en el grupo. Cada elemento de dicho vector correspondería al cumpleaños de cada persona. Este es un número pseudoaleatorio entero entre 1 y 365, generado con np.random.uniform().

La clase Grupo tiene un método llamado Coincidencia, en el cual verifico si al menos dos personas en el mismo grupo tienen el mismo cumpleaños. Esto lo logro casteando a tipo set la lista de cumpleaños, eliminando los elementos repetidos. Si la diferencia entre la cantidad de personas en el grupo y la cantidad de elementos en el set es mayor que 0, significa que al menos hay un cumpleaños repetido. Si hay un cumpleaños repetido retorna 1 (caso favorable) y, si no lo hay, 0.

Por último, generé una función llamada Probablilidad. En esta función se considera la cantidad de grupos a evaluar con un tamaño específico de personas N. Dentro de la función, para cada iteración, genero un objeto de la clase Grupo con N personas y llamo al método Coincidencia. Con np.sum() sumo la cantidad de casos favorables durante todas las iteraciones y, finalmente, lo divido por la cantidad de grupos evaluados.

Para generar el gráfico y las tablas presentadas, consideré 1000 grupos de un tamaño fijo de personas. La cantidad de personas por grupo que tomé varió entre 10 y 60, aumentando de a 10.

| Cantidad de personas por grupo | Probabilidad (%) |
|---|---|
| 10.0 | 10.5 |
| 20.0 | 44.0 |
| 30.0 | 67.9 |
| 40.0 | 87.7 |
| 50.0 | 96.0 |
| 60.0 | 99.5 |




## XI. EJERCICIO 15

Para el ejercicio 15, implementé una clase llamada Noiser. Se inicializa asignando 2 umbrales minV y maxV que serán indicados por el usuario (yo usé 2 y 5 respectivamente). Tanto minV como maxV fueron designados como privados, utilizando un doble guión bajo durante la asignación. Luego, utilizando el método *__call__* generé un método tal que al recibir un valor escalar x, suma a x una componente pseudoaleatoria generada con np.random.rand() entre los umbrales mencionados.

Luego, utilizando np.vectorize(Noiser(minV,maxV)) permito que pueda aplicarse el functor Noiser a cada elemento de un array. Verifiqué el correcto funcionamiento con x = np.linspace(1, 10, 10).