Simulation – Models and Time Loops

15-110 – Wednesday 11/13

Learning Goals

Build **simulations** to study how systems change over time

- Design **components** and **rules** that can form a simulation for a given problem
- Create graphical simulations using Tkinter and a **time loop**

Simulations and Models

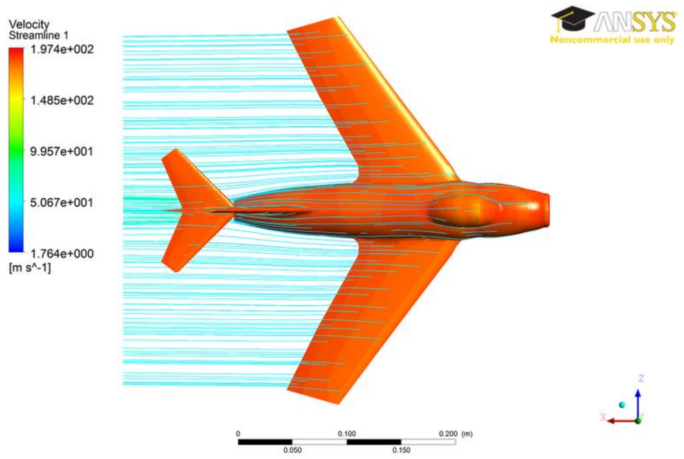
What are simulations?

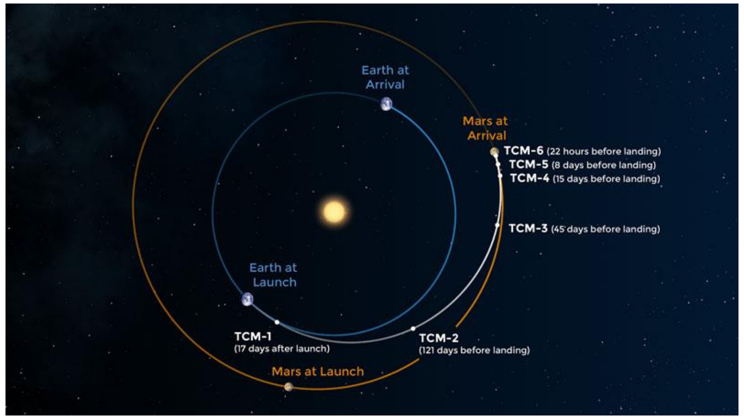
A **simulation** is an automated imitation of a real-world event.

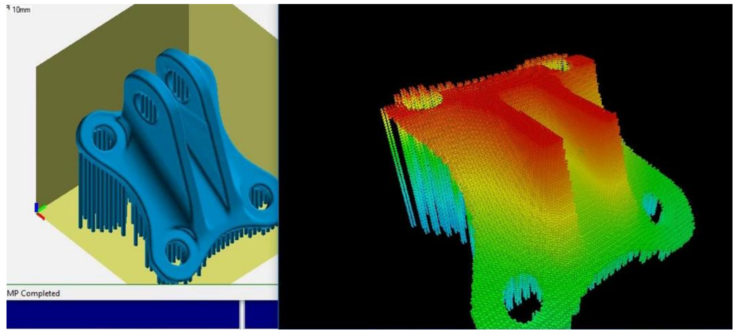
By running simulations on different starting inputs, and by interacting with them while they run, we can test how the event will change under different circumstances.

Examples of Simulations

Simulation is used across many different fields, including training people, testing designs, and predicting results.




Earth at Arrival
Mars at Arrival
TCM-6 (22 hours before landing)
TCM-5 (8 days before landing)
TCM-4 (15 days before landing)
TCM-3 (45 days before landing)
Earth at Launch
TCM-1 (17 days after launch)
TCM-2 (121 days before landing)
Mars at Launch



A model of a highway junction created with AnyLogic Road Traffic Library.
The cars are coming from each entry at constant rates.
At the Gardena exit there are periodic congestions that cause traffic jams.

Simulations vs. Real-world Experiments

Simulations share a lot in common with real world experiments. Major differences include:

- Experiments run in **real time**; simulations can be **sped up, slowed down, or paused**.
- Experiments can be **expensive**; simulations are fairly **cheap**.
- Experiments include **all possible factors**; simulations only include **factors we program in**.

Example Simulations

You can explore simulations across a variety of fields on the site NetLogo.

- Ant colony movements
- Flocking behavior
- Gravitational forces
- Climate change
- Fire spreading
- Rumor mills

Simulations Run on Models

How do we program a simulation? You need to design a good **model**, which will mimic the part of the real world you want to study. The simulation represents how the system represented by the model changes **over time**, or how it changes **based on events**.

Models are composed of two parts:

- The **components** of the system (information that describes the world at an exact moment).
- The **rules** of the system (how the components change as time passes).

Components are like variables, and rules are like functions!

Example Model

Problem: how will increasing the price of bread over the course of a few months affect how many people buy bread?

Model Components: current price; delta change in price; overall consumer count; distribution of consumer incomes

Model Rules: supply/demand relationship for bread; relationship between income and max amount willing to pay

Activity: Design a Model

Problem: if a zombie outbreak occurs, what proportion of the population will be human vs. zombie over the course of a year?

What are the components of this model? What are the rules?

Coding a Simulation

Simulation Parts in Code

We'll implement simulations in this class **graphically**, like in NetLogo. We'll use Tkinter again to do this!

Our simulation code will be composed of three parts:

- Making the **initial components**, by storing the starting component values in a shared data structure
- Implementing a **time loop**, which will repeatedly run the model's rules to update the components
- Graphically drawing a **view**, which will repeatedly display the current state of the components

Making the Components

We'll represent our components in code in a **dictionary** called **data**. The keys will take the place of variable names, while the value will be the actual component values.

For example, to store the price of bread, we could set **data["price"] = 5.00**.

By storing all of the components in one structure, we can pass the same structure around to all the functions we write, using **aliasing**. This will let us update data in one function, then display the updated components in another.

Running the Rules

To run the simulation's rules, we need to repeatedly call the rules at regular intervals. We'll do this by creating a **time loop**, and calling a function within that time loop.

To make a time loop, we'll used the built-in function **canvas.after**. This function lets us repeatedly call the same function (like recursion), but pauses before it makes the call. That lets us recurse/loop infinitely, while not freezing the window.

In the actual rules function, we'll update the values in **data**, to change them over time.

Displaying the Model

Finally, to display the whole model, we'll use Tkinter to draw graphics that represent the components visually. By referring to values in **data** in this function, we can make graphics based on pre-defined components.

We'll erase and re-draw the graphics window every time the rules of the simulation run. By changing the components a little bit at a time, this makes the display appear to be updating smoothly!

Simulation Functions

To implement these three steps, we'll use a new **simulation framework** that you can find linked on the course website. In this framework, you can update three functions that correspond to the three steps:

- **makeModel(data)** makes the original components. *data* is the model dictionary
- **runRules(data, call)** runs the rules to update data. *call* is an integer, and represents the number of times runRules has been called
- **makeView(data, canvas)** displays the model. *canvas* is a Tkinter canvas

You are not responsible for the code in timeLoop(data, canvas, call) or runSimulation(width, height, timeRate). But you can change the window size by changing the width and height, and speed up/slow down the simulation by changing timeRate.

Simple Example – color-changing ball

Let's start with a simple simulation. Say we want to draw a circle, and have the color of the circle change over time.

The **model** should hold any component values that might change. In this case, that's the **color** of the circle.

The **rules** should describe how the model changes over time. In this case, we **change the color** every call to runRules().

The **view** should draw a circle in the middle of the window, and set its color based on the color in the model.

Simple Example Code

```
def makeModel(data):
    # put variables in data here
    data["color"] = "red"

def makeView(data, canvas):
    # (200, 200) is center point
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50, fill=data["color"])

def runRules(data, call):
    if data["color"] == "red":
        data["color"] = "green"
    elif data["color"] == "green":
        data["color"] = "blue"
    else:
        data["color"] = "red"
```

Advanced Example – Zombie Outbreak

Now let's try something a little harder, and simulate a **zombie outbreak**.

Model: humans and zombies move around on a 2D grid. Start with 20 humans and 1 zombie.

View: humans and zombies will both be squares. Humans are purple, zombies are green.

Rules: every step, move each zombie in a random direction on the grid. If a zombie is touching (bordering) a human, turn the human into a zombie.

Zombie Outbreak Code

Check the course website after lecture for the final zombie outbreak code!

Note that when you program an advanced simulation, you usually want to start by programming just the model and view, then test by running the code. Then try adding one rule at a time, testing each one as you go. This makes it easier to identify bugs early on.

Using Simulations

Once we've programmed a robust simulation, we can **change the starting state** to see how it changes the simulation. This is especially useful when we want to **predict** certain things about the world.

We can check predictions more quickly by making timeRate smaller (calling the simulation more often).

For example: how long will it take for the whole world to become zombies...

- In our current code?
- If we start with more or fewer humans?
- If we start with more zombies?

Learning Goals

Build **simulations** to study how systems change over time

- Design **components** and **rules** that can form a simulation for a given problem
- Create graphical simulations using Tkinter and a **time loop**