Simple dynamical modelling

Last time

- Introduction to Julia syntax
- Performance
- Some Julian tips:
  - Return from function, don't print
  - Types are important, but don't worry too much right now
  - Multiple dispatch: Different *methods* of same function for related functionality

Goals for today

- Simple models of **dynamics**
- **Simulate** to produce **data**
- **Explore data** by **plotting** and **interacting**
- **Stability** analysis

Dynamics

- Many models in sciences, engineering, economics…
  concern **dynamics**:
  *how system changes (**evolves**) over time*
- Examples:
  - stock market
  - chemical reactions
  - noise in jet engine
  - interactions of genes
  - motion of galaxy

Modelling

- What is a **model**?
  Computational / mathematical description of small piece of world
- Why is it necessary to model?
- Full system (=world) is too complicated.
- Isolate effects of interest.
- Start simple, understand behaviour.
- Make more complicated (add additional effects), understand how results change.
- Repeat!

Modelling radioactivity and populations

- Best models can represent different phenomena.
- Simple model that can describe **radioactivity**, **population dynamics**, …
- How does population of bacteria / number of radioactive atoms / stock price grow or decay over time?

- **Suppose** ("hypothesis"): bacteria reproduce every certain period of time (20 mins for Escherichia coli)
- **Suppose**: each bacterium / atom produces, *on average* λ offspring.
- Death is taken account of by λ ($=$ birth rate $-$ death rate).
- If population is *small*, randomness is important – see later classes.
- If population is *large* then *(population at end of season)* $= \lambda \times$ *(population at start).*
- Note: We are leaving out effects of *space* in this model and just counting total number of individuals.

Mathematical model

- Mathematical description of model.
- Notation: x_n is population at nth time step.
- **Initial condition**: population at start, x_0
- Mathematical description of dynamics: specify x_n in terms of x_0, x_1, ..., x_{n-1}.
- Often will depend only on *one previous step*.

Mathematical model II

- After step 1: $x_1 = \lambda \cdot x_0$
- After step 2: $x_2 = \lambda \cdot x_1$

  etc.
- Generalize: **recurrence relation**

$$x_{n+1} = \lambda x_n$$

- In many applications, instead have **differential equations** – how a quantity changes *continuously* in time, expressed using time derivatives.
- But numerical solution methods often use discrete time steps anyway!

Computational thinking

- Want to know system's **behaviour** – **dynamics** / evolution in time.
- Our growth model is simple enough that can be solved *analytically* – formula for x_n at time n (**exercise**)
- But in general mathematical models *cannot* be solved analytically – e.g. Robert May, *Nature* 1976
- Instead use computer as *tool* to investigate behavior.

Computational thinking II

- How *map* model $\longrightarrow$ **computational experiment**?
- Create a small “world” inside computer.
- Start off by representing the **initial data**: x_0, λ and final time N as variables.
- Need a way to store all the x_n

Computational thinking III

- Solution: math x_n corresponds to computation `x[n]`, where `[n]` means "index into suitable data structure"
- 6.0001: use Python dictionary or list for this
- Using dictionary:

```
x0 = 20.0  # initial population
λ = 1.2    # type as \lambda<TAB>
N = 20     # final time

x = Dict(0 => x0)  # empty (untyped) dictionary

for n in 1:N     # 1:N is a Range object
    x[n+1] = λ * x[n]
end
```

- `1 => x0` is a `Pair` representing fact that `1` maps to `x0`
- Using Julia `Vector` (1D array) instead:

```
x = zeros(N+1)  # Vector of zeros
x[1] = x0

for n in 2:N+1
    x[n+1] = λ * x[n]
end
```

- We *preallocated* an array to store data in since we knew how much storage we needed.
- Note that *arrays have index starting at 1* in Julia.
- Package `OffsetArrays.jl` allows other indexing.

Creating data over time

- Often create / read data and store incrementally, e.g. when don't know how much storage we need
- Need a data structure that knows how to automatically grow
- `Dict` allows this
- Common alternative if ordered data, e.g. in time: create empty `Vector` and add data to it
- `push!(vv, a)`; cf. `vv.append(a)` in Python.
- `!` is convention in Julia: function that *modifies its argument*

- Argument must be *mutable*, e.g. `Vector`
- Disadvantage: loses explicit index `n`; implicit as "the index of the last element that was added"
- Advantage: *Think in a new way*, less mathematically and more computationally:
  - "current value of `x`"
  - "next value of `x` that will be produced"
- Suggestion: Call vector `xs` since stores many values of `x`; allows using `x` for current value:

```
x = x0        # current value
xs = [x0]     # initialize Vector with initial value

for n in 1:N
    next_x = λ * x
    push!(xs, next_x)

    x = next_x
end
```

- Key step: *update current value* at end of loop
- New value will be used as input in next loop iteration.
- Mathematically: often write x' for next value: $x' = \lambda\, x$
- Julia: `x′ = λ * x` – write as `\prime<TAB>`
- **Julia looks like math**

Anonymous functions

- Math: $f_\lambda(x) = \lambda\, x$
- Function of one variable, x, with one parameter, λ
- Julia: "the function that takes x to $2x$" is written

```
x -> 2x
```

- Called an **anonymous function** since it has no name.

