POLITECNICO MILANO 1863

IMAGE AND SOUND PROCESSING GROUP (ISPG)

CREATIVE PROGRAMMING AND COMPUTING

Lab: Reactive Agents

REACTIVE AGENTS

- Definition of the instrument
- Music composition with Python
- Physics-related reactive agents

DEFINITION OF THE INSTRUMENT

MUSIC COMPOSITION WITH PYTHON

- In this class we will use Python to generate a time sequence of values, using a deterministic function with a spice of randomness
- We then map the generated values, usually within a range, in meaningful parameters for a composition, usually pitch and durations
- Then we send those pitches to SuperCollider to generate a meaningful music composition
- Let's first create the instrument in SuperCollider

MUSIC COMPOSITION WITH PYTHON

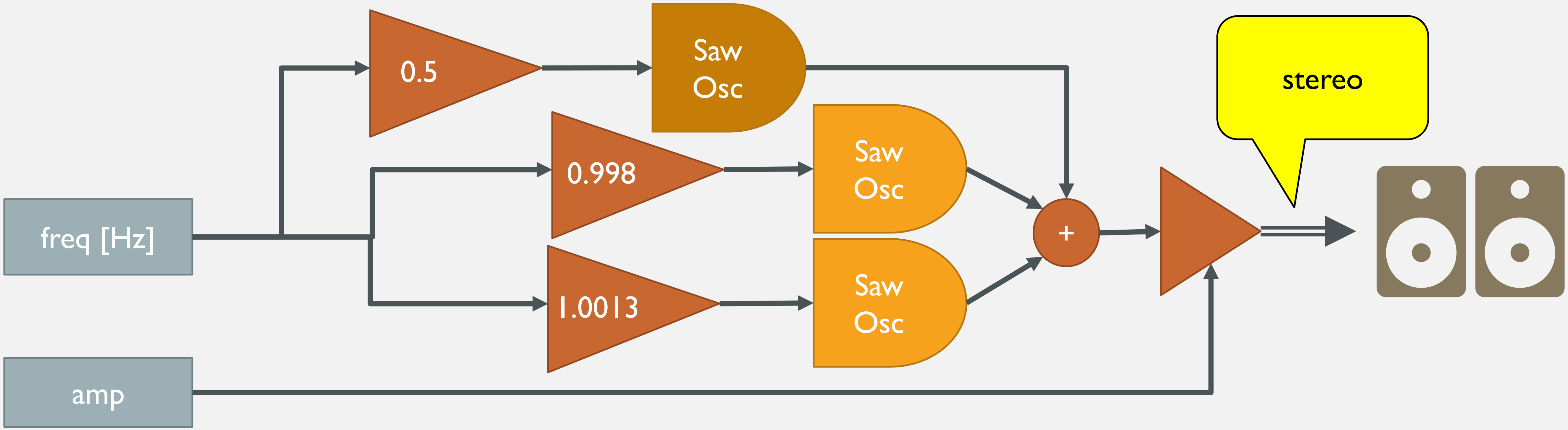
Previously on MAE…

- 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

MUSIC COMPOSITION WITH PYTHON

- 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)

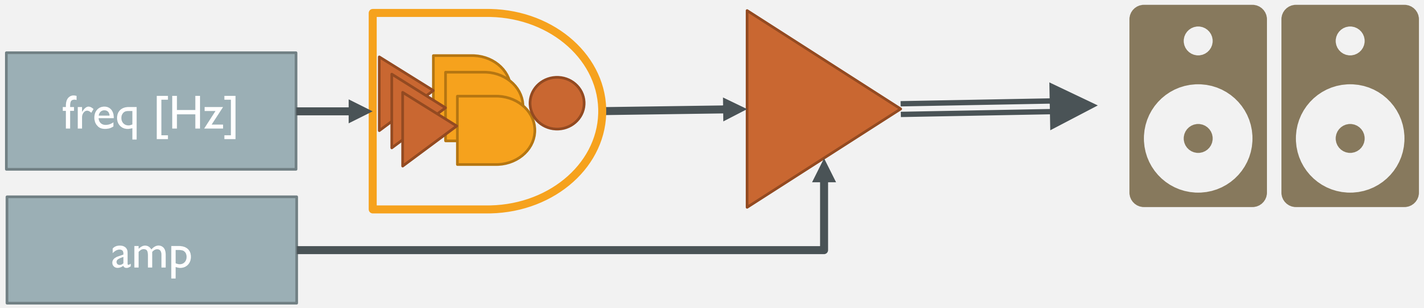


MUSIC COMPOSITION WITH PYTHON

Let's make a big block out of it

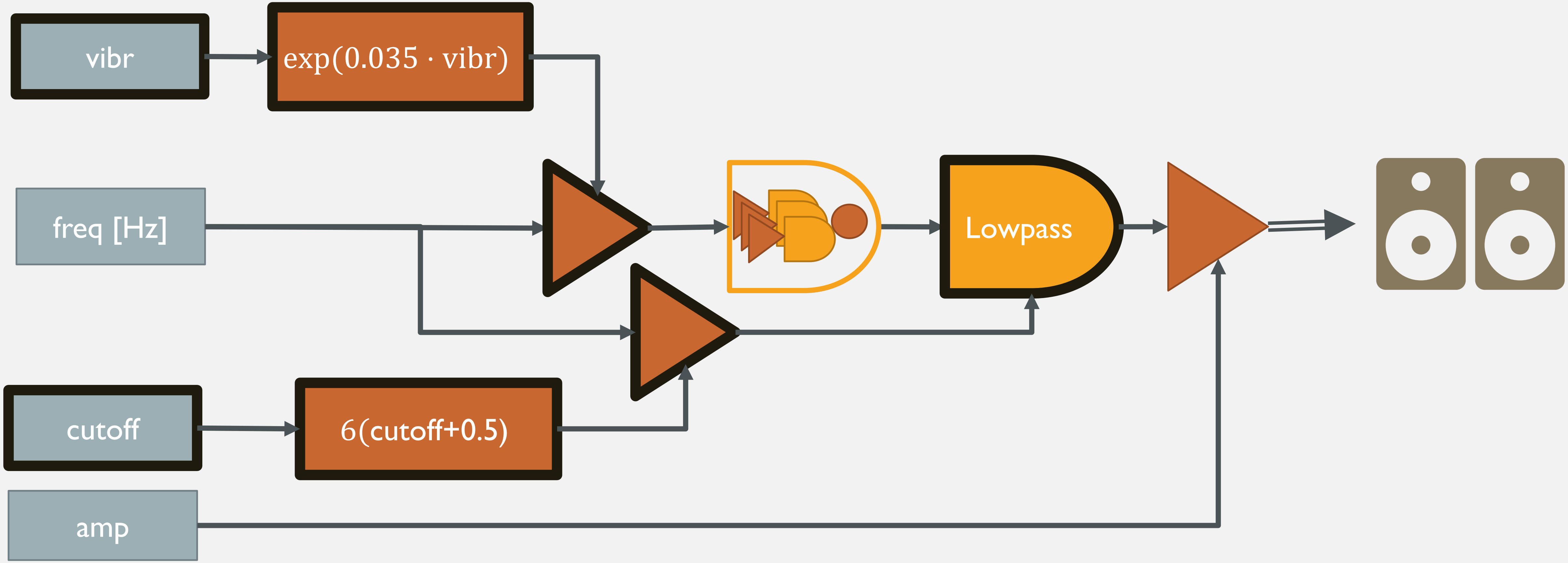
- Now, this is an interesting instrument
- Let's add a few elements more:
  - A vibrato
  - A wah-wah effect

```
# moogs.scd

(SynthDef("moog1", {
 arg freq=440, amp=0;
 var osc1, osc2, osc_sub;
 osc1=Saw.ar(1.0013*freq);
 osc2=Saw.ar(0.998*freq);
 osc_sub=Saw.ar(0.5*freq);
 osc=osc1+osc2;
 Out.ar([0,1], amp*osc);}).add;)

(var instr = Synth(\moog1);
instr.set(\amp,1);)
```



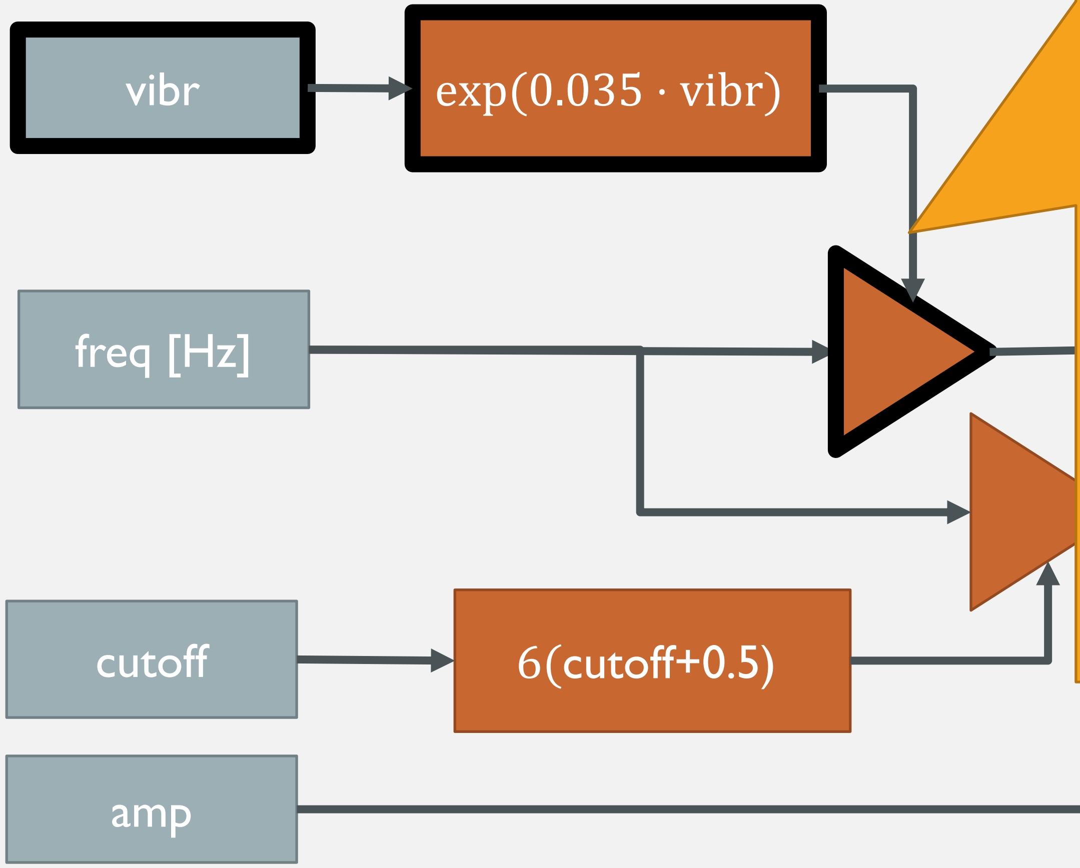
MUSIC COMPOSITION WITH PYTHON

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



MUSIC COMPOSITION WITH PYTHON

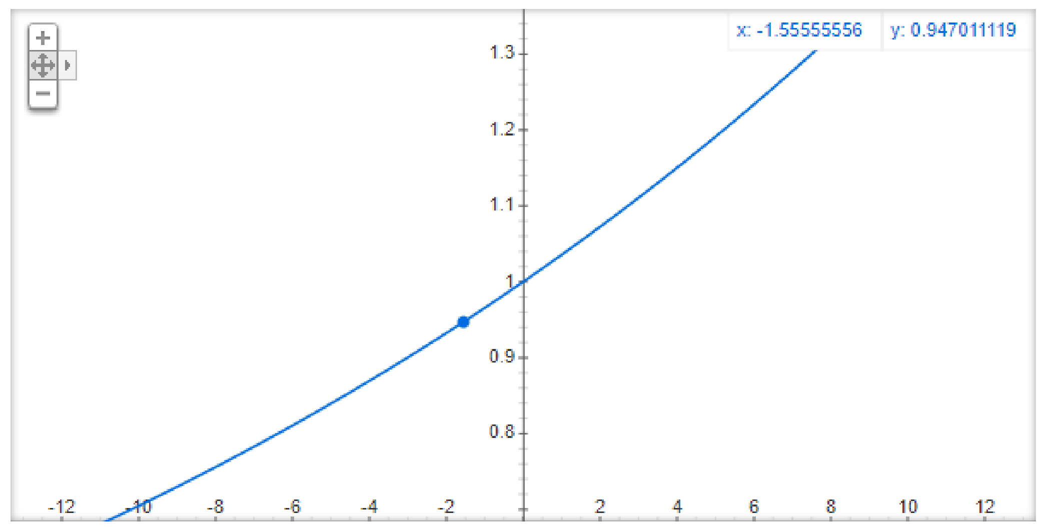
Let’s change the moog to include vibrato and wah-wah effect



This is the responsible for the vibrato

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

Vibr oscillates around 0 → pitch oscillates around freq → vibrato



This is the wah-wah like

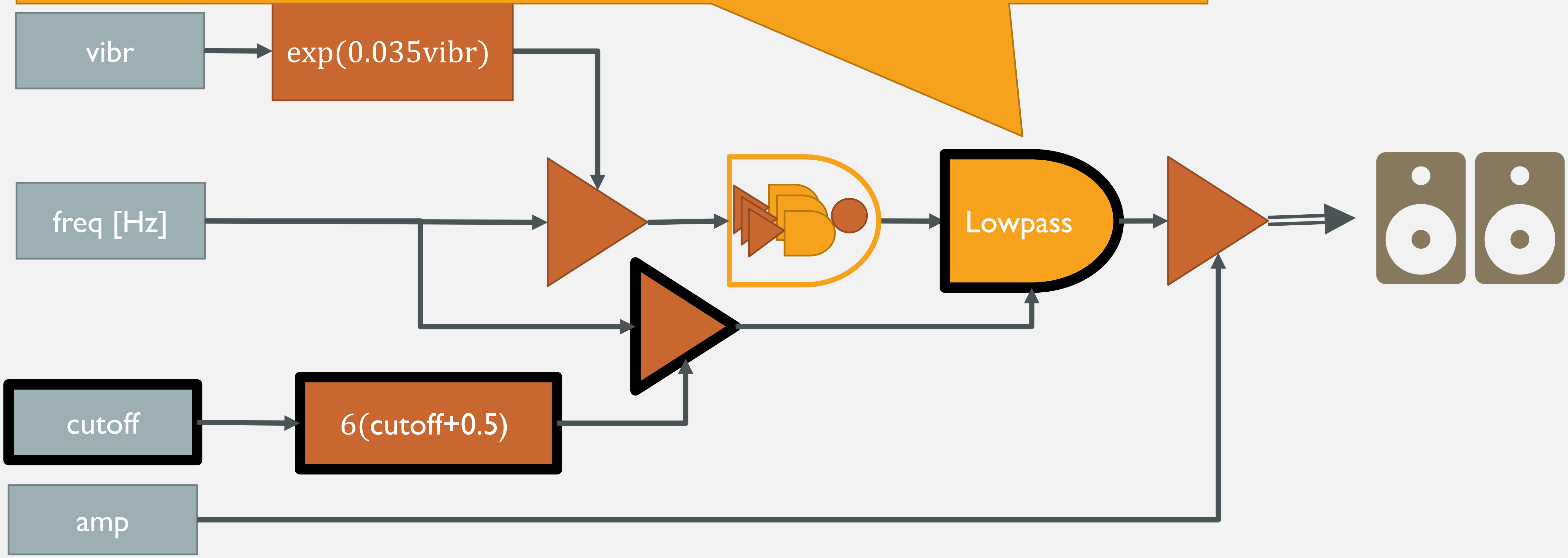
We insert a lowpass filter that cuts a variable number of harmonics.

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

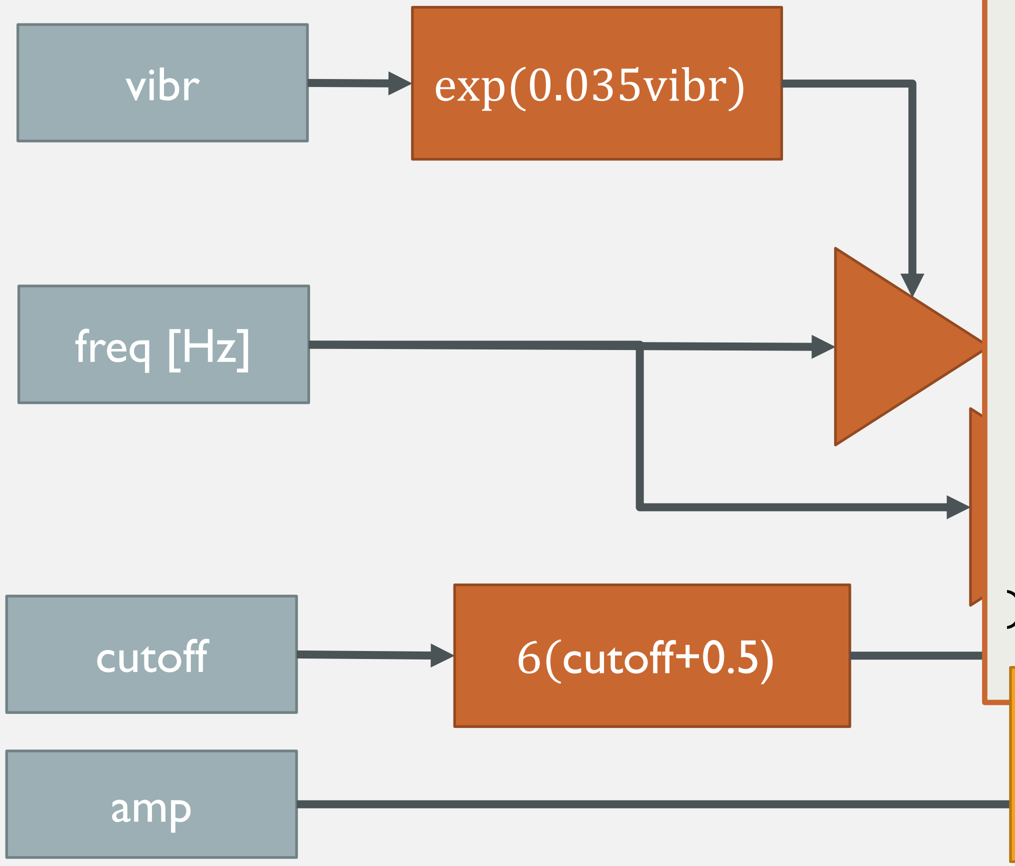
By varying *cutoff* dynamically we create an open-close effect that sounds like a wah-wah

```
BLowPass.ar(in, freq);
```



MUSIC COMPOSITION WITH PYTHON

Let's change the moog to include vibrat



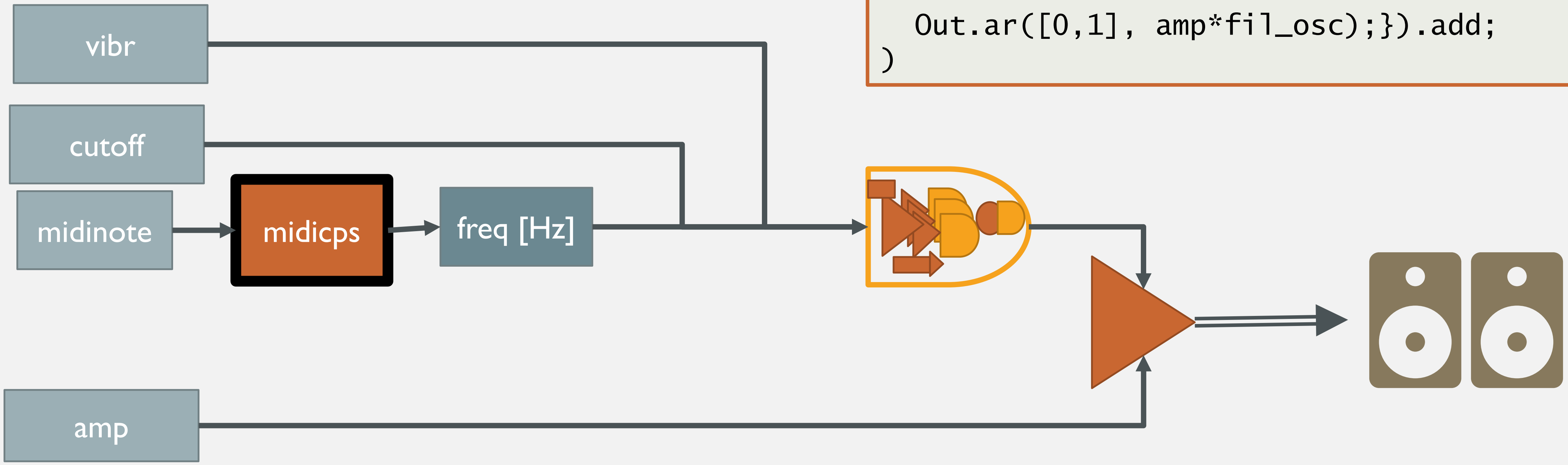

```
(SynthDef("moog", {
 arg vibr=0, cutoff=0.5, freq=440, amp=0;
  var osc1, osc2, osc3, f0, cutoff_freq, fil_osc;
  f0=exp(vibr*(0.035*vib_int))*freq;

  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], amp*fil_osc);}).add;
)
```

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

MUSIC COMPOSITION WITH PYTHON

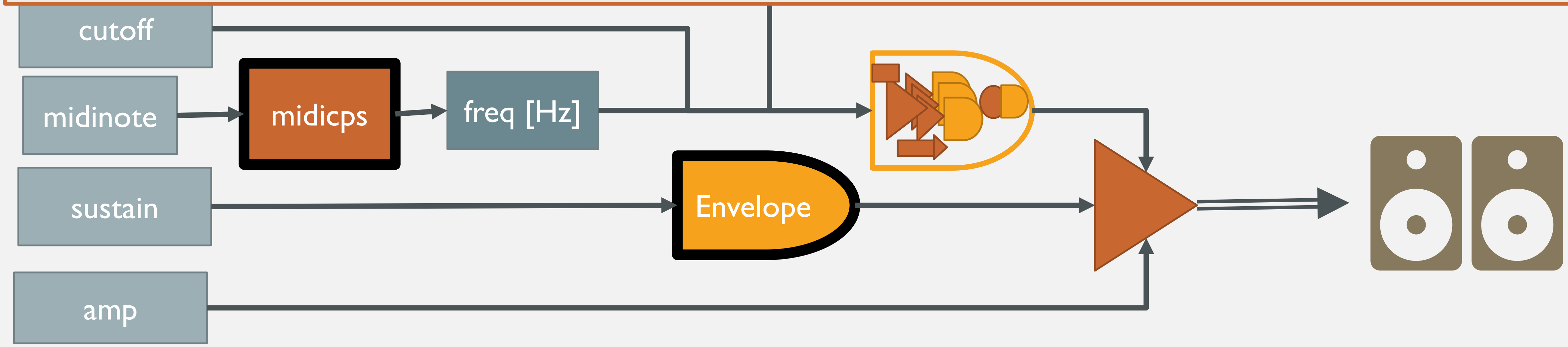
Lastly we replace freq with midinote

```
# moogs.scd
(SynthDef("moog", {
  freq=midicps(midinote);

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

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



MUSIC COMPOSITION WITH PYTHON

```
# moogs.scd
(SynthDef("moog", {
    arg vibr=0, cutoff=0.5, midinote=60, amp=0, sustain=1;
    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));
    fil_osc= fil_osc;
    Out.ar([0,1], amp*fil_osc);}).add;)
```



MUSIC COMPOSITION WITH PYTHON

Let's first test it

Nice, but boring

We need to add the OSC receiver

```
# moogs.scd
(
~instr=Synth(\moog);
)

// setting the note
(
~instr.set(\midinote, 62, \amp, 1);
)

//setting the cutoff
(
~instr.set(\cutoff, 3);
)
```

MUSIC COMPOSITION WITH PYTHON

This is the PORT. Look carefully at it

Let's first test it

We will just use note and amp for now, but bear with me

Nice, but boring

We need to add the OSC receiver

Let's test it with Python

OSC

```
# moogs.scd
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",);
)
```

MUSIC COMPOSITION WITH PYTHON

MUSIC COMPOSITION WITH PYTHON

The architecture of our Python script for automatic music composition is the following:

- Look at compute_music.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
pars : information from the user

MUSIC COMPOSITION WITH PYTHON

- See the class Status on the right
  - *id* is the ID, and starts from `ID_TART=-1`
  - *midinote* is the current midinote
  - *dur* is the duration of the note in beats with reference to BPM
  - *amp* is the amplitude
  - *pars* is a dictionary where you can store any further information you want, such as, for example
    - the ten past status
    - A histogram of the notes currently played
    - An offset for the midinote to change tonality
    - Etc.

```python
# compute_music.py

class Composition:
    def __init__(self):
        self.id=-1
        self.midinote=-1
        self.dur=0
        self.amp=0
        self.BPM=120.
        self.pars={}
```

MUSIC COMPOSITION WITH PYTHON

Idea:

- we define a compotisioin and its initial set of parameters (note, effect, etc)
- we write a function that at each step changes the parameters depending on what is playing in that moment
- We send the new parameters to the instrument
- we sleep to the time required to play the note (the duration)

We change the generated music by changing the function that updates the composition

See the rest of the architecture

MUSIC COMPOSITION WITH PYTHON

The architecture of our Python script for automatic music composition is the following:

- Look at compute_music.py

main
- Create a number of agents (melodic lines) passing them the compositional algorithm
- Start them
- Wait for interruption from user

Agent(thread)

Attributes

instr: an instr OSC

comp the composition at the current moment

func: the function to update the composition

Methods

planning: calls func with the comp to update the notes

action (main thread function): while the thread is active :
- it calls *planning* to update the composition
- it sends the data of the notes to the instrument
- It sleeps for the duration of the note

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

pars : information from the user

MUSIC COMPOSITION WITH PYTHON

The architecture of our Python script for automatic music composition is the following:

- Look at compute_music.py

This is the script you need to run!

main

- Create a number of agents (melodic lines) passing them the compositional algorithm
- Start them
- Wait for interruption from user

Agent(thread)

Attributes

instr: an instr OSC

comp the composition at the current moment

func: the function to update the composition

Methods

planning: calls func with the comp to update the notes

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

- it calls *planning* to update the composition
- it sends the data of the notes to the instrument
- It sleeps for the duration of the note

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

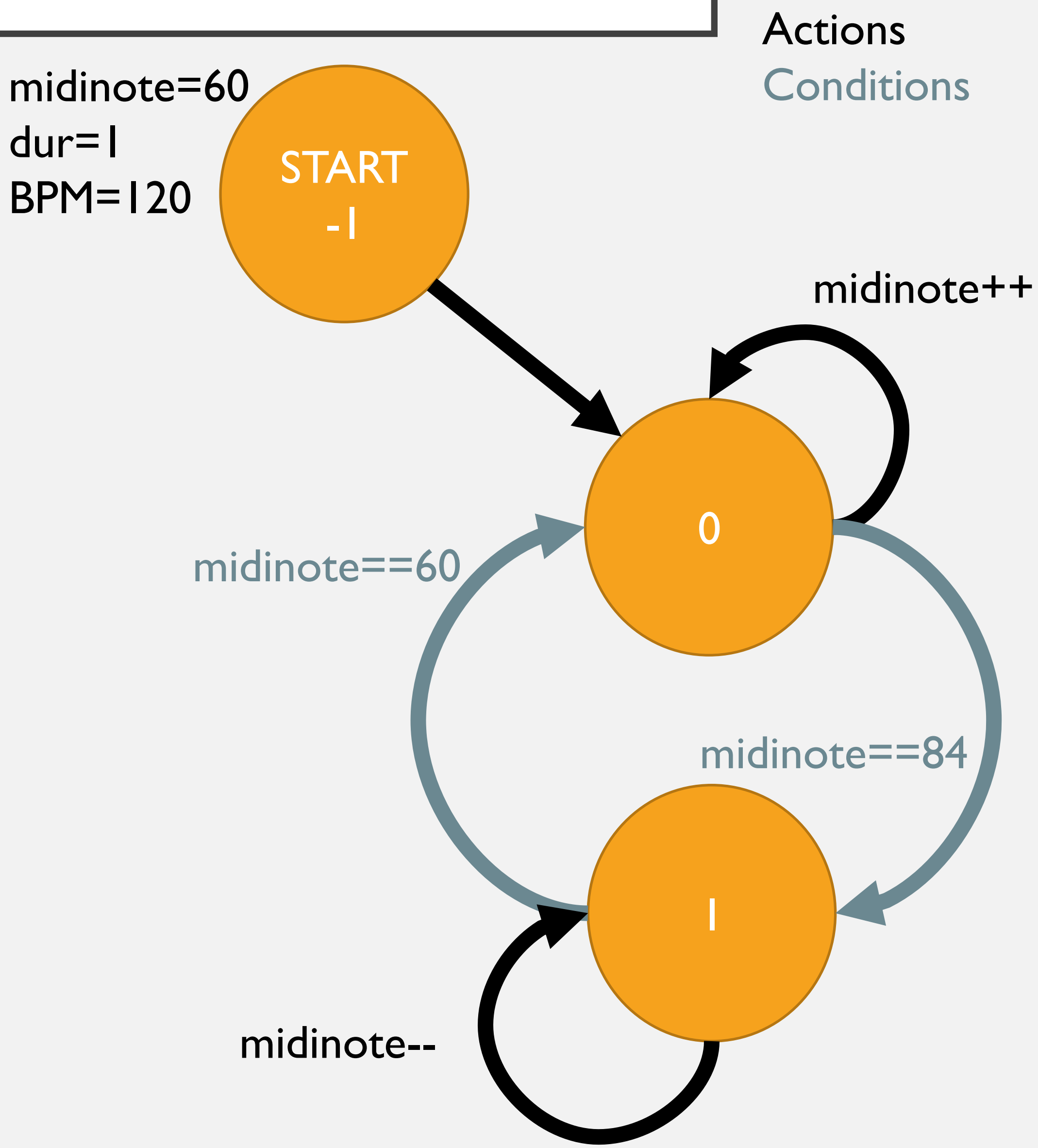
BPM: beats per minute

pars : information from the user

MUSIC COMPOSITION WITH PYTHON

Exercize 1:

- 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…

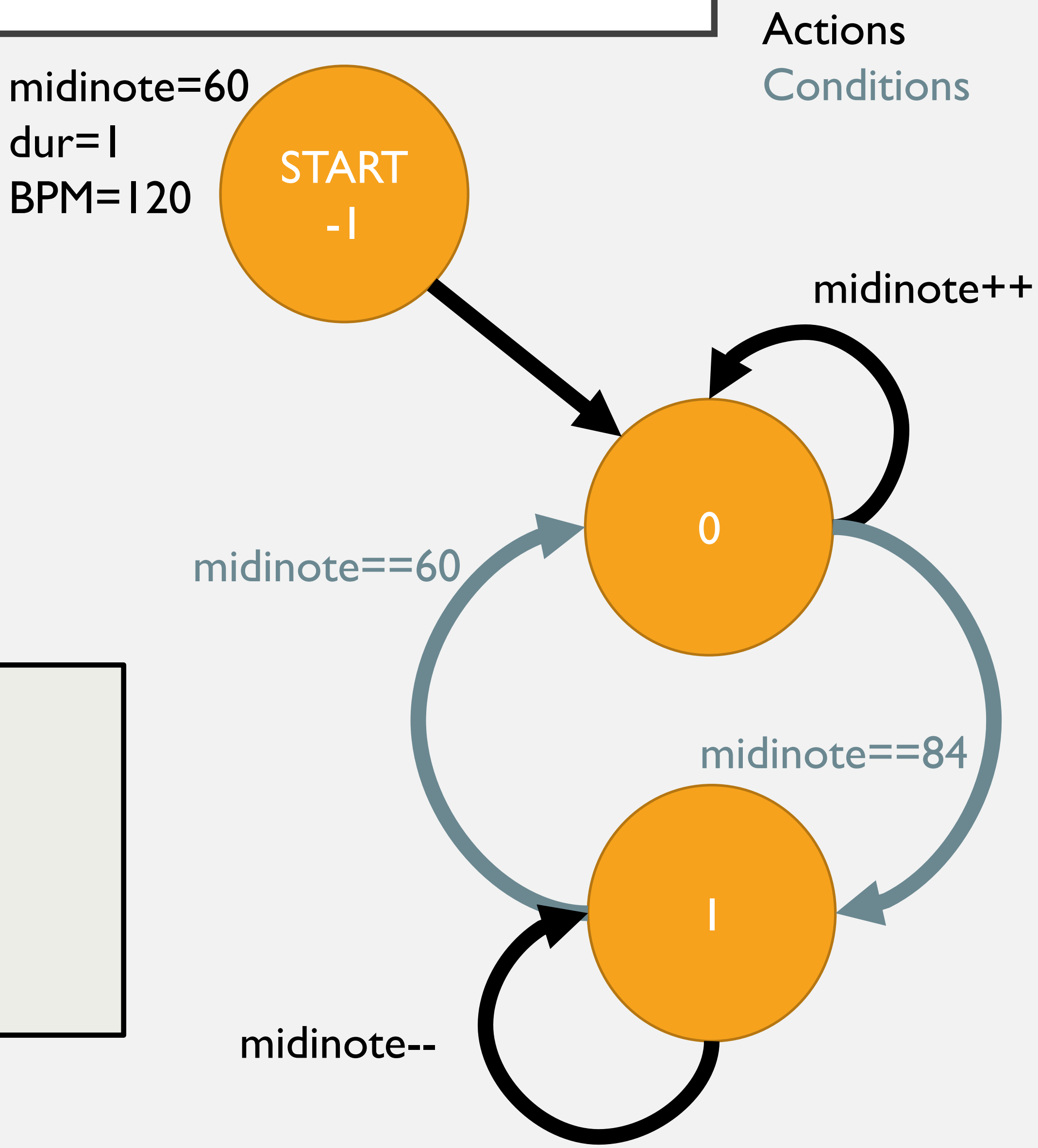


MUSIC COMPOSITION WITH PYTHON

Exercize 1:

```
# your_code.py
def simple_next(comp):
    if comp.id==STATUS_START:
        comp.midinote=60
        comp.current=0
        comp.dur = 1
        comp.amp =1
        comp.BPM = 120
    elif comp.id==0:
        # your code
```

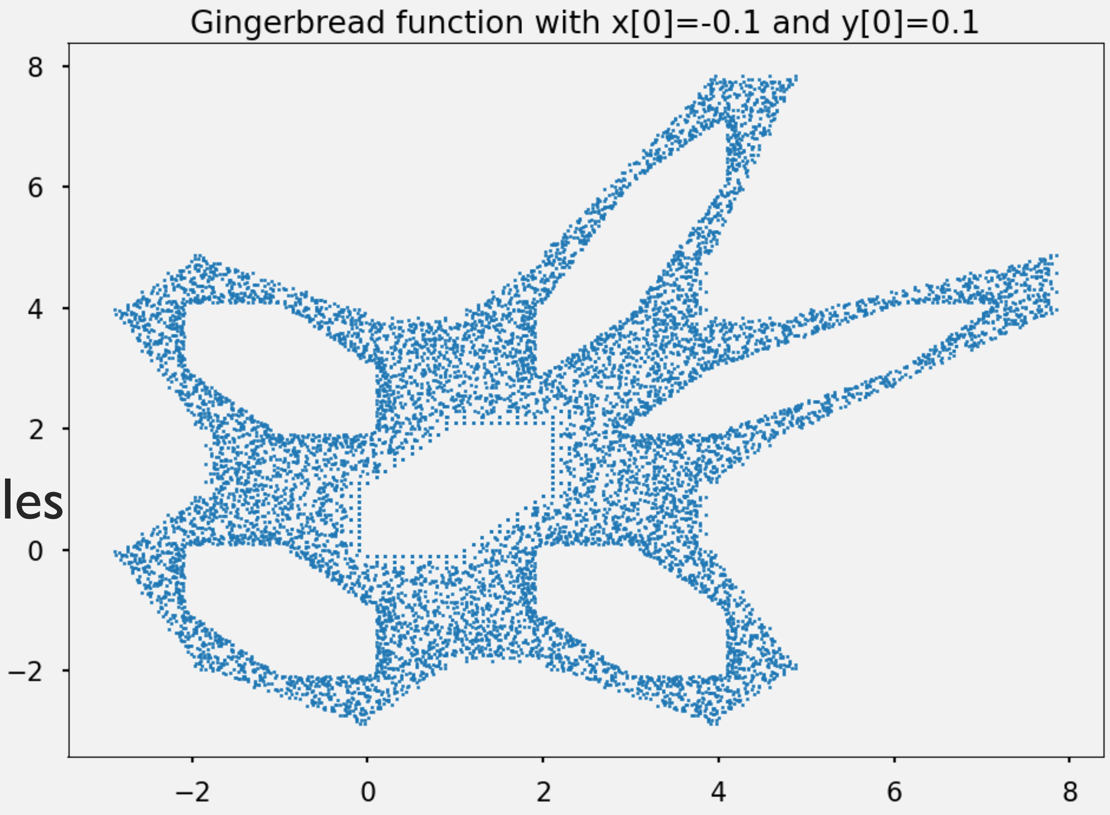
```
# main.py

#...
n_agents=1
agents=[_ for _ in range(n_agents)]
agents[0] = Agent(57120, "/note", 60, simple_next)
#...
```



MUSIC COMPOSITION WITH PYTHON

Exercize 2: Let's use the Gingerbread function

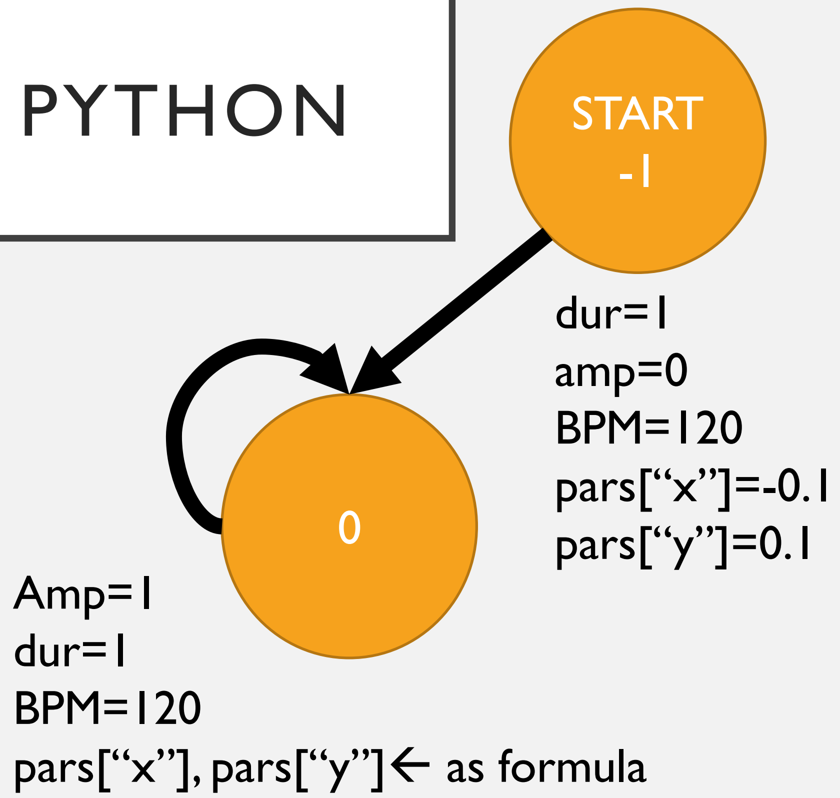
- Two variables $x(n)$ and $y(n)$ with n is a count
  - $x(0) = -0.1$ and $y(0) = 0.1$
- At each step
  - $x(n+1) = 1 - y(n) + |x(n)|$
  - $y(n+1) = x(n)$
- We need to map $x(n)$ and/or $y(n)$ to music variables
  - They range between -3 and 8 in this case



MUSIC COMPOSITION WITH PYTHON

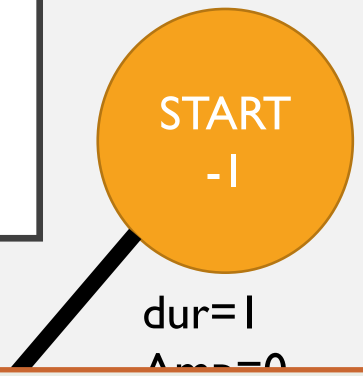
Exercize 2: Let's use the Gingerbread function

- At beginning, initialize the status and pars in status with
  - The value of x and y
  - Possible values of duration
  - A list of midinotes named *notes*
    - suggestion: the pentatonic scale C D E G A at different octaves
- At each step you update the values of x and y following the formula
- midinote is taken as an element of pars["notes"] given the value of y[n]



pars["y"]	-3					…		8
pars["notes"]	60	62	64	67	69	72	…	84

MUSIC COMPOSITION WITH PYTHON



Exercize 2: Let's use the Gingerbread function

```
# your_code.py
def map_(x_in, range_in, range_out):
        # my code, map x_in from range_in to range_out

def gingerbread(comp):
    def next_gingerbread(comp):
        # your code
    if comp.current ==ID_START:
        comp.pars["notes"] = # your code
        comp.dur, status.BPM, status.amp = # your code
        comp.pars["range_y"]=[-3, 8]
        comp.pars["y"], comp.pars["x"]=0.1, -0.1
        i=map_(comp.pars["y"], comp.pars["range_y"],
                                  [0, len(comp.pars["notes"])])
        comp.midinote=comp.pars["notes"][int(i)]
        comp.id=0
    elif comp.id==0:
        comp.amp=1
        # your code
```

MUSIC COMPOSITION WITH PYTHON

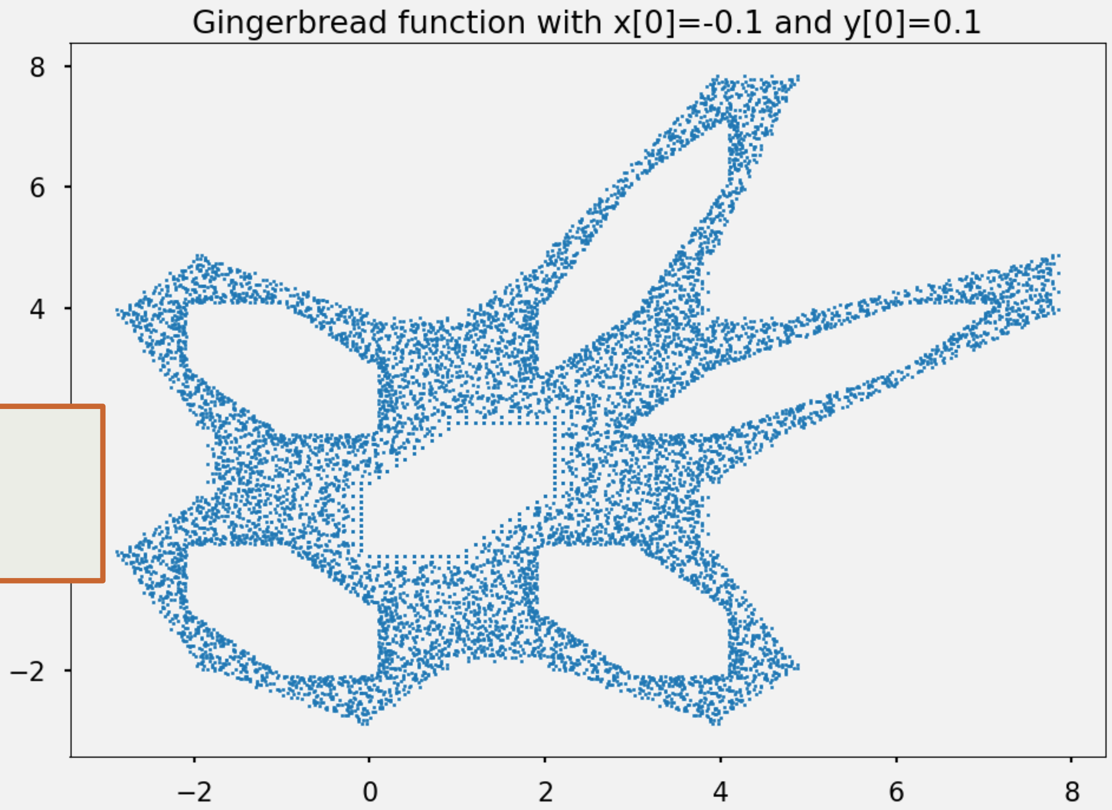
This is nice but it is too predictable

The rule is deterministic

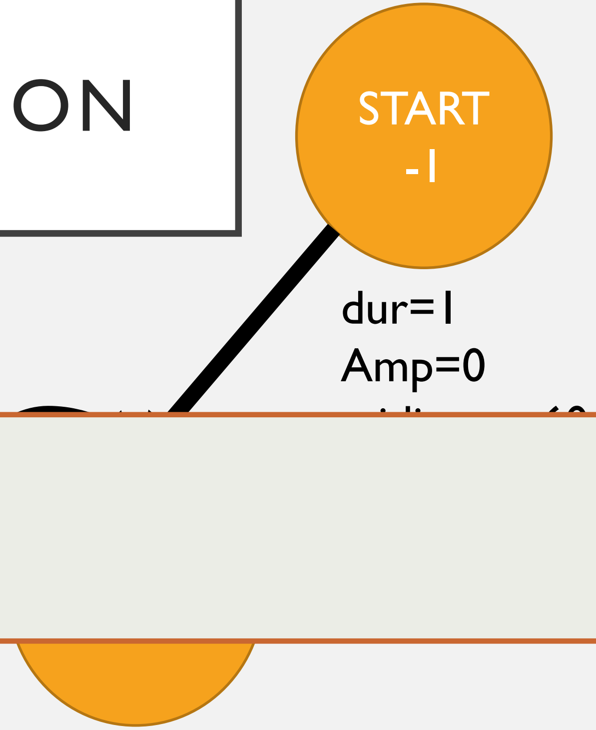
Let's add "creativity" by introducing randomicity in the gingerbread function

Or in the mapping between gingerbread and melody

```
import random

random_value=random.random() # between 0 and 1
```



MUSIC COMPOSITION WITH PYTHON



Exercize 2: Add some randomness in the Gingerbread function

```python
# your_code.py
def gingerbread_randomness(comp):
    # your code
```

PLAYING WITH PHYSICS

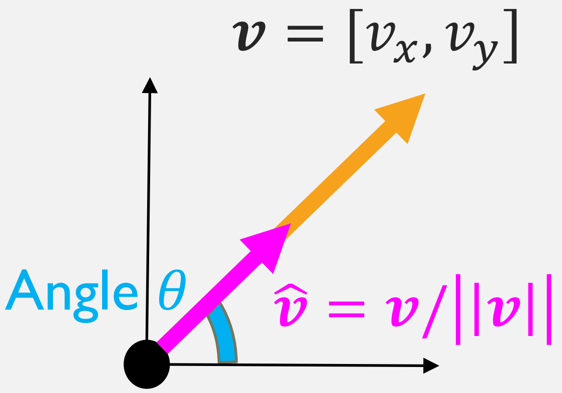
PLAYING WITH PHYSICS

- We have seen how we can use a reactive agent to create a composition by following strict rules, and how to add randomness to make compositions change
- In Processing, we will see a different kind of reactive agents that follow the physics law
- Processing displays a screen in two dimensions, hence we will start from vectors in two dimensions

THE BASIC OF VECTORS

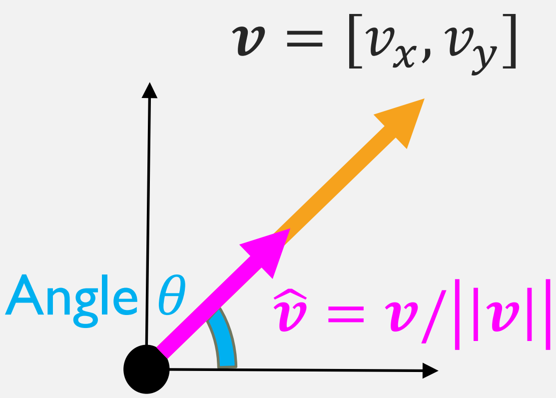
- We will see how to play with motion in 2D with Processing

- 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
- Agent has position , velocity and acceleration; mouse is converted into a Pvector
  - $\boldsymbol{p}(n) = [p_x, p_y]$; $\boldsymbol{v}(n) = [v_x, v_y]$; $\boldsymbol{a}(n) = [a_x, a_y]$;
- We model the following as:
  - Compute the vector that points from the object to the mouse locations
    - $\boldsymbol{m}(n) = \boldsymbol{mouse} = [mouse_x, mouse_y]$
    - $\boldsymbol{\Delta}_{\mathbf{m}}(\mathrm{n}) = \boldsymbol{m}(n) - \boldsymbol{p}(n)$
  - Normalize the vector and scale it to a given constant value (use c=CONSTANT_ACC=10)
    - $\widetilde{\boldsymbol{\Delta}}_m(n) = {}^{\boldsymbol{\Delta}_{\mathbf{m}}(n)}/_{||\boldsymbol{\Delta}_{\mathbf{m}}||(n)}\, c$
  - Assign the vector to object's acceleration; update velocity and position accordingly
    - $\boldsymbol{a}(n+1) = \widetilde{\widetilde{\boldsymbol{\Delta}}_m}(n); \boldsymbol{v}(n+1) = \boldsymbol{v}(n) + \boldsymbol{a}(n); \boldsymbol{p}(n) = \boldsymbol{p}(n) + \boldsymbol{v}(n)$
- Finally we 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()
}
```

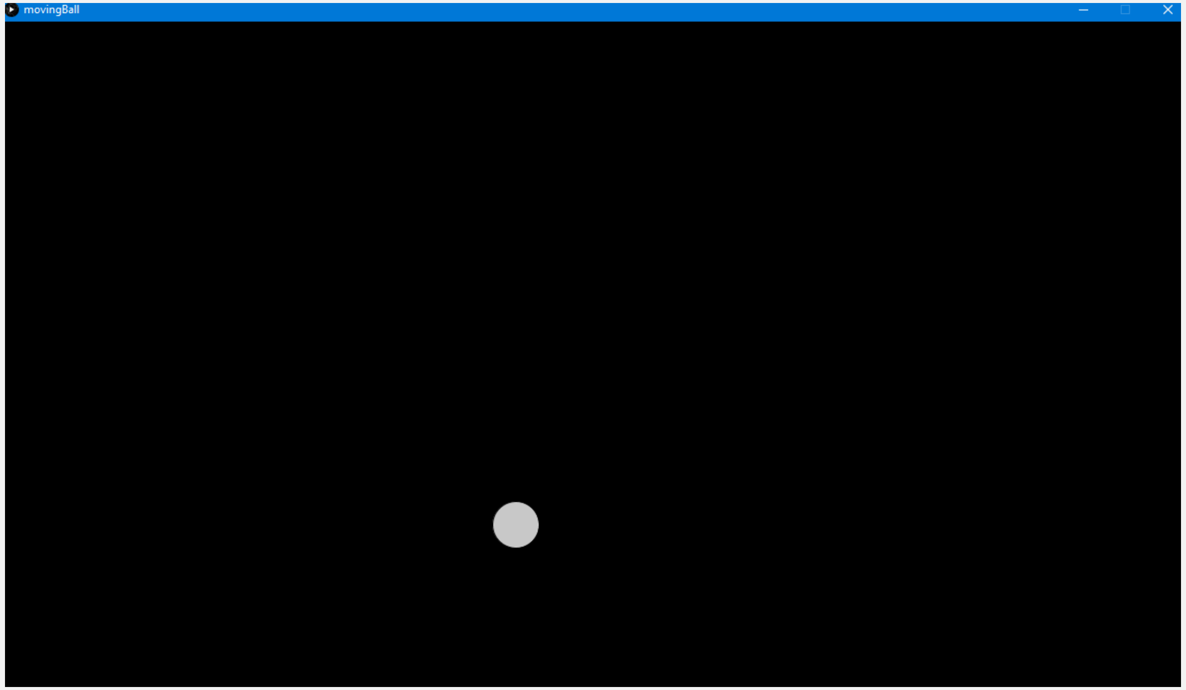
```
# AgentMover.pde
int RADIUS_CIRCLE=50; int LIMIT_VEL=50; int ACC=2;

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 delta = 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

What if we change the constant value?

Try it!



PLAYING WITH PHYSICS

- We have seen how we can use a reactive agent to create a composition by following strict rules, and how to add randomness to make compositions change
- we still have two available controls to change the *timbre* of our instrument
  - The cutoff and the vibrato
- We will control the timbre with reactive agents inspired by physics

- We will create a system controlled by physics law and then we will map its behavior to our instrument
- First, we need to install the library **oscP5** on processing

PLAYING WITH PHYSICS

- We h
  strict
- we st
  - The
- We w
- We w … to
  our i
- First,

```
# movingBall.pde
import oscP5.*; import netP5.*;
int PORT = 57120;
OscP5 oscP5;
NetAddress ip_port;
AgentMover mover;
void setup(){
  mover=new AgentMover();
  oscP5 = new OscP5(this,55000);
  ip_port = new NetAddress("127.0.0.1",PORT);
  size(1280, 720);
  background(0);
}

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); }
```

PLAYING WITH PHYSICS

- Let’s close the circle!

- We need to find a nice mapping between the agent and the cutoff or vibr.

PLAYING WITH PHYSICS

- Let's close the circle!

- We need to find a nice mapping between the agent and the cutoff or vibr.
- We know vibr should range around 0
- We know cutoff is designed to be between 0 and 1
- Let's map position of the agent with cutoff and vibr
  - `float cutoff=constrain((this.position.y/height),0,1);`
  - `float vibrato=constrain(this.position.x/width-0.5, -0.5, 0.5);`

constrain is a clipping

PLAYING WITH PHYSICS

```
# AgentMover.pde

class AgentMover{
  PVector pos, vel, acc;
  float vibrato=0;
  float cutoff=0.5;
  /* all the code*/
  void computeEffect(){
    /* your code here*/
    this.vibrato= /* your code*/
    this.cutoff= /* your code*/
  }
```

tween the agent and the cutoff or vibr.

d 0

- We know cutoff is designed to be between 0 and 1
- Let's map position of the agent with cutoff and vibr
  - `float cutoff=constrain((this.position.y/height),0,1);`
  - `float vibrato=constrain(this.position.x/width-0.5, -0.5, 0.5);`

PLAYING WITH PHYSICS

```
# AgentMover.pde

class AgentMover{
  PVector pos, vel, acc;
  float vibrato=0;
  float cutoff=0.5;
  /* all the code*/
  void computeEffect(){
    /* your code here*/
    this.vibrato= /* your code*/
    this.cutoff= /* your code*/
  }
```


```
# movingBall.pde
/* …*/

void draw(){
  background(0);
  mover.update();
  mover.computeEffect();
  sendEffect(mover.cutoff,
mover.vibrato);
  mover.draw();
}
```

tween t

d 0

- We know cutoff is designed to be between
- Let's map position of the agent with cutoff
  - float cutoff=constrain((this.posi
  - float vibrato=constrain(this.posi

PLAYING WITH PHYSICS

- Let's close the circle!

We also need to change the receiving function from super collider

PLAYING WITH PHYSICS

- Let's close the circle!

midinote, amp

We also need to change the receiving function from super collider

If the first argument is «note», we change the midinote and amp(Python)

If the first argument is «effect», we change the cutoff and vibrato (Processing)

```
# moogs.scd
NetAddr("127.0.0.1",57120);
(OSCdef('OSCreceiver',{
    arg msg;
    var note, amp, cutoff, vibr;

    msg.postln;

    if(msg[1].asString()=="note"){
        note=msg[2];
        amp=msg[3];
        ~instr.set(\midinote,note, \amp,amp);}{};

    if(msg[1].asString()=="effect"){
        cutoff=msg[2];
        vibr=msg[3];
        ~instr.set(\vibr,vibr, \cutoff,cutoff);}{};
    },
    "/note_effect",);)
```

NEWTON'S LAWS

We can keep playing by translating Newtons' laws into Processing

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);}
}
```

PHYSICS IN FLUIDS

- Let's rewrite the previous example adding fluid resistance, a.k.a. viscous force or drag force

$$F_d = -0.5\rho v^2 A\, C_d\ \hat{\boldsymbol{v}}$$

- ρ is the fluid density;
- v^2 is the squared magnitude of velocity;
- A is the frontal area of the object pushing through fluid;
- C_d is the coefficient of drag;

PHYSICS IN FLUIDS

- Let's rewrite the previous example adding fluid resistance, a.k.a. viscous force or drag force

$$F_d = -0.5\rho||v||^2 A\, C_d\ \hat{\boldsymbol{v}}$$

- ρ is the fluid density; start with 0.1
- $||v||^2$ is the squared magnitude of velocity;
- A is the frontal area of the object pushing throu
  - (in our case it's half a circle's perimeter)
- C_d is the coefficient of drag; start with 0.1

Let's first repeat the previous exercise adding dra

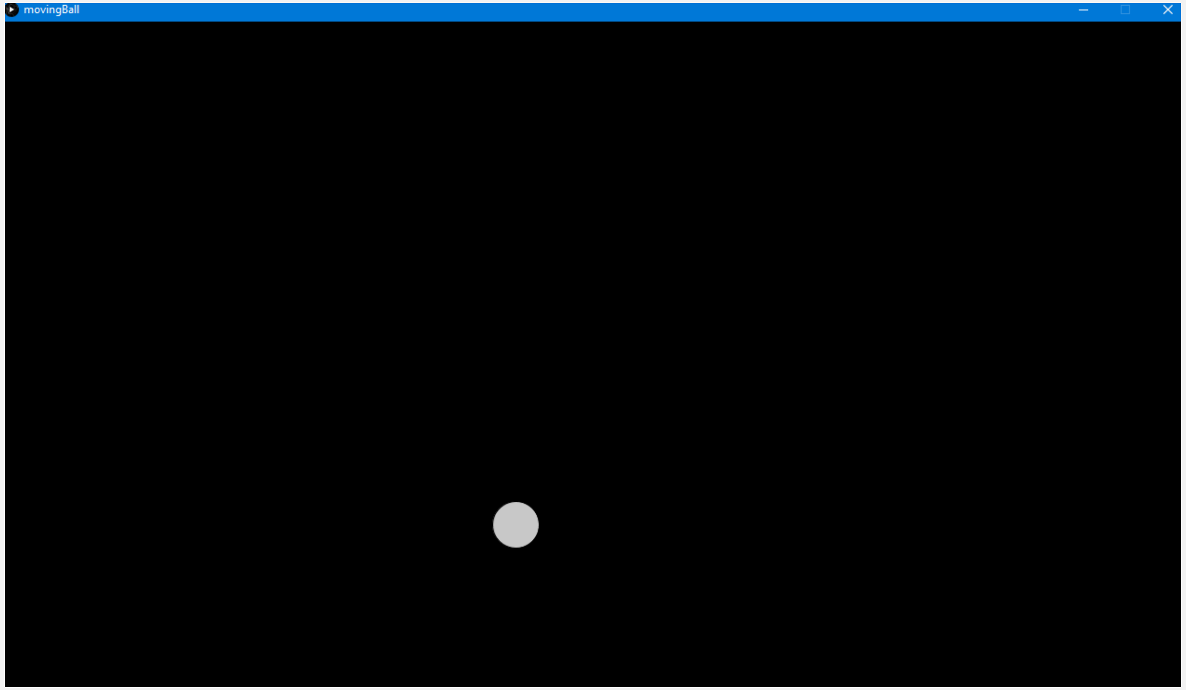
```
# moving_ball_fluid.pde
/*...*/
Pvector computeDragForce(AgentMover mover){
  /* your code here*/}
void draw(){
  background(0);
  dragForce = computeDragForce(mover);
  mover.applyForce(dragForce)
  mover.update();
  mover.computeEffect();
  sendEffect(mover.cutoff, mover.vibrato);
  mover.draw();
}
```

PHYSICS IN FLUIDS

You can play and test different fluid densities

Try to slowly change fluid densities at every frame

How would you use it?

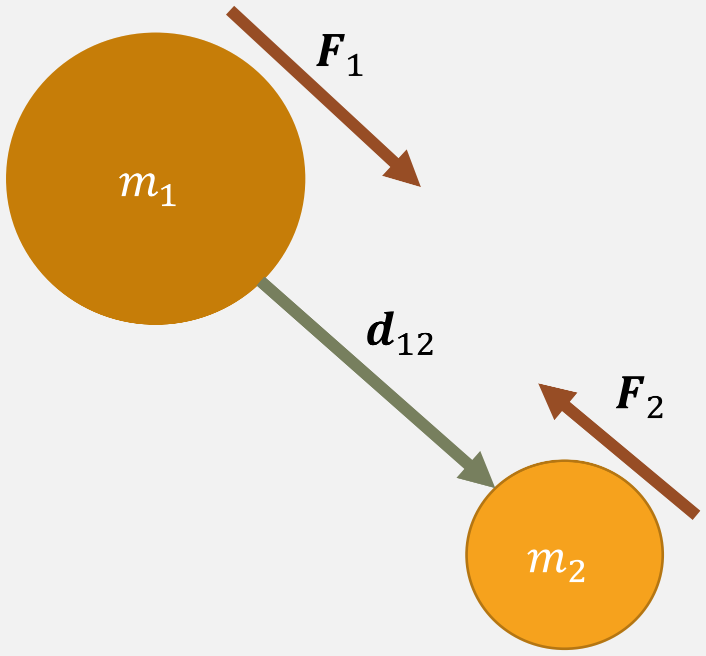


GRAVITY AND ATTRACTION

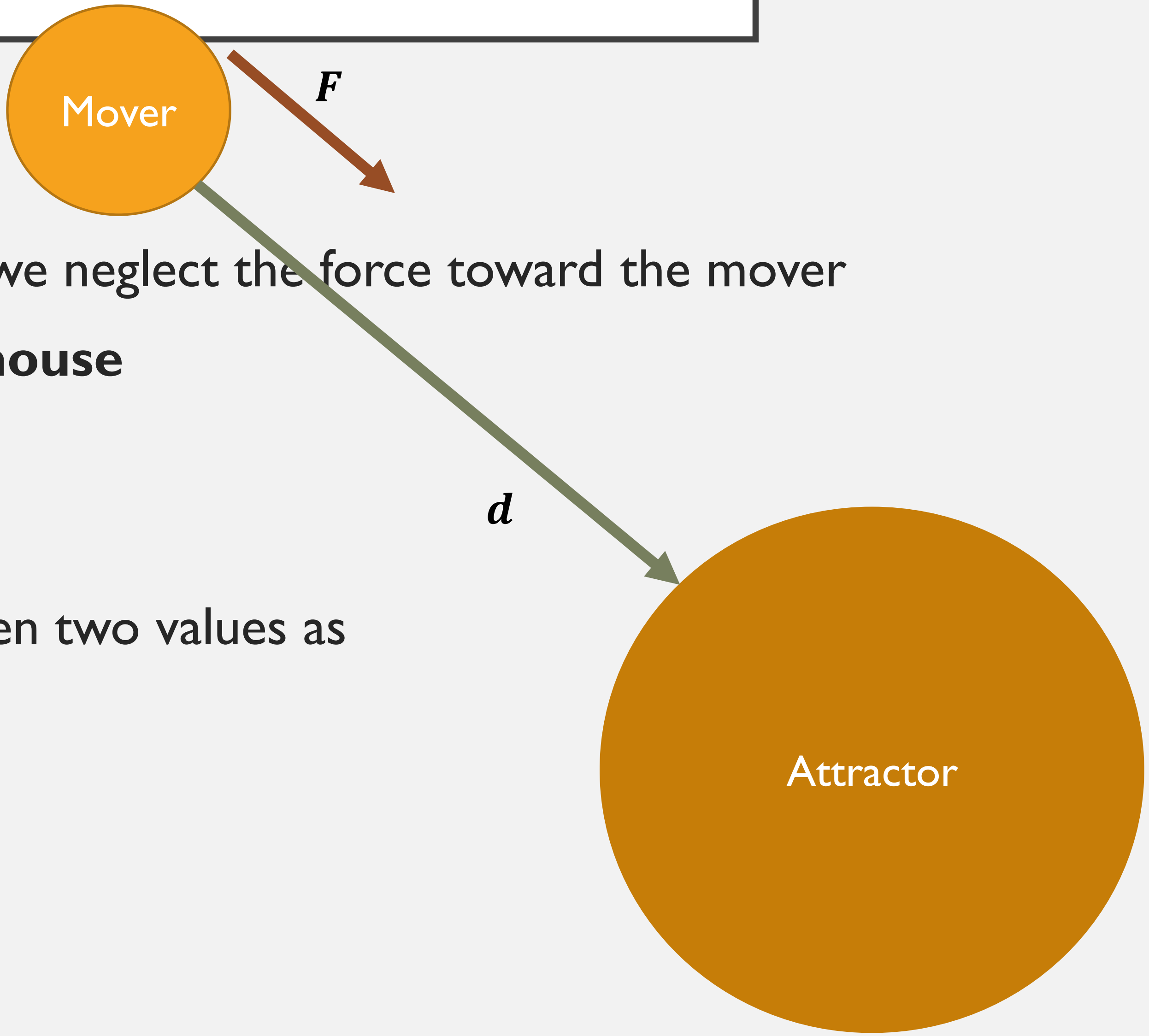
- Gravity occurs among any pair of objects following the formula

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

  - $G = 6.67428 \cdot\ 10^{-11}$ *is* the universal gravity constant. We will set it to 1
  - $\hat{\boldsymbol{d}}_{12}$ 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**
- Since $m_1 > m_2$, we have $\|\boldsymbol{a}_1\| = \left\|\frac{\boldsymbol{F}_1}{m_1}\right\| < \left\|\frac{\boldsymbol{F}_2}{m_2}\right\| = \|\boldsymbol{a}_2\|$
- Between two objects of greatly different masses, the acceleration of the bigger is irrelevant



GRAVITY AND ATTRACTION



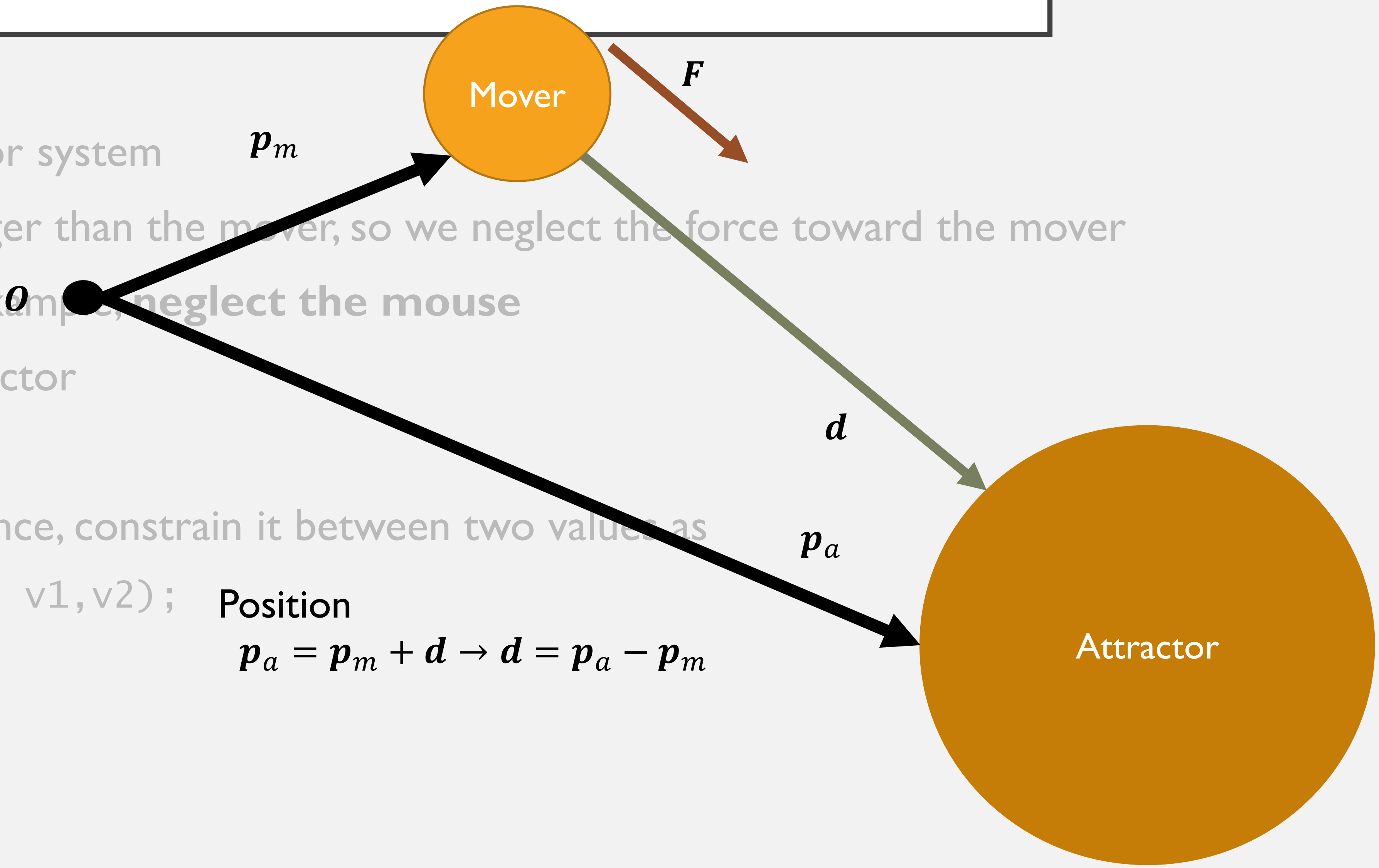
Let's define a mover-attractor system

- 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
- 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
- Place the mover
- when computing the distance, constrain it between two values as
  - `dist=constrain(dist, v1,v2);`



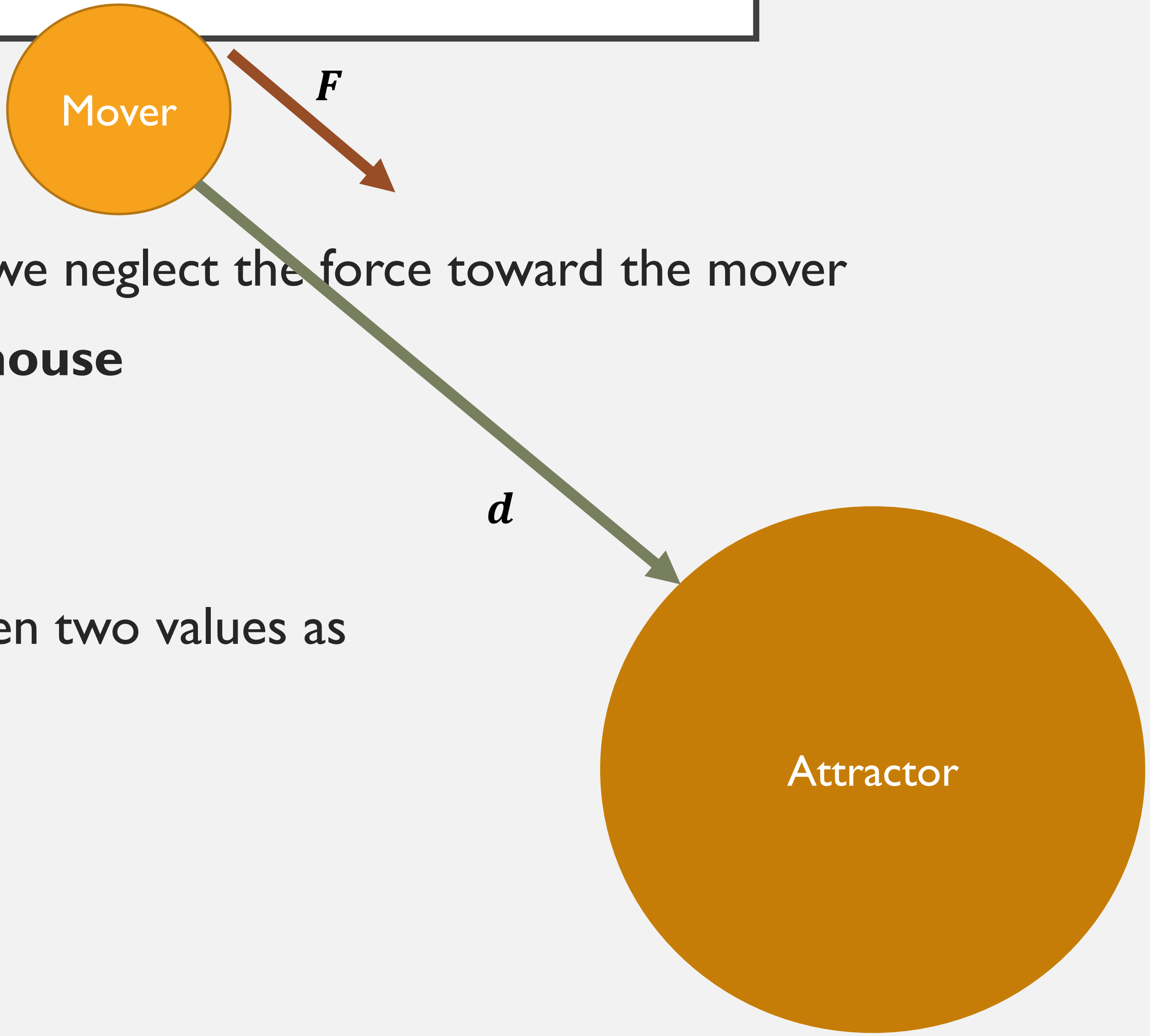
Position

$$\boldsymbol{p}_a = \boldsymbol{p}_m + \boldsymbol{d} \rightarrow \boldsymbol{d} = \boldsymbol{p}_a - \boldsymbol{p}_m$$

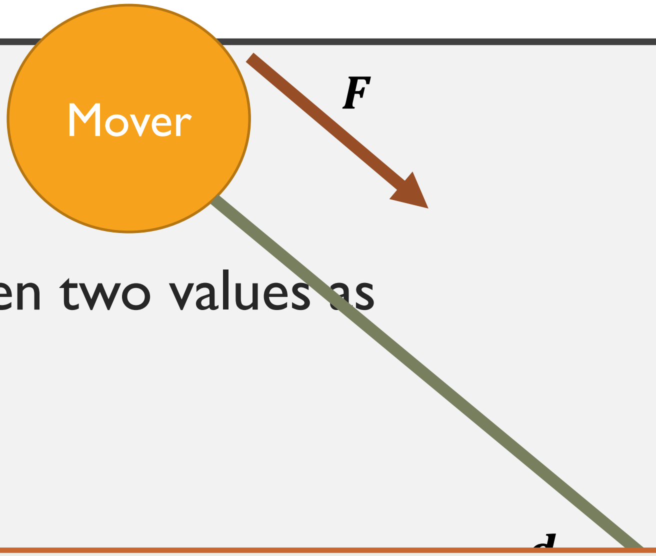
GRAVITY AND ATTRACTION

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
- 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

- when computing the distance, constrain it between two values as
  - `dist=constrain(dist, v1,v2);`
- Let's map the distance to the cutoff frequency as
  - `cutoff=map(dist, v1,v2,0,1);`
  - You need to change mover's method `computeEffect` so it takes also a float argument `dist` as input

```
# moving_ball_fluid.pde

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

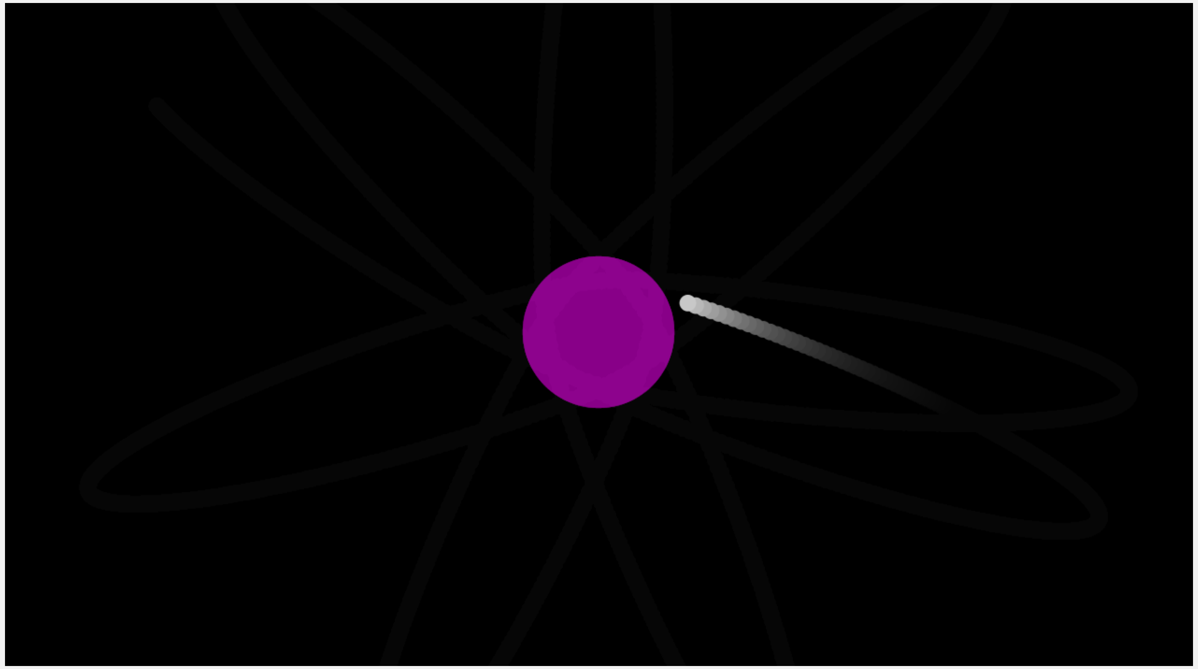
GRAVITY AND ATTRACTION

Play with transparency

What if the attractor follows the mouse?

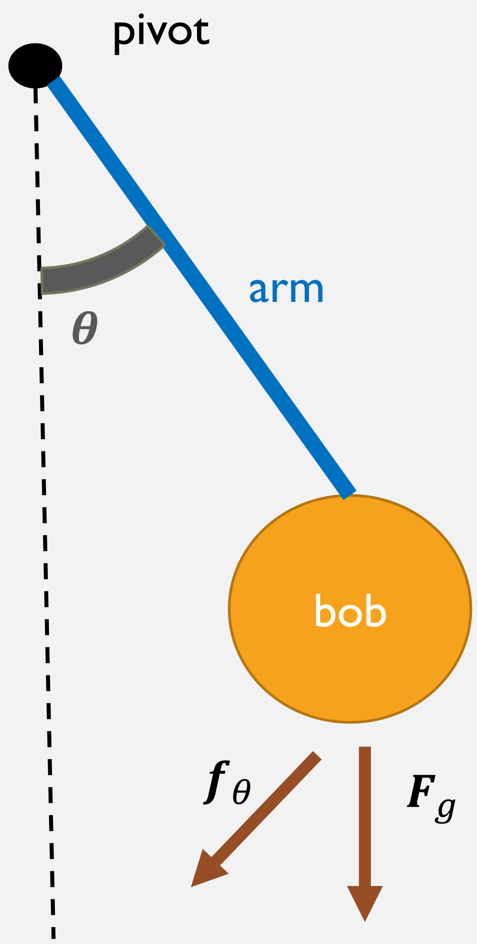
If you change the sign of the attraction, you get a repeller, from which your movers run away

What if you build an attractor in the middle and a repeller that follows your mouse?



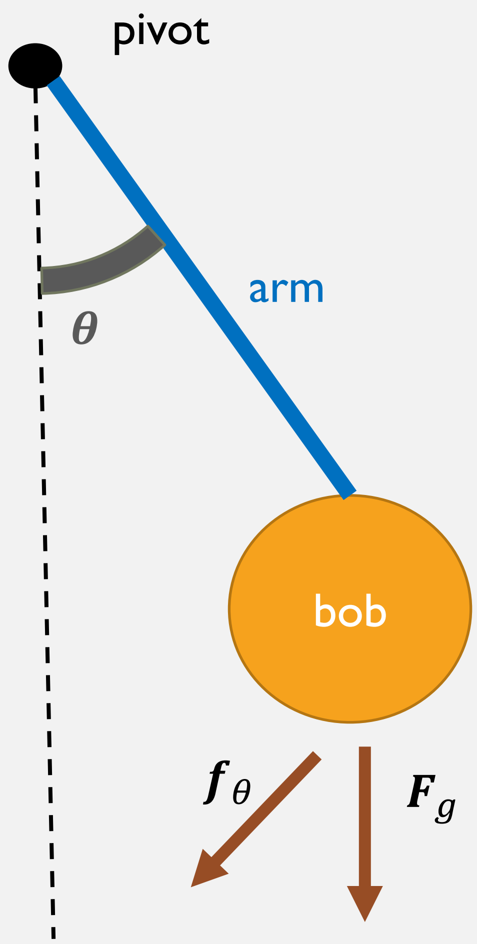
TRIGONOMETRY AND FORCES
THE PENDULUM

- 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 position of the mass, we will act on the angle of the arm**
- Hence we will talk of angle, angular velocity, angular acceleration
  - We are in the 1-dimensional world



TRIGONOMETRY AND FORCES THE PENDULUM

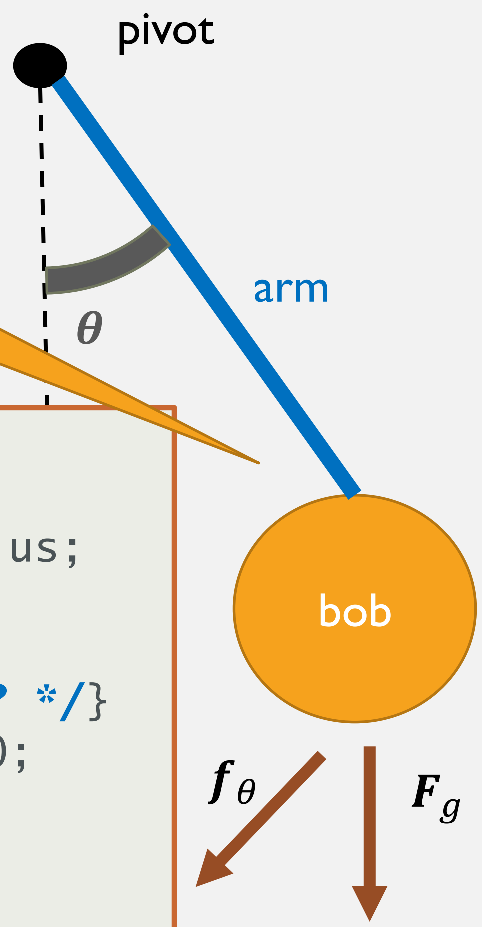
- We are in the 1-dimensional world: $a_\theta, v_\theta, \theta$ are scalars
- $f_\theta = \left(-Gm\frac{\sin(\theta)}{r}\right)$ with $G = 9.8$ and r length of the arm
- a_θ is updated by adding $\frac{f_\theta}{m}$ with m the mass of the "bob"



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/sin(theta)

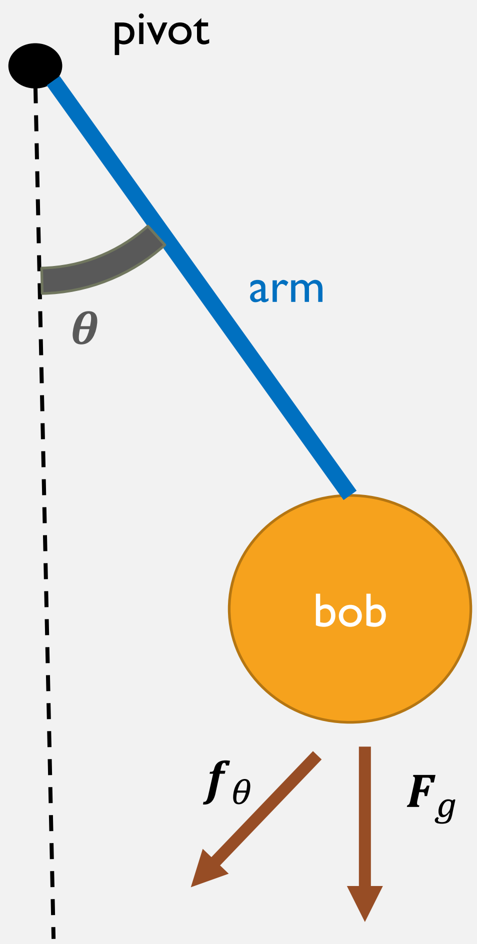



```
# AgentPendulum.pde
class AgentPendulum{
  PVector pivotPos, massPos; float angle, aVel, aAcc; float r, mass, radius;
  AgentPendulum(float x, float y, float r, float mass){
    this.pivotPos = new PVector(x, y);
    this.mass=mass; this.r=r; this.angle=random( -PI/2, -PI/4); /* other? */}
  void 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();
}
```

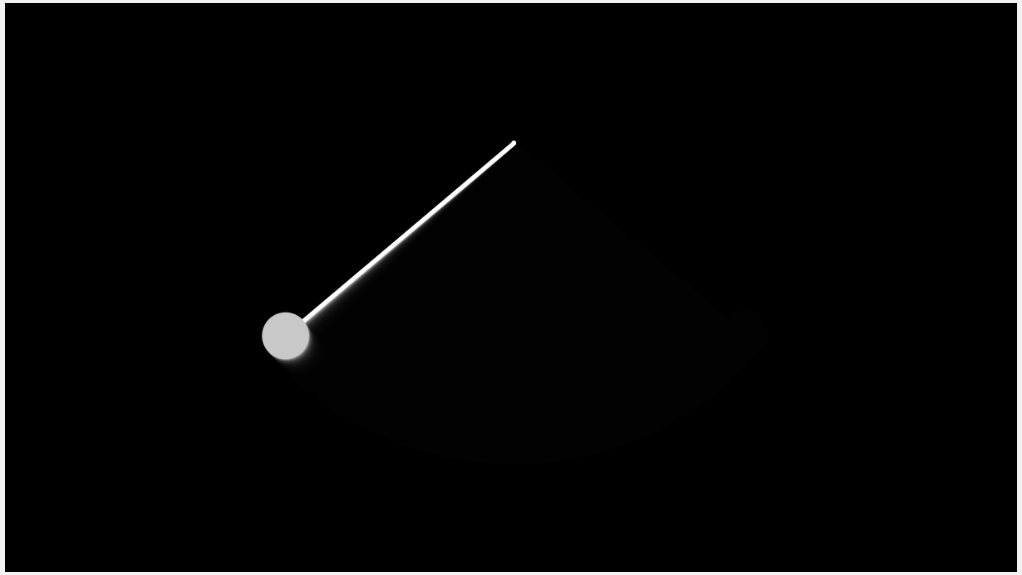


TRIGONOMETRY AND FORCES THE PENDULUM

The pendulum will go on forever because in our model it does not exist a dumping

Try multiplying velocity times 99% at every step

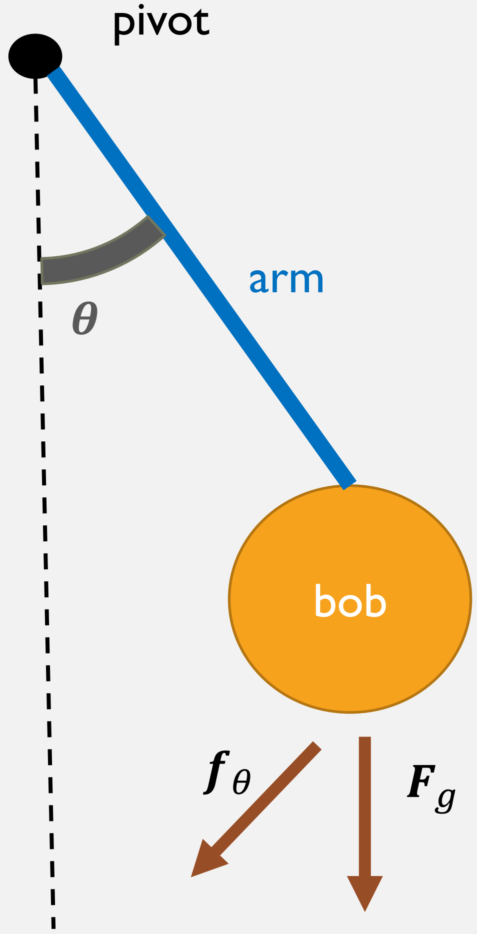
If it’s too fast, try to play with its parameters and slow it down



TRIGONOMETRY AND FORCES
THE PENDULUM

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

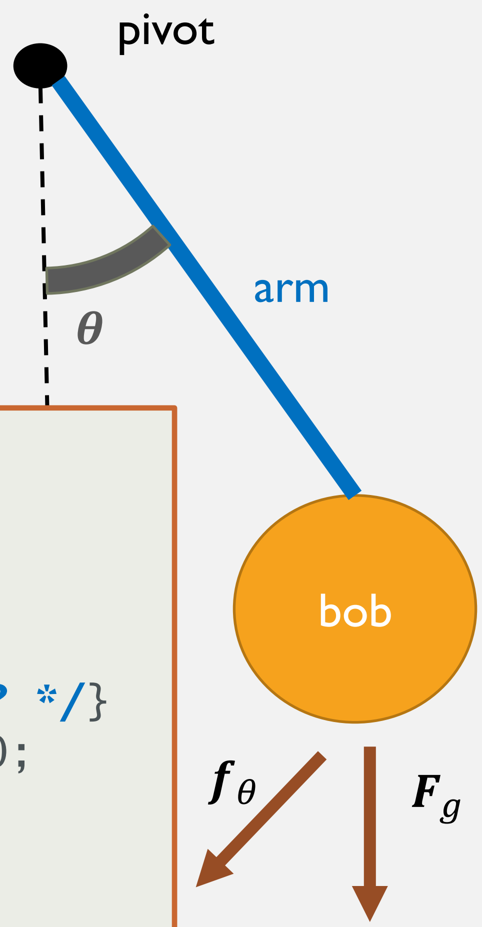
- The pendulum oscillates around the theta = 0
- Hence, we can map theta into the vibrato!
- Cutoff with magnitude of angular velocity



TRIGONOMETRY AND FORCES
THE PENDULUM

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

- The pendulum oscillates around the theta = 0
- Hence, we can map theta into the vibrato!
- Cutoff with magnitude of angular velocity



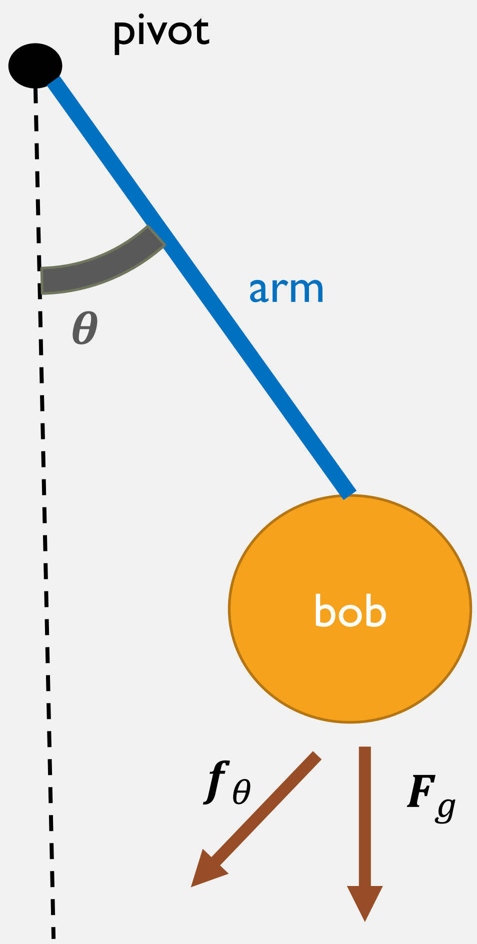

```
# AgentPendulum.pde
class AgentPendulum{ /* ... */
float vibrato, cutoff;
  AgentPendulum(float x, float y, float r, float mass){
    this.pivotPos = new PVector(x, y);
    this.mass=mass; this.r=r; this.angle=random( -PI/2, -PI/4); /* other? */}
  void update(){this.aVel+=this.aAcc; this.angle+=this.aVel;  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

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

- The pendulum oscillates around the theta = 0
- Hence, we can map theta into the vibrato!
- Cutoff with magnitude of angular velocity

```
# pendulum.pde
void draw(){rectMode(/* ... */
  float force= computeForce(pendulum);
  pendulum.applyForce(-1*G*sin(pendulum.angle)/pendulum.r);
  pendulum.update();
  pendulum.computeEffect(dist);
  sendEffect(pendulum.cutoff, pendulum.vibrato);
  pendulum.draw();
}
```



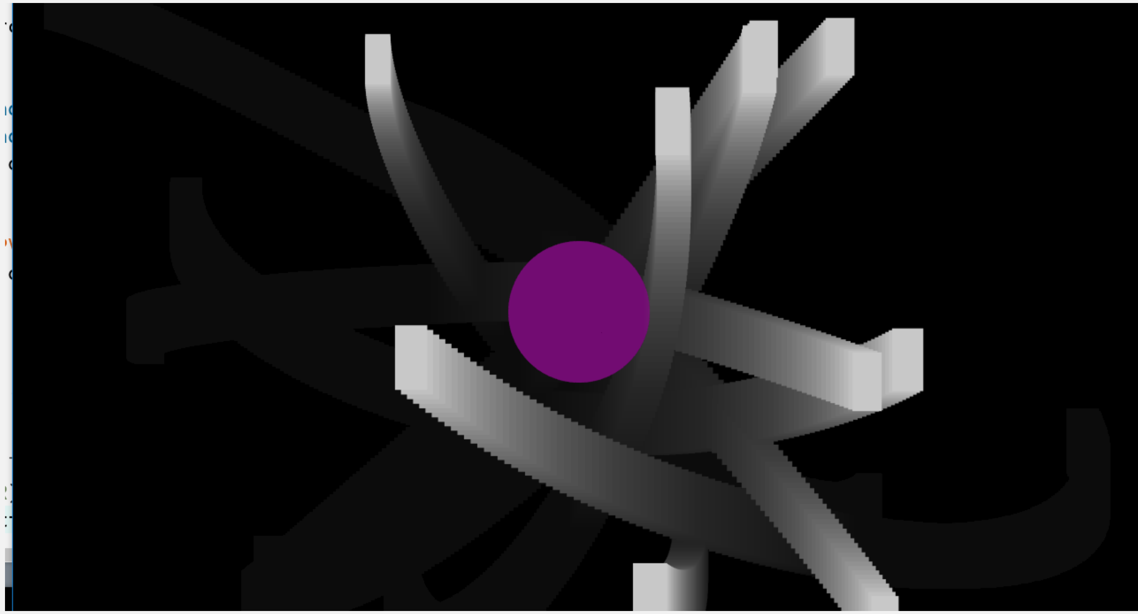
ONE LAST THING

PLAYING WITH ANGLES

Homework! Back to the exercise about attractor

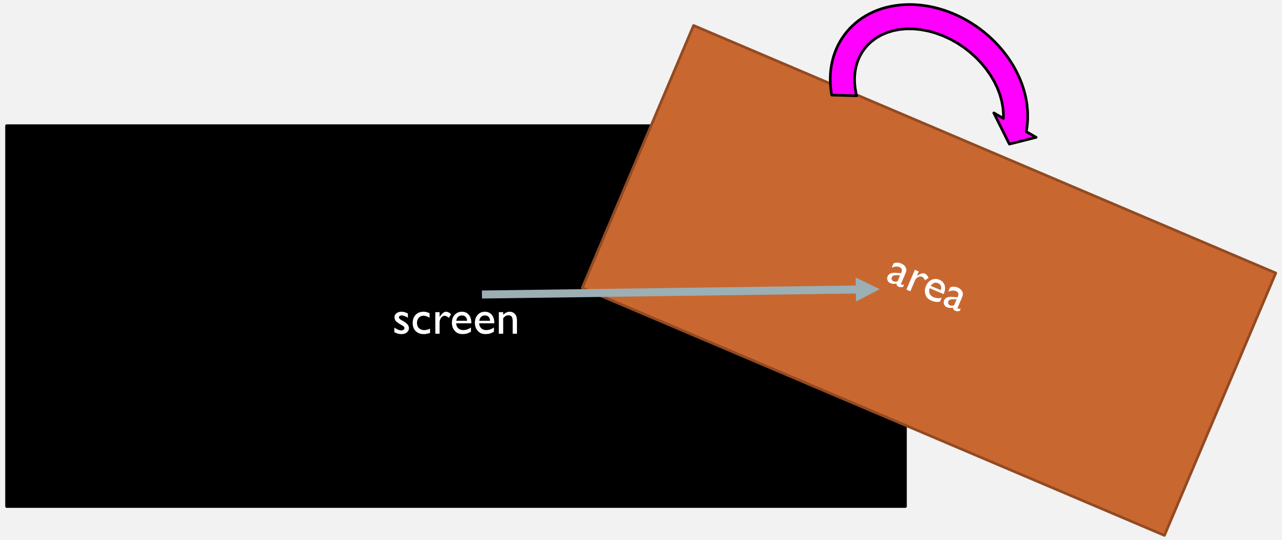
- We like circles because when they don't have a direction.
- Let's use rectangle instead of circles for one moment in the attractor example
  - rectMode(CENTER); rect(this.position.x, this.position.y, 0.5*this.radius_circle, this.radius_circle);
- Not that cool anymore ☹

- Let's rotate the object in order to point the movement direction, i.e., the direction of velocity
  - AgentMover.velocity.heading()
  - https://processing.org/reference/PVector_heading_.html
- How to use the angles for rotation?



PLAYING WITH ANGLES

- How to use the angles for rotation?
- We rotate (and translate) the whole screen!



- pushMatrix() and popMatrix() saves and restore the current screen

```
# AgentMover.pde
class AgentMover{
  /* ...*/
  void action(){
    this.planning();
    fill(200);
    rectMode(CENTER);
    pushMatrix();
    rectMode(CENTER);
    translate(position.x,position.y);
    rotate(/* angle? */);
    rect(0,0, this.radius_circle,
              0.5*this.radius_circle);
    popMatrix();
  }
}
```

PLAYING WITH ANGLES

Can you think of a musical property that you could map with the angle?