TopEdu

# Starter code for *ten* research projects

A guide for students and mentors: how to run each starter, what it should show you, and where your own research begins.

START HERE

# How to use this guide

Each TopEdu project comes with one short Python file. It solves the project's equation with simple methods you can read line by line, checks itself against an answer that is known exactly, and ends with a list of first experiments. The research is what you do after that.

## What is in the download

| File | What it is |
|---|---|
| `01_chip_cooling.py` | Project 1: Where should the hot chip go? |
| `02_soil_temperature.py` | Project 2: How deep does summer go? |
| `03_critical_patch.py` | Project 3: How small can a nature reserve be? |
| `04_tissue_oxygen.py` | Project 4: How thick can lab-grown tissue be? |
| `05_pipe_resonance.py` | Project 5: The missing notes of a closed pipe |
| `06_basin_resonance.py` | Project 6: Why earthquakes pick certain buildings |
| `07_damped_string.py` | Project 7: Why a plucked note changes as it fades |
| `08_beam_torsion.py` | Project 8: Why a ruler twists so easily |
| `09_cantilever_beam.py` | Project 9: When the textbook beam formula breaks |
| `10_solver_race.py` | Project 10: The race between solvers |
| `check_all.py` | Runs all ten and confirms each one passes its check. Run it once after installing. |
| `requirements.txt` | The two libraries the files need: numpy and matplotlib. |

## Set up in ten minutes

1. **Install Python 3** from python.org (on a Mac it may already be there: type `python3 --version` in Terminal).
2. **Install the two libraries.** In Terminal (Mac) or PowerShell (Windows), in the folder you unzipped:

   ```
   python3 -m pip install -r requirements.txt
   ```

   On Windows, type `py` where this guide says `python3`.
3. **Check everything works:**

   ```
   python3 check_all.py
   ```

   It should end with `10 of 10 pass`, in about a minute.
4. **Run your project:**

   ```
   python3 01_chip_cooling.py --show
   ```

`--show` opens the plots in a window. Without it they are saved in the `output` folder.

**No computer of your own?** A school Chromebook works with Google Colab, which is free: open a new notebook, upload the file with the folder icon on the left, and run `!python 01_chip_cooling.py` in a cell. The plots appear in the `output` folder in the same panel.

## How every starter is built

All ten files have the same five parts, in this order, so once you have read one you can find your way around the others.

| Part | What it holds | What you do with it |
|---|---|---|
| The opening note | The question, the model in words, the units, how to run it | Read it first, and say it back to your mentor without notes |
| The setup | Named numbers in capitals, such as `LENGTH` or `KAPPA`, set to the online lab's defaults | This is where your experiments start |
| The solver | One or two short functions that step or solve the equation on a grid | Read every line; ask about any you cannot explain |
| The check | A comparison with a known exact answer, and one `CHECK ... PASS` line | Never edit the check to make it pass. A FAIL after your change is information |
| YOUR TURN | Four first experiments, as comments at the bottom | Your weeks 4 to 9 grow out of these |

## The rules that make it research

- **Predict before you run.** Write your guess, and why, in your log first. A wrong prediction you can explain is worth more than a right one you cannot.
- **Keep a log.** Date, what you changed, what you expected, what happened. Your paper is written from it.
- **Check the grid.** Run your key result again with twice the points. If the answer moves, report by how much.
- **Compare with the world.** Every project has a measurement or a public data set. The gap between the model and the world is where the research is.
- **Own every line.** Your mentor explains; you write and run. Name everyone and everything that helped, including this starter.

START HERE

# Five ideas you will meet

The starters use only a handful of numerical ideas. Each is one paragraph here; your mentor can show you more.

### *A grid and finite differences*

The computer cannot store a temperature at every point, so it stores one every $h$ metres. A second derivative becomes a difference of neighbours: `u''(x) ≈ (u[i+1] − 2 u[i] + u[i−1]) / h²`. Halving $h$ cuts that error about four times: this is called second-order accuracy, and project 10 measures it.

### *Stepping in time, and the speed limit*

Heat and wave starters move forward in small time steps. A step that is too large makes the numbers blow up. For heat, the step must stay below about `h² / (2 × diffusivity)`; for waves, a wave must not cross more than one grid cell per step (the CFL rule). Each file says where its limit is: try breaking it once, on purpose.

