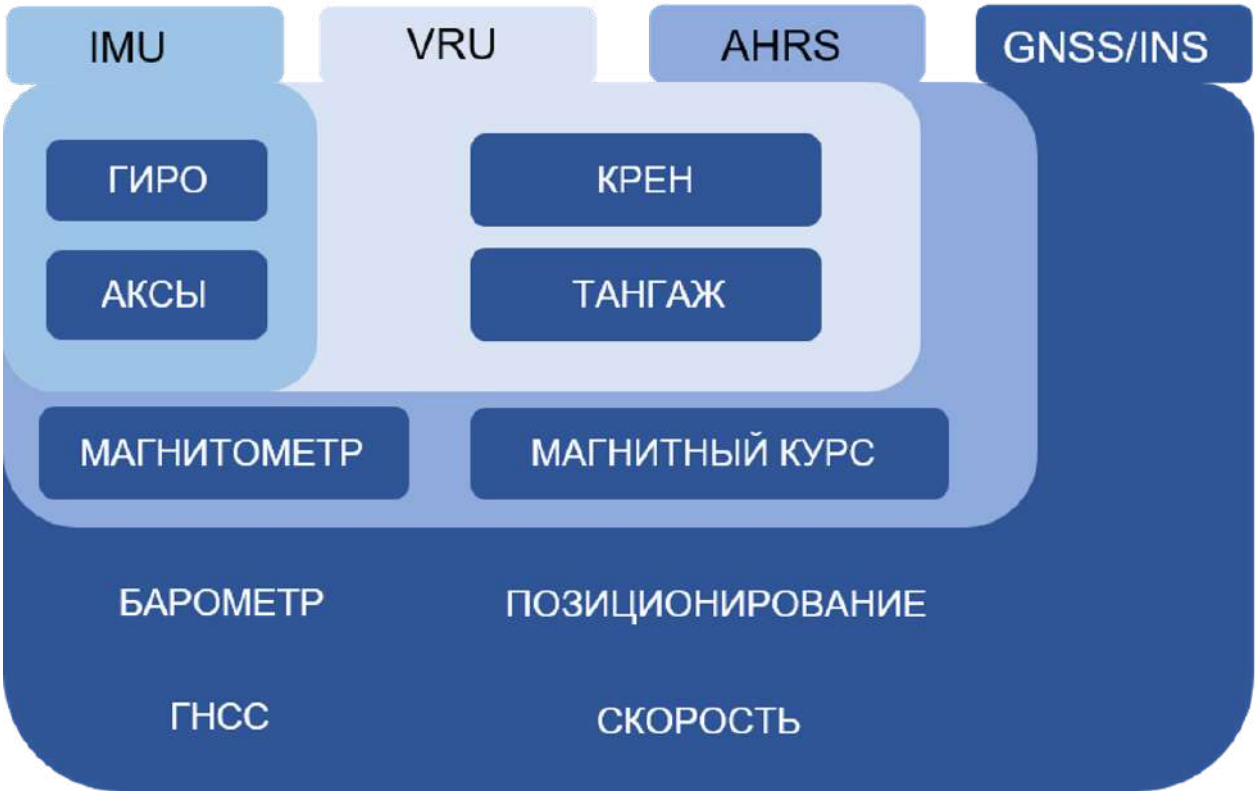
Четырнадцатое практическое занятие

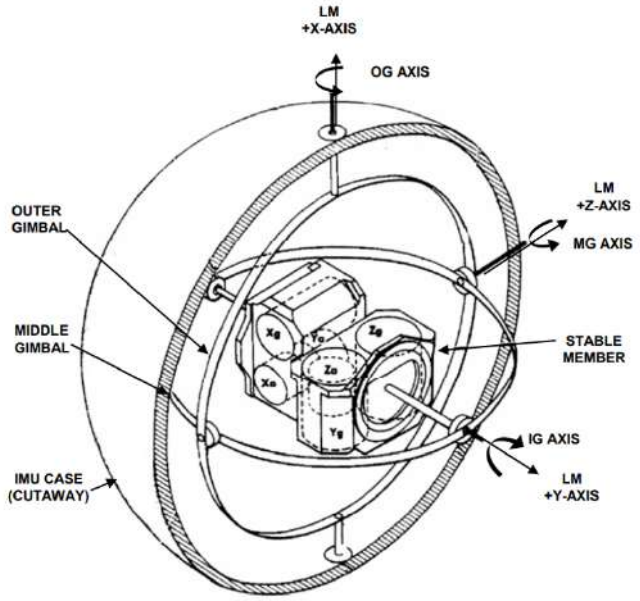
Работа с IMU, разработка VRU

Инерционные датчики используются как компонент навигационных модулей, определяющих положение устройств в пространстве:



IMU (Inertial Measurement Unit)

IMU (Inertial Measurement Unit) – это датчик, который измеряет **ускорение** и **угловую скорость** объекта, а в некоторых случаях еще и его **ориентацию** в пространстве.



Основные компоненты IMU:

1. **Акселерометр** – измеряет линейное ускорение по осям (X, Y, Z).
2. **Гироскоп** – определяет угловую скорость (вращение вокруг осей).

Применение IMU:

- **Дроны и робототехника** – стабилизация полета, навигация.
- **Смартфоны и VR/AR-устройства** – поворот экрана, трекинг движений.
- **Автомобили** – системы стабилизации, беспилотные технологии.
- **Авиация и космос** – инерциальные навигационные системы.

Недостатки:

- **Накопление ошибок (дрейф)** – из-за интегрирования шумов датчиков позиция со временем "уплывает".
- **Требует калибровки** – чувствителен к температурным изменениям и помехам.

VRU (vertical reference unit)

В результате комплексной обработки данных с акселерометров и гироскопов становится возможной создание **VRU (vertical reference unit)** - системы, которая

определяет **вертикальное положение** (ориентацию) объекта относительно силы тяжести. Она используется для измерения **крена (roll)** и **тангажа (pitch)** объекта, но обычно не учитывает **рыскание (yaw)**.

Основные функции VRU:

- Измеряет наклон объекта относительно земной вертикали (используя гравитацию).
- Обычно включает **акселерометры** и **гироскопы**.
- Может быть частью более сложных систем, таких как **AHRS (Attitude and Heading Reference System)**.

Ограничения VRU:

- Не определяет **рыскание (курс)** – для этого нужен магнитометр или GPS.
- Точность зависит от вибраций и ускорений (как и у IMU).

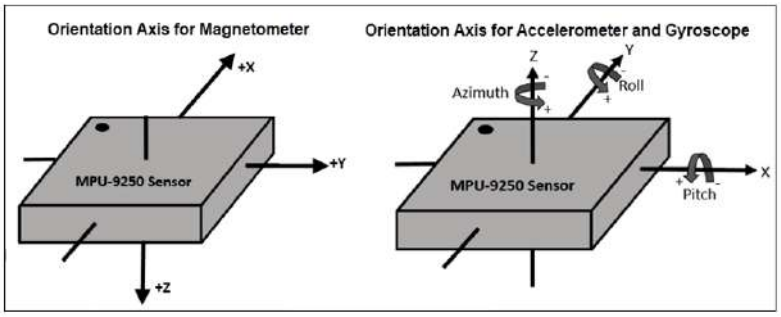
Более продвинутые системы (например, **AHRS – Attitude and Heading Reference System**) комбинируют IMU с фильтрами (например, **Калмана**) для более точного определения ориентации.