- Now can write `julia` `f(λ) = x -> λ * x`
- So `f(3)` *is an (anonymous) function*

Exploring data: Visualization

- Now want to explore *how data behaves*.
- "Exploratory data analysis": visualize by plotting
- We choose `Plots.jl` package (out of several options).
- Install package (one time) with

  ```
  using Pkg
  Pkg.add("Plots")
  ```

  or

  ```
  ]add Plots
  ```

- `Pkg` is *built-in* Julia package manager.
- In REPL, `]` enters package mode: prompt `(v1.2) pkg>`.

Plotting

- Once installed, load in *each* session:

```
using Plots
```

- Can now plot data as *points* (`scatter`) or lines (`plot`):

```
scatter(x)
```

- If x coordinates not given, Julia plots the data against time.

Automate: put it into a function

- Have generated data for specific values of λ and x_0.
- How is dynamics *affected* by x_0 and λ?
- Must change them and recalculate.
- *Never* copy, paste and modify values by hand!
- Instead, introduce *abstraction*: **create a function**

```julia
"""
Simulate growth with rate `λ`, initial value `x0` and time `N`.
"""
function growth(λ, x0, N)

    x = zeros(N+1)
    x[1] = x0

    for n in 1:N
        x[n+1] = λ * x[n]
    end

    return x
end
```

- Docstrings (within `"""`) are placed *above* function body.
- Can now easily experiment with different values by executing the function:

```
growth(1.2, 10.0, 10)
growth(2.0, 10.0, 10)
```

- **Exercise**: What happens if `x0` and `\lambda` are integers instead of `Float64`?
- Which argument is which? It's difficult to remember.
- Use **keyword arguments** (NB: semicolon, `;`):

```
growth(; λ=1.1, x0=20.0, N=10) = growth(λ, x0, N)
```

- Call with

```
growth(N=20)
```

```
growth(N=20, x0=1.0)
```

- Note that we have *added a method* to the function `growth`:

```
methods(growth)
```

Automating plotting

- It's tempting to add plotting to the data generation.
- But we shouldn't do so:
  always separate *data generation and plotting*
- Reasons for this:
  - Data generation is slow; may want to plot data in different ways.
  - Plotting is slow or inconvenient – may not want to plot, e.g. if running non-interactively
- **Exercise**: Write a function `plot_growth` that takes the same parameters, generates the data and then plots.

Interactive visualizations

- It's still clumsy to modify values and re-plot by hand. Can we do this more intuitively and *interactively*?
- `Interact.jl` package: interactive widgets
- Can be used inside Juno and Jupyter notebook
- Install and load `Interact.jl`. Then try

```
@manipulate for i in 1:10
    i^2
end
```

- *Looks like* standard `for` loop, with `i^2` calculated at each iteration.
- But it is doing something different
- `@manipulate` is a **macro** – think of as a “super-function”
- Macro takes piece of Julia code and modifies it
- Returns new piece of Julia code, which is executed *in place of* old code; Julia never sees old code, only modified code.
- NB: In normal `for` would need to output; in a `@manipulate` the last calculated value is displayed automatically

- Can use `@macroexpand` to see new code. Here it generates the widgets.
- Can manipulate more than one variable at a time:

```
using Interact
@manipulate for a in 0:10, b in 1:11
    HTML( (a, b) )   # use HTML representation of output
end
```

- **Exercise**: Use `@manipulate` to visualize the dynamics of the simple growth model as λ and x_0 are varied. What are suitable (physically realistic) bounds for those variables?

  There is a key *critical value* of λ where the *dynamics changes qualitatively*. What is it? This is called a **bifurcation**.

Nonlinear dynamics

- General dynamics with single-step function f:

$$x_{n+1} = f(x_n)$$

- **Discrete-time dynamical system**
- Apply f to previous output at each step.
- Example: $f = \cos$.
- "Repeatedly press the cos key on your calculator". What happens?
- **Exercise**: Implement this.

Fixed points

- Behaviour is *completely different* from previous: iterates **converge** to a **fixed point**:

$$x_n \to x^* \text{ as } n \to \infty$$

- **Fixed point**: value that *does not change* when f is applied:

$$x^* = f(x^*)$$

- **Transcendental equation**: *impossible* to find explicit form for solution.
- But iterative method successfully solves this equation (with certain precision).

Rate of convergence

- How “good” is this method?
- Measure *distance* of x_n from x^*, i.e. $\delta_n := |x_n - x^*|$
- **Exercise**: Implement this.
- Find that it converges “quickly”.
- How characterize *rate* of convergence, i.e. how *fast* does δ_n decrease as function of n?
- Plot data differently: **log scale**
- `yscale=:log` in `Plots.jl`

Stability analysis

- What can we say about $\delta_n := |x_n - x^*|$ analytically?
- For large n, we know x_n is *close to* x^*, so

$$x_{n+1} = x^* + \delta_{n+1} = f(x_n) = f(x^* + \delta_n) \qquad (1)$$
$$\simeq f(x^*) + \delta_n f'(x^*), \qquad (2)$$

- Approximately satisfies *linear* dynamics:

$$\delta_{n+1} = \lambda\, \delta_n,$$

with $\lambda = f'(x^*)$ (constant).

- *We already understand this!*
- **Computing and mathematics often consist of trying to reduce a new problem to a problem you already know how to solve!**
- Behaviour of nonlinear systems near fixed points can often be reduced to analyzing the *linearized* system.

Review

- Modelling allows us to study isolated phenomenon in the computer
- Produce data incrementally and store it in a `Vector` using `push!`
- Examine its behaviour by plotting
- Make it interactive to quickly scan possible behaviours
- Reduce a problem to one you already know how to solve if you can!