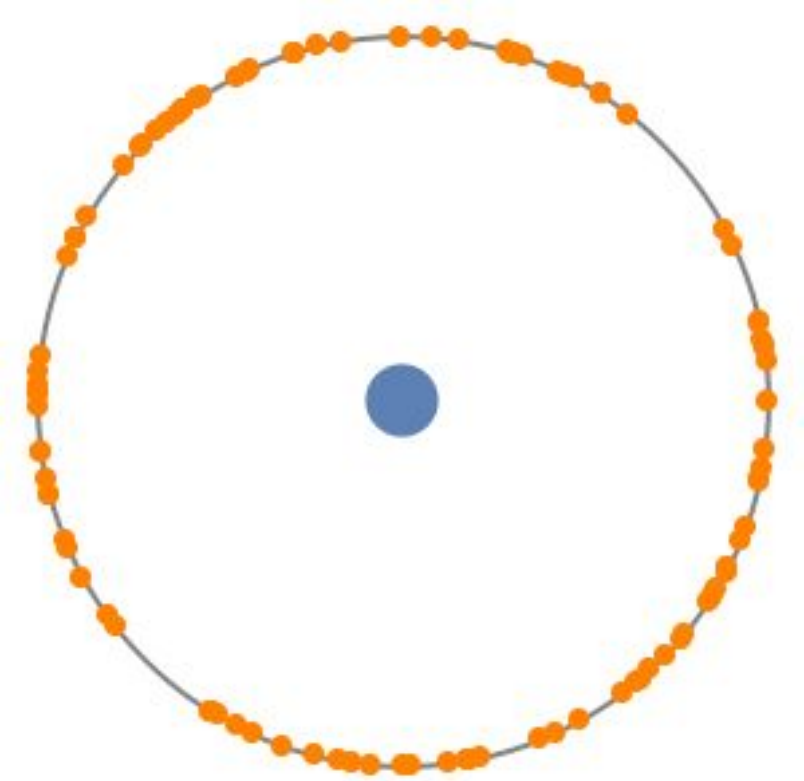
Модель Курамото



Научный руководитель: М.В. Царьков

Участники проекта

В проекте принимали участие:

- Владислав Ухватов
- Сергей Кузьменков
- Лев Оганисян
- Артем Воронин
- Григорий Дашков
- Георгий Хлевтов

Цукина Курамото

Введение

Чтобы достичь поставленную цель **необходимо**:

1. Подробно изучить язык программирования Python и его особенности
2. Изучить библиотеки Python, которые могут понадобиться для проекта
3. Составить план действий
4. Протестировать готовый проект и в случае необходимости сделать нужные доработки

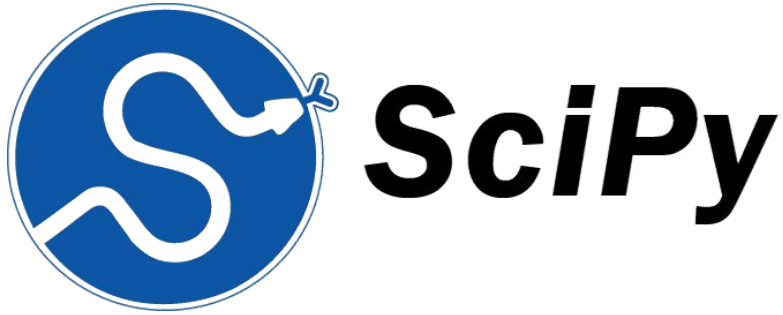
Актуальность проекта

Синхронизация – одно из фундаментальных нелинейных явлений природы. Это явление можно рассматривать как метод самоорганизации взаимодействующих систем.

Во многих естественных ситуациях взаимодействуют более двух объектов. Поэтому, если два осциллятора способны к подстройке частоты, то можно ожидать такой способности от большого количества осцилляторов.

Актуальность проекта

Для решения нашей задачи будут использованы ведущие методы численного моделирования и графического построения, основывающиеся на таких библиотеках Python, как Django, Networkx, Scipy, Numpy



С помощью этих библиотек мы сможем воспроизвести нашу задачу в реальность. Мы напишем веб сервис, который через модель Курамото будет описывать поведение линейных гармонических осцилляторов с затуханием, иначе можно сказать “физических маятников”.