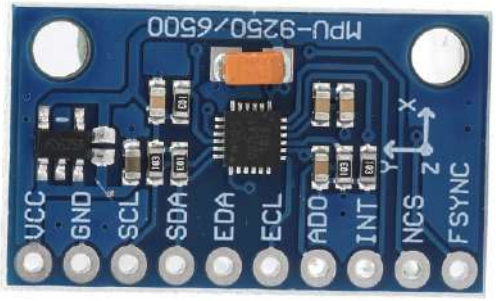
AHRS (Attitude and Heading Reference System)

AHRS (Attitude and Heading Reference System) – это система, которая определяет **полную ориентацию объекта в пространстве**, включая:

- **Крен (roll)** – наклон влево/вправо.
- **Тангаж (pitch)** – наклон вперёд/назад.
- **Рыскание (yaw)** – направление (курс) относительно севера.

В отличие от **IMU** (которая просто измеряет ускорение и вращение) и **VRU** (которая определяет только крен и тангаж), AHRS **вычисляет точную 3D-ориентацию**, комбинируя данные нескольких датчиков.





Из чего состоит AHRS?

1. **Гироскоп** – измеряет угловую скорость (вращение).

2. **Акселерометр** – определяет ускорение и гравитационный вектор (чтобы найти "вертикаль").
3. **Магнитометр** – служит цифровым компасом, определяя курс (yaw) относительно магнитного севера.
4. **Алгоритмы фильтрации** (чаще всего **фильтр Калмана**) – устраняют шумы и дрейф гироскопов.

Проблемы AHRS

1. **Магнитные помехи** – металл или электромагнитные поля искажают показания магнитометра.
2. **Дрейф гироскопа** – со временем накапливается ошибка (фильтр Калмана ее уменьшает, но не устраняет полностью).
3. **Вибрации** – могут влиять на точность акселерометра.

Для более точной навигации AHRS часто комбинируют с **GNSS** и **барометром**.

Вывод:

- **Если нужно просто ускорение и вращение → IMU.**
- **Если нужен крен и тангаж без курса → VRU.**
- **Если нужна полная 3D-ориентация → AHRS.**

Работа с датчиком MPU6050

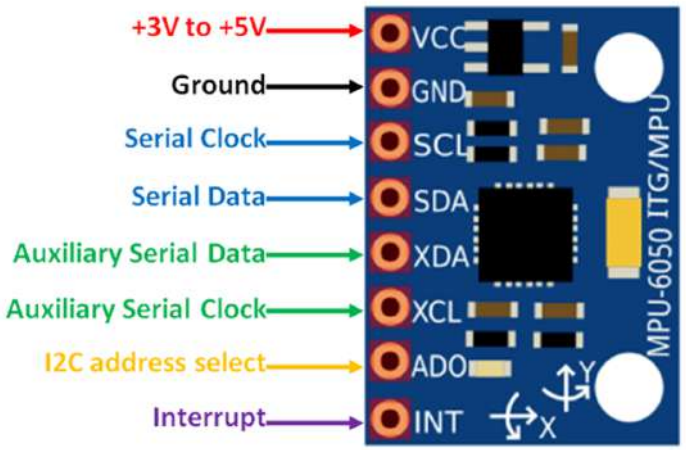
Характеристики датчика

MPU-6050 – это популярный **IMU-датчик (Inertial Measurement Unit)** от InvenSense (ныне TDK), который объединяет **3-осевой гироскоп** и **3-осевой акселерометр** в одном чипе. Благодаря низкой цене и простоте подключения, он широко используется в робототехнике, дронах, VR-устройствах и других проектах, где требуется измерение движения.

Основные техтнические характеристики

Параметр	Значение
Акселерометр	3 оси (±2g, ±4g, ±8g, ±16g)
Гироскоп	3 оси (±250, ±500, ±1000, ±2000 °/c)
Интерфейс	I²C (по умолчанию адрес **0x68**), можно SPI
Напряжение	3.3–5 В
Частота вывода	До 1 кГц
Дополнительно	Встроенный **DMP (Digital Motion Processor)** для расчёта углов

Параметр	Значение
Температура	-40°C до +85°C



Библиотека для работы с датчиком

В качестве исходного кода для работы с датчиком MPU 6050 были взяты материалы из этого репозитория.

Интересующийся читатель может самостоятельно изучить содержание этого репозитория и вдохновиться множеством примеров для работы с различными датчиками и RP Pico.

⚠ Однако в рамках наших занятий я рекомендую воспользоваться незначительно исправленным и дополненным кодом, который Вы можете скачать, перейдя по этой ссылке

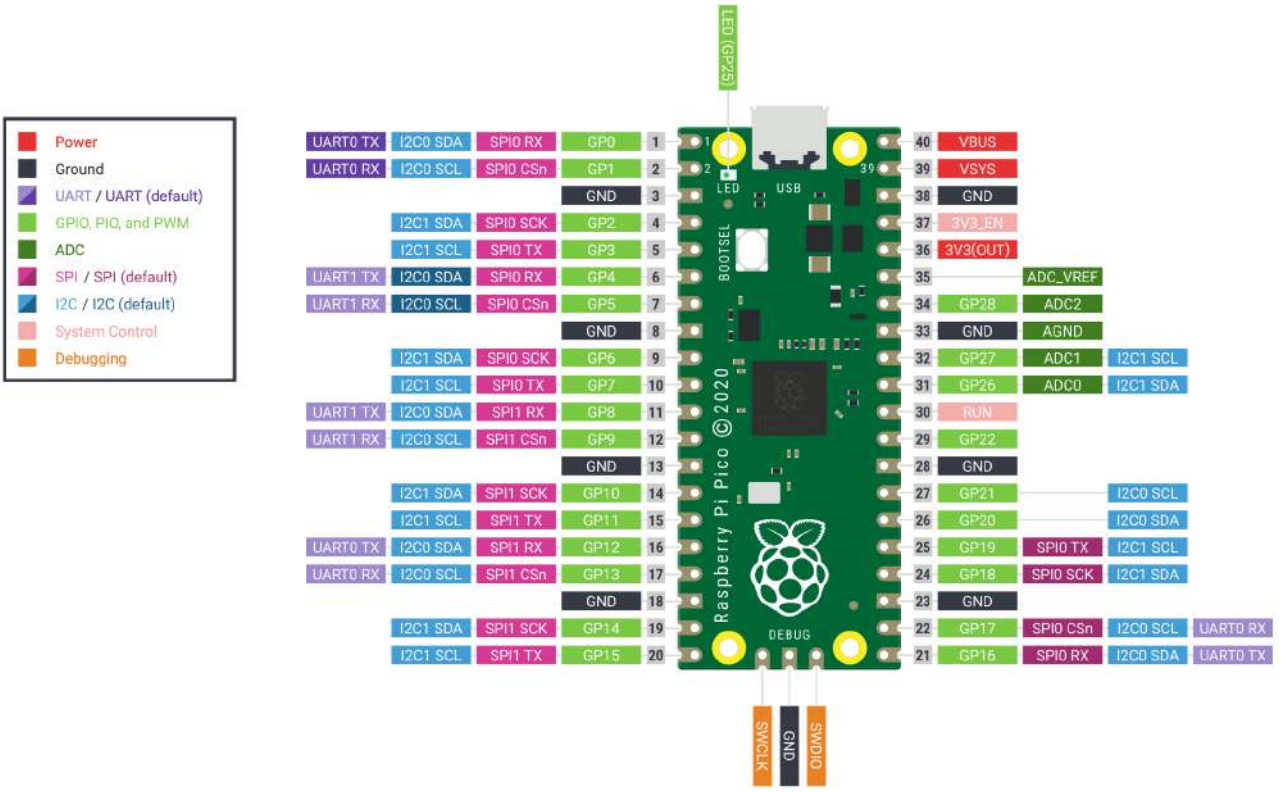
ⓘ В дальнейшем скаченный файл `imu.py` необходимо поместить в корень ваше рабочего проекта и записать в память микроконтроллера (аналогично файлу `ssd1306.py` с которым мы работали при подключении экрана)

Схема подключения

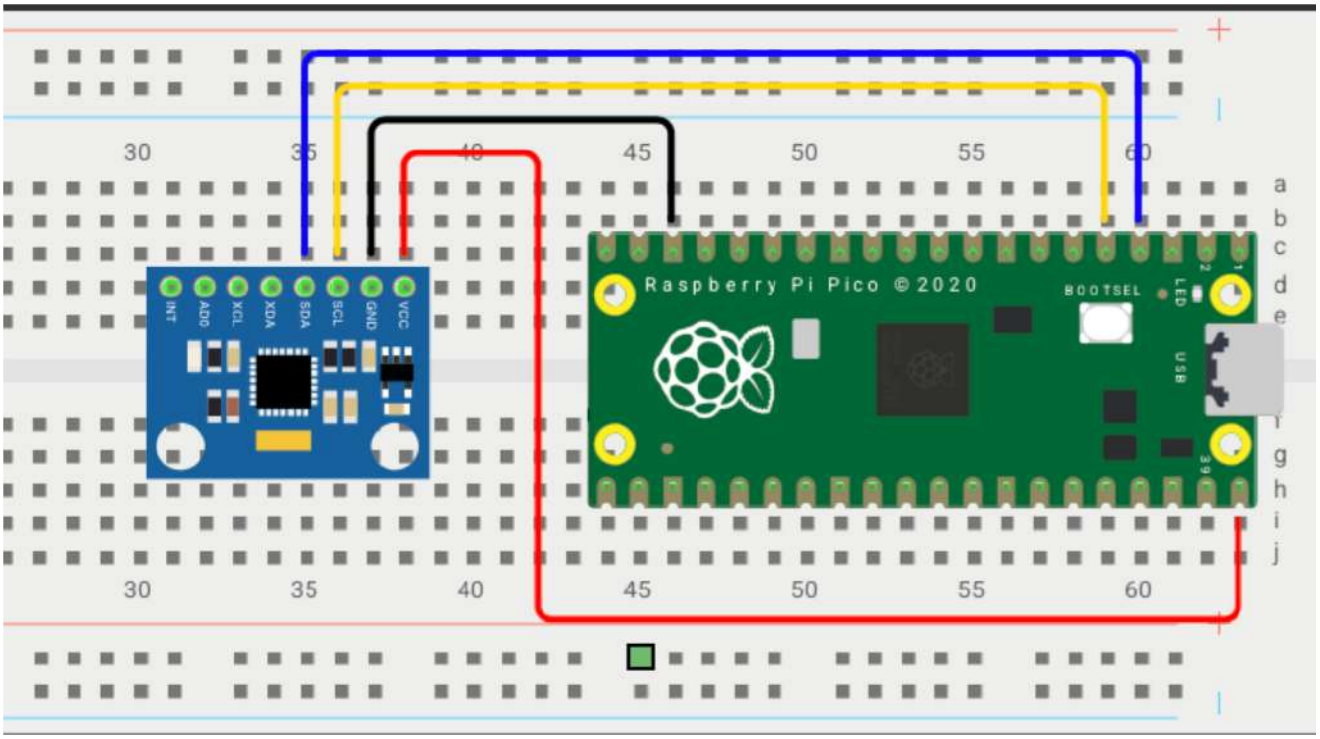
Датчик MPU 6050 подключается по I2C соединению (аналогично дисплею ssd1306).

Напомним, что для подключения устройства по I2C необходимо присоединить его к SDA и SDL шинам.

RP Pico поддерживает работу двух I2C независимых I2C шин, что следует из хорошо известной нам схемы подключения:



В дальнейшей работе будем использовать подключение к первой `1` I2C шине на пинах 2 и 3:



Код для инициализации I2C соединения по указанной схеме:

```
from machine import I2C, Pin
```

```
I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))
```

Получение данных с датчика

Получить "сырые" данные с датчика можно запустив следующий код:

```
from imu import MPU6050
from machine import I2C, Pin
import time

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

mpu = MPU6050(i2c)

while True:
    x_accel, y_accel, z_accel = mpu.accel.xyz
    x_gyro, y_gyro, z_gyro = mpu.gyro.xyz

    print(time.ticks_ms() / 1000)
    print(f"ACCEL: {x_accel, y_accel, z_accel}")
    print(f"GYRO: {x_gyro, y_gyro, z_gyro}")
    print()
    time.sleep(.05)
```

> Отметим, что работа с датчиком осуществляется через класс `MPU6050`, который загружается из предварительно установленного на микроконтроллер файла `imu.py`:

```
from imu import MPU6050
```

после чего устройство необходимо инициализировать, передав в него объект предварительно созданного i2C соединения:

```
mpu = MPU6050(i2c)
```

Далее в бесконечном цикле будет происходить опрос датчика:

```
x_accel, y_accel, z_accel = mpu.accel.xyz
x_gyro, y_gyro, z_gyro = mpu.gyro.xyz
```

И вывод полученной информации в консоль стандартным способом.

Но разберем немного подробнее методы получения информации с датчика. Датчик MPU 6050 способен выдать нам информацию с встроенных в него трехосевых акселерометров (через атрибут `accel`) и трехосевых гироскопов (через атрибут `gyro`).

Оба эти атрибута возвращают нам экземпляры класса `Vector3d`, из которого мы можем получить интересующие нас компоненты ускорений и угловых скоростей по осям (через атрибуты `x`, `y` и `z` или все сразу через атрибут `xyz` (как в предлагаемом примере)).

После получения информации с датчика (`mpu.accel.xyz` и `mpu.gyro.xyz`) значения из векторов распаковываются в соответствующие им переменные, через которые мы будем выполнять всю последующую работу.

Расчет крена и тангажа

Что это за такие странные названия?

Крен (roll) и **тангаж (pitch)** – это углы, описывающие **ориентацию объекта в пространстве** (самолёта, дрона, корабля и т. д.).

1. Крен (Roll) – наклон влево/вправо

- **Определение:** Угол поворота объекта вокруг его **продольной оси** (нос–хвост).
- **Пример:** Когда самолёт "заваливается" на крыло в повороте – это крен.
- **Откуда название:** Происходит от голландского *"krengen"* – "наклонять, кренить". В русский язык пришло через морскую терминологию.

2. Тангаж (Pitch) – наклон вперёд/назад

- **Определение:** Угол поворота вокруг **поперечной оси** (крылья).
- **Пример:** Когда самолёт **задирает нос** вверх или **опускает** вниз – это тангаж.
- **Откуда название:** От французского *"tangage"* (морской термин), которое, в свою очередь, происходит от *"tangon"* – шест для управления парусом.

Откуда взялись эти термины?

Оба слова пришли из **авиации и мореплавания**:

- **Крен** изначально использовался в мореходстве (наклон корабля на бок).

- **Тангаж** тоже морской термин, но позже перекочевал в авиацию. В русский язык они попали в **XIX–XX веках** вместе с развитием техники.

Как запомнить?

- **Крен = Крыло вниз/вверх**.
- **Тангаж = Тянем нос**.

Расчет с помощью акселерометра

Акселерометр – это датчик, который измеряет **ускорение** объекта, включая гравитацию.

Принцип работы акселерометра зависит от типа, но чаще всего используется **микроэлектромеханическая система (MEMS)**:

- Внутри есть **маленькая масса (инерционный элемент)** на пружинках.
- При ускорении масса смещается, и датчик фиксирует это изменение (ёмкостной, пьезоэлектрический или другой метод).
- Данные преобразуются в электрический сигнал.

Если представить груз закрепленный по трем осям, в покое на него постоянно будет действовать ускорение свободного падения **(g ≈ 9.8 м/с²)**.

При этом оно всегда будет направлено вниз по отвесной линии.

Если при этом плавно повернуть инерциальный датчик, то вектор этого ускорения будет распределен вдоль осей так, что $A_x^2 + A_y^2 + A_z^2 = 1$ (В идеальном случае).

Зная компоненты вектора ускорения свободного падения вдоль осей, возможно рассчитать искомые углы как:

$$pitch = \arctan\left(\frac{A_y}{A_z}\right)$$

$$roll = \arctan\left(\frac{A_x}{A_z}\right)$$

Применив полученные знания, можем получить следующий код:

```
import math

from imu import MPU6050
from machine import I2C, Pin
import time

I2C_ID = 1
SDA_PIN = 2
```

```
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

mpu = MPU6050(i2c)

while True:
    x_accel, y_accel, z_accel = mpu.accel.xyz

    accel_pitch = math.degrees(math.atan(y_accel / z_accel))
    accel_roll = math.degrees(math.atan(x_accel / z_accel))

    print("Accel_pitch:", accel_pitch, "\tAccel_roll:", accel_roll,
sep="\t")

    time.sleep(.05)
```

Акселерометр позволяет получить независимые значения углов крена и тангажа в любой момент времени. Однако, точность подобных определений зачастую не является достаточной, так как на измерения будут существенно влиять вибрации и внешние ускорения, прилагаемые к датчику.

Из-за этого крен и тангаж, определяемые по акселерометру, **обладают постоянной, но большой погрешностью**!

Расчет с помощью гироскопа

В электронике (смартфоны, дроны) используются **микроэлектромеханические (MEMS) гироскопы**. Их работа основана на **эффекте Кориолиса**:

Упрощённая схема:

1. Внутри чипа есть **колеблющаяся масса** (как маятник).
2. При повороте устройства **сила Кориолиса** отклоняет массу перпендикулярно колебаниям.
3. Датчик фиксирует это отклонение и преобразует его в **угловую скорость** (градусы/секунду, °/с).

Таким образом гироскоп сам по себе не может указать нам истинные углы крена и тангажа, однако может помочь в их отслеживании, так как фиксирует **скорость изменения этих углов в каждый момент времени**.

Интегрируя эти значения, мы можем определить накопленный от стартовой точки разворот вдоль осей:

$$pitch = \sum_{t=0}^{t_n} G_y \cdot \Delta t_{i-(i-1)}$$

$$roll = \sum_{t=0}^{t_n} G_x \cdot \Delta t_{i-(i-1)}$$

Код для получения относительных углов крена и тангажа с помощью гироскопов:

```
import math

from imu import MPU6050
from machine import I2C, Pin
from time import ticks_ms, ticks_diff, sleep

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

mpu = MPU6050(i2c)

last_time = ticks_ms()
gyro_pitch, gyro_roll = 0, 0

while True:
    x_gyro, y_gyro, z_gyro = mpu.gyro.xyz
    current_time = ticks_ms()
    dt = ticks_diff(current_time, last_time) / 1000
    last_time = current_time

    gyro_pitch += y_gyro * dt
    gyro_roll += x_gyro * dt

    print("Gyro_pitch:", gyro_pitch, "\tGyro_roll:", gyro_roll, x_gyro,
y_gyro, sep="\t")

    sleep(.05)
```

Однако и здесь полученный результат сложно назвать идеальным - **ошибки в определении угловых скоростей неизбежно будут накапливаться при добавлении новых значений к предыдущему результату!**

Из-за этого значения углов будут постепенно **дрейфовать** ("уплывать") от истинных значений.

Обобщив все это, можем увидеть:

Акселерометр	Гироскоп
Измеряет **линейное ускорение**	Измеряет **угловую скорость** (вращение).
Чувствителен к **гравитации**	Не реагирует на гравитацию.
Обладает большой, но постоянной ошибкой	Быстро накапливает ошибку (дрейф), но в "моменте" точнее чем акселерометр

Из чего можем сделать вывод, что поодиночке оба эти датчика неспособны выдать нам удовлетворительное решение, однако совместная обработка данных с этих устройств может позволить нивелировать их частные недостатки.

Комбинированное решение

Скомбинировать данные с гироскопа и акселерометра в одно общее решение, можно применив к ним **алгоритмы фильтрации**:

- фильтр Калмана;
- комплементарный фильтр;
- и др.

Фильтр Калмана

Фильтр Калмана – это алгоритм математической обработки данных, который позволяет **оценивать состояние динамической системы** (например, положение, скорость, ориентацию), учитывая **шум измерений** и **неточности модели**.

Простыми словами: Это "умный" способ **объединить показания датчиков** (например, гироскопа и акселерометра) с **предсказанием движения**, чтобы получить более точный результат, чем каждый датчик по отдельности.

Как работает фильтр Калмана?

Алгоритм состоит из двух этапов: **1. Прогноз (Предсказание)** На основе **модели системы** предсказывается новое состояние.

- Например: если дрон вращается с угловой скоростью 10°/с, через 0.1 сек его угол изменится на 1°. **2. Коррекция (Обновление)** Предсказание корректируется с учетом **реальных измерений** от датчиков.
- Если гироскоп показал 9.5°/с, а акселерометр – 10.2°/с, фильтр "усреднит" эти значения, учитывая их точность.

Формулы упрощённо:

$$\begin{cases} Прогноз: x_k = A \cdot x_k - 1 + B \cdot u_k \\ Коррекция: x_k = x_k + K \cdot (z_k - H \cdot x_k) \end{cases}$$

Где:

- x_k – состояние системы (например, угол).
- z_k – измерение датчика.
- K – **коэффициент Калмана** (определяет, кому верить больше: прогнозу или датчику).

Ограничения:

- **Требует точной модели системы** – если модель неверна, фильтр будет ошибаться.
- **Вычислительная сложность** – для матричных операций нужна достаточная мощность.
- **Не работает с нелинейными системами** – для них используют **Extended Kalman Filter (EKF)** или **Unscented Kalman Filter (UKF)**.

Другими словами, фильтр Калмана оправдан тогда, когда у нас есть возможность обоснованно предсказать, как будут (или не будут) изменяться значения:

Например:

- Самолет не может вдруг двигаться хвостом вперед;
- Автомобиль - двигаться перпендикулярно направлению движения и т.п.

Так как программируемое нами VRU может изменять значения произвольно (так как ни один математик не предскажет, как нам захочется повернуть его в следующий момент), в решении нашей задачи нам целесообразнее воспользоваться более простым **комплементарным фильтром**.

Комплементарный фильтр

Complementary Filter (комплементарный фильтр) – это алгоритм, который объединяет данные **гироскопа** и **акселерометра**, чтобы получить более точную оценку угла наклона. В отличие от фильтра Калмана, он требует меньше вычислений и идеально подходит для систем с ограниченными ресурсами (например, Arduino, Rp Pico, дроны).

Принцип работы Фильтр работает по простой идее:

- **Гироскоп** точен в **краткосрочной перспективе** (нет задержек), но **дрейфует** со временем.
- **Акселерометр** точен в **долгосрочной перспективе** (не дрейфует), но **шумит** при резких движениях.

Complementary Filter "смешивает" эти данные:

$$Угол = \alpha \cdot (Предыдущий\ угол + Гироскоп \cdot dt) + (1–\alpha) \cdot Акселерометр$$

где:

- α ($0 < \alpha < 1$) – коэффициент доверия гироскопу (обычно 0.9–0.98).
- dt – время между измерениями (например, 0.01 сек при 100 Гц).

Как выбирается α?

- Чем ближе **α к 1**, тем сильнее фильтр доверяет гироскопу (меньше шума, но больше дрейфа).
- Чем ближе **α к 0**, тем больше учитывается акселерометр (меньше дрейфа, но больше шума).

Запишем все это в виде кода:

```
import math

from imu import MPU6050
from machine import I2C, Pin
from time import ticks_ms, ticks_diff, sleep

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3

i2c = I2C(id=I2C_ID,
          sda=Pin(SDA_PIN),
          scl=Pin(SCL_PIN),
          )

mpu = MPU6050(i2c)

last_time = ticks_ms()
complex_pitch, complex_roll = 0, 0

A = 0.9

while True:
    x_accel, y_accel, z_accel = mpu.accel.xyz
    x_gyro, y_gyro, z_gyro = mpu.gyro.xyz

    current_time = ticks_ms()
    dt = ticks_diff(current_time, last_time) / 1000
    last_time = current_time

    accel_pitch = math.degrees(math.atan(y_accel / z_accel))
    accel_roll = math.degrees(math.atan(x_accel / z_accel))
```

```
    complex_pitch = A * (complex_pitch + y_gyro * dt) + (1 - A) *
accel_pitch
    complex_roll = A * (complex_roll + x_gyro * dt) + (1 - A) * accel_roll

    print("Complex_pitch:", complex_pitch, "\tComplex_roll:",
complex_roll, sep="\t")
    sleep(.05)
```

Разработка класса `Level`

Пока все, что мы написали - это простой процедурный код, так как в нем много повторов, его невозможно нормально включить в работу с другими устройствами и т.д. Мы все это уже не раз обсуждали ранее...

И что же делать? Ответ у нас уже тоже есть - применит ООП!

Разработаем класс `Level`, который позволит создать нам собственный электронный уровень!

Название придумано!

Начнем:

```
class Level:
    pass
```

И что дальше? Можем ли мы унаследовать его от ранее использованного класса `MPU6050`? Технически конечно, но тем самым мы запутаем пользователя, так как дадим ему возможность получать необработанные "сырые" данные с гироскопов и акселерометров, а это плохо. К тому же мы нарушим **принцип подстановки Лисков** из SOLID, так как нам потребуется передавать значения коэффициента `a` для комплементарного фильтра...

Короче, раз наследование - это плохо, значит композиция - это хорошо!

Таким образом:

```
class Level:

    def __init__(self, side_str, a=0.95, device_addr=None, transposition=
(0, 1, 2), scaling=(1, 1, 1)):
        self._imu = MPU6050(side_str, device_addr, transposition, scaling)
        self.a = a
        self._last_time = None
        self._accel_pitch, self._accel_roll = 0, 0
```

```
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0
```

Теперь добавим в наш класс методы для расчета предварительно инициированных нулями атрибутов для крена и тангажа:

```
class Level:

    def __init__(self, side_str, a=0.95, device_addr=None, transposition=
(0, 1, 2), scaling=(1, 1, 1)):
        self._imu = MPU6050(side_str, device_addr, transposition, scaling)
        self.a = a
        self._last_time = None
        self._accel_pitch, self._accel_roll = 0, 0
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0

    def _calk_pitch_roll(self):
        accel_data = self._imu.accel.xyz
        gyro_data = self._imu.gyro.xyz
        dt = self._get_dt()
        self._get_accel_pitch_roll(accel_data)
        self._get_gyro_pitch_roll(gyro_data, dt)
        self._get_complex_pitch_roll(gyro_data, dt)

    def _get_dt(self):
        ...

    def _get_accel_pitch_roll(self, accel_data):
        ...

    def _get_gyro_pitch_roll(self, gyro_data, dt):
        ...

    def _get_complex_pitch_roll(self, gyro_data, dt):
        ...
```

При вызове метода `_calk_pitch_roll` будет выполняться опрос данных с датчика и последовательно вычисляться значения крена и тангажа по каждому из ранее рассмотренных способов!

Чего не хватает (кроме кода расчетов который здесь затерт для компактности представления кода)? Открытого интерфейса для пользователя, позволяющего получить данные с датчика!

Добавим их:

```
class Level:

    def __init__(self, side_str, a=0.95, device_addr=None, transposition=
(0, 1, 2), scaling=(1, 1, 1)):
        self._imu = MPU6050(side_str, device_addr, transposition, scaling)
        self.a = a
        self._last_time = None
        self._accel_pitch, self._accel_roll = 0, 0
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0

    ...

    def get_data_str(self):
        self._calk_pitch_roll()
        data = {"A": (self._accel_pitch, self._accel_roll),
                "G": (self._gyro_pitch, self._gyro_roll),
                "C": (self._complex_pitch, self._complex_roll),
                }
        data = json.dumps(data)
        return data

    def send_pitch_roll_data(self):
        data = self.get_data_str()
        sys.stdout.write(f"{data}\n".encode("utf-8"))
```

Метод `get_data_str` опрашивает датчик, формирует по текущим значениям углов словарь и переводит его в строку, используя JSON сериализацию.

Метод `send_pitch_roll_data` запрашивает данные в виде строки и отправляет их по серийному соединению.

Соберем все это вместе и получим следующий код:

```
from math import atan, degrees
import sys
import time
import json
from machine import Pin, I2C
from time import ticks_ms, ticks_diff
from imu import MPU6050


class Level:

    def __init__(self, side_str, a=0.95, device_addr=None, transposition=
(0, 1, 2), scaling=(1, 1, 1)):
        self._imu = MPU6050(side_str, device_addr, transposition, scaling)
```

```
        self.a = a
        self._last_time = None
        self._accel_pitch, self._accel_roll = 0, 0
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0

    def _calk_pitch_roll(self):
        accel_data = self._imu.accel.xyz
        gyro_data = self._imu.gyro.xyz
        dt = self._get_dt()
        self._get_accel_pitch_roll(accel_data)
        self._get_gyro_pitch_roll(gyro_data, dt)
        self._get_complex_pitch_roll(gyro_data, dt)

    def _get_dt(self):
        current_time = ticks_ms()
        if self._last_time is None:
            dt = 0
        else:
            dt = ticks_diff(current_time, self._last_time) / 1000
        self._last_time = current_time
        return dt

    def _get_accel_pitch_roll(self, accel_data):
        x_accel = accel_data[0]
        y_accel = accel_data[1]
        z_accel = accel_data[2]
        try:
            self._accel_pitch = degrees(atan(y_accel / z_accel))
            self._accel_roll = degrees(atan(x_accel / z_accel))
        except ZeroDivisionError:
            pass
        return self._accel_pitch, self._accel_roll

    def _get_gyro_pitch_roll(self, gyro_data, dt):
        x_gyro = gyro_data[0]
        y_gyro = gyro_data[1]
        self._gyro_pitch += y_gyro * dt
        self._gyro_roll += x_gyro * dt
        return self._gyro_pitch, self._gyro_roll

    def _get_complex_pitch_roll(self, gyro_data, dt):
        x_gyro = gyro_data[0]
        y_gyro = gyro_data[1]
        self._complex_pitch = self.a * (self._complex_pitch + y_gyro * dt)
+ (1 - self.a) * self._accel_pitch
        self._complex_roll = self.a * (self._complex_roll + x_gyro * dt) +
(1 - self.a) * self._accel_roll
        return self._complex_pitch, self._complex_roll
```

```
    def get_data_str(self):
        data = {"A": (self._accel_pitch, self._accel_roll),
                "G": (self._gyro_pitch, self._gyro_roll),
                "C": (self._complex_pitch, self._complex_roll),
                }
        data = json.dumps(data)
        return data

    def send_pitch_roll_data(self):
        self._calk_pitch_roll()
        data = self.get_data_str()
        sys.stdout.write(f"{data}\n".encode("utf-8"))


if __name__ == "__main__":
    I2C_ID = 1
    SDA_PIN = 2
    SCL_PIN = 3
    IRQ_PIN = 16

    i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))
    level = Level(i2c)

    while True:
        time.sleep(0.05)
        level.send_pitch_roll_data()
```

Скачать его можно перейдя по ссылке:

Скопировав этот код в фаил `main.py` и записав его на микроконтроллер, мы сможем увидеть в консоли поток углов рассчитанных тремя способами:

- "A" - с акселерометра;
- "G" - с гироскопа;
- "C" - комплексное решение.

Но как увидеть их нагляднее?

Нужно построить график!

К тому же мы уже делали нечто подобное пару занятий назад...

В классе мы уже отправляем данные в требуемом виде: json-строка, разделенная символом переноса на новую строку. Значит нужно только написать клиентскую часть!

Кстати вот она:

```
import math

import numpy as np
import serial
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import json

from CONFIG import RP_PORT

s = serial.Serial(port=RP_PORT, baudrate=115200)


def read_data():
    data = s.readline().strip()
    data = data.decode(encoding="utf-8")
    try:
        data = json.loads(data)
    except json.decoder.JSONDecodeError:
        return None
    return data


fig, ax = plt.subplots()

circle_90 = plt.Circle((0, 0), 90, edgecolor='black', facecolor='none',
linewidth=2)
circles = [circle_90]
for degree in range(15, 90, 15):
    circles.append(plt.Circle((0, 0), degree,
                              edgecolor='black', facecolor='none',
linewidth=1, alpha=0.25))


def init_bace_graf():
    ax.set_xlim(-100, 100)
    ax.set_ylim(-100, 100)
    ax.set_aspect('equal')
    ax.axis('off')
    ax.clear()
    for circle in circles:
        ax.add_patch(circle)
    for angle in np.arange(0, 360, 30):
        rad = np.radians(angle)
```

```
        x, y = np.cos(rad), np.sin(rad)
        ax.plot([100 * x, x], [100 * y, y], 'k-', linewidth=1, alpha=0.25)


data_type = {
    "A": "red",
    "G": "green",
    "C": "blue",
}


def update(i):
    init_bace_graf()
    data = None
    while data is None:
        data = read_data()
    for type, pr_data in data.items():
        if type not in data_type:
            continue
        pitch, roll = pr_data
        circle_imu = plt.Circle((pitch, roll), 25,
edgecolor=data_type[type], facecolor='none', linewidth=1)
        ax.add_patch(circle_imu)
        ax.scatter(pitch, roll, marker="o", s=100, c=data_type[type])


ani = animation.FuncAnimation(fig, update, interval=1)
plt.show()
```

Запустить этот код необходимо в отдельном проекте, имеющем доступ к пакетам перечисленным в преамбуле файла!

Все это делается аналогично тому, как мы разбирали при работе с серийным подключением!

> ⓘ Обратите внимание на словарь `data_type` - комментируя перед запуском файла строки в нем, Вы сможете отключать визуализацию не интересных вам источников данных.

Например:

```
data_type = {
    "A": "red",
    # "G": "green",
    "C": "blue",
}
```

Оставит для визуализации только углы с акселерометра и комплексное решение.

Ссылка на файл с кодом для визуализации:

Разработка класса `Level` с прерываниями

Вроде все в порядке, но наш код имеет решающий изъян:

⚠ Запуск расчета выполняется непосредственно во время запроса!

Чем это чревато?

- Если мы будем запрашивать данные редко - данные с гироскопа перестанут соответствовать реальным углам поворота, так как мы будем игнорировать все изменения угловых скоростей произошедшие между опросами датчика!
- Из-за этого и комплементарный фильтр перестанет работать!

Кажется: ну и что? Ведь сейчас все работает! А если нужно опрашивать данные чаще, я просто уменьшу задержку в цикле опроса датчика:

```
if __name__ == "__main__":
    I2C_ID = 1
    SDA_PIN = 2
    SCL_PIN = 3
    IRQ_PIN = 16

    i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))
    level = Level(i2c)

    while True:
        time.sleep(0.05)
        level.send_pitch_roll_data()
```

Ну попробуйте... На первый взгляд, ничего не произойдет, но при попытке визуализации данные начнут обновляться на графике с огромной задержкой из-за того, что в процессе его построения они считываются и отрисовываются построчно и график не успевает обновляться требуемое количество раз в секунду.

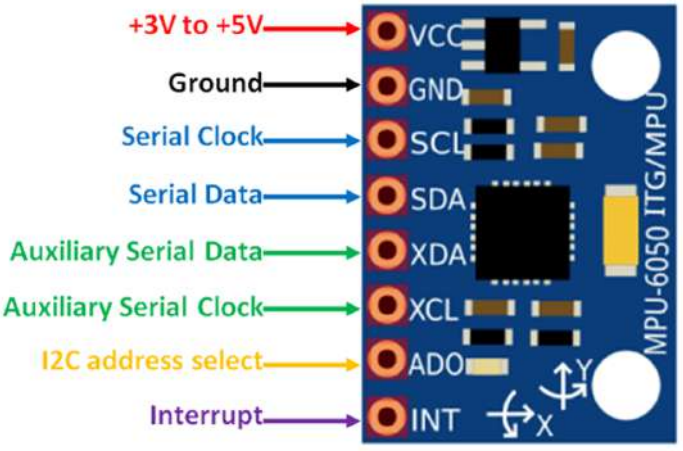
Что же делать?

Необходимо усовершенствовать наш класс `Level`, чтобы расчеты выполнялись независимо от внешних и внутренних запросов и не блокировали возможность прочих операций!

Когда-то у нас уже был некоторый опыт в решении подобных задач.... Когда мы работали с кнопками и прерываниями!

Инициализация прерываний со стороны датчика

Взглянем еще раз на схему датчика:



Хмммм... есть пин `Interrupt` - наверное, он как раз для этого!

Как он должен работать: в момент, когда датчик готов передать данные на микроконтроллер, он должен сформировать высокий сигнал на этом пине, что, в свою очередь, будет служить сигналом для микроконтроллера выполнить запрос данных с датчика.

Вроде все логично, но используемый нами ранее код не формирует на этом пине необходимый нам сигнал!

Для устранения этого досадного упущения в файл `imu.py` был добавлен слудующий класс:

```
class MPU6050WithInterrupt(MPU6050):

    def __init__(self, side_str, device_addr=None, transposition=(0, 1,
2), scaling=(1, 1, 1),
                 active_high=True, latch=False, open_drain=True):
        super().__init__(side_str, device_addr, transposition, scaling)
        self.enable_data_ready_interrupt()
        self.configure_int_pin(active_high, latch, open_drain)

    def enable_data_ready_interrupt(self):
        try:
            self._write(0x01, 0x38, self.mpu_addr)
```

```
        except OSError:
            raise MPUException(self._I2Cerror)

    def configure_int_pin(self, active_high=True, latch=False,
open_drain=False):
        config = 0x00
        if not active_high:
            config |= 0x80  # ACTL = 1 (активный низкий)
        if latch:
            config |= 0x20  # LATCH_INT_EN = 1 (удерживать INT)
        if open_drain:
            config |= 0x40  # OPEN = 1 (открытый сток)

        try:
            self._write(config, 0x37, self.mpu_addr)  # INT_PIN_CFG (0x37)
        except OSError:
            raise MPUException(self._I2Cerror)
```

Что мы видим? Класс `MPU6050WithInterrupt` наследуется от класса `MPU6050` и посылает на сам датчик команду начать посылать сигнал прерывания на выделенный на это пин.

ⓘ Отметим, что такой пример наследования является идеальным исполнением **принципа подстановки Лисков**, так как ничем не нарушает логику работы стандартного класса `MPU6050`, а это значит, что экземпляры класса `MPU6050WithInterrupt` легко могут заменить `MPU6050` во всем ранее написанном коде и ничего не сломают!

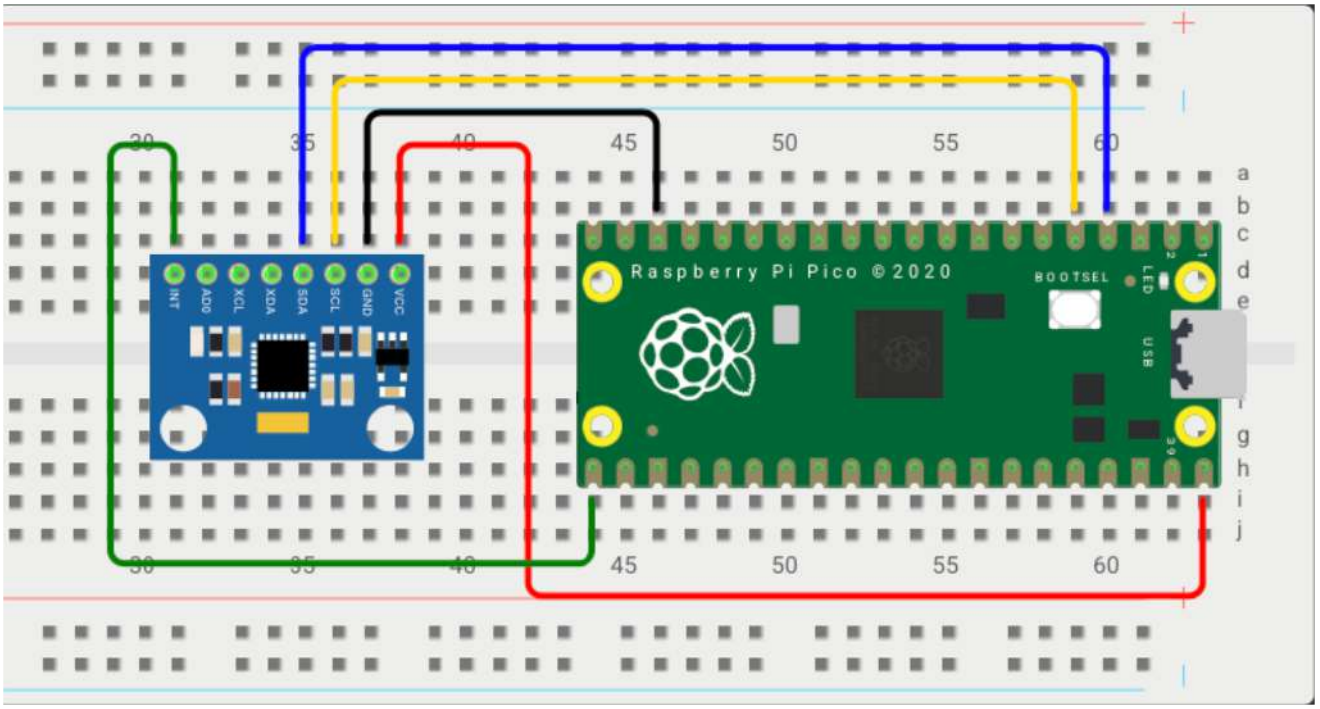
> Если Вы скачали файл `imu.py` по ссылке с яндекс диска, то этот файл уже присутствует в нем!

Проверить работу прерывания можно, запустив следующий код:

```
from imu import MPU6050WithInterrupt
from machine import I2C, Pin

I2C_ID = 1
SDA_PIN = 2
SCL_PIN = 3
IRQ_PIN = 16

i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))

imu = MPU6050WithInterrupt(i2c)

# Настройка GPIO для прерывания
int_pin = Pin(IRQ_PIN, Pin.IN, Pin.PULL_UP)
```

```
# Обработчик прерывания
def imu_interrupt(pin):
    accel_data = imu.accel.xyz  # "Сырые" значения (int)
    gyro_data = imu.gyro.xyz  # "Сырые" значения (int)
    print(accel_data, gyro_data)


# Назначить обработчик на прерывание (по фронту или уровню)
int_pin.irq(trigger=Pin.IRQ_RISING, handler=imu_interrupt)
```

Схема для подключения с прерыванием



После запуска кода консоль с огромной скоростью начнет наполняться данными с устройства.

> ⚠ Данные выводимые в консоль на большой скорости и частые прерывания могут помешать перезаписи файлов на микроконтроллере и "подвесить" его. Если такое произошло, временно отключите пины на датчике и снимите провод с пина который фиксирует прерывания!

Изменения класса `Level` для работы с прерываниями

Так как изменений в коде, относительно предыдущего варианта, не очень много рассмотрим сразу весь файл:

```
from math import atan, degrees
import sys
import time
import json
from machine import Pin, I2C
```

```
from time import ticks_ms, ticks_diff

from imu import MPU6050WithInterrupt, MPUException


class Level:

    def __init__(self, side_str, irq_pin, a=0.95, device_addr=None,
transposition=(0, 1, 2), scaling=(1, 1, 1),
                 active_high=True, latch=False, open_drain=True):
        self._imu = MPU6050WithInterrupt(side_str, device_addr,
transposition, scaling,
                                         active_high, latch, open_drain)
        self._irq_pin = Pin(irq_pin, Pin.IN, Pin.PULL_UP)
        self._irq_pin.irq(trigger=Pin.IRQ_RISING, handler=lambda pin:
self._imu_interrupt())
        self.a = a
        self._last_time = None
        self._temperature_c = 0
        self._accel_pitch, self._accel_roll = 0, 0
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0

    def _imu_interrupt(self):
        try:
            accel_data = self._imu.accel.xyz
            gyro_data = self._imu.gyro.xyz
            self._temperature_c = self._imu.temperature
            dt = self._get_dt()
            self._get_accel_pitch_roll(accel_data)
            self._get_gyro_pitch_roll(gyro_data, dt)
            self._get_complex_pitch_roll(gyro_data, dt)
        except MPUException as e:
            return

    def _get_dt(self):
        current_time = ticks_ms()
        if self._last_time is None:
            dt = 0
        else:
            dt = ticks_diff(current_time, self._last_time) / 1000
        self._last_time = current_time
        return dt

    def _get_accel_pitch_roll(self, accel_data):
        x_accel = accel_data[0]
        y_accel = accel_data[1]
        z_accel = accel_data[2]
        try:
            self._accel_pitch = degrees(atan(y_accel / z_accel))
```

```
            self._accel_roll = degrees(atan(x_accel / z_accel))
        except ZeroDivisionError:
            pass
        return self._accel_pitch, self._accel_roll

    def _get_gyro_pitch_roll(self, gyro_data, dt):
        x_gyro = gyro_data[0]
        y_gyro = gyro_data[1]
        self._gyro_pitch += y_gyro * dt
        self._gyro_roll += x_gyro * dt
        return self._gyro_pitch, self._gyro_roll

    def _get_complex_pitch_roll(self, gyro_data, dt):
        x_gyro = gyro_data[0]
        y_gyro = gyro_data[1]
        self._complex_pitch = self.a * (self._complex_pitch + y_gyro * dt)
+ (1 - self.a) * self._accel_pitch
        self._complex_roll = self.a * (self._complex_roll + x_gyro * dt) +
(1 - self.a) * self._accel_roll
        return self._complex_pitch, self._complex_roll

    def get_data_str(self):
        data = {"A": (self._accel_pitch, self._accel_roll),
                "G": (self._gyro_pitch, self._gyro_roll),
                "C": (self._complex_pitch, self._complex_roll),
                "tC": self._temperature_c,
                }
        data = json.dumps(data)
        return data

    def send_pitch_roll_data(self):
        data = self.get_data_str()
        sys.stdout.write(f"{data}\n".encode("utf-8"))


if __name__ == "__main__":
    I2C_ID = 1
    SDA_PIN = 2
    SCL_PIN = 3
    IRQ_PIN = 16

    i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))
    level = Level(i2c, IRQ_PIN)

    while True:
        time.sleep(0.05)
        level.send_pitch_roll_data()
```

Ссылка на файл с кодом:

Превые изменения встречают нас в методе `__init__`:

```
    def __init__(self, side_str, irq_pin, a=0.95, device_addr=None,
transposition=(0, 1, 2), scaling=(1, 1, 1),
                 active_high=True, latch=False, open_drain=True):
        self._imu = MPU6050WithInterrupt(side_str, device_addr,
transposition, scaling,
                                         active_high, latch, open_drain)
        self._irq_pin = Pin(irq_pin, Pin.IN, Pin.PULL_UP)
        self._irq_pin.irq(trigger=Pin.IRQ_RISING, handler=lambda pin:
self._imu_interrupt())
        self.a = a
        self._last_time = None
        self._temperature_c = 0
        self._accel_pitch, self._accel_roll = 0, 0
        self._gyro_pitch, self._gyro_roll = 0, 0
        self._complex_pitch, self._complex_roll = 0, 0
```

Что мы поменяли:

- при создании экземпляра класса нам необходимо указать номер пина, к которому мы подключим вывод `Interrupt` датчика;
- `elf._imu = MPU6050WithInterrupt()` теперь агрегирует класс `MPU6050WithInterrupt`, а не `MPU6050`;
- после инициализации инерциальной системы `_imu` выполняется включения пина `irq_pin` в режим прерывания, аналогично тому, как мы делали это с кнопкой;
- при срабатывании прерывания будет выполняться метод `self._imu_interrupt()`, определяющий логику необходимых при прерывании действий.

Остальная часть инициализирующего метода осталась без изменений, за исключением введения нового атрибута `_temperature_c`, в который будет записываться информация о температуре датчика.

Рассмотрим теперь метод `self._imu_interrupt()` запускаемый при наступлении прерывания:

```
    def _imu_interrupt(self):
        try:
            accel_data = self._imu.accel.xyz
```

```
            gyro_data = self._imu.gyro.xyz
            self._temperature_c = self._imu.temperature
            dt = self._get_dt()
            self._get_accel_pitch_roll(accel_data)
            self._get_gyro_pitch_roll(gyro_data, dt)
            self._get_complex_pitch_roll(gyro_data, dt)
        except MPUException as e:
            return
```

По своей логике он аналогичен использованному в предыдущей версии класса `Level` методу:

```
    def _calk_pitch_roll(self):
        accel_data = self._imu.accel.xyz
        gyro_data = self._imu.gyro.xyz
        dt = self._get_dt()
        self._get_accel_pitch_roll(accel_data)
        self._get_gyro_pitch_roll(gyro_data, dt)
        self._get_complex_pitch_roll(gyro_data, dt)
```

Не считая опрос температуры датчика:

```
self._temperature_c = self._imu.temperature
```

Главным отличием является конструкция `try-except`, определяющая логику выполнения действий при возникновении ошибки `MPUException`.

Исключение `MPUException` - возникает тогда, кода читаемые с датчика данные содержат в себе ошибку - из-за плохого контакта пинов на I2C шине или внешних помех на проводах.

Ранее количество опросов датчика было кратно меньше и эти редкие по своей природе события не успевали возникнуть, хотя корректнее было бы обернуть и предыдущий код подобной конструкцией.

Методы:

```
    def _get_dt(self):
        ...

    def _get_accel_pitch_roll(self, accel_data):
        ...

    def _get_gyro_pitch_roll(self, gyro_data, dt):
        ...
```

```
    def _get_complex_pitch_roll(self, gyro_data, dt):
        ...
```

Остались полностью без изменений, так как все логика вычислений осталась прежней, а эти методы больше ни за что не отвечают.

Методы доступные пользователю:

```
    def get_data_str(self):
        data = {"A": (self._accel_pitch, self._accel_roll),
                "G": (self._gyro_pitch, self._gyro_roll),
                "C": (self._complex_pitch, self._complex_roll),
                "tC": self._temperature_c,
                }
        data = json.dumps(data)
        return data

    def send_pitch_roll_data(self):
        data = self.get_data_str()
        sys.stdout.write(f"{data}\n".encode("utf-8"))
```

тоже изменились косметически:

- в `get_data_str()` нам теперь не нужно в ручную "включать" опрос датчика, так как теперь все это будет происходить "в фоне";
- ну и в словарь `data` мы добавили строку `"tC": self._temperature_c`, сохраняющую текущую температуру.

Часть запуска тоже изменилась лишь тем, что в константе `IRQ_PIN = 16` мы указываем к какому пину будет приводиться прерывание:

```
if __name__ == "__main__":
    I2C_ID = 1
    SDA_PIN = 2
    SCL_PIN = 3
    IRQ_PIN = 16

    i2c = I2C(id=I2C_ID, sda=Pin(SDA_PIN), scl=Pin(SCL_PIN))
    level = Level(i2c, IRQ_PIN)

    while True:
        time.sleep(0.05)
        level.send_pitch_roll_data()
```

Ну и поменялась сигнатура создания объекта класса `Level`:

```
level = Level(i2c, IRQ_PIN)
```

Сам опрос датчика как и передача информации в консоль так и осталась с задержкой в 0.05 секунд, но теперь она никак не связана с периодичностью опроса датчика.

Код отрисовки графика уровня остается, естественно, прежним, как и все правила его запуска.

Выполнив все необходимые манипуляции, Вы сможете отметить, что скорость реагирования "комплексного" решения резко возросла, а значит можно поэксперементировать с коэффициентом `a` при инициации уровня, увеличив долю точного гироскопа и повысив, тем самым, точность результирующего комплексного решения.