POLITECNICO MILANO 1863

IMAGE AND SOUND PROCESSING GROUP (ISPG)

CREATIVE PROGRAMMING AND COMPUTING

Lab: Reactive Agents

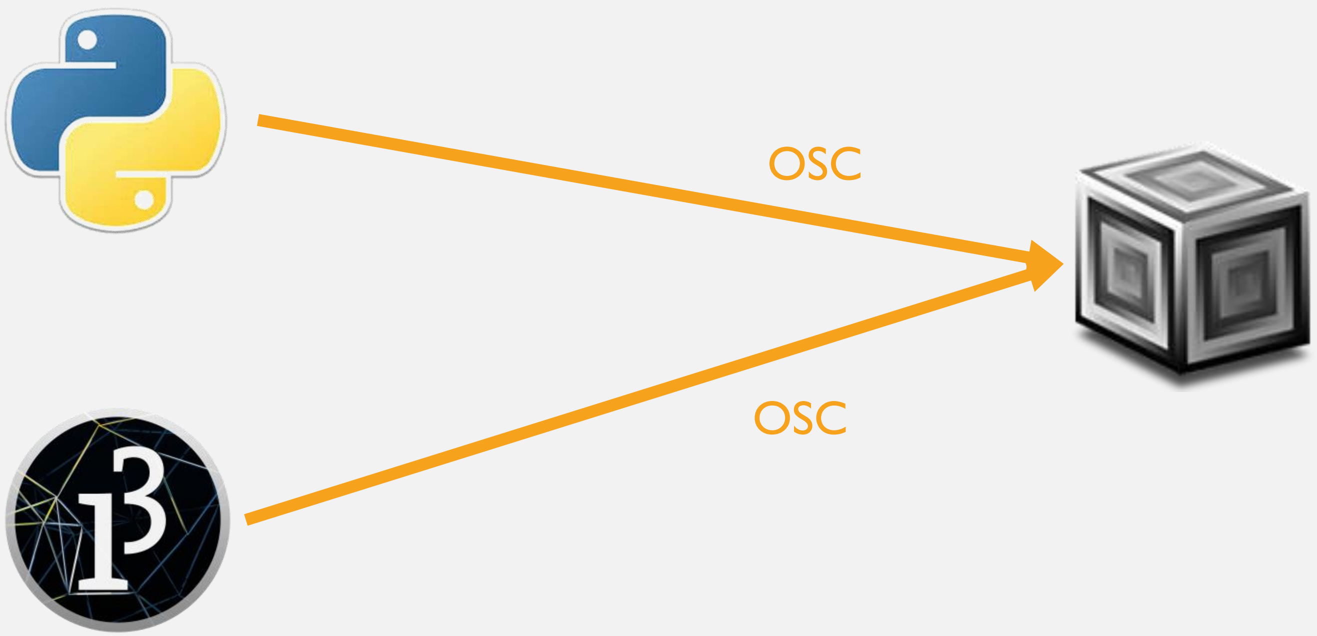
REACTIVE AGENTS

- Reactive agents behave like …
- We will see two kind of reactive agents:
  - Music composition with Python
  - Physics-related reactive agents
- We want to connect our agent to some effect
  - A musical instrument with SuperCollider

OVERVIEW

REACTIVE AGENTS

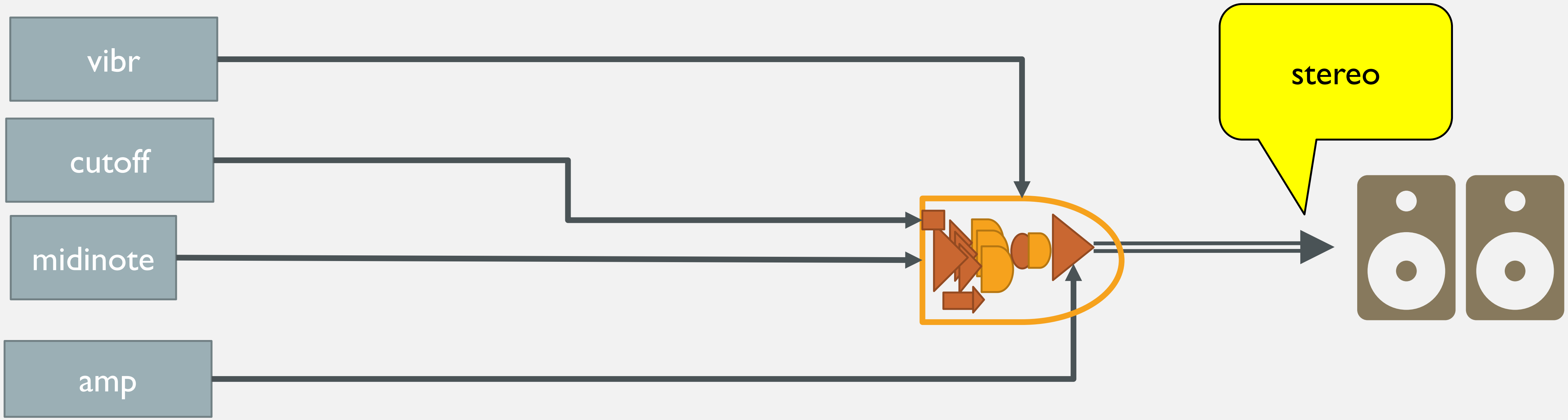
- We first create an instrument in SuperCollider
- We will control the instrument with Python and Processing using the OSC protocol
- Depending on what we need to do, we will control different parameters



DEFINITION OF THE INSTRUMENT

In super_collider_instrument you can find moogs.scd, with the definition of an instrument:

- Arguments are: midinote, vibr (vibrato), cutoff, amp (amplitude)



MUSIC COMPOSITION WITH PYTHON

We will use Python to create an automatic composition:

- create an automatic composition
- send the parameters of the composition to the OSC instrument

Exercize 0

Run and make sure `example.py` works with `moogs.scd`

PLAYING WITH PHYSICS

We will first test OSC in Processing in `testOSC.pde`

- we install the library **oscP5**
- first test: we map mouse's x and y position into cutoff and vibrato
  - MouseX ranges between 0 and width → vibrato is designed to oscillate around 0
  - MouseY ranges between 0 and height →cutoff is designed to range between 0 and 1
- Mapping:
  - `float cutoff =  map(mouseY, 0, height,0,1);`
  - `float vibrato = map(mouseX, 0, width, -0.5, 0.5);`

Exercize 0

- Run and make sure `testOSC.pde` works with `moogs.scd`

MUSIC COMPOSITION WITH PYTHON

requires the package python-osc

MUSIC COMPOSITION WITH PYTHON

We will use Python to create an automatic composition:

- create an automatic composition
  - define a composition starting from an initial set of parameters (note, effect, duration, amplitude, etc.)
  - write a function that at each step changes the parameters depending on what is playing in that moment
  - Compositions are then generated by different algorithms and functions
- send the parameters of the composition to the OSC instrument
  - sleep to the time required to play the note (the duration)

Let’s see how

MUSIC COMPOSITION WITH PYTHON

We will use Python to create an automatic composition:

- create an automatic composition → class Composition

- send the parameters of the composition to the OSC instrument → class InstrOsc

MUSIC COMPOSITION WITH PYTHON

```
# classes.py
```

InstrOsc

Attributes

name of the message

client for OSC

Methods

*send(*data)* send via OSC

Composition

Attributes

midinote current note

dur duration of the note in beats

amp: amplitude

BPM: beats per minute

Methods

next(): compute the next note of the composition ABSTRACT

Agent(thread)

Attributes

instr: an InstrOSC

composer the Composition at the current moment

Methods

action (main thread function): while the thread is active :

- it calls *composer.next()* to update the composition
- it sends the data of the notes to the instrument
- it sleeps for the duration of the note

Your code here

Actual Composition

- inherits Composition
- **adds attributes, if needed**
- **Implements next()**

MUSIC COMPOSITION WITH PYTHON

```
# classes.py

class Composition:
    def __init__(self, id=ID_START,
                 midinote=ID_START,
                      dur=0,
                      amp=0,
                      BPM=120):
        self.id = id
        self.midinote = midinote
        self.dur = dur
        self.amp = amp
        self.BPM = BPM
    def next(self):
       pass
```

- See the class *Composition* on the left
  - *id* is the ID, and starts from `ID_START=-1`
  - *midinote* is the current midinote
  - *dur* is the duration of the note in beats with reference to BPM
  - *amp* is the amplitude
- next() is an *abstract* method: it is not implemented
  - we create a child class to use different compositions

MUSIC COMPOSITION WITH PYTHON

```
# <ex_i>.py
class Bach(Composition):
    def __init__(self, BPM=60):
        Composition.__init__(self,
                             BPM=BPM)

    def next(self):
        # your code here
```

- *Composition* is a generic composition
- Your actual composition is a *child* that inherits it
- In the init, it initialize the class
- In next() you implement the composition

MUSIC COMPOSITION WITH PYTHON

Example of next(): play a random note and every 10 notes randomly changes the BPM

```
# example.py
class Random_Next(Composition):
    def __init__(self, BPM=60):
        Composition.__init__(self, BPM=BPM)
    def next(self):
        if self.id ==ID_START:
            self.id=10  // start the countdown
        if self.id >0:  // countdown is not over: play a random note
            self.midinote = int(np.random.randint(60,84))
            self.dur =  float(2**np.random.randint(-3, 0))
            self.amp = float(np.random.choice([0,1,1,1]))
            self.id-=1
        else: // countdown is over: change BPM, start a new countdown
            self.BPM=np.random.randint(60, 120)
            self.id=10
            self.next()
```