Постановка задачи

В проекте стоял ряд задач необходимых к выполнению, для хорошей реализации, мы должны были распределить обязанности между небольшими командами в нашей группе. Наш конечный план действий:

1. Собрать сведения о модели Курамото.
2. Написать GUI для взаимодействия пользователя с интерфейсом.
3. Перенести описание модели в Python.
4. Соединить распределенные между членами команды задания воедино, собрав окончательную версию проекта.
5. Радоваться результату.😁

Расчёт

- Самая популярная форма модели имеет следующие управляющие уравнения:

$$\frac{d\theta_i}{dt} = \omega_i + \frac{K}{N}\sum_{j=1}^{N}\sin(\theta_j - \theta_i), \qquad i = 1 \dots N$$

Где система состоит из *N* осцилляторов предельного цикла с фазами θ_i и константа *K*

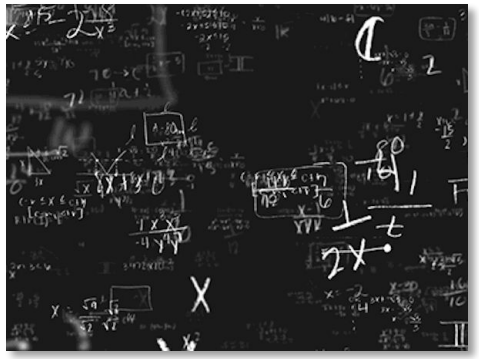
- В систему можно добавить шум. В этом случае исходное уравнение изменяется на:

$$\frac{d\theta_i}{dt} = \omega_i + \zeta_i + \frac{K}{N}\sum_{j=1}^{N}\sin(\theta_j - \theta_i)$$

Где ζ_i это колебание и функция времени.

- После преобразований основное уравнение становится:

$$\frac{d\theta_i}{dt} = \omega_i - Kr\sin(\theta_i)$$



Наш код

Это основной класс модели Курамото, который и отвечает за расчеты

```python
class Kuramoto:
    def __init__(self, coupling: float = 1, dt: float = 0.01, total_time: float = 10, nodes_count=None,
                 vibration_array=None):
        if nodes_count is None and vibration_array is None:
            raise ValueError("nodes_count or vibration_array must be specified")

        self.dt = dt
        self.total_time = total_time
        self.coupling = coupling

        if vibration_array is not None:
            self.natfreqs = vibration_array
            self.nodes_count = len(vibration_array)
        else:
            self.nodes_count = nodes_count
            self.natfreqs = normal(size=self.nodes_count)

    def derivative(self, angles_vector, time, connectivity_matrix):
        assert len(angles_vector) == len(self.natfreqs) == len(
            connectivity_matrix), 'Input dimensions do not match, check lengths'
        angles_i, angles_j = meshgrid(angles_vector, angles_vector)
        dxdt = self.natfreqs + self.coupling / connectivity_matrix.sum(axis=0) * (
                connectivity_matrix * sin(angles_j - angles_i)).sum(axis=0)
        return dxdt

    def integrate(self, angles_vector, connectivity_matrix):
        time = linspace(0, self.total_time, int(self.total_time / self.dt))
        time_series = odeint(self.derivative, angles_vector, time, args=(connectivity_matrix,))
        return time_series.T

    def run(self, connectivity_matrix=None, angles_vector=None):
        assert (connectivity_matrix == connectivity_matrix.T).all(), 'connectivity_matrix must be symmetric'
        if angles_vector is None:
            angles_vector = self.init_angles()
        return self.integrate(angles_vector, connectivity_matrix)

    def mean_frequency(self, activity_matrix, connectivity_matrix):
        assert len(connectivity_matrix) == activity_matrix.shape[0], 'connectivity_matrix does not match act_mat'
        _, steps_count = activity_matrix.shape
        dxdt = zeros_like(activity_matrix)
        for time in range(steps_count):
            dxdt[:, time] = self.derivative(activity_matrix[:, time], time, connectivity_matrix)
        return numpy_sum(dxdt * self.dt, axis=1) / self.total_time

    @staticmethod
    def phase_coherence(angles_vector):
        total = sum([(E ** (1j * i)) for i in angles_vector])
        return abs(total / len(angles_vector))

    def init_angles(self):
        return 2 * PI * random(size=self.nodes_count)
```