### *Edges*

A fixed value on an edge (a cold edge, a rock wall, a clamped string) is set directly. An insulated or free edge uses a mirror point outside the grid, a copy of its neighbour, so nothing crosses it.

### *Hearing frequencies*

The pipe, basin and string starters record a signal and find its notes with a Fourier transform (`numpy.fft`). A parabola through the tallest point and its two neighbours finds a peak more precisely than the spacing of the transform.

### *Fitting a straight line to logarithms*

If something grows like a power, `y = C xᵖ`, then `log y` is a straight line in `log x` with slope $p$. Exponential fades become straight lines in `log y` against time. `numpy.polyfit` fits the line. You will use this in almost every project.

CONTENTS

# The ten projects


| | | |
|---|---|---|
| 1 | Where should the hot chip go? | Heat and engineering |
| 2 | How deep does summer go? | Earth science and climate |
| 3 | How small can a nature reserve be? | Mathematical biology |
| 4 | How thick can lab-grown tissue be? | Biophysics and medicine |
| 5 | The missing notes of a closed pipe | Acoustics and music |
| 6 | Why earthquakes pick certain buildings | Earthquake science |
| 7 | Why a plucked note changes as it fades | Physics and music |
| 8 | Why a ruler twists so easily | Mechanics and structures |
| 9 | When the textbook beam formula breaks | Civil and mechanical engineering |
| 10 | The race between solvers | Numerical analysis and computing |


Each project has the same pages: the question, the objectives, the model in the starter, what you should see, your first experiments, the data, the real-world test, and the honest limit.

PROJECT 1 · HEAT AND ENGINEERING

# Where should the hot chip go?

*Your phone throttles when it gets hot. Find out why where the chip sits matters so much.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `01_chip_cooling.py` | Algebra II, exponentials; Fourier series are met along the way | A laptop. Optional: a small metal plate and low-cost thermistors, with an adult |

## The question

A hot chip sits on a metal plate cooled along one edge. How much does moving the chip, or spreading its heat, change the peak temperature, and can you predict the best layout for two chips?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 01_chip_cooling.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: Compare the first steady-state runs with the series solution you derive for the plate.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Sweep the chip position and size, record the peak temperature, then add a second heat source and search for the coolest layout. Then step away from the screen: heat a small aluminium plate with a power resistor, cool one edge in ice water, and read temperatures with low-cost thermistors.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: material properties are uniform, so the model cannot represent a plate made of two metals.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

A chip heats a square metal plate. The bottom edge (y = 0) is held cold, the other three edges are insulated. How hot does the plate get, and how much does the chip's position change that?

Units are scaled: the plate is 1 by 1, and temperature is "degrees above the cold edge". Run: python 01_chip_cooling.py (add --show to open the plots in a window) Needs: Python 3 with numpy and matplotlib (pip install numpy matplotlib)