MUSIC COMPOSITION WITH PYTHON

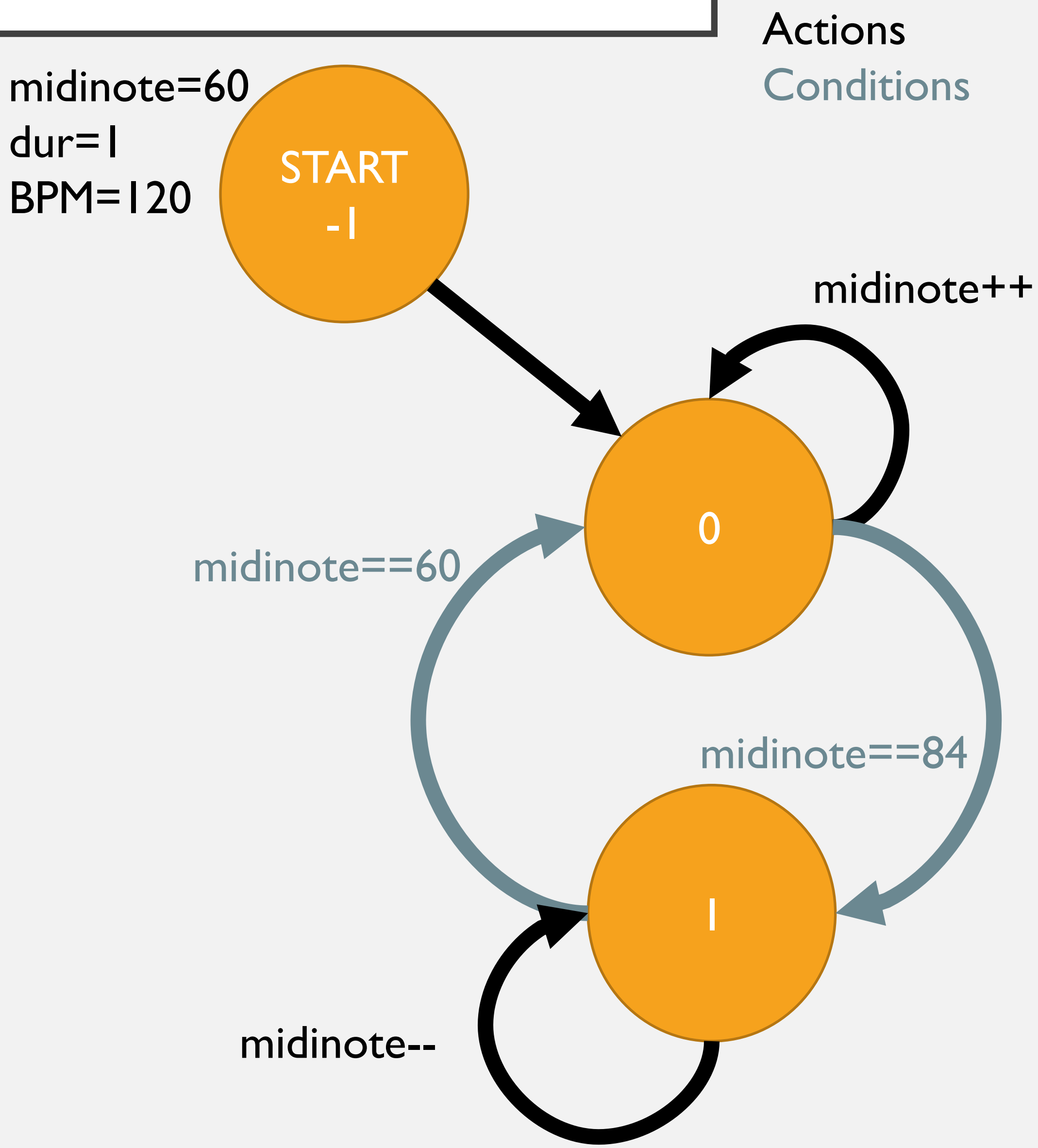
Exercize 0

- Run and make sure `example.py` works with `moogs.scd`

MUSIC COMPOSITION WITH PYTHON

Exercize 1: `simple_next.py`

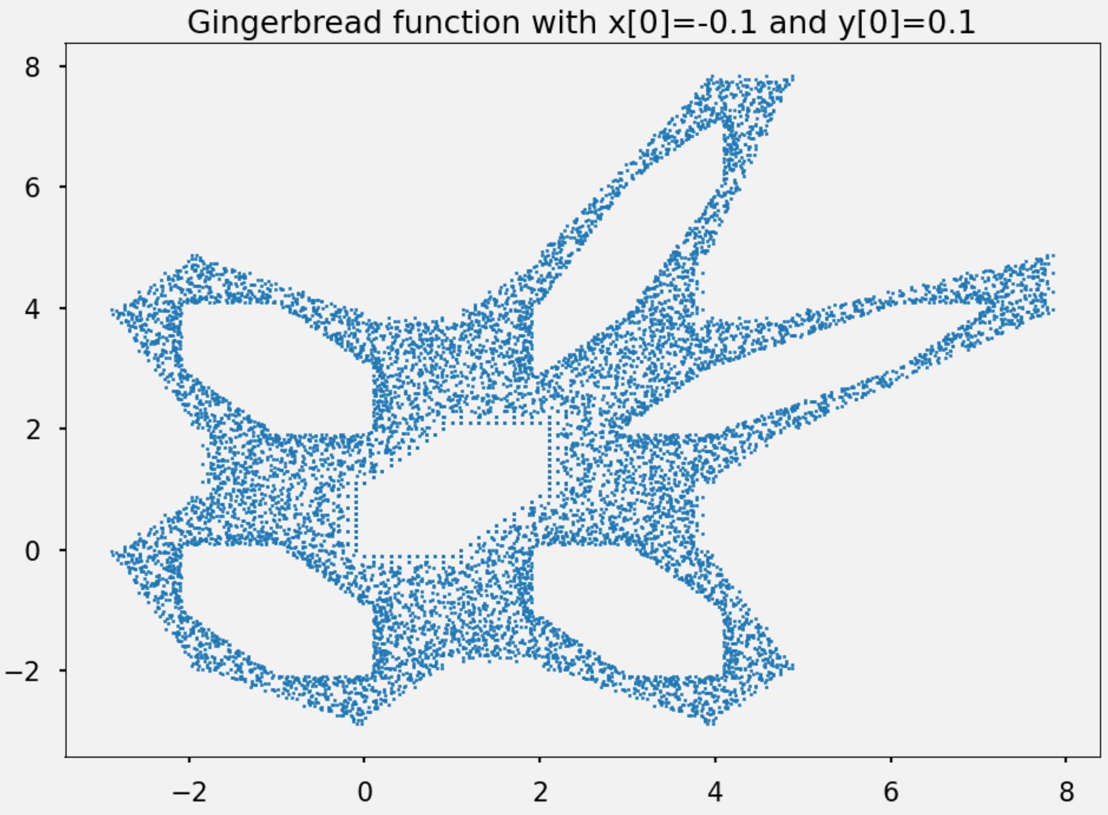
- Implement this silly algorithm
- starts with midinote = 60 and duration = 1
- Increase midinote at each step
  - until it is equal to 84
- Decrease midinote at each step
  - until it is equal to 60
- And so on…



MUSIC COMPOSITION WITH PYTHON

Exercize 2: Let's use the Gingerbread function

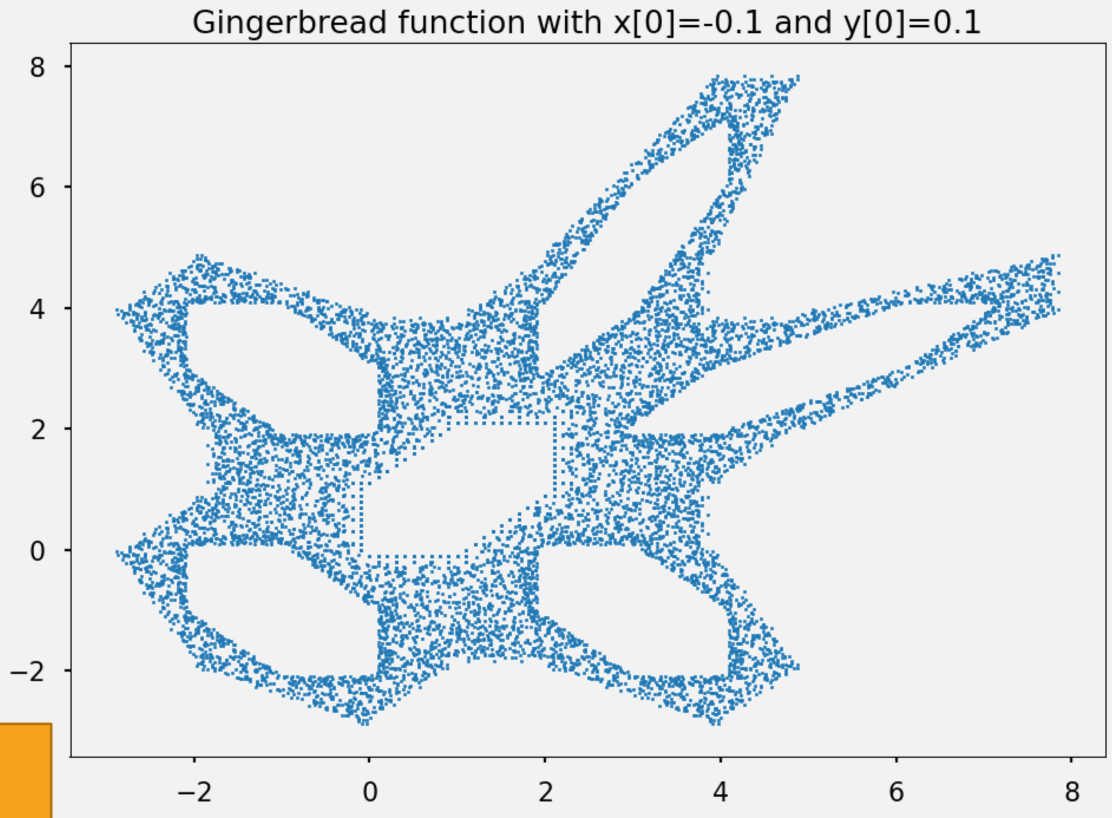
- Two variables $x(n)$ and $y(n)$ with n is a count
  - $x(0) = -0.1$ and $y(0) = 0.1$
- At each step
  - $x(n+1) = 1 - y(n) + |x(n)|$
  - $y(n+1) = x(n)$
- map $x(n)$ and/or $y(n)$ to duration and midinote
  - They range between -3 and 8 in this case
  - remap them into meaningful value



MUSIC COMPOSITION WITH PYTHON

Exercize 2: Let's use the Gingerbread function

- Two variables $x(n)$ and $y(n)$ with n is a count
  - $x(0) = -0.1$ and $y(0) = 0.1$
- At each step
  - $x(n+1) = 1 - y(n) + |x(n)|$
  - $y(n+1) = x(n)$
- map $x(n)$ and/or $y(n)$ to duration and midinote
  - They range between -3 and 8 in this case
  - remap them into meaningful value



Challenge: add "creativity" to the GingerBread by adding randomicity

PLAYING WITH PHYSICS

PLAYING WITH PHYSICS

- We used Python to create a reactive agent to create a composition
  - following strict rules
  - We can add randomness to make compositions change
  - We control *note*, *duration* and *amplitude*

- We will use Processing to create reactive agents inspired by physics
  - And use those agents to control the timbre of the instrument
  - We will control the *cutoff* and the *vibrato*

PLAYING WITH PHYSICS

We will first test OSC in Processing in `testOSC.pde`

- we install the library **oscP5**
- We will use mouse's x and y position
  - MouseX ranges between 0 and width
  - MouseY ranges between 0 and height
- We need to find a nice mapping between them and the cutoff or vibr.
  - vibr is designed to oscillate around 0
  - cutoff is designed to range between 0 and 1
- We map position of the agent with cutoff and vibr
  - `float cutoff =  map(mouseY, 0, height,0,1);`
  - `float vibrato = map(mouseX, 0, width, -0.5, 0.5);`

constrain is a clipping

PLAYING WITH PHYSICS

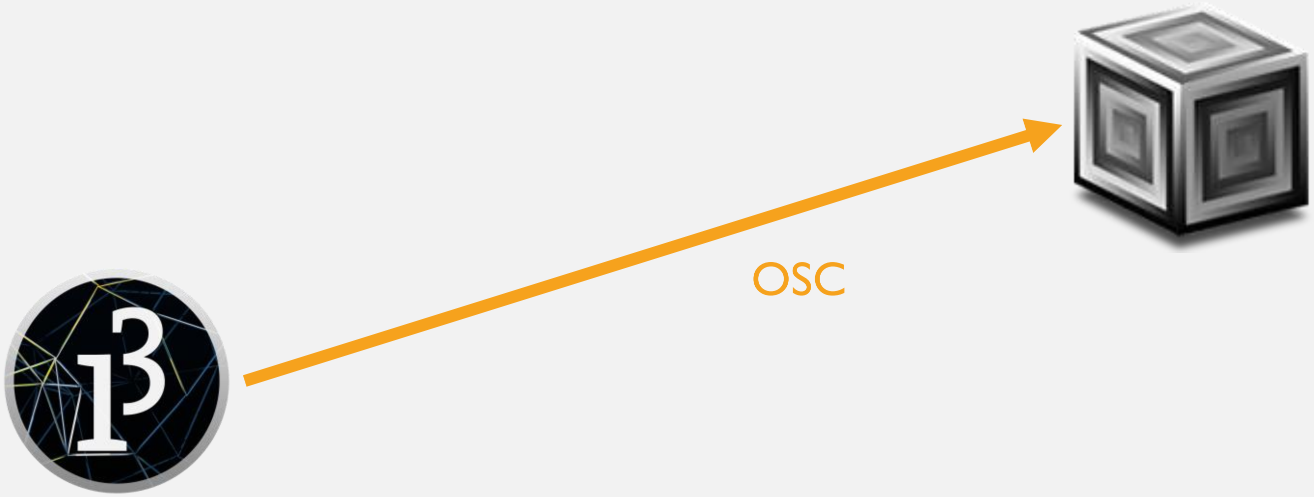
```
# testOSC.pde
import oscP5.*; import netP5.*;
int PORT = 57120;
OscP5 oscP5; NetAddress ip_port;

void setup(){
  oscP5 = new OscP5(this,55000);
  ip_port = new NetAddress("127.0.0.1",PORT);
  size(1280, 720); background(0);
}

void draw(){
  background(0);
  rectMode(CENTER); fill(255);
  ellipse(mouseX, mouseY, 10, 10);
  float cutoff =  map(mouseY, 0, height,0,1);
  float vibrato = map(mouseX, 0, width, -0.5, 0.5);
  sendEffect(cutoff, vibrato);
}
```

```
void sendEffect(float cutoff, float vibrato){
  OscMessage effect =
              new OscMessage("/note_effect");
  effect.add("effect");
  effect.add(cutoff);
  effect.add(vibrato);
  oscP5.send(effect, ip_port);
}
```

MUSIC COMPOSITION WITH PYTHON

Exercize 0

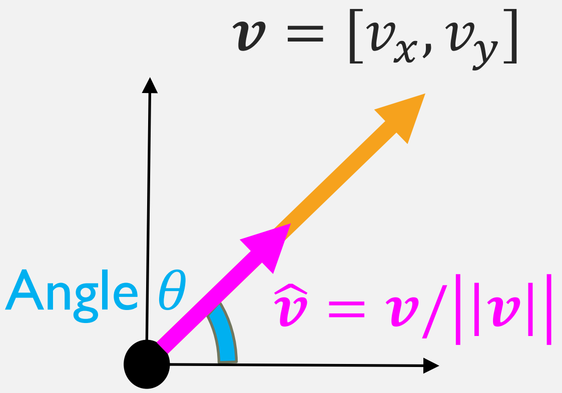
- Run and make sure `testOSC.pde` works with `moogs.scd`