Наш код

```python
class KuramotoHandler:
    def __init__(self, data: dict, handler=None, time=None):
        if handler is not None:
            self.handler = handler
        if time is not None:
            self.time = time
        self.data = data
        self.objects_name = tuple(data['objects'].keys())

    def connectHandler(self, handler):
        self.handler = handler
        return self

    def setTime(self, time):
        self.time = time
        return self

    def build(self):
        return self.__build(self.data['objects'])

    def __calculate(self, vibration_array: list, fps: int = 60):
        model = Kuramoto(coupling=3, dt=0.01, total_time=self.time, vibration_array=vibration_array)
        calculations = model.run(connectivity_matrix=to_array(to_binomial(n=len(self.objects_name), p=1)))
        return self.handler(matrix=calculations, names=self.objects_name).collect(fps * self.time)

    def __build(self, objects):
        vibration_array = [objects[self.objects_name[object_index]]['frequency'] for object_index in
                           range(len(objects))]
        return self.__calculate(vibration_array=vibration_array, fps=self.data['fps'])
```

Так же наша модель состоит из его хендлера, который отвечает за обработку данных, перед тем, как они передадутся в класс модели

Наш код

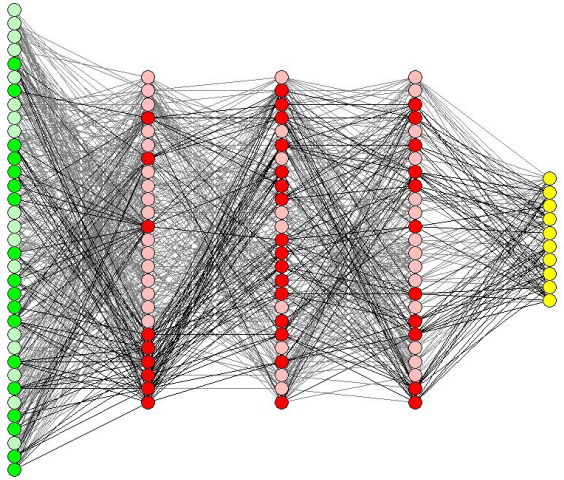
Некоторые из наших функций

```
def derivative(self, angles_vector, time, connectivity_matrix):
    assert len(angles_vector) == len(self.natfreqs) == len(
        connectivity_matrix), 'Input dimensions do not match, check lengths'
    angles_i, angles_j = meshgrid(angles_vector, angles_vector)
    dxdt = self.natfreqs + self.coupling / connectivity_matrix.sum(axis=0) * (
                connectivity_matrix * sin(angles_j - angles_i)).sum(axis=0)
    return dxdt

def integrate(self, angles_vector, connectivity_matrix):
    time = linspace(0, self.total_time, int(self.total_time / self.dt))
    time_series = odeint(self.derivative, angles_vector, time, args=(connectivity_matrix,))
    return time_series.T

def run(self, connectivity_matrix=None, angles_vector=None):
    assert (connectivity_matrix == connectivity_matrix.T).all(), 'connectivity_matrix must be symmetric'
    if angles_vector is None:
        angles_vector = self.init_angles()
    return self.integrate(angles_vector, connectivity_matrix)

def mean_frequency(self, activity_matrix, connectivity_matrix):
    assert len(connectivity_matrix) == activity_matrix.shape[0], 'connectivity_matrix does not match act_mat'
    _, steps_count = activity_matrix.shape
    dxdt = zeros_like(activity_matrix)
    for time in range(steps_count):
        dxdt[:, time] = self.derivative(activity_matrix[:, time], time, connectivity_matrix)
    return numpy_sum(dxdt * self.dt, axis=1) / self.total_time

@staticmethod
def phase_coherence(angles_vector):
    total = sum([(E ** (1j * i)) for i in angles_vector])
    return abs(total / len(angles_vector))

def init_angles(self):
    return 2 * PI * random(size=self.nodes_count)
```

Результаты расчёта

Заключение

Синхронизация является развивающейся областью исследований и останется популярной в течение долгого времени. Именно по этим причинам данная модель и была использована в данной работе.



Исследование синхронизации в нейронных сетях представляет собой актуальную проблему, которая является основой построения мощных вычислительных сетей.

Написанный нами в ходе этого проекта сервис, в теории, может быть полезен для тех, кто без длинных и нудных расчетов хочет познакомиться с таким чудным явлением, как синхронизация.

Заключение

Данная работа опубликована в открытом источнике astromodel.ru

Прочитать нашу статью о модели Курамотор можно по данной ссылке

Спасибо за внимание!)