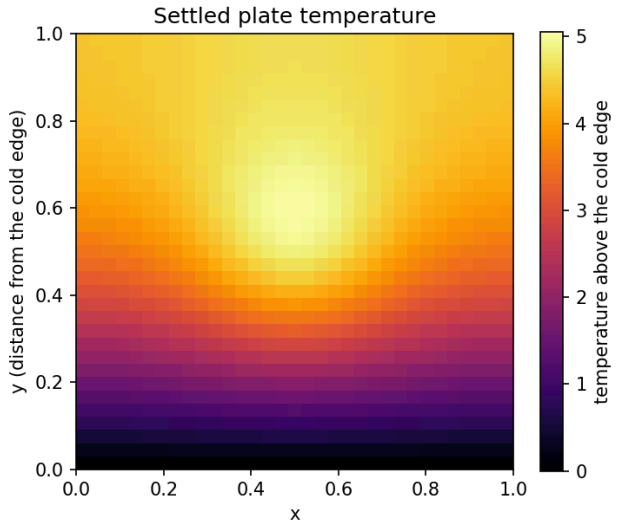
## What the starter does

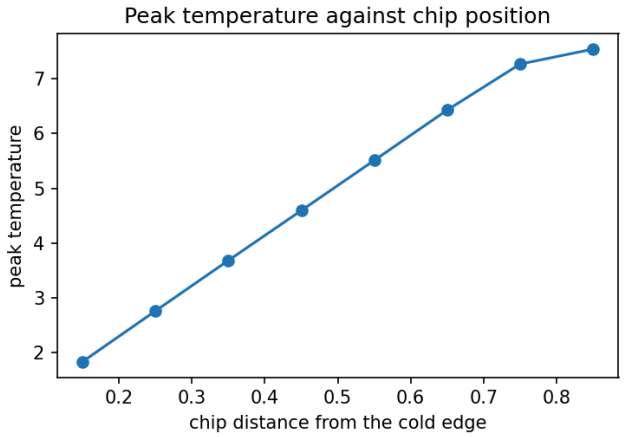
- Stage 1 (weeks 1 to 3): run this file. It solves the heat equation step by step until the plate settles, then checks the peak temperature against an exact series answer. You should see a CHECK line that says PASS.
- Stage 2 (weeks 4 to 9): change the chip, then answer your own question in the YOUR TURN section at the bottom.

## What you should see

Run `python3 01_chip_cooling.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
settled after 32248 steps; peak 5.0544 at x = 0.50, y = 0.59
series answer at that point: 5.0495
CHECK peak temperature against the series: error 0.10% -> PASS (limit 1%)
  chip at distance 0.15 from the cold edge: peak 1.837
  chip at distance 0.25 from the cold edge: peak 2.763
  chip at distance 0.35 from the cold edge: peak 3.684
  chip at distance 0.45 from the cold edge: peak 4.595
  chip at distance 0.55 from the cold edge: peak 5.510
  chip at distance 0.65 from the cold edge: peak 6.428
  chip at distance 0.75 from the cold edge: peak 7.267
  chip at distance 0.85 from the cold edge: peak 7.540
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Write your prediction first: if the chip size doubles with the same total power, does the peak halve? (Hint: to keep the total power fixed, scale POWER by (old size / new size)^2.) Then run it.
2. Add a second chip: heat_source(x, y, 0.3, 0.4) + heat_source(x, y, 0.7, 0.6). Search for the two positions with the lowest peak. Is the best layout symmetric? Is it what you guessed?

3. Grid check: run with N = 17, 33 and 65. How much does the peak change? Report that in your paper.
4. Real world: heat an aluminium plate with a power resistor and cool one edge in ice water. Which of your predictions survive? The gap is the interesting part.

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Chip position, x | 0.15 to 0.85 plate widths | 0.5 |
| Distance from the cooled edge | 0.15 to 0.85 plate widths | 0.5 |
| Chip size | 0.06 to 0.25 plate widths | 0.12 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Heat, 2D)** | peak temperature 7.3284 at (0.25, 0.8125) |
| **Known answer** | series solution 7.3271 |
| **Error** | 0.02% (limit 1.00%): passed |
| **A first finding** | moving the chip from 0.7 to 0.2 plate widths from the cooled edge changes the peak from 7.33 to 2.44 |

## The real world

Heat a small aluminium plate with a power resistor, cool one edge in ice water, and read temperatures with low-cost thermistors.

## The honest limit

Material properties are uniform, so the model cannot represent a plate made of two metals.

How the full project checks itself: Compare the first steady-state runs with the series solution you derive for the plate. The online lab runs the same project on the research solver (Heat, 2D).

PROJECT 2 · EARTH SCIENCE AND CLIMATE

# How deep does summer go?

*Dig down far enough and it is winter in summer. Measure how far, using real climate data.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `02_soil_temperature.py` | Algebra II, trigonometry, logarithms | A laptop and free public data |

## The question

Seasonal temperature swings fade and arrive later as you go underground. From real sensor data, what is your soil's thermal diffusivity, and how does it change with wetness or with the site?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 02_soil_temperature.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: The model has an exact answer: amplitude falls as e^(-z/d) and the delay grows as z/d, with d = $\sqrt{(2\kappa/\omega)}$. Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start. Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Drive the surface with a yearly temperature cycle, measure the amplitude and the delay at each depth, and fit the diffusivity to data from a real station. Then step away from the screen: download free soil temperatures at 5 to 100 cm from NOAA's U.S. Climate Reference Network and compare stations with different soils. Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: soil is one uniform layer here, so a mismatch at some depths is itself a finding to explain: layers or moisture may be the cause. Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

The ground's surface warms and cools once a year. Underground the swing gets smaller and arrives later. How much smaller, and how much later, tells you the soil's thermal diffusivity (kappa).

Units: depth in metres, time in years, temperature in degrees C away from the yearly average. Run: python 02_soil_temperature.py [--show] or python 02_soil_temperature.py my_station.csv Needs: numpy and matplotlib

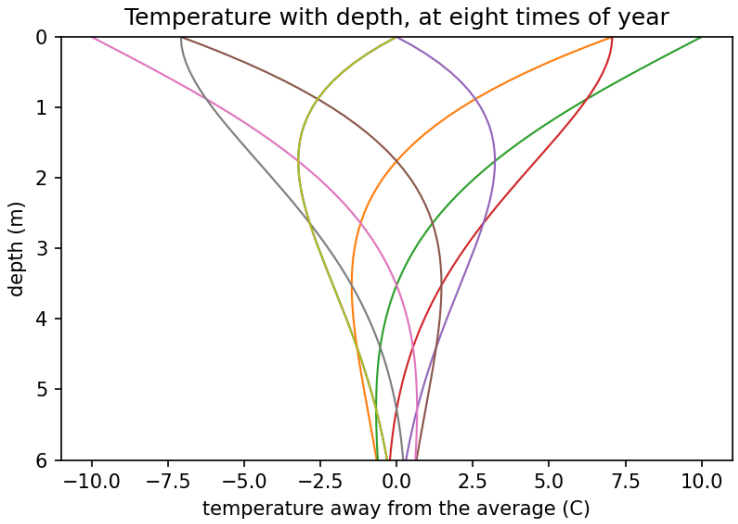
## What the starter does

- Stage 1: run this file. It steps the heat equation through one year in a 20 m column of soil, driven by a yearly cycle at the surface, and compares the whole field with the exact answer. The CHECK line should say PASS.
- Stage 2: fit kappa from temperatures at several depths, first from the model, then from a real station.

## What you should see

Run `python3 02_soil_temperature.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
largest error over the year: 1.97e-03 degrees C (0.020% of the swing)
CHECK whole year against the exact answer -> PASS (limit 1% of the swing)
  depth 0.05 m: swing  9.78 C,   1.3 days behind the surface
  depth 0.10 m: swing  9.56 C,   2.6 days behind the surface
  depth 0.20 m: swing  9.15 C,   5.1 days behind the surface
  depth 0.50 m: swing  8.00 C,  13.0 days behind the surface
  depth 1.00 m: swing  6.40 C,  25.9 days behind the surface
kappa from shrinking: 15.81   kappa from lag: 15.73   (the model used 15.8)
damping depth d = 2.24 m: below about 3d = 6.7 m the year is almost invisible
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Predict first: if the soil were twice as diffusive (wet), how much deeper would the 1 C swing reach? Then run with KAPPA = 31.6 and check.
2. Download two years of soil temperatures from a USCRN station, put them in the simple CSV layout described in read_station, and run: python 02_soil_temperature.py my_station.csv
3. Compare two stations with different soils or rainfall. Do "kappa from shrinking" and "kappa from lag" agree? If not, why might a real soil break this one-layer model?

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Thermal diffusivity | 5 to 40 m²/year | 15.8 |
| Surface swing | 2 to 25 °C | 10 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Heat, 1D)** | largest error against the exact solution over a year: 1.44e-3 °C; swing at depth 2.24 m reaches 3.58 °C over 8 frames |
| **Known answer** | amplitude 10 °C at the surface, 3.68 °C at one damping depth |
| **Error** | 0.01% (limit 1.00%): passed |

## The real world

Download free soil temperatures at 5 to 100 cm from NOAA's U.S. Climate Reference Network and compare stations with different soils.

## The honest limit

Soil is one uniform layer here, so a mismatch at some depths is itself a finding to explain: layers or moisture may be the cause.

How the full project checks itself: The model has an exact answer: amplitude falls as e^(−z/d) and the delay grows as z/d, with d = √(2κ/ω). The online lab runs the same project on the research solver (Heat, 1D).

PROJECT 3 · MATHEMATICAL BIOLