Conception d'Applications Interactives

développement d'IHM en python/TkInter

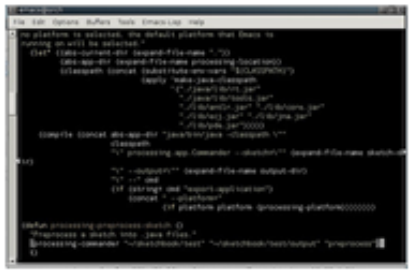
Alexis NEDELEC

Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

Interfaces Homme-Machine

Interagir avec un ordinateur

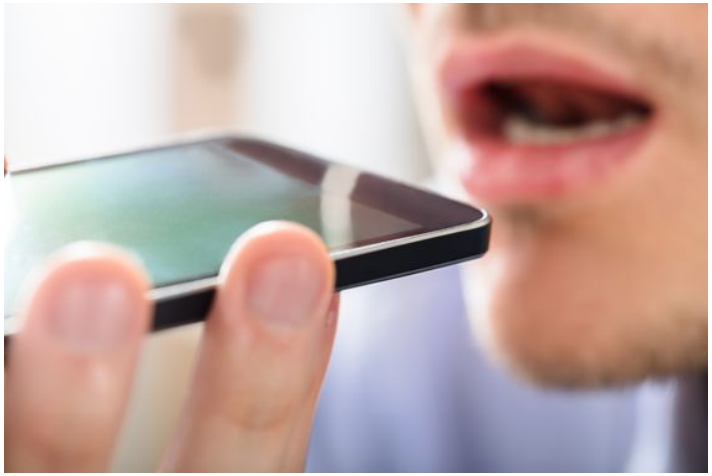
- CLI (Command Line Interface) : interaction clavier
- GUI (Graphical User Interface) : interaction souris-clavier
- NUI (Natural User Interface) : interaction tactile, capteurs

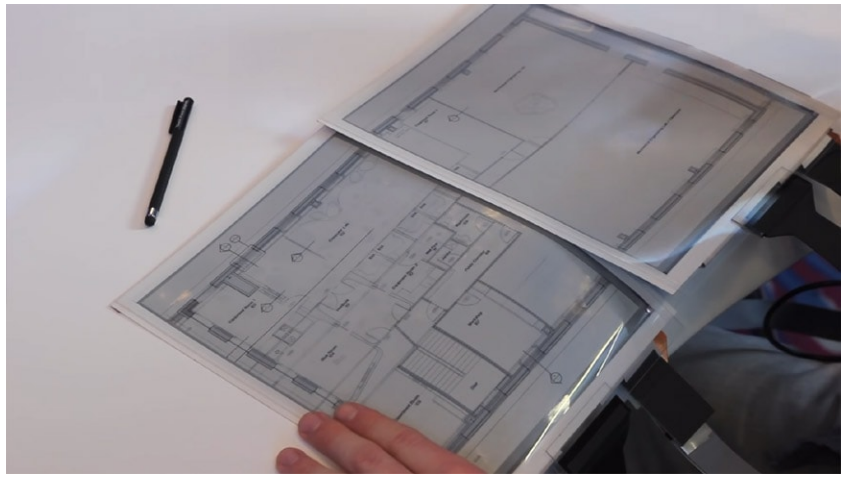


Interfaces Homme-Machine

Interagir avec un ordinateur

- VUI (Voice User Interface) : interaction vocale
- OUI (Organic User Interface) : interaction biométrique
- ...





Interfaces Homme-Machine

Objectifs du cours

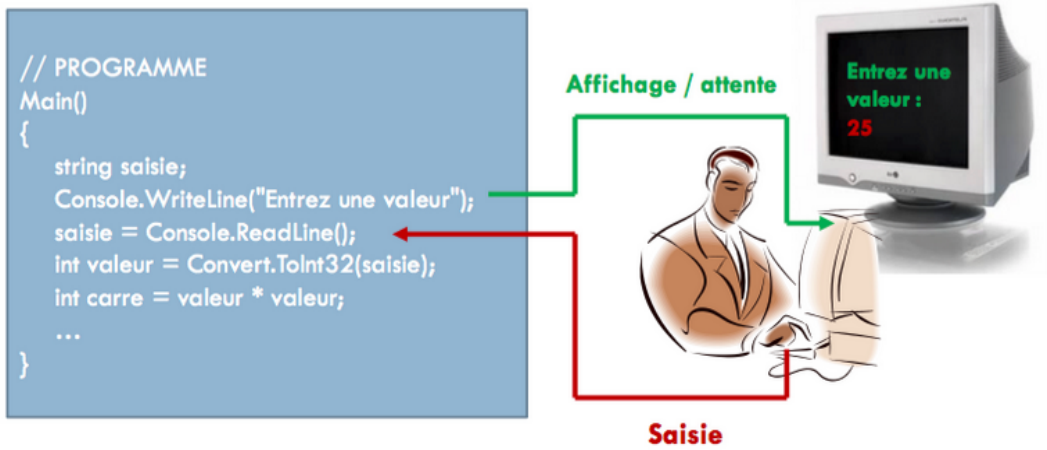
Savoir développer des IHM avec une bibliothèque de composants

1. paradigme de programmation événementielle (Event Driven)
2. interaction WIMP (Window Icon Menu Pointer)
3. bibliothèque de composants graphiques (Window Gadgets)
4. développement d'applications GUI (Graphical User Interface)
5. patrons de conception (Observer,MVC)

Programmation événementielle

Programmation classique : trois étapes séquentielles

1. initialisation
   - modules externes, ouverture fichiers, connexion serveurs ...
2. traitements de données
   - affichage, modification, appel de fonctions ...
3. terminaison : sortir “proprement” de l’application



Programmation événementielle

Programmation d'IHM : l'humain dans la boucle ... d'événements

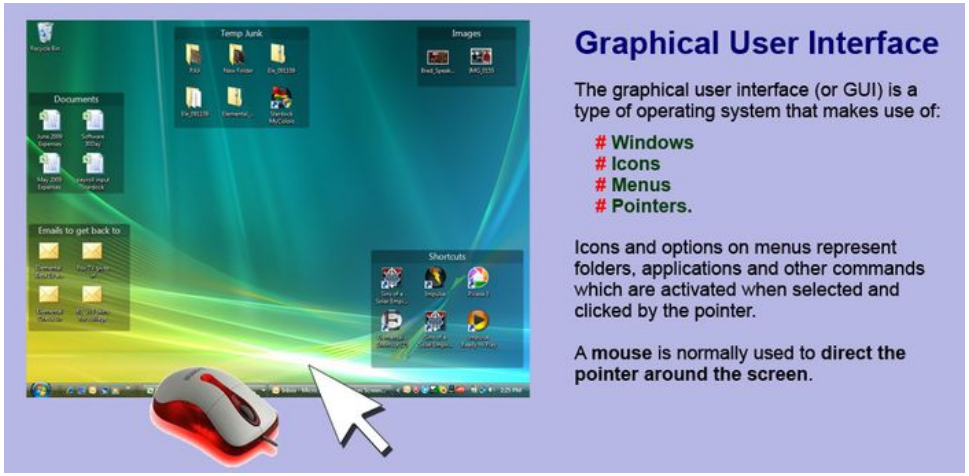
1. initialisation
   - modules externes, ouverture fichiers, connexion serveurs ...
   - création de composants graphiques
2. traitements de données par des fonctions réflexes (actions)
   - affichage de composants graphiques
   - liaison composant-événement-action
   - attente d'action utilisateur, dans une boucle d'événements
3. terminaison : sortir "proprement" de l'application

```
// PROGRAMME
Main()
{
  ...
  while(true)  // tantque Mamie s'active
  {
     // récupérer son action (faire une maille ...)
     e = getNextEvent();
     // traiter son action (agrandir le tricot ...)
     processEvent();
  }
  ...
}
```

Programmation événementielle

API pour développer des IHM

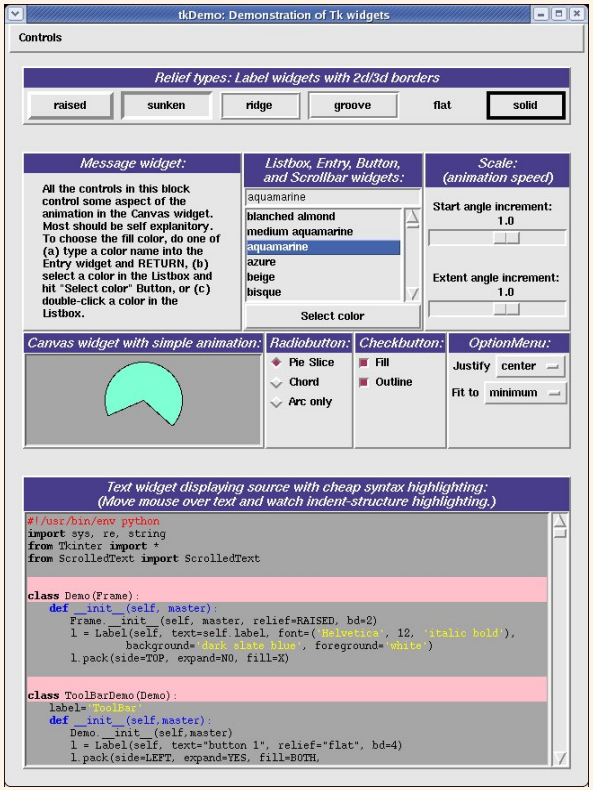
- Java : AWT,SWT,Swing,JavaFX,...,JGoodies, QtJambi ...
- C,C++ : Xlib, GTk, Qt, MFC, ...
- Python : TkInter, wxWidgets, PyQt, Pyside, Kivy,libavg...
- JavaScript : Angular, React, Vue.js, JQWidgets ...
- ...



https://www.ictlounge.com/html/operating_systems.htm

Python/TkInter

TkInter : Tk (de Tcl/Tk) pour python



Documentation python :

`https://docs.python.org/fr/3/library/tk.html`

Python/TkInter

Hello World : création d'IHM

```
from Tkinter import Tk,Label,Button
mw=Tk()
label_hello=Label(mw,
                  text="Hello World !",fg="blue")
button_quit=Button(mw,
                   text="Goodbye World", fg="red",
                   command=mw.destroy)
label_hello.pack()
button_quit.pack()
mw.mainloop()
```

Hello World

Création de composants graphiques

- `mw=Tk()`
- `label_hello=Label(mw, ...)`
- `button_quit=Button(mw, ...)`

Interaction sur un composant

- `button_quit=Button( ..., command=mw.destroy)`

Positionnement des composants

- `label_hello.pack(), button_quit.pack()`

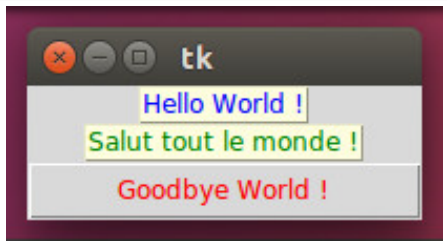
Entrée dans la boucle d'événements

- `mw.mainloop()`

Hello World

Personnalisation d'IHM : fichier de configuration

```
from Tkinter import Tk,Label,Button
mw=Tk()
mw.option_readfile("hello.opt")
label_hello=Label(root,text="Hello World !")
label_bonjour=Label(root,name="labelBonjour")
button_quit=Button(root,text="Goodbye World !")
label_hello.pack()
label_bonjour.pack()
button_quit.pack()
mw.mainloop()
```



Hello World

Configuration d'options de composants graphiques

```
*Button.foreground: red
*Button.width:20
*Label.foreground: blue
*labelBonjour.text: Salut tout le monde !
*labelBonjour.foreground: green
*Label.background: light yellow
*Label.relief: raised
```

Composants graphiques

Widgets : **W**indow ga**dgets**

Fonctionnalités des widgets, composants d'IHM

- affichage d'informations (label, message...)
- composants d'interaction (button, scale ...)
- zone d'affichage, saisie de dessin, texte (canvas, entry ...)
- conteneur de composants (frame)
- fenêtres secondaires de l'application (toplevel)

Composants graphiques

TkInter : fenêtres, conteneurs

- `Toplevel` : fenêtre secondaire de l'application
- `Canvas` : afficher, placer des "éléments" graphiques
- `Frame` : surface rectangulaire pour contenir des widgets
- `Scrollbar` : barre de défilement à associer à un widget

TkInter : gestion de textes

- `Label` : afficher un texte, une image
- `Message` : variante de label pour des textes plus importants
- `Text` : afficher du texte, des images
- `Entry` : champ de saisie de texte

Composants graphiques

Tkinter : gestion de listes

- `Listbox` : liste d'items sélectionnables
- `Menu` : barres de menus, menus déroulants, surgissants

Tkinter : composants d'interactions

- `Menubutton` : item de sélection d'action dans un menu
- `Button` : associer une interaction utilisateur
- `Checkbutton` : visualiser l'état de sélection
- `Radiobutton` : visualiser une sélection exclusive
- `Scale` : visualiser les valeurs de variables

Fabrice Sincère, cours sur python, notamment TkInter

Gestion d'événements

Structuration d'un programme

```
def callback(event) :
  """ definition des comportements (actions) """
   pass
def gui(parent) :
  """
  Creation des composants (arbre  de composants)
  Positionnement des composants  (layout manager)
  Liaison  Composant-Evenement-Action
  """
  pass
if __name__ =="__main__" :
    mw=Tk()
    gui(mw)
    mw.mainloop()
```

Gestion d'événements

Interaction par défaut : composant-<Button-1>-action

- option `command` : "click gauche", exécute la fonction associée

```
button_quit=Button(mw,...,command=mw.destroy)
```

Paramétrer l'interaction : composant-événement-action

- implémenter une `fonction "réflexe"` (action)

```
def callback(event) :
    mw.destroy()
```

- liaison (bind) "Composant-Evenement-Action" :

```
button_quit.bind("<Button-1>",callback)
```

Gestion d'événements

Types d'événements

représentation générale d'un événement :

- `<Modifier-EventType-ButtonNumberOrKeyName>`

Exemples

- `<Control-KeyPress-A>` (`<Control-Shift-KeyPress-a>`)
- `<KeyPress>`, `<KeyRelease>`
- `<Button-1>`, `<Motion>`, `<ButtonRelease>`

Principaux types

- `Expose` : exposition de fenêtre, composants
- `Enter`, `Leave` : pointeur de souris entre, sort du composant
- `Configure` : l'utilisateur modifie la fenêtre
- ...

Gestion d'événements

Fonctions réflexes associées

```
def callback(event) :
  # action needs information from :
  #  - user: get data from devices (mouse,keyboard ...)
  #  - widget: get or set widget options
  #  - the application: to manage shared data
```

`callback` : récupération d'informations

- liées à l'utilisateur (argument `event`)
- liées au composant graphique :
  - `event.widget` : on récupère le widget lié à l'événement
  - `configure()` : on peut fixer des valeurs aux options de widget
  - `cget()` : on peut récupérer une valeur d'option de widget
- liées à l'application (paramétrer la fonction réflexe)

Gestion d'événements

callback : informations liées aux périphériques

```
def callback(event) :
  # user : get data from mouse coordinates
  print("pointer coordinates on screen ",
        event.x_root,event.y_root)
  # widget: set widget options
  event.widget.configure(text="x="+str(event.x) \
                         +"y="+str(event.y))
```

callback : informations liées à l'application

```
def mouse_location(event,label):
 #  application : to manage shared data (label)
 label.configure(text= "Position : X=" + str(event.x)\
                       + ", Y="  + str(event.y))
```

Gestion d'événements

création de l'IHM : arbre de composants

```
def gui(parent) :
  canvas=tk.Canvas(parent,
                   width=200,height=150,
                   bg="light yellow")
  data=tk.Label(parent,text="Mouse Location")
  hello=tk.Label(parent,text="Hello World !",
                 fg="blue")
  button_quit=tk.Button(parent,text="Goodbye World",
                        fg="red")
  hello.pack()
  canvas.pack()
  data.pack()
  button_quit.pack()
```

Gestion d'événements

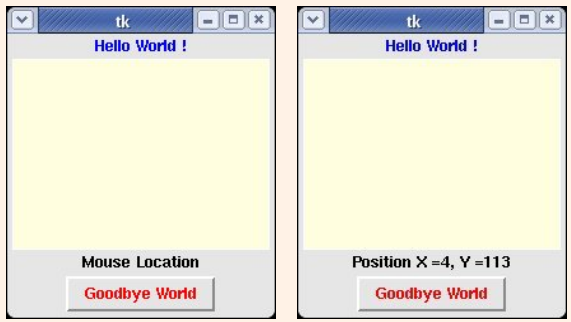
création de l'IHM : interaction

```
button_quit.bind("<Button-1>",app_exit)
canvas.bind("<Motion>",
            lambda event,label=data : \
              mouse_location(event,label))
...
```

Transmission des données de l'application (`data`) :

- fonctions anonymes (`lambda`)

Utilisation de l'IHM

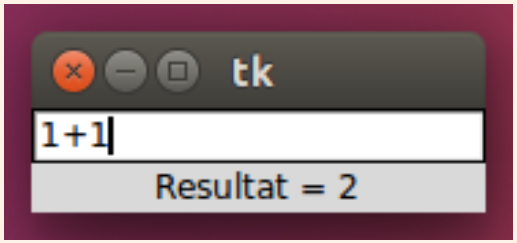


Gestion d'événements

Traitement des données utilisateur : `eval()`

```
from math import *
def evaluer(event,label,entry):
    label.configure(text= "Resultat = " \
                          + str(eval(entry.get())))
```

Utilisation de l'IHM



Gestion d'événements

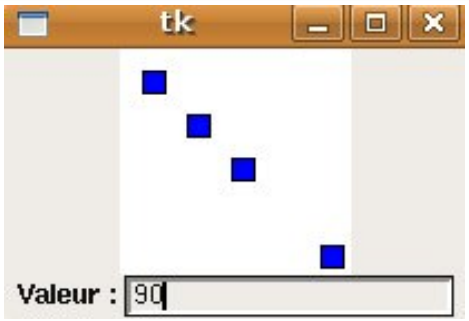
Création de l'IHM

```
def gui(parent) :
    entry=tk.Entry(parent)
    label=tk.Label(parent)
    entry.pack()
    label.pack()
    entry.bind("<Return>",lambda event,label=label,\
                                        entry=entry : \
                            evaluer(event,label,entry))
```

Gestion d'événements

Communication entre composants : event_generate()

```
def set_value(event):
  event.widget.event_generate("<Control-Z>")
def display(event,canvas,entry):
  x=int(entry.get())
  canvas.create_rectangle(x,x,x+10,x+10,fill="blue")
```




```
entry.bind("<Return>", set_value)
parent.bind("<Control-Z>",lambda ... : display(...))
```

Gestion d'événements

Communication entre composants : création de l'IHM

```
def gui(parent) :
  canvas=tk.Canvas(parent,...,bg="white",bd=1)
  label=tk.Label(parent,text = "Valeur :")
  entry=tk.Entry(parent)
  canvas.pack()
  label.pack(side="left")
  entry.pack(side="left")
  entry.bind("<Return>",set_value)
  parent.bind("<Control-Z>",lambda event,
                                  canvas=canvas,
                                  entry=entry : \
                              display(event,canvas,entry))
```

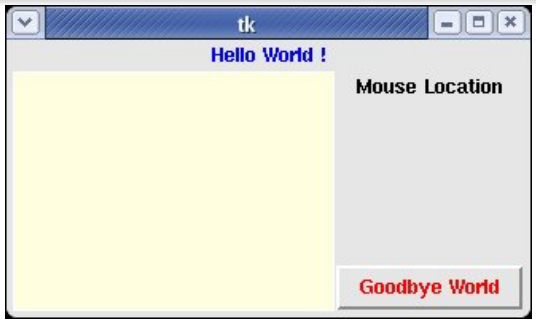
Positionnement de composants

TkInter : Layout manager

- `pack()` : agencer les widgets les uns par rapport aux autres
- `grid()` : agencer sous forme de frille (ligne/colonne)
- `place()` : positionner les composants géométriquement

`pack()` : "coller" les widgets par leur côté

```
hello.pack()
canvas.pack(side="left")
label.pack(side="top")
quit.pack(side="bottom")
```



Positionnement de composants

Regroupement de composants : Frame

```
def gui(parent) :
  # IHM : creation des composants
  frame=tk.Frame(parent,bg="yellow")
  canvas=tk.Canvas(frame,width=200,height=150,
                   bg="light yellow")
  label=tk.Label(frame,text="Mouse Location")
  hello=tk.Label(frame,text="Hello World !",fg="blue")
  quit=tk.Button(frame,text="Goodbye World",fg="red",
                 command=mw.destroy)
```

Positionnement de composants

Regroupement de composants : Frame

```
# IHM : Gestionnaires de positionnement
frame.pack(fill="both",expand=1)
hello.pack()
canvas.pack(fill="both",expand=1)
label.pack()
quit.pack()
```

Positionnement de composants

grid() :agencement ligne/colonne

```
# IHM : creation des composants
nom_label=tk.Label(parent,text="Nom :")
prenom_label=tk.Label(parent,text="Prenom :")
nom_entry=tk.Entry(parent)
prenom_entry=tk.Entry(parent)
# IHM : Gestionnaires de positionnement (layout manager)
nom_label.grid(row=0)
prenom_label.grid(row=1)
nom_entry.grid(row=0,column=1)
prenom_entry.grid(row=1,column=1)
```

Positionnement de composants

place() : positionnement géométrique

```
# IHM : creation des composants
msg=tk.Message(parent,text="Place : \n \
          options de positionnement de widgets",
          justify="center",bg="yellow",relief="ridge")
okButton=tk.Button(mw,text="OK")
# IHM : Gestionnaires de positionnement (layout manager)
msg.place(relx=0.5,rely=0.5,
          relwidth=0.75,relheight=0.50,anchor="center")
okButton.place(relx=0.5,rely=1.05,in_=msg,anchor="n")
```

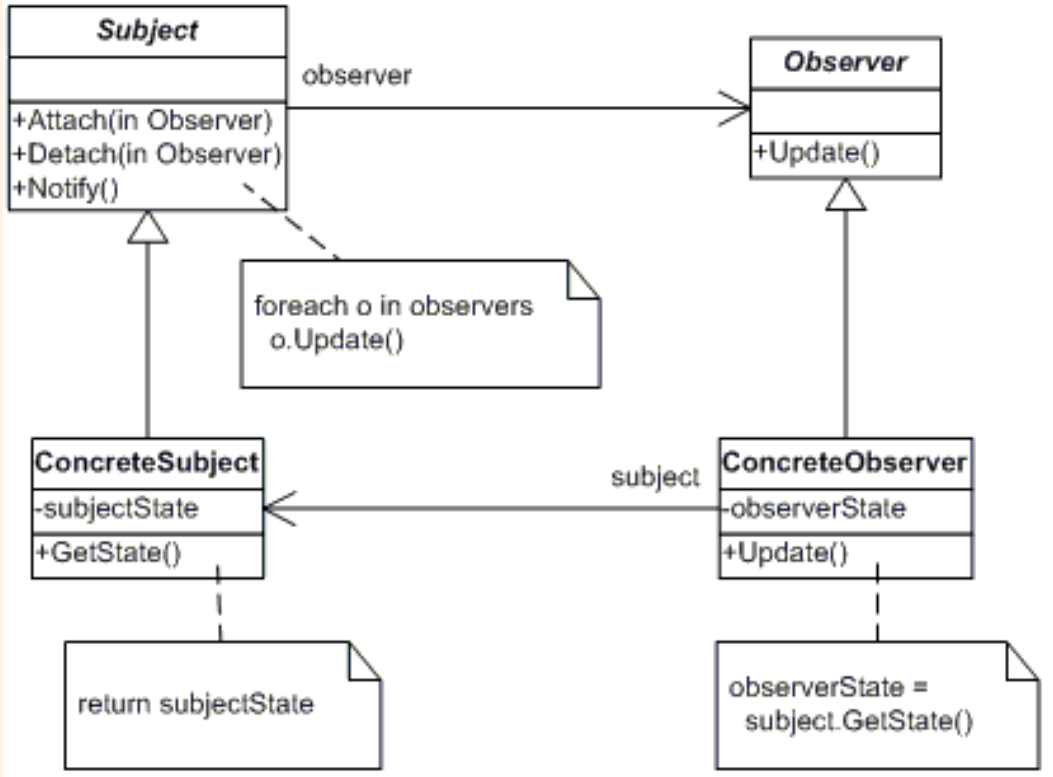
Patrons de conception

Programmer des IHM "proprement"

- Patrons de conception (Design Pattern)
- Modèle `Observer`
  - observateurs (`Observer`)
  - d'observable (`Subject`)
- Modèle `Observer` avec IHM
- Modèle MVC pour IHM
  - M : le modèle (les données)
  - V : l'observation du modèle
  - C : la modification du modèle

Modèle Observer

Observateur-Sujet observé



Modèle Observer

Subject : informer les Observer

```
class Subject(object):
    def __init__(self):
        self.observers=[]
    def notify(self):
        for obs in self.observers:
            obs.update(self)
```

En cas de modification des données du modèle :

- `notify()` : demander aux observateurs de se mettre à jour

Modèle Observer

Subject : ajouter/supprimer des Observer

```
    def attach(self, obs):
        if not hasattr(obs,"update"):
            raise ValueError("Observer must have \
                              an update() method")
        self.observers.append(obs)
    def detach(self, obs):
        if obs in self.observers :
            self.observers.remove(obs)
```

Modèle Observer

Observer : mise à jour

```
class Observer:
    def update(self, subject):
        raise NotImplementedError
```

Lorsque l'observable (Subject) est modifié :

- update() : on se met à jour

Modèle Observer

Exemple : Distributeur de billets

```
class ATM(Subject):
    def __init__(self,amount):
        Subject.__init__(self)
        self.amount=amount
    def fill(self,amount):
        self.amount=self.amount+amount
        self.notify()                    # obs.update(self)
    def distribute(self,amount):
        self.amount=self.amount-amount
        self.notify()                    # obs.update(self)
```

Modèle Observer

Exemple : Distributeur de billets

```
class Amount(Observer):
    def __init__(self,name):
        self.name=name
    def update(self,subject):
        print(self.name,subject.amount)
```

Modèle Observer

Exemple : Distributeur de billets

```
if __name__ == "__main__" :
    amount=100
    dab = ATM(amount)
    obs=Amount("Observer 1")
    dab.attach(obs)
    obs=Amount("Observer 2")
    dab.attach(obs)
    for i in range(1,amount/20) :
        dab.distribute(i*10)
    dab.detach(obs)
    dab.fill(amount)
```

MVC

Trygve Reenskaug

"MVC was conceived as a general solution to the problem of users controlling a large and complex data set. The hardest part was to hit upon good names for the different architectural components. Model-View-Editor was the first set. After long discussions, particularly with Adele Goldberg, we ended with the terms Model-View-Controller."

Smalltalk

"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse."

MVC

Exemple : variable de contrôle TkInter (IntVar())

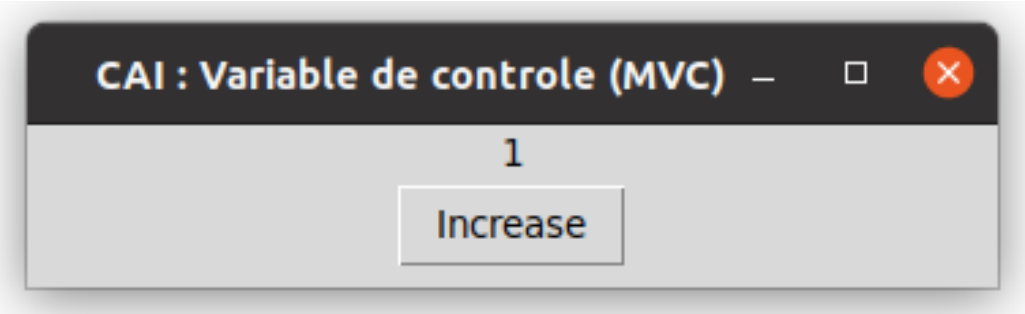
```
root=tk.Tk()
model=tk.IntVar()
model.set(1)
view=tk.Label(root,textvariable=model)
ctrl=tk.Button(root, text="Increase")
ctrl.bind("<Button-1>",lambda event,counting=model:\
                        increase(event,counting))
view.pack()
ctrl.pack()
root.mainloop()
```

- Modèle : les données de l'application (`IntVar()`)
- Vue : la visualisation des données (`Label()`)
- Contrôleur : la modification du modèle (`Button()`)

MVC

Exemple : Variables de contrôles TkInter

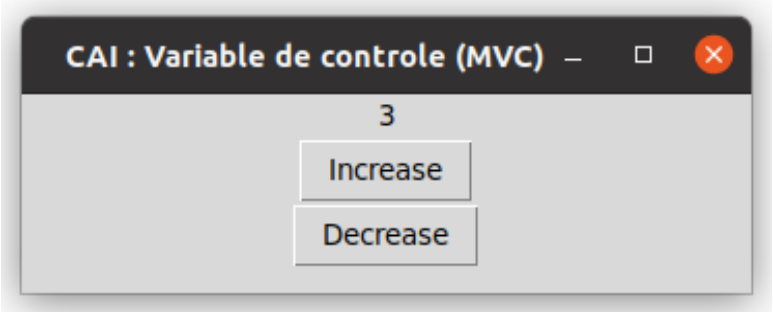
```
def increase(event,counting):
  value=counting.get()+1
  counting.set(value)
```



Exemple : Variables de contrôles TkInter

```
def decrease(event,counting):
  value=counting.get()-1
  counting.set(value)
```

MVC



Exemple : Variables de contrôles TkInter

```
if __name__ =="__main__" :
   ...
   ctrl_down=tk.Button(root, text="Decrease")
   ctrl_down.bind("<Button-1>",
                  lambda event,counting=model:\
                        decrease(event,counting))
   ...
```

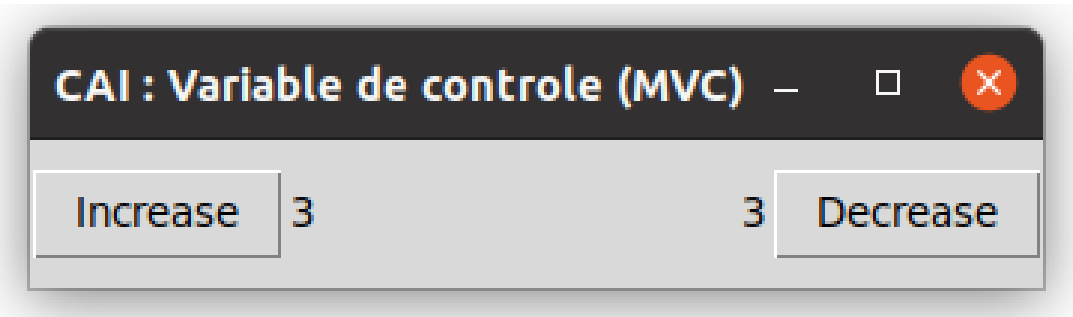
Un modèle, une vue, **plusieurs contrôleurs**.

MVC

Un modèle, **plusieurs vues**, plusieurs contrôleurs.

Exemple : Variables de contrôles TkInter

```
...
ctrl.pack(side="left")
view.pack(side="left")
...
view_down=tk.Label(root,textvariable=model)
ctrl_down.pack(side="right")
view_down.pack(side="right")
...
```

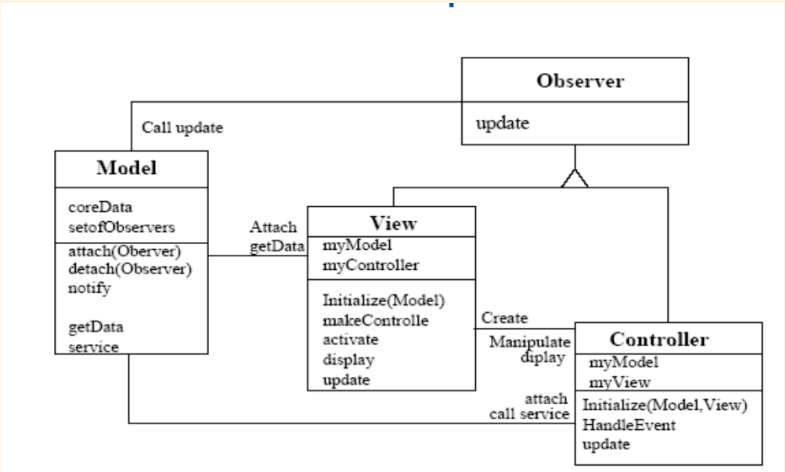


MVC

Modèle-Vue-Contrôleur

- Modèle : données de l'application (logique métier)
- Vue : présentation des données du modèle
- Contrôleur : modification (actions utilisateur) des données

MVC : diagramme de classes UML

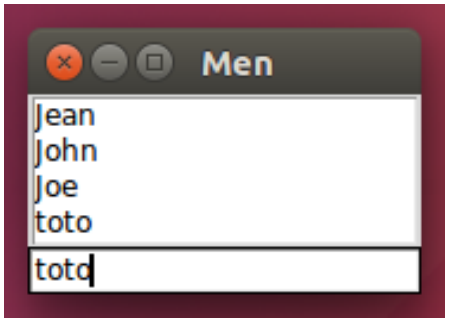


O. Boissier, G. Picard (SMA/G2I/ENS Mines Saint-Etienne)

MVC

Exemple : gestion d'une liste de noms

```
if __name__ == "__main__":
    mw=tk.Tk()
    mw.title("Men")
    names=["Jean", "John", "Joe"]
    model = Model(names)
    view = View(mw)
    view.update(model)
    model.attach(view)
    ctrl = Controller(model,view)
```



Modèle

Insertion, suppression de noms

```
class Model(Subject):
    def __init__(self, names=[]):
        Subject.__init__(self)
        self.__data = names
    def get_data(self):
        return self.__data
    def insert(self,name):
        self.__data.append(name)
        self.notify()                # obs.update(self)
    def delete(self, index):
        del self.__data[index]
        self.notify()                # obs.update(self)
```

Vue : l'Observer du modèle

Visualisation du modèle : update()

```
class View(Observer):
    def __init__(self,parent):
        self.parent=parent
        self.list=tk.Listbox(parent)
        self.list.configure(height=4)
        self.list.pack()
        self.entry=tk.Entry(parent)
        self.entry.pack()
    def update(self,model):
        self.list.delete(0, "end")
        for data in model.get_data():
            self.list.insert("end", data)
```

Contrôleur : du Subject à l'Observer

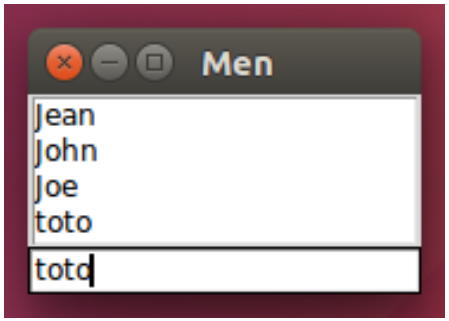
Contrôle du modèle : action utilisateur

```
class Controller(object):
     def __init__(self,model,view):
          self.model,self.view = model,view
          self.view.entry.bind("<Return>",
                                self.enter_action)
          self.view.list.bind("<Delete>",
                               self.delete_action)
     def enter_action(self, event):
          data = self.view.entry.get()
          self.model.insert(data)
     def delete_action(self, event):
          for index in self.view.list.curselection():
             self.model.delete(int(index))
```

Test IHM

Un modèle, une vue, un contrôleur

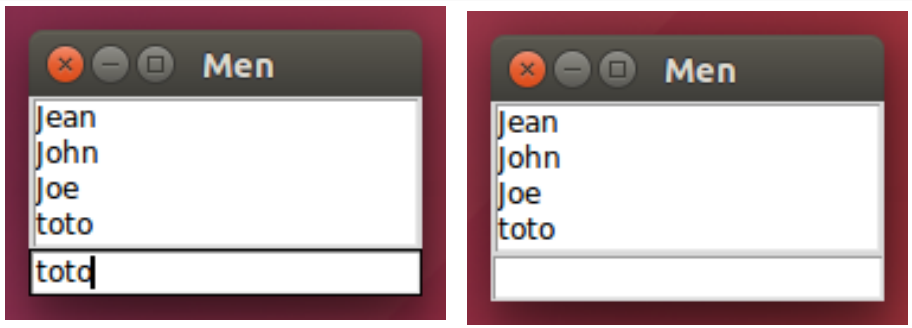
```
if __name__ == "__main__":
    mw=tk.Tk()
    mw.title("Men")
    names=["Jean", "John", "Joe"]
    model = Model(names)
    view = View(mw)
    view.update(model)
    model.attach(view)
    ctrl = Controller(model,view)
```



Test IHM

Un modèle, des vues, des contrôleurs

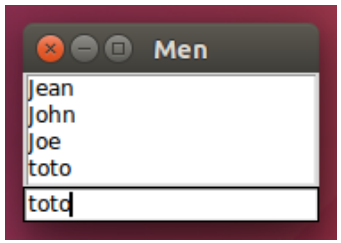
```
...
top = tk.Toplevel()
top.title("Men")
view = View(top)
view.update(model)
model.attach(view)
ctrl = Controller(model,view)
```

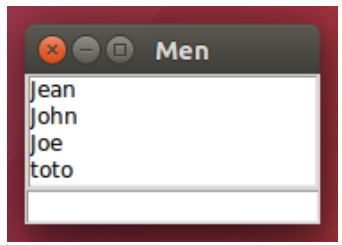


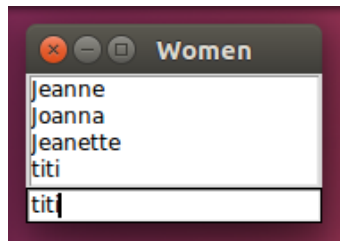
Test IHM

Des modèles, des vues, des contrôleurs

```
top = tk.Toplevel()
top.title("Women")
names=["Jeanne", "Joanna", "Jeanette"]
model = Model(names)
view = View(top)
view.update(model)
model.attach(view)
ctrl = Controller(model,view)
```



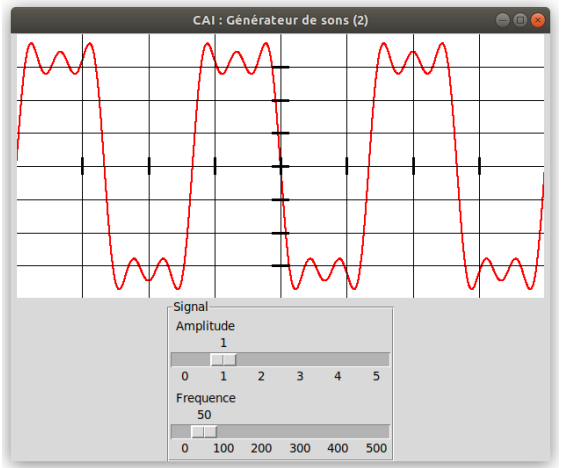




Labos : Objectifs

Modéliser, Visualiser et Contrôler des signaux

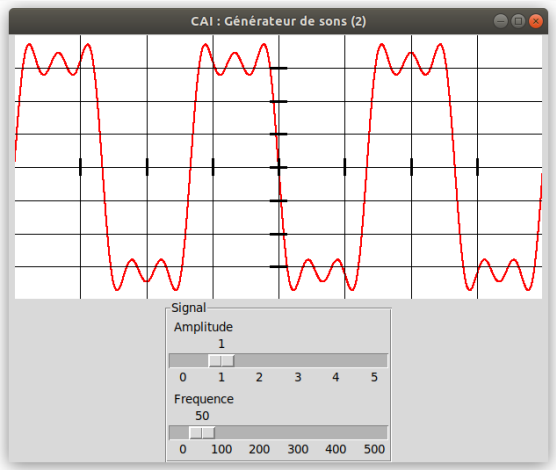
```
root=tk.Tk()
model_X=Generator("X")
view=Screen(root)
model_X.attach(view)
ctrl=Controller(root,model_X)
...
```

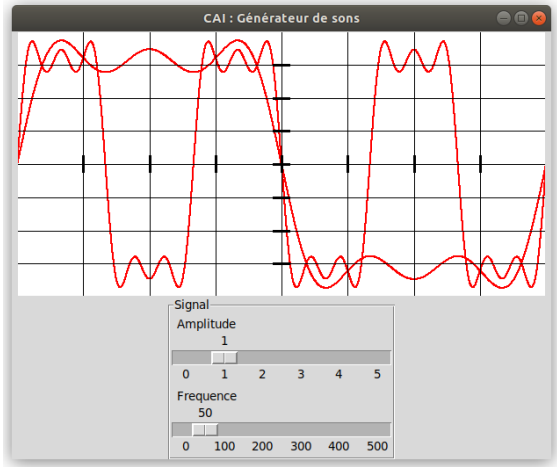


Labos : Objectifs

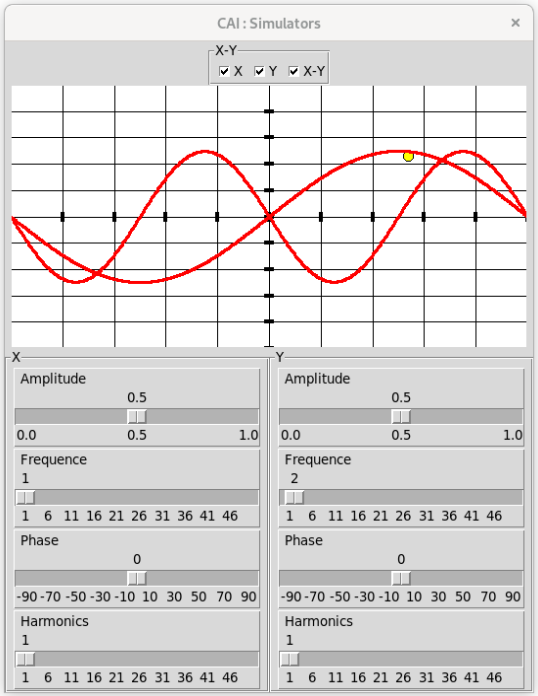
Modéliser, Visualiser et Contrôler des signaux

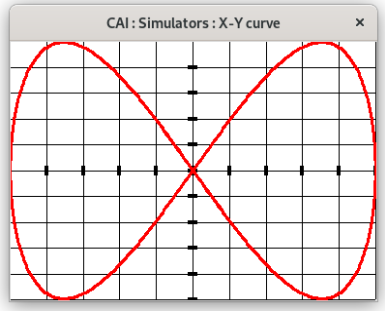
```
...
top = tk.Toplevel()
view=Screen(top)
model_X.attach(view)              # premier signal
model_Y=Generator("Y")            # deuxieme signal
model_Y.attach(view)
ctrl=Controller(top,model_Y)
...
```





Labos : Objectifs



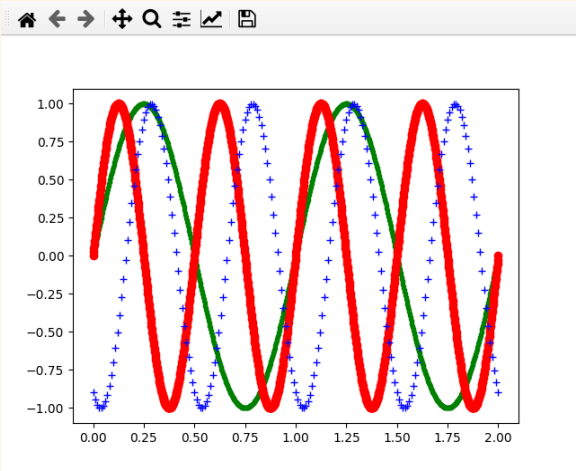


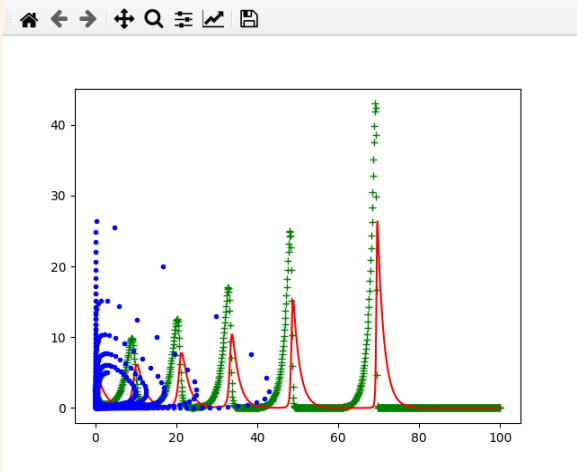
Modéliser, Visualiser et Contrôler des signaux

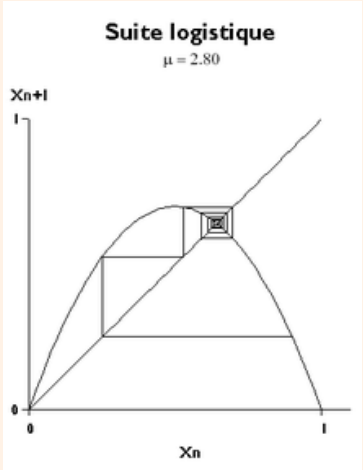
- contrôler tous les paramètres (amplitude,fréquence ...)
- charger, sauvegarder les paramètres des signaux
- sauvegarder les images représentées dans les vues
- animer un spot sur la trajectoire des courbes

Labos : Objectifs

Modéliser, Visualiser et Contrôler des signaux





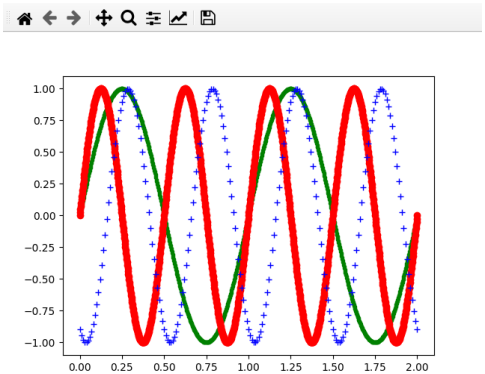


Générateur de signal

Mouvement vibratoire harmonique

$$e = \sum_{h=1}^{n} (a/h)\ sin(2\ \pi\ (f * h)\ t + \phi)$$

- a, f, p : amplitude, fréquence, phase
- h : nombre d'harmoniques ($h > 0$)



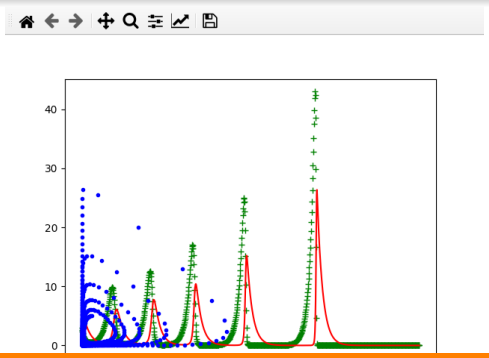
Générateur de signal

Équations de Lotka-Volterra (évolution proies-prédateurs)

$$\frac{dx(t)}{dt} = x(t).(\alpha - \beta.y(t))$$

$$\frac{dy(t)}{dt} = -y(t).(\gamma - \delta.x(t))$$

- $\mathrm{d}x(t)/\mathrm{d}t, \mathrm{d}y(t)/\mathrm{d}t$: variation au cours du temps
- α, δ : taux de reproduction des proies, prédateurs
- β, γ : taux de mortalité des proies, prédateurs

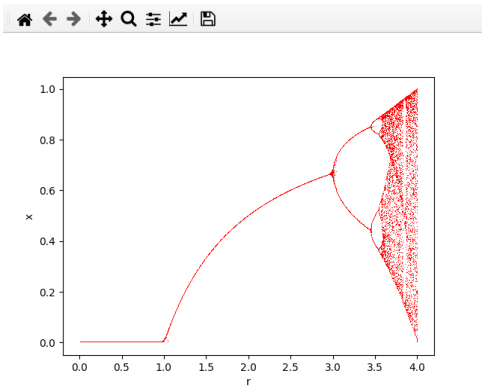


Générateur de signal

Suite logistique

$$x_{n+1} = \mu.x_n.(1 - x_n), x_0 \in [0, 1]$$

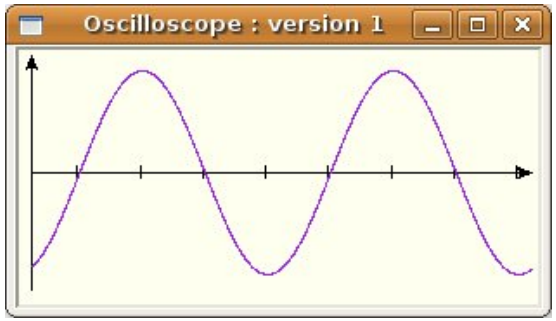
- x_n : rapport population sur population maximale
- μ : comportement asymptotique
  - convergence $(0 < \mu < 1; 1 < \mu < 2; 2 < \mu < 3)$
  - périodicité $(3 < \mu < 3.57)$
  - chaotique $(\mu >= 3.57)$



Modèle de signal

Son pur : mouvement vibratoire sinusoïdal

$$e = a\ sin(2\ \pi\ f\ t + \phi)$$



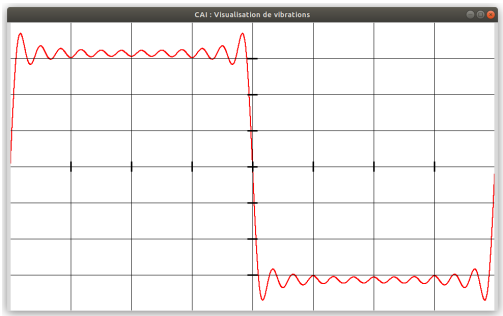
- e, t : élongation , temps
- a, f, ϕ : amplitude,fréquence,phase

Modèle de signal

Son complexe : mouvement vibratoire avec harmoniques

$$e = \sum_{h=1}^{n} (a/h)\ sin(2\ \pi\ (f * h)\ t + \phi)$$

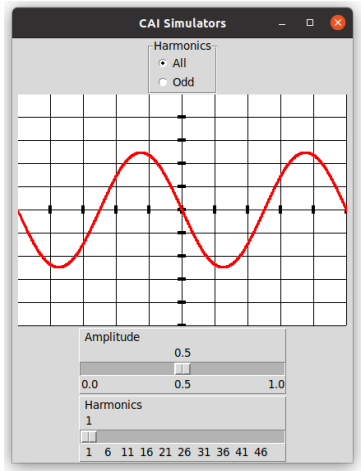
- e, t : élongation , temps
- a, f, ϕ : amplitude,fréquence,phase
- h : nombre d'harmoniques



Gestion du signal

Tout dans la classe (Screen)

```
root=tk.Tk()
view=Screen(root)
view.generate()
view.create_grid()
view.plot_signal(view.get_signal(),view.get_name())
view.packing()
root.mainloop()
```



Gestion du signal

Tout dans la classe (Screen)

```
class Screen :
  def __init__(self,parent,bg="white",name="signal",..):
    self.parent=parent
    self.name=name
    self.mag,self.freq,self.phase=mag,freq,phase
    self.harmonics=harmonics
    ...
    self.canvas=tk.Canvas(parent,bg=bg)
    self.width=int(self.canvas.cget("width"))
    self.height=int(self.canvas.cget("height"))
    ...
```

Génération du signal

Calcul de vibration

```
def vibration(self,t):
  a,f,p=self.mag,self.freq,self.phase
  sum=0
  for h in range(1,self.harmonics+1) :
    sum=sum+(a*1.0/h)*sin(2*pi*(f*h)*t-p)
  return sum
```

Génération du signal

Calcul du signal

```
def generate(self,period=1):
    del self.signal[0:]
    echantillons=range(int(self.samples)+1)
    Te=period/self.samples
    for t in echantillons :
        self.signal.append(
                            [t*Te,self.vibration(t*Tech)]
                            )
    return self.signal
```

Visualisation de signal

Dimensionnement du signal à visualiser

```
def plot_signal(self,signal,name="signal",color="red"):
  width,height=self.width,self.height
  if signal and len(signal)>1:
    self.canvas.delete(name)
    plot=[(x*width,height/2*(y+1)) \
                  for (x,y) in signal]
    self.canvas.create_line(plot,\
                            fill=color,smooth=1,\
                            width=3,tags=name)
```

Visualisation de signal

Création de la grille de visualisation

```
def create_grid(self):
  width,height=self.width,self.height
  tiles=self.tiles
  tile_x=width/tiles
  for t in range(1,tiles+1): # lignes verticales
    x=t*tile_x
    self.canvas.create_line(x,0,
                            x,height,
                            tags="grid")
    self.canvas.create_line(x,height/2-5,
                            x,height/2+5,
                            width=4,tags="grid")
```

Visualisation de signal

Création de la grille de visualisation

```
tile_y=height/tiles
for t in range(1,tiles+1): # lignes horizontales
  y=t*tile_y
  self.canvas.create_line(0,y,
                          width,y,
                          tags="grid")
  self.canvas.create_line(width/2-5,y,
                          width/2+5,y,
                          width=4,tags="grid")
```

Visualisation de signal

Redimensionnement de la visualisation

```
self.canvas.bind("<Configure>", self.resize)

  def resize(self, event):
    self.width = event.width
    self.height = event.height
    self.canvas.delete("grid")
    self.create_grid()
    self.plot_signal(self.signal,self.name)
```

Contrôle du signal

Création de contrôleur (Scale)

```
def create_controls(self):
  self.mag_var=tk.DoubleVar()
  self.mag_var.set(self.get_magnitude())
  self.scale_mag=
      tk.Scale(self.parent,
               variable=self.mag_var,
               label="Amplitude",
               orient="horizontal",length=250,
               from_=0,to=1,resolution=0.1,
               tickinterval=0.5,
               sliderlength=20,relief="raised",
               command=self.cb_update_magnitude)
```

Contrôle du signal

Création de contrôleur (RadioButton)

```
frame=tk.LabelFrame(self.parent,text="Harmonics")
self.radio_var=tk.IntVar()
btn=tk.Radiobutton(frame,text="All",
                   variable=self.radio_var,value=1,
                   command=self.cb_activate_button)
btn.select()
btn.pack(anchor ="w")
btn=tk.Radiobutton(frame,text="Odd",
                   variable=self.radio_var,value=2,
                   command=self.cb_activate_button)
btn.pack(anchor ="w")
frame.pack()
```

Contrôle du signal

Manipulation des contrôleurs

```
def cb_update_magnitude(self,event):
  self.set_magnitude(self.mag_var.get())
  self.generate()
  self.plot_signal(self.signal,self.name)

def cb_activate_button(self):
  self.harmo_odd_even=self.radio_var.get()
```

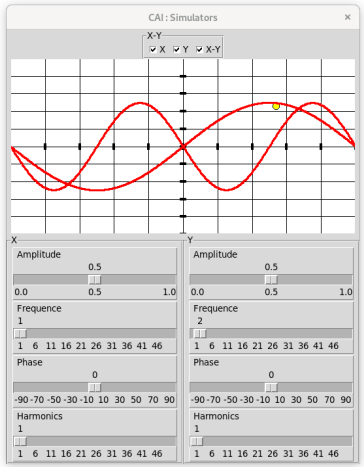
Modèle MVC :

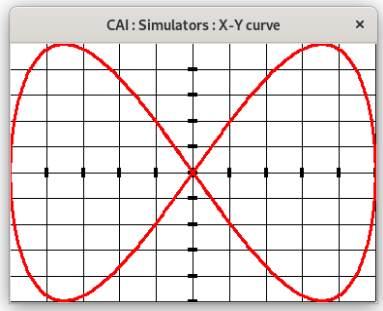
- plusieurs modèles pour une vue
- plusieurs vues pour un modèle
- différents contrôles sur plusieurs modèles

Modéliser, Visualiser et Contrôler des signaux

Modèle MVC

- `observer.py` : patron de conception (`Subject`,`Observer`)
- `generator.py` : modèle de signal (héritage `Subject`)
- `screen.py` : visualisation de signaux (héritage `Observer`)
- `controls.py` : contrôle de modèles de signaux
- `main.py` : intégrer le tout





Conclusion

Création d'Interfaces Homme-Machine

- un langage de programmation (python)
- une bibliothèque de composants graphiques (TkInter)
- gestion des événements (composant-événement-action)
- programmation des actions (callbacks, fonctions réflexes)
- création de nouveaux composants, d'applications
- mise en œuvre des patrons de conception (Observer,MVC)
- critères ergonomiques des IHM (Norme AFNOR Z67-110)

Bibliographie

Documents

- Gérard Swinnen : “Apprendre à programmer avec Python 3”
  `https://inforef.be/swi/python.htm`
- Guido van Rossum : “Tutoriel Python”
  `https://bugs.python.org/file47781/Tutorial_EDIT.pdf`
- Mark Pilgrim :
  “An introduction to Tkinter” (1999)
- John W. Shipman :
  “Tkinter reference : a GUI for Python” (2006)
- John E. Grayson :
  “Python and Tkinter Programming” (2000)
- Bashkar Chaudary :
  “Tkinter GUI Application Develoment Blueprints” (2015)

Bibliographie

Adresses “au Net”

- https://inforef.be/swi/python.htm
- https://docs.python.org/fr/3/library/tk.html
- https://wiki.python.org/moin/TkInter
- https://www.jchr.be/python/tkinter.htm
- https://www.thomaspietrzak.com/teaching/IHM