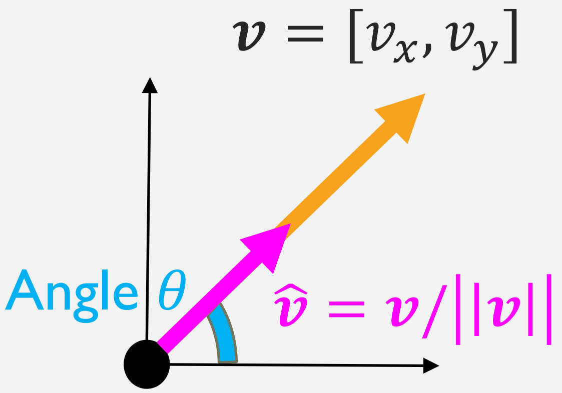
THE BASIC OF VECTORS

- We will see how to play with motion in 2D with Processing

- The basic element are **vectors**, as in linear algebra and in physics:
  - 2D world → 2-component vector $\boldsymbol{v} = [v_x, v_y]$
  - We compute magnitude $||\boldsymbol{v}||$ and with respect to the origin $\boldsymbol{O} = [0, 0]$
  - We compute angle $\angle\boldsymbol{v}$ with respect to axis $\hat{\boldsymbol{x}} = [1\ 0]$
  - The versor is $\hat{\boldsymbol{v}} = \boldsymbol{v}/||\boldsymbol{v}||$



THE BASIC OF VECTORS



- We will see how to play with motion in 2D with Processing
- The basic element are **vectors**, as in linear algebra and in physics:
- Processing provides the object **PVector** as
  - `PVector v = new PVector(v_x,v_y);`
  - PVector is defined for 3D world; when initialized with 2 values, the third is automatically set to 0
  - PVector have a set of methods, such as add(Pvector), mag(), mult(float), normalize(), etc.
  - See https://processing.org/reference/PVector.html

THE BASIC OF VECTORS

- Let's create a processing script to display a circle that follows the mouse
- Agent has position , velocity and acceleration; mouse is converted into a Pvector
  - $\boldsymbol{p}(n) = [p_x, p_y]$; $\boldsymbol{v}(n) = [v_x, v_y]$; $\boldsymbol{a}(n) = [a_x, a_y]$;
- Remember that acceleration is
  - $\boldsymbol{a}(n) = \frac{\boldsymbol{v}(n) - \boldsymbol{v}(n-1)}{\Delta \mathrm{t}}$
- Δt is a constant, we can choose any value: we choose 1
- Therefore we can write $\boldsymbol{a}(n) = \boldsymbol{v}(n) - \boldsymbol{v}(n-1) \rightarrow \boldsymbol{v}(n) = \boldsymbol{a}(n) + \boldsymbol{v}(n-1)$
- And $\boldsymbol{p}(n) = \boldsymbol{v}(n) + \boldsymbol{p}(n-1)$

THE BASIC OF VECTORS

- Let's create a processing script to display a circle that follows the mouse
- Algorithm:
  1. Compute the vector that points from the object's to the mouse's locations
     - $\boldsymbol{m}(n) = \boldsymbol{mouse} = [mouse_x, mouse_y]$
     - $\boldsymbol{\Delta}_{\mathbf{m}}(\mathrm{n}) = \boldsymbol{m}(n) - \boldsymbol{p}(n)$
  2. Normalize the vector and scale it to a given constant value (use c=CONSTANT_ACC=10)
     - $\widetilde{\boldsymbol{\Delta}}_m(n) = {}^{\boldsymbol{\Delta}_{\mathbf{m}}(n)}/_{||\boldsymbol{\Delta}_{\mathbf{m}}||(n)}\, c$
  3. assign the vector to object's acceleration; update velocity and position accordingly
     - $\boldsymbol{a}(n) = \widetilde{\boldsymbol{\Delta}}_m(n); \boldsymbol{v}(n) = \boldsymbol{v}(n-1) + \boldsymbol{a}(n); \boldsymbol{p}(n) = \boldsymbol{p}(n-1) + \boldsymbol{v}(n)$
  4. draw the object

THE BASIC OF VECTORS

```
# movingBall.pde
AgentMover mover;
void setup(){
  mover=new AgentMover();
  size(1280, 720);
  background(0);
}

void draw(){
  background(0);
  mover.update();
  mover.draw()
}
```

```
# AgentMover.pde
int RADIUS_CIRCLE=50; int LIMIT_VEL=50; int ACC=2;

class AgentMover{
  PVector pos, vel, acc;
  AgentMover(){
    this.pos = new PVector(random(width), random(height));
    this.vel= new PVector(random(-2, 2), random(-2, 2));
    this.acc = new PVector(random(2), random(2));}
  void update(){
    PVector mouse = new PVector(mouseX, mouseY);
    /* your code here. */
    this.vel.add(this.acc);
    this.vel.limit(LIMIT_VEL);
    this.pos.add(this.velocity);  }
  void draw(){
    fill(200);
    ellipse(this.pos.x, this.pos.y, // ... }
}
```

Look at mouseX, mouseY,
.sub(), .normalize(), .mult()

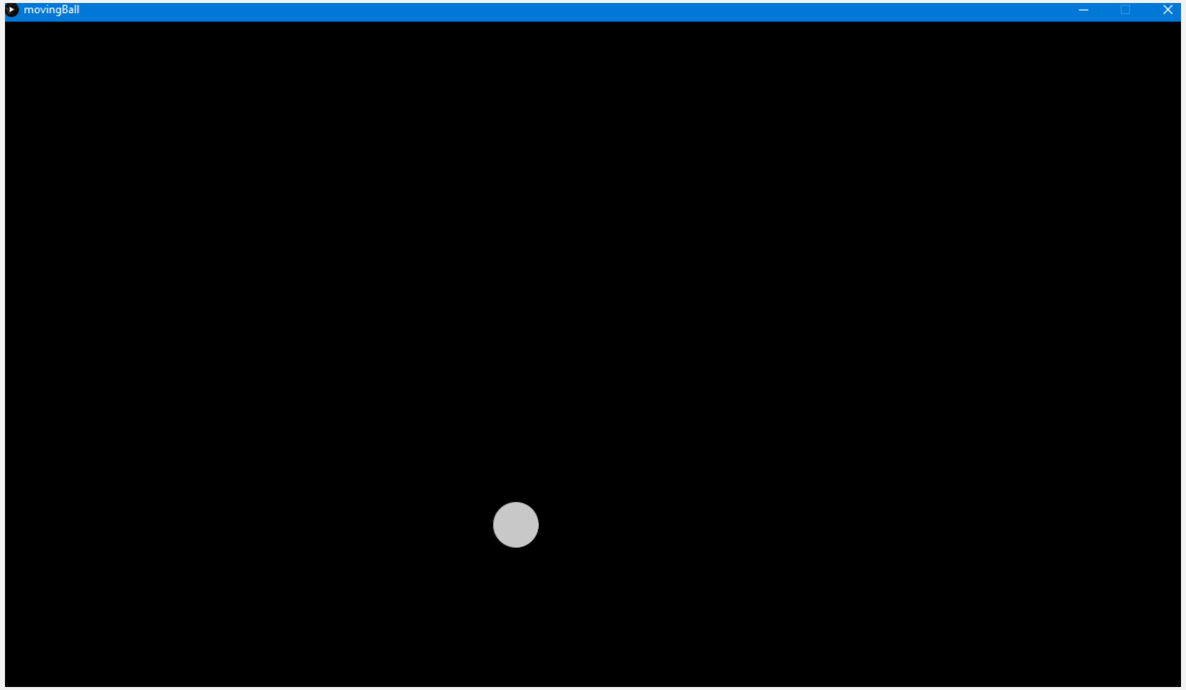
THE BASIC OF VECTORS

- Let's create a processing script to display a circle that follows the mouse
- Algorithm:
  1. Compute the vector that points from the object's to the mouse's locations
     - $\boldsymbol{m}(n) = \boldsymbol{mouse} = [mouse_x, mouse_y]$
     - $\boldsymbol{\Delta}_{\mathbf{m}}(\mathrm{n}) = \boldsymbol{m}(n) - \boldsymbol{p}(n)$
  2. Normalize the vector and scale it to a given constant value (use c=CONSTANT_ACC=10)
     - $\widetilde{\boldsymbol{\Delta}}_m(n) = {}^{\boldsymbol{\Delta}_{\mathbf{m}}(n)}/_{||\boldsymbol{\Delta}_{\mathbf{m}}||(n)}\, c$
  3. assign the vector to object's acceleration; update velocity and position accordingly
     - $\boldsymbol{a}(n) = \widetilde{\boldsymbol{\Delta}}_m(n); \boldsymbol{v}(n) = \boldsymbol{v}(n-1) + \boldsymbol{a}(n); \boldsymbol{p}(n) = \boldsymbol{p}(n-1) + \boldsymbol{v}(n)$
  4. draw the object
  5. **Map mover's position to vibrato and cutoff in `computeEffect()`**

THE BASIC OF VECTORS

What if we change the constant value?

Try it!



NEWTON'S LAWS

We can keep playing with Physics introducing the concept of **force**.

We translate Newtons' laws into Processing PVector

1. First Law: a object's PVector velocity will remain constant (even 0) if in a state of equilibrium, i.e., the sum of the force applied to it is zero
2. Second Law: Object's Pvector acceleration equals Pvector force divided by object's mass
   - We can assume the mass is equal to 1
3. Third Law: if we calcolate a Pvector force of object A on object B,
   - we must apply the force PVector.mult(f, -1) that B exerts on A

NEWTON'S LAWS

- In order to dealing with forces, we implement a method
  applyForce(Pvector force)
  to our AgentMover

- Remember that after applied the acceleration, we need to reset it
  - the object will keep moving due to velocity

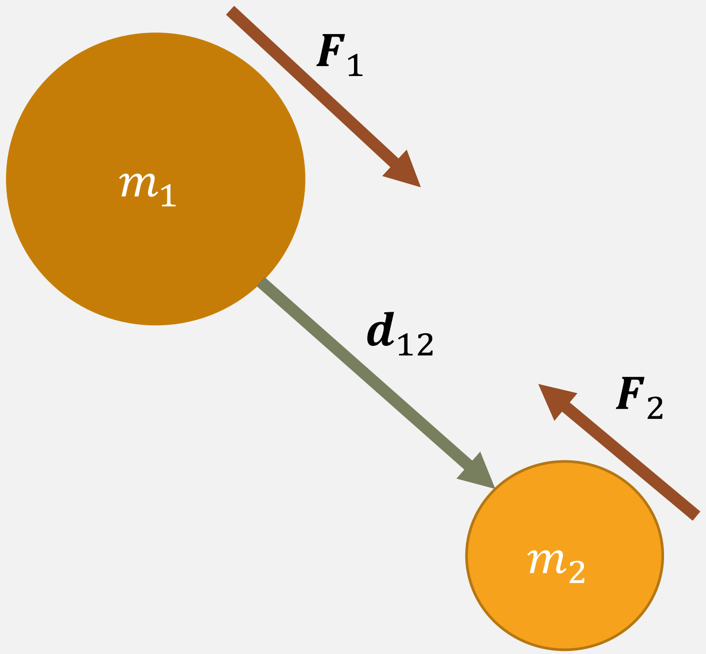
```
# AgentMover.pde
int RADIUS_CIRCLE=50; int LIMIT_VEL=50;
int ACC=2;

class AgentMover{
  PVector position, velocity,
  acceleration; float mass;
  AgentMover(){ /* ... */ }
  void update(){
    /* ...
    this.velocity.add(this.acceleration);
    this.position.add(this.velocity);
    this.acceleration.mult(0);}
  void computeEffect(){/*…*/}
  void draw() { /* ... */ }
  void applyForce(Pvector force) {
     PVector f = force.get()
     f.div(this.mass);
     this.acceleration.add(f);}
}
```

GRAVITY AND ATTRACTION

- Gravity occurs among any pair of objects following the formula

$$\boldsymbol{F}_1 = \frac{G m_1 m_2}{||\boldsymbol{d}_{12}||^2}\,\widehat{\boldsymbol{d}}_{12} = -\boldsymbol{F}_2$$

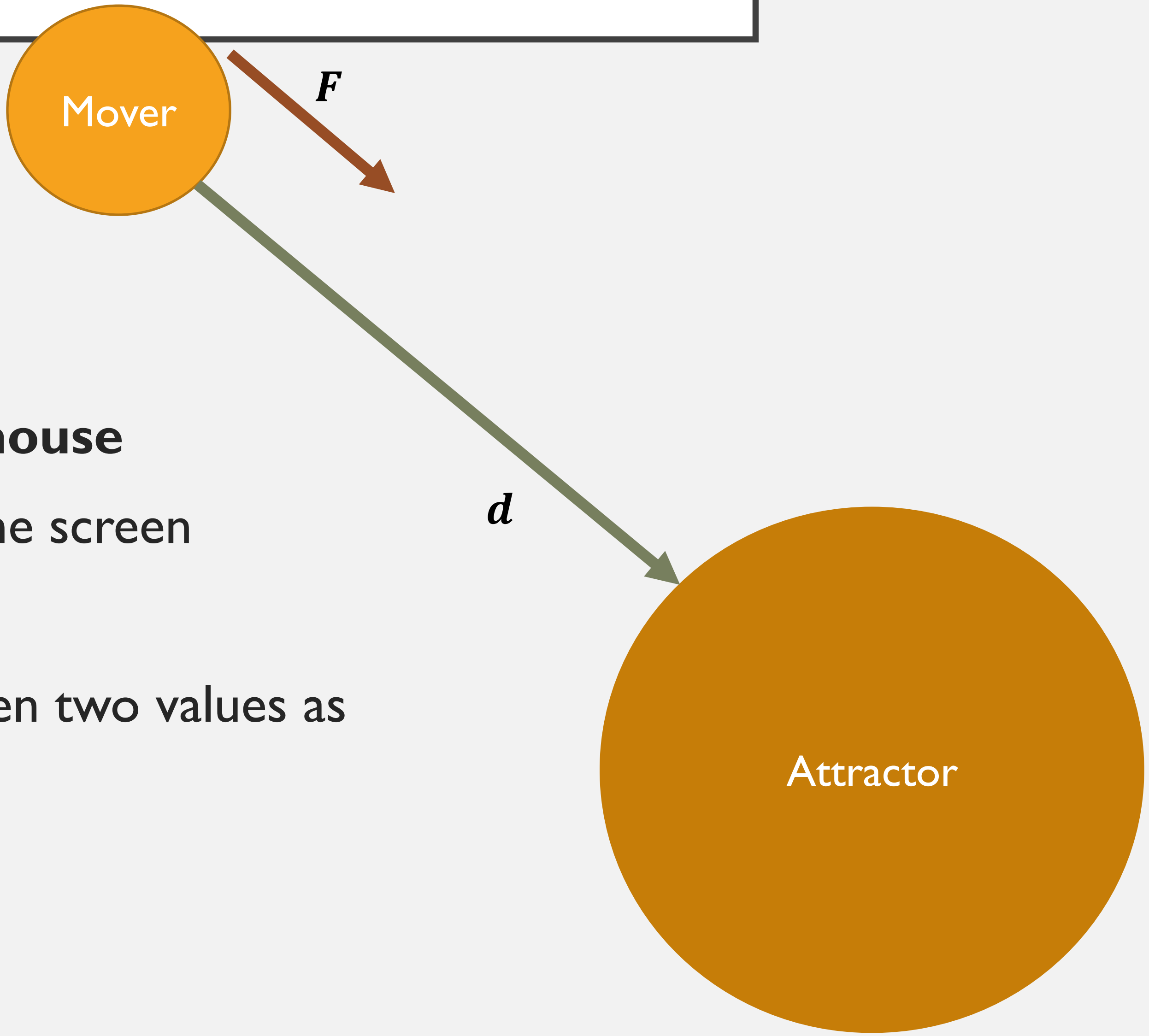
  - $G = 6.67428 \cdot\ 10^{-11}$ *is* the universal gravity constant. We will set it to 1
  - $\widehat{\boldsymbol{d}}_{12}$ is the vector going from object with mass m_1 to object with mass m_2
- The two objects are attracted with each other with the **same force, but opposite directions**
- If $m_1 \gg m_2$, we have $||\boldsymbol{a}_1|| = \left\|\frac{\boldsymbol{F}_1}{m_1}\right\| \ll \left\|\frac{\boldsymbol{F}_2}{m_2}\right\| = ||\boldsymbol{a}_2||$
  - For two objects of greatly different masses, the acceleration of the bigger is neglectable



GRAVITY AND ATTRACTION

Let's define a mover-attractor system

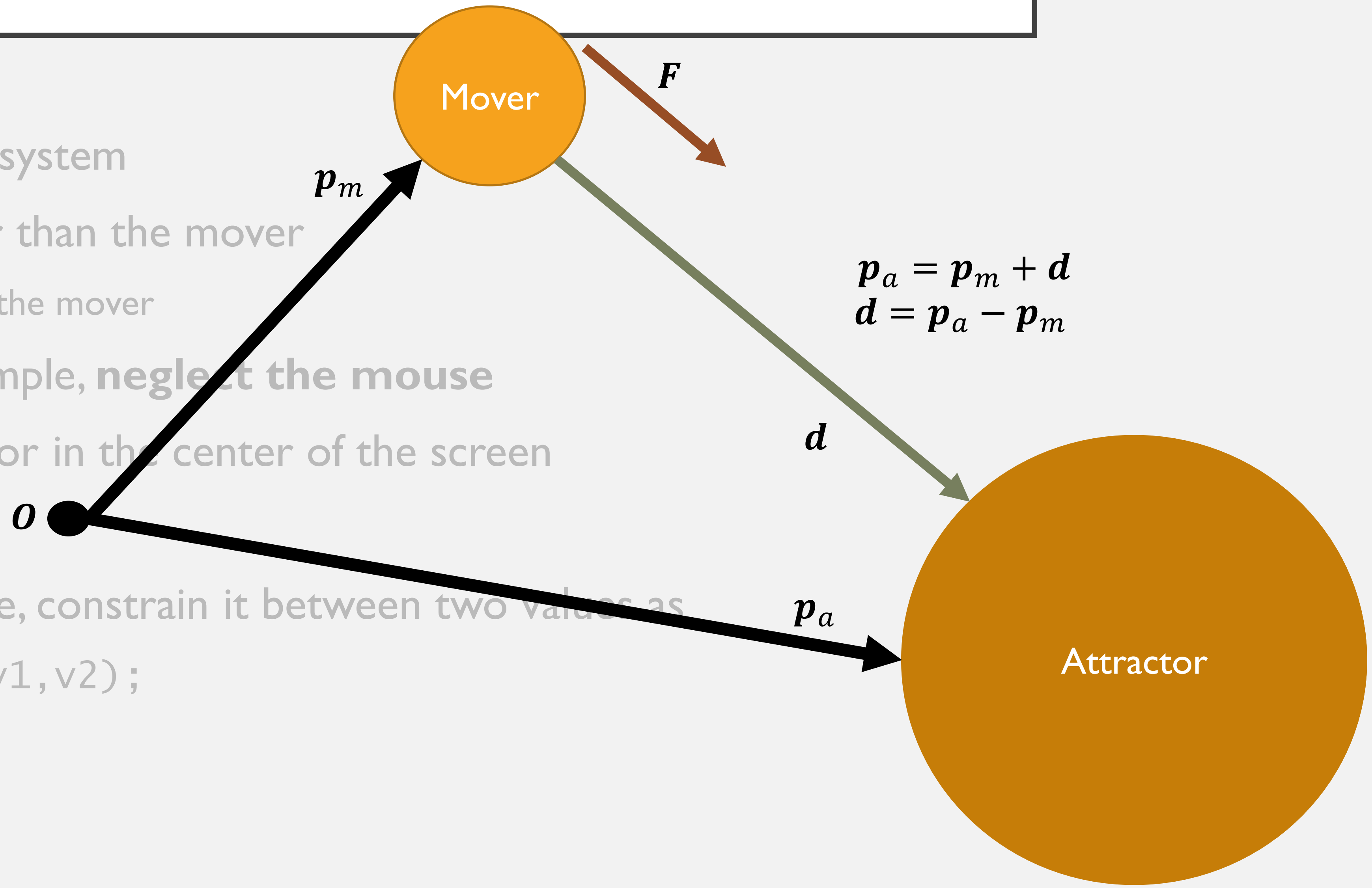
- The attractor is much bigger than the mover
  - so we neglect the force toward the mover
- Start from the previous example, **neglect the mouse**
- Place (and show) the attractor in the center of the screen
- Place the mover
- when computing the distance, constrain it between two values as
  - `dist=constrain(dist, v1,v2);`



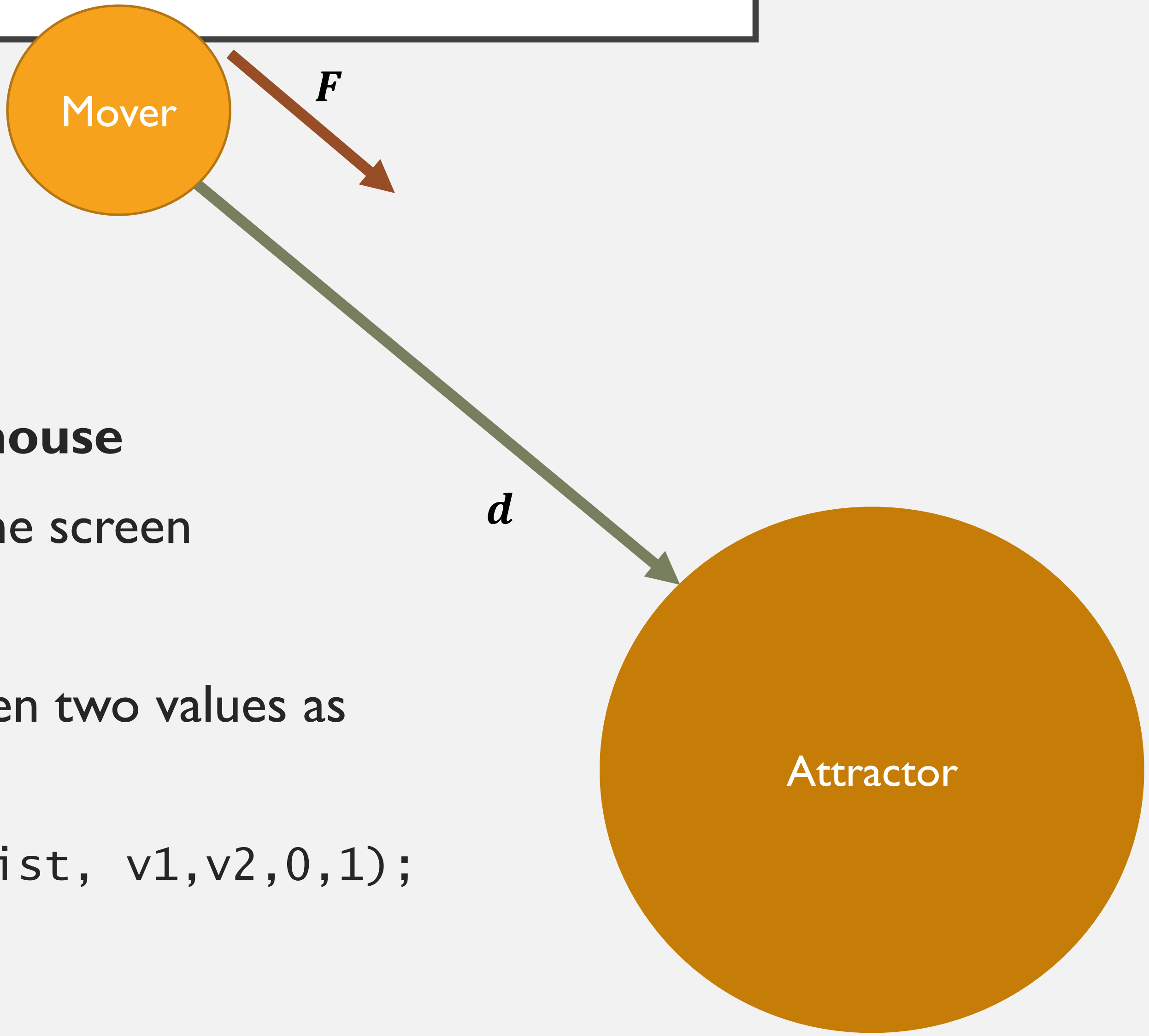
GRAVITY AND ATTRACTION

Let's define a mover-attractor system

- The attractor is much bigger than the mover
  - so we neglect the force toward the mover
- Start from the previous example, **neglect the mouse**
- Place (and show) the attractor in the center of the screen
- Place the mover
- when computing the distance, constrain it between two values as
  - `dist=constrain(dist, v1,v2);`



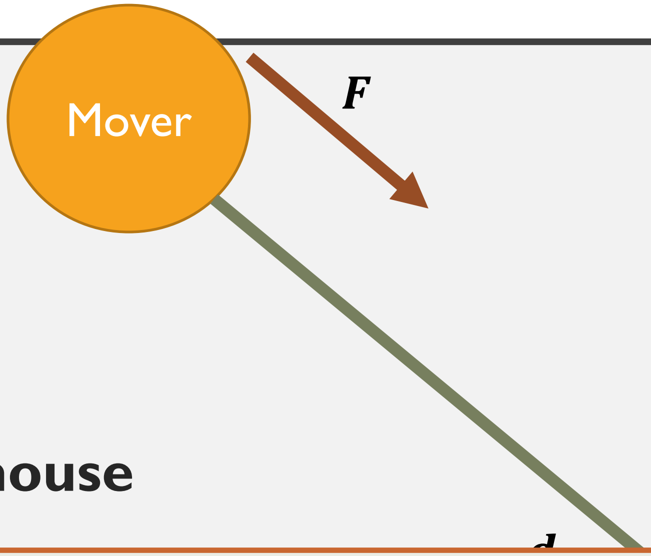
GRAVITY AND ATTRACTION



Let's define a mover-attractor system

- The attractor is much bigger than the mover
  - so we neglect the force toward the mover
- Start from the previous example, **neglect the mouse**
- Place (and show) the attractor in the center of the screen
- Place the mover
- when computing the distance, constrain it between two values as
  - `dist=constrain(dist, v1,v2);`
  - map the distance to the cutoff value as `cutoff=map(dist, v1,v2,0,1);`

GRAVITY AND ATTRACTION



Let's define a mover-attractor system

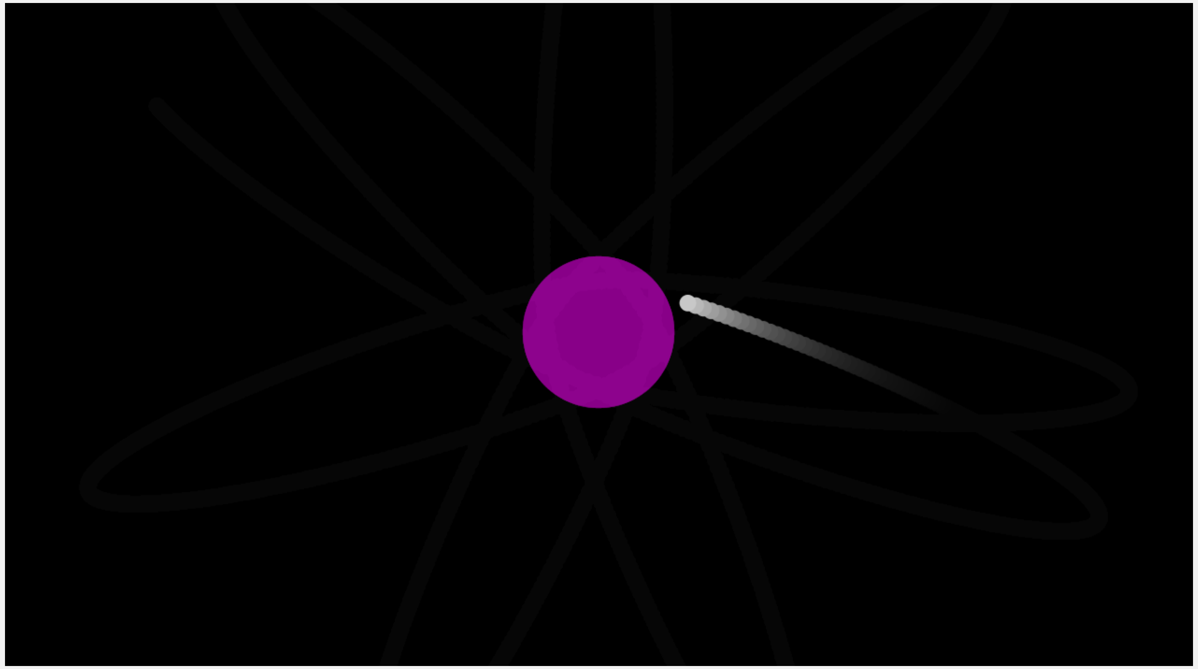
- The attractor is much bigger than the mover
  - so we neglect the force toward the mover
- Start from the previous example, **neglect the mouse**
- Place (and show) the attractor in the cer
- Place the mover
- when computing the distance, constrain
  - `dist=constrain(dist, v1,v2);`
  - map the distance to the cutoff value as `cutof`

```
# moving_ball_fluid.pde

PVector computeGravityForce(AgentMover mover){
  /* your code here*/
}
void draw(){
  PVector gravityForce=computeGravityForce(mover);
  mover.computeEffect(dist);
  /* … */
}
```

GRAVITY AND ATTRACTION

Suggestions:

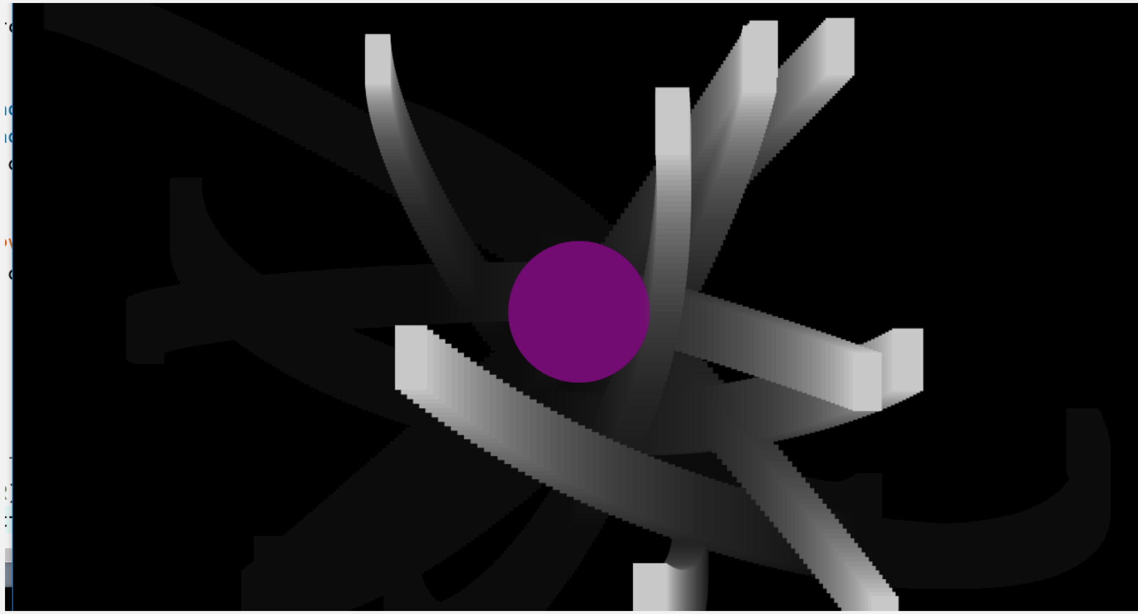
- Play with transparency
- Make the attractor follow the mouse
- If you change the sign of the force, you get a **repeller**, from which your movers run away
- Build a system with multiple attractors and repellers
    - Make attractors and repellers slowly move in the screen



PLAYING WITH ANGLES

Challenge: what if we use rectangle instead of circles for the agent?

- we need to rotate the object towards the direction of velocity
  - AgentMover.velocity.heading() → angle
  - https://processing.org/reference/PVector_heading_.html
- By rotating the whole area with pushMatrix() and popMatrix()
  - saves and restore the current screen

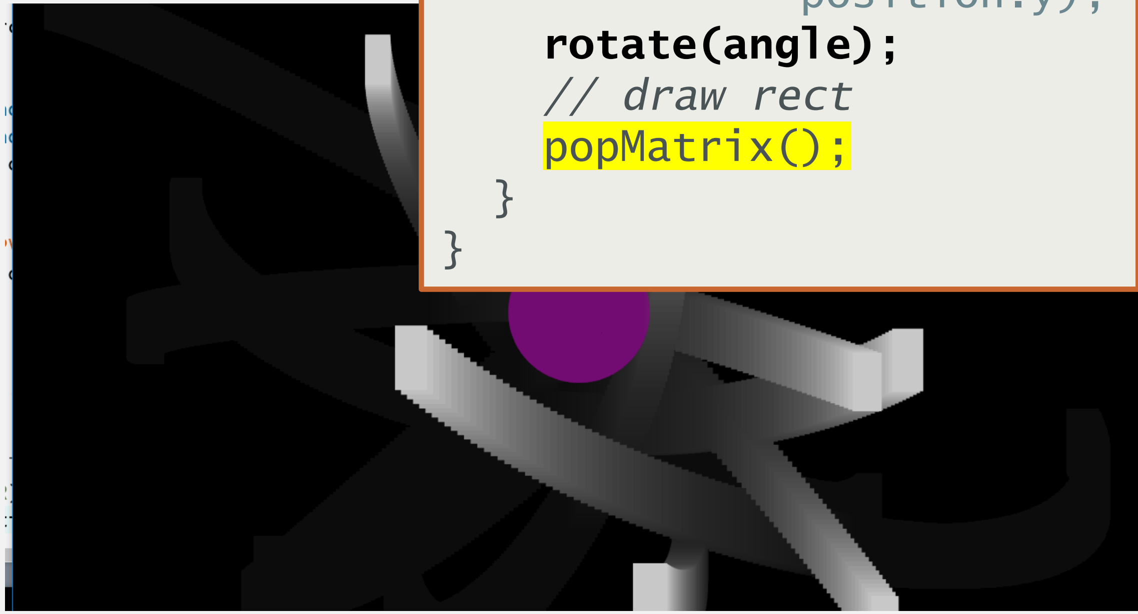


PLAYING WITH ANGLES

Challenge: what if we use rectangle instead of circles for the agent?

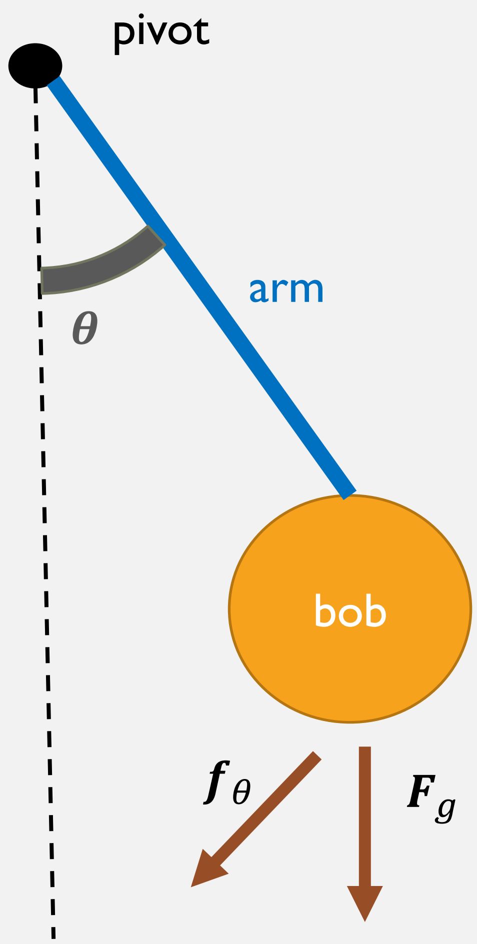
- we need to rotate the object towards the direction of velocity
  - AgentMover.velocity.heading() → angle
  - https://processing.org/reference/PVector_heading_.html
- By rotating the whole area with pushMatrix() and popMatrix()
  - saves and restore the current screen

```
# AgentMover.pde
class AgentMover{
  /* ...*/
  void action(){
    this.planning();
    pushMatrix();
    rectMode(CENTER);
    translate(position.x,
              position.y);
    rotate(angle);
    // draw rect
    popMatrix();
  }
}
```



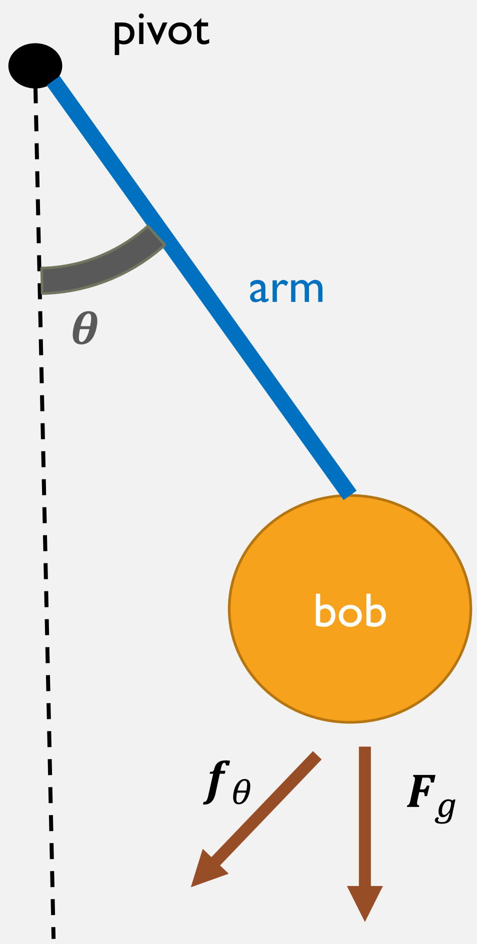
TRIGONOMETRY AND FORCES
THE PENDULUM

- Let's use forces to model a pendulum
- A pendulum is composed of a **pivot**, an arm and a mass
- When moved from rest (equilibrium), we create an angle $\boldsymbol{\theta}$
- The acceleration of the mass toward the center depends on the force of gravity $\boldsymbol{F_g}$ and the angle $\boldsymbol{\theta}$
- **Instead of acting on the 2D position of the mass, we will act on the 1D angle of the arm**
  - angle instead of position
  - ,angular velocity and angular acceleration



TRIGONOMETRY AND FORCES
THE PENDULUM

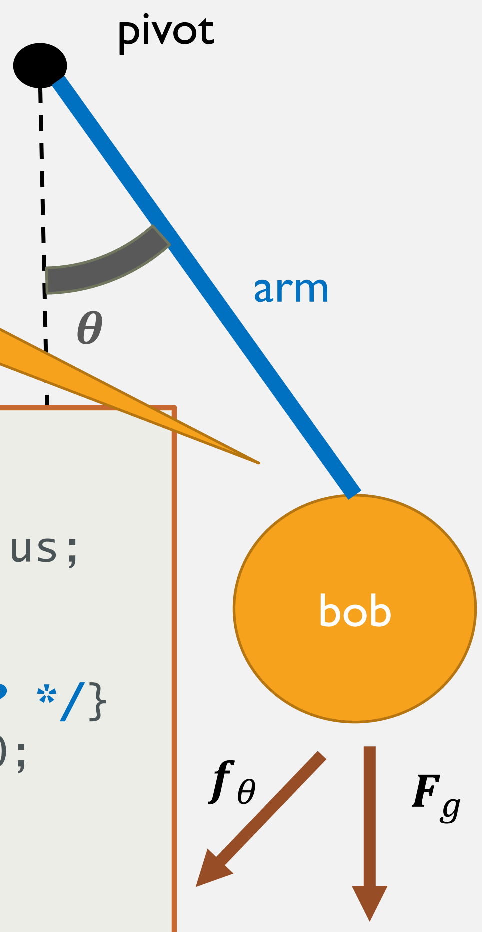
- We are in the 1-dimensional world: $a_\theta, v_\theta, \theta$ are scalars
- $f_\theta = \left(-Gm\frac{\sin(\theta)}{r}\right)$ with $G = 9.8$ and r length of the arm
- a_θ is updated by adding $\frac{f_\theta}{m}$ with m the mass of the "bob"



TRIGONOMETRY AND FORCES
THE PENDULUM

- We are in the 1-dimensional world: $a_\theta, v_\theta, \theta$
- $f_\theta = \left(-Gm\frac{\sin(\theta)}{r}\right)$ with $G = 9.8$ and r length
- a_θ is updated by adding $\frac{f_\theta}{m}$ with m the mass of the "bob"

We compute massPos as pivotPos + r cos or sin(theta)

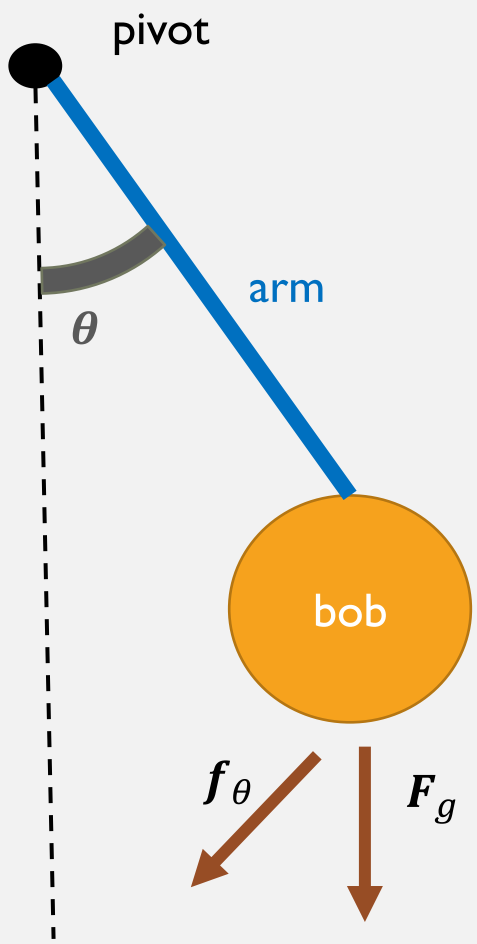



```
# AgentPendulum.pde
class AgentPendulum{
  PVector pivotPos, massPos; float angle, aVel, aAcc; float r, mass, radius;
  AgentPendulum(float x, float y, float r, float mass){
    this.pivotPos = new PVector(x, y);
    this.mass=mass; this.r=r; this.angle=random( -PI/2, -PI/4); /* other? */}
  void update(){this.aVel+=this.aAcc; this.angle+=this.aVel;  this.aAcc=0;
    this.massPos.set(this.r*sin(this.angle), this.r*cos(this.angle));
   this.massPos.add(this.pivotPos); }
  void applyForce(float force){ /* your code */ }
  void draw(){/* 1) draw pivot ; 2) draw arm with line; 3) draw mass */}
}
```

TRIGONOMETRY AND FORCES
THE PENDULUM

- We are in the 1-dimensional world: $a_\theta, v_\theta, \theta$ are scalars
- $f_\theta = \left(-Gm\frac{\sin(\theta)}{r}\right)$ with $G = 9.8$ and r length of the arm
- a_θ is updated by adding $\frac{f_\theta}{m}$ with m the mass of the "bob"

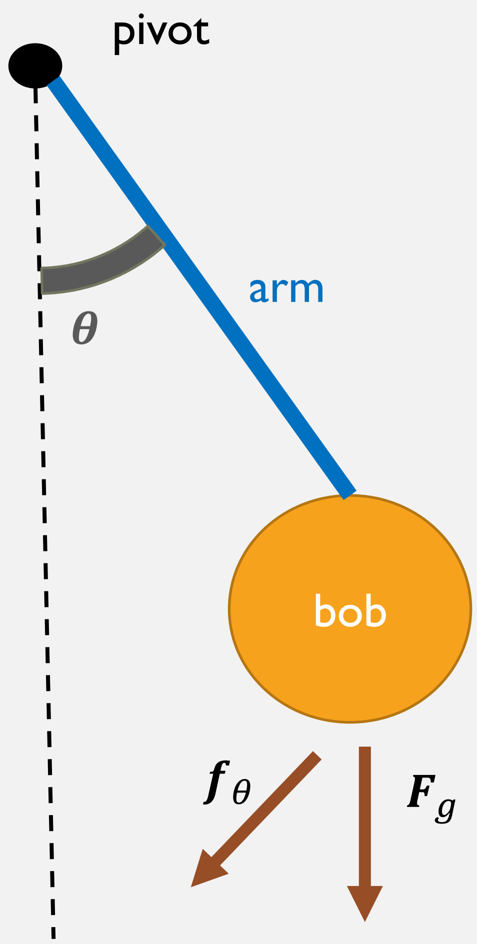
```
# pendulum.pde
AgentPendulum pendulum; float G=9.8; int MASS_TO_PIXEL=10;
void setup(){size(1280, 720); background(0);
  pendulum=new AgentPendulum(width/2, height/4, height/2, 100); }
float computeForce(AgentPendulum pendulum){/* your code */}
void draw(){rectMode(/* ... */
  float force= computeForce(pendulum);
  pendulum.applyForce(-1*G*sin(pendulum.angle)/pendulum.r);
  pendulum.update();
  pendulum.draw();
}
```



TRIGONOMETRY AND FORCES
THE PENDULUM

What kind of musical parameter can you map into a pendulum movement?

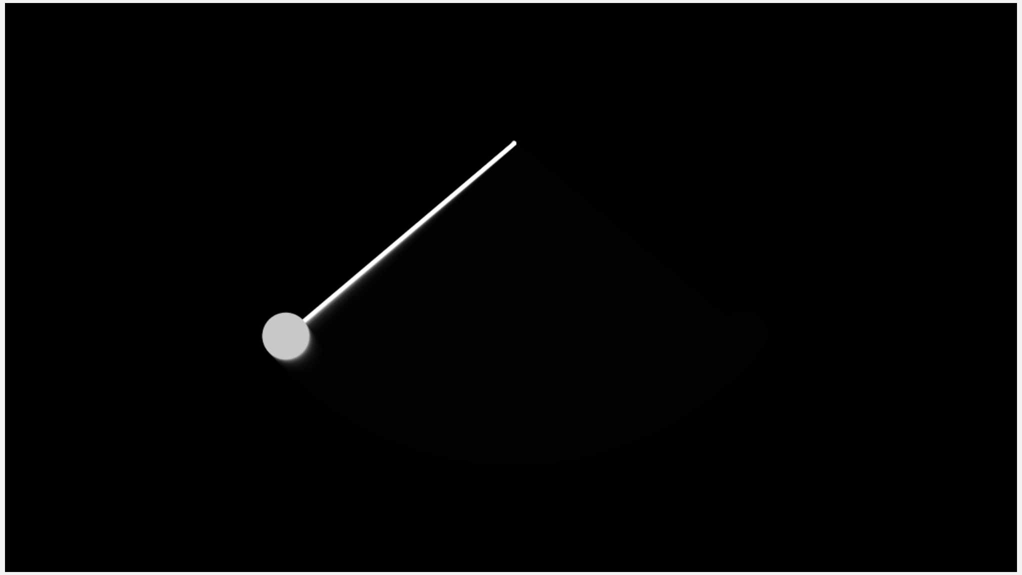
- The pendulum oscillates around the theta = 0
  - map theta into the vibrato
  - map magnitude of angular velocity into cutoff



TRIGONOMETRY AND FORCES THE PENDULUM

The pendulum will go on forever

- Apply a dumping 0.99 to velocity at every step
- If the motion is too fast, which parameters you can change to slow it down?



DEFINITION OF THE INSTRUMENT

DEFINITION OF THE INSTRUMENT

Previously on MAE…

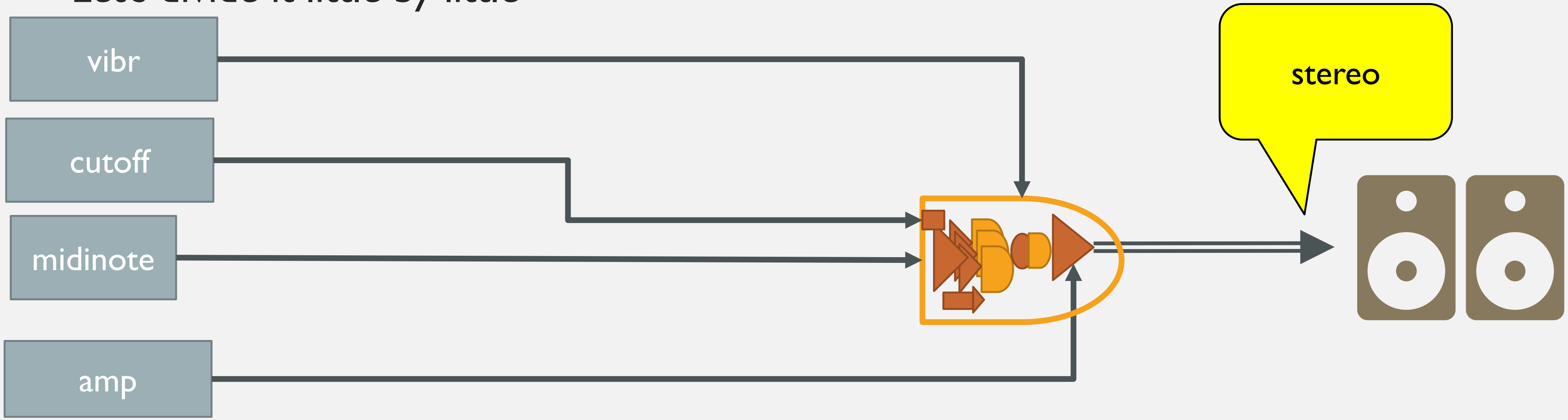
- Super Collider is a framework composed of two elements
  - A language with editor and interpreter
  - A server for sound synthesis
- You use the language to create instruments and melodies, patterns, effects
- Than you send the commands to the server for executing it
- Useful shortcuts (in macOS CMD=CTRL)

- CTRL + B → server boot
- CTRL+N → new script
- CTRL+ENTER → execute line/block (in round parenthesis)
- CTRL + . → stop execution
- CTRL + / → comment/decomment line(s)
- CTRL + SHIFT + P →clean the interpreter window

DEFINITION OF THE INSTRUMENT

In super_collider_instrument you can find moogs.scd, with the definition of an instrument:

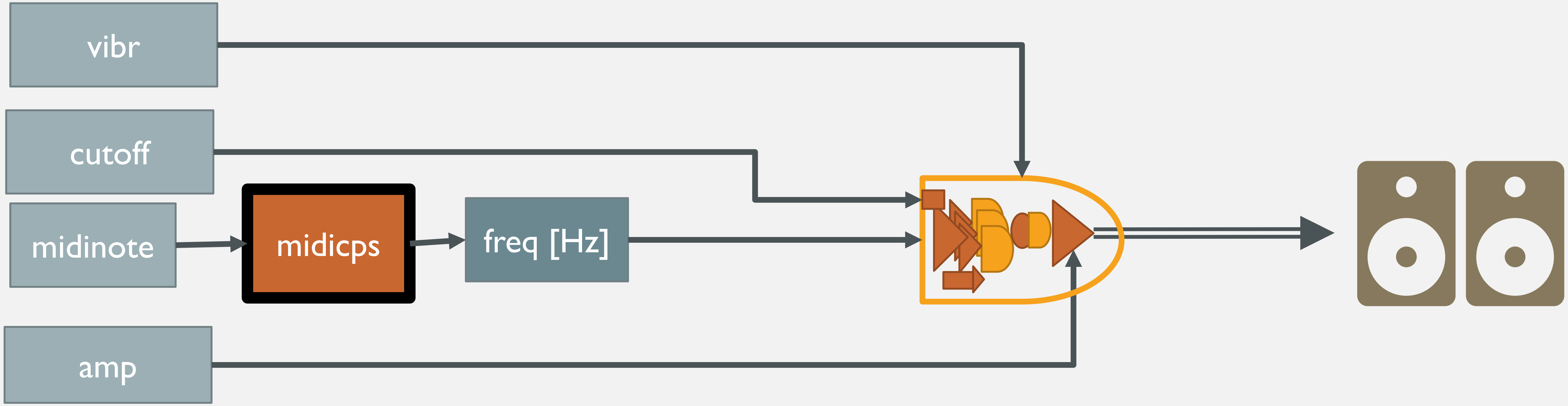
- `arg vibr=0, cutoff=0.5, midinote=60, amp=0;`
- Arguments are: midinote, vibr (vibrato), cutoff, amp (amplitude)
- Let's divide it little by little



DEFINITION OF THE INSTRUMENT

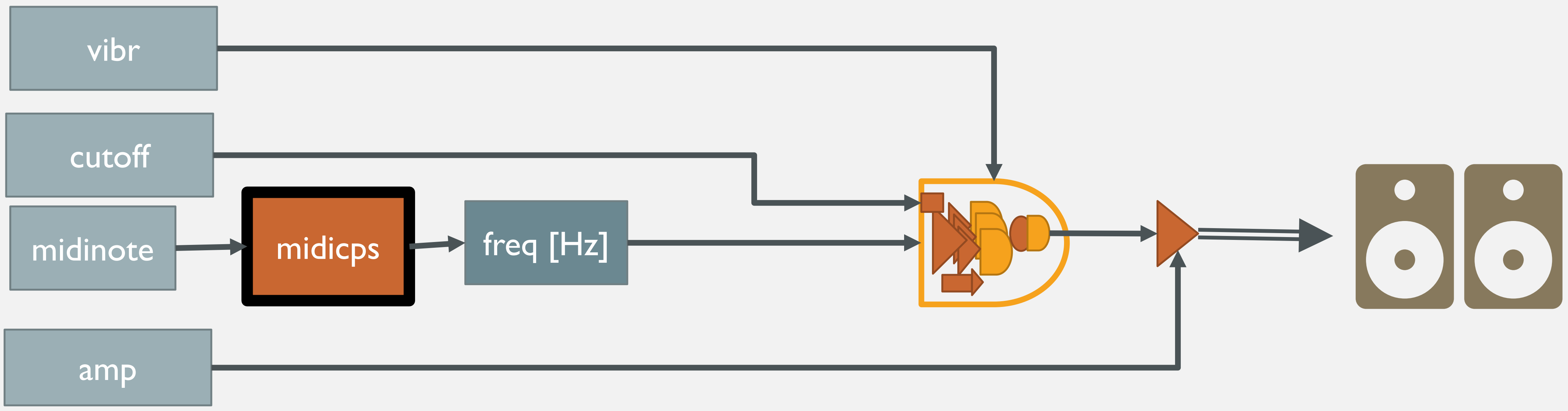
First: midinote is converted to a frequency in Hz with the `midicps` function

- Input a midinote, returns the pitch frequency



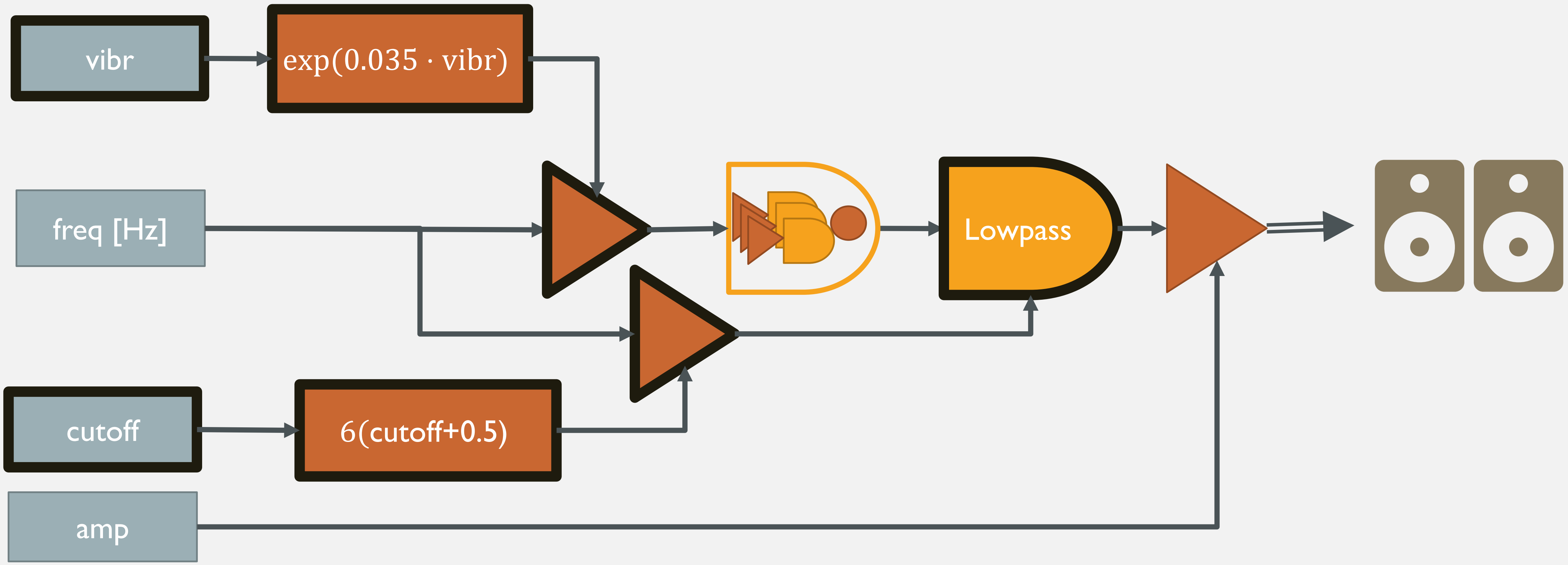
DEFINITION OF THE INSTRUMENT

Secondly, the amplitude is multiplied to the output of the instrument to control the volume



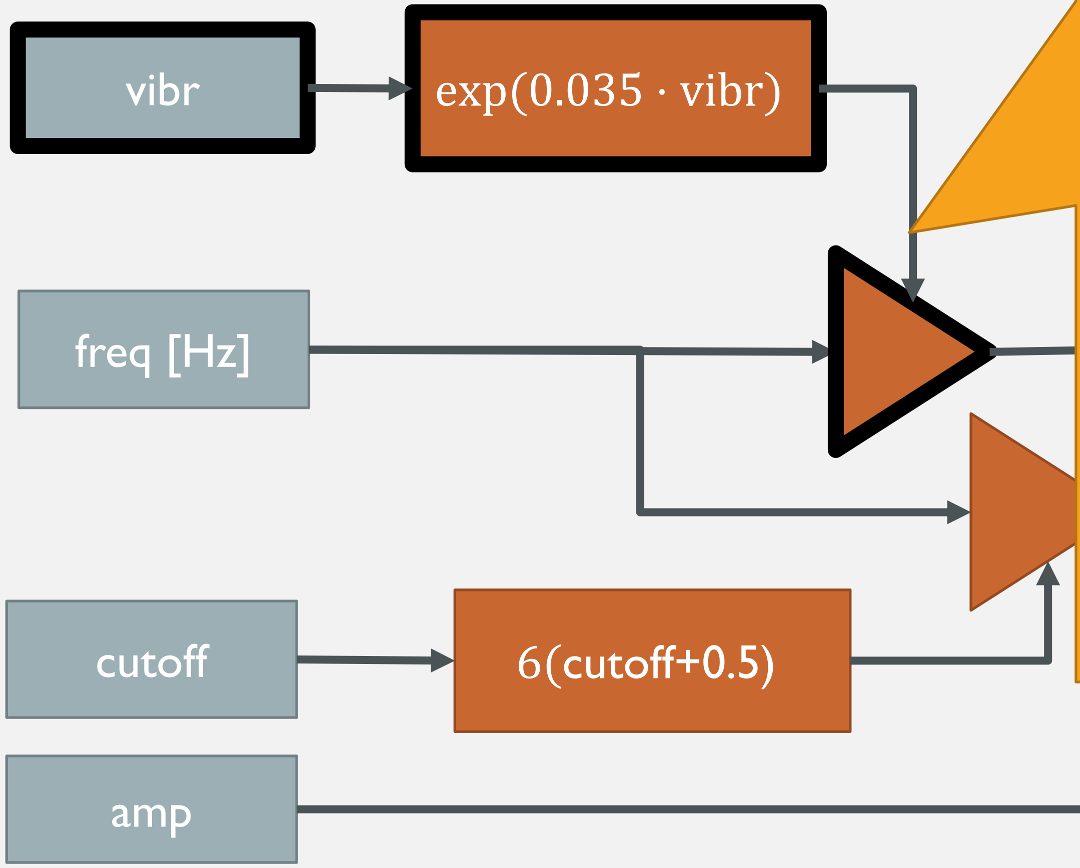
DEFINITION OF THE INSTRUMENT

Vibrato is a number around 0 that allows to jiggle around the correct pitch



DEFINITION OF THE INSTRUMENT

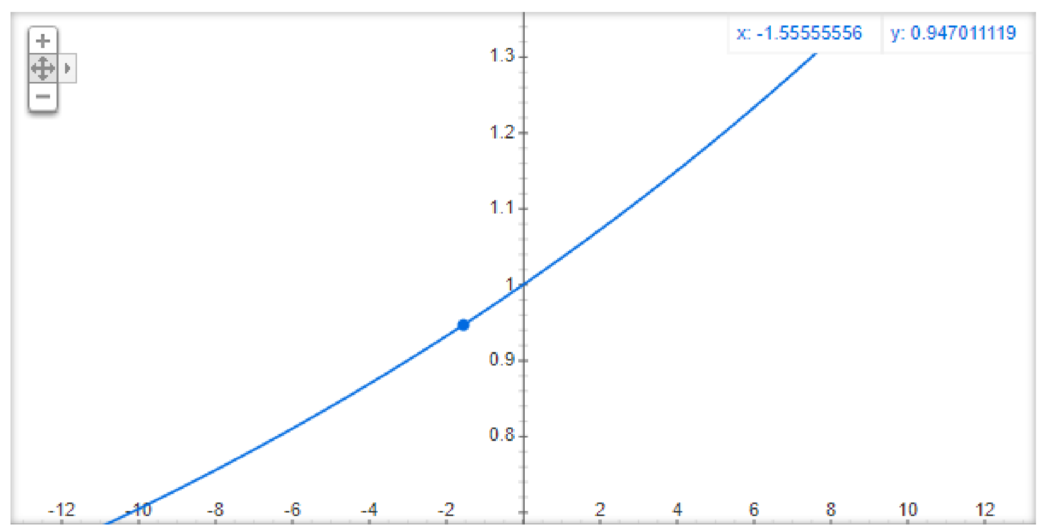
Let's change the moog to include vibrato and wah-wah effect



This is the responsible for the vibrato

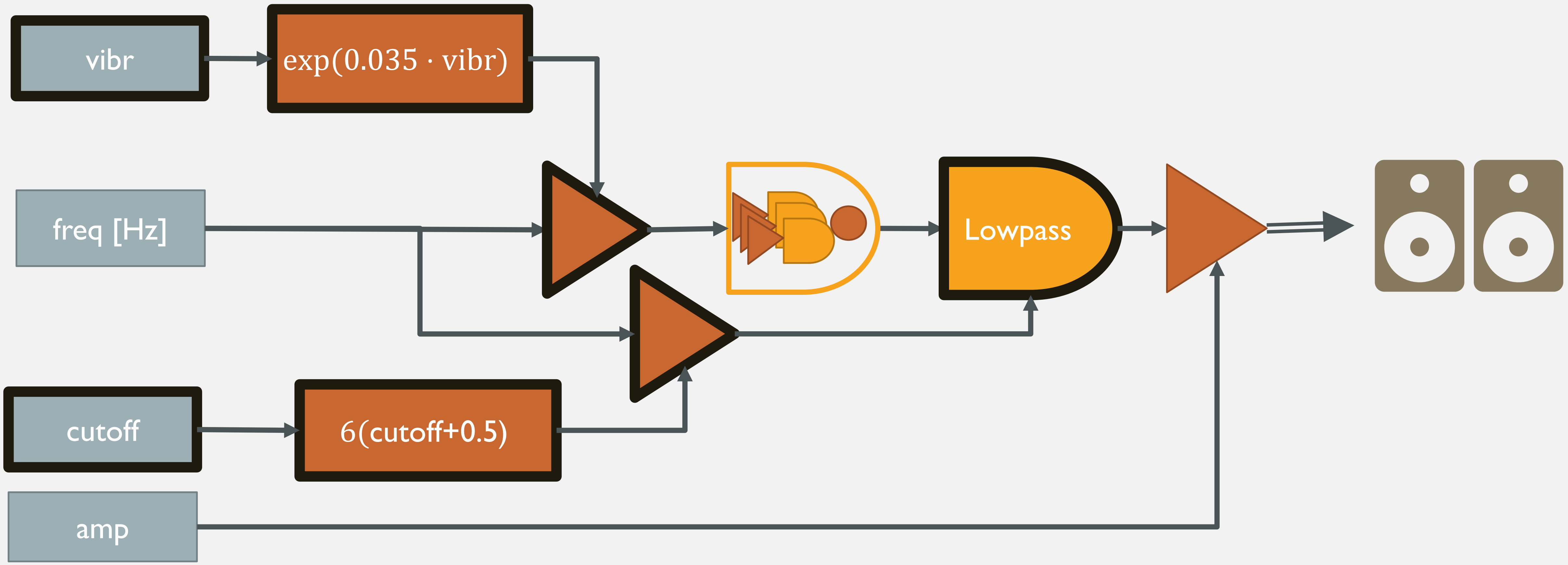
When vibr is 0 → pitch = freq → no vibrato.

Vibr oscillates around 0 → pitch oscillates around freq → vibrato



DEFINITION OF THE INSTRUMENT

Cutoff is a factor that controls how many frequencies to keep in a lowpass, and gives a wah-wah effect



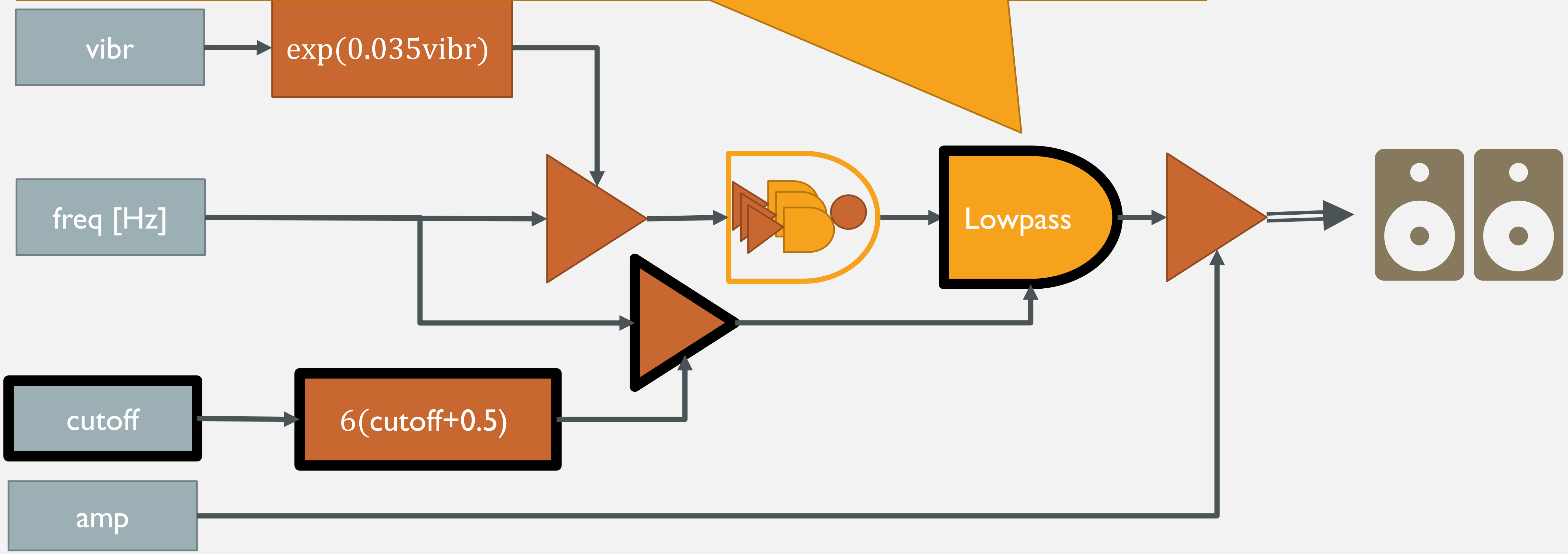
This is the wah-wah like

We insert a lowpass filter that cuts a variable number of harmonics.

Cutoff=0 → cutoff frequency = 6 (cutoff+0.5) freq = 3 freq → we keep three harmonics

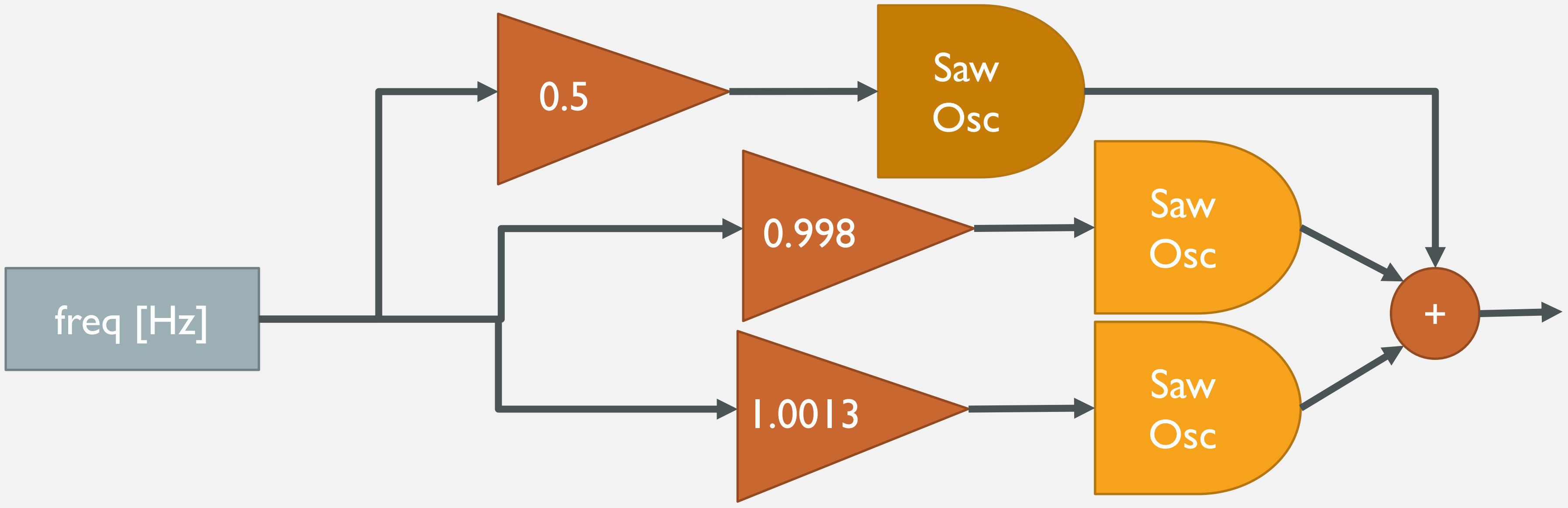
Cutoff = 1 → cutoff frequency = 9 freq → we keep nine harmonics

By varying *cutoff* dynamically we create an open-close effect that sounds like a wah-wah



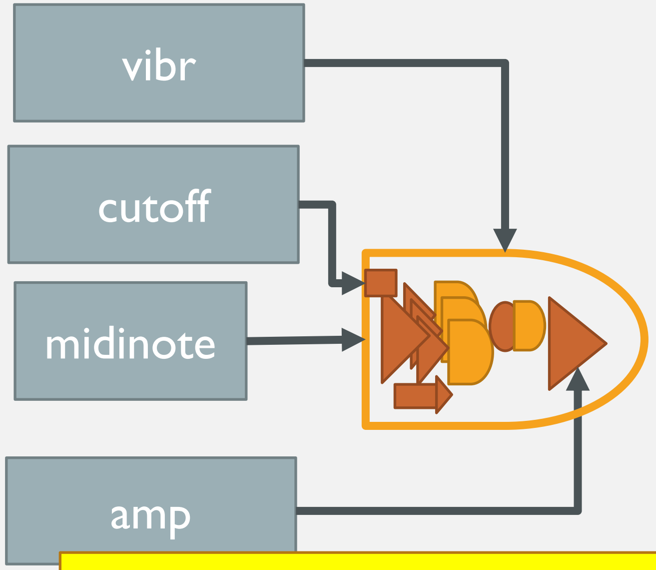
DEFINITION OF THE INSTRUMENT

- Lastly, the instrument is based on three saw oscillators:
  - two generate the tone (slightly out-of-tune to create baptiments)
  - One is a sub-harmonic (half the pitch frequency)



DEFINITION OF THE INSTRUMENT

Final implementation

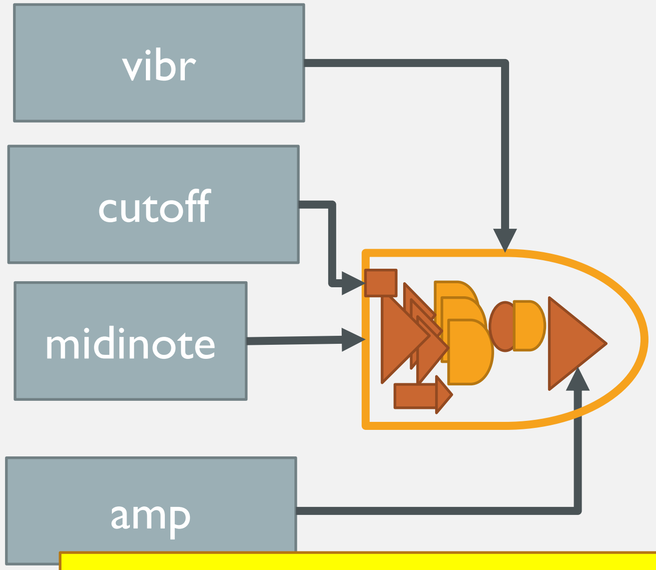



```
# moogs.scd

(SynthDef("moog", {
    arg vibr=0, cutoff=0.5, midinote=60, amp=0;
    var osc1, osc2, osc3, f0, vib_int, cutoff_freq, fil_osc, freq;
    freq=midicps(midinote);
    f0=exp(vibr*0.035)*freq;

    osc1=Saw.ar(f0*1.0013);
    osc2=Saw.ar(f0*0.998);
    osc3=Saw.ar(f0*0.5);

    cutoff_freq=((cutoff+0.5)*6)*freq;

    fil_osc=BLowPass.ar(in:osc1+osc2+osc3,
                        freq:cutoff_freq.min(20000));

    Out.ar([0,1]          fil_osc);}).add;
)
```

Why?

DEFINITION OF THE INSTRUMENT

Final implementation




```
# moogs.scd

(SynthDef("moog", {
    arg vibr=0, cutoff=0.5, midinote=60, amp=0;
    var osc1, osc2, osc3, f0, vib_int, cutoff_freq, fil_osc, freq;
    freq=midicps(midinote);
    f0=exp(vibr*0.035)*freq;

    osc1=Saw.ar(f0*1.0013);
    osc2=Saw.ar(f0*0.998);
    osc3=Saw.ar(f0*0.5);

    cutoff_freq=((cutoff+0.5)*6)*freq;

    fil_osc=BLowPass.ar(in:osc1+osc2+osc3,
                        freq:cutoff_freq.min(20000));

    Out.ar([0,1]        fil_osc);}).add;
)
```

This will choose the minimum between the cutoff_freq and 20,000 Hz and therefore avoid the cutoff frequency gets too high

DEFINITION OF THE INSTRUMENT

```
# moogs.scd
// test it
(
~instr=Synth(\moog);
)
// setting the note
(
~instr.set(\midinote, 62, \amp, 1);
)
//setting the cutoff
(
~instr.set(\cutoff, 3);
)
```

```
# moogs.scd
// Creating an OSC receiver (Python only)
NetAddr("127.0.0.1",57120);

(
OSCdef('OSCreceiver',
    {
        arg msg;
        var note, amp, cutoff, vibr;
        msg.postln;
        note=msg[2];
        amp=msg[3];
        ~instr.set(\midinote,note,
                         \amp,amp);
    },
    "/note_effect",);
)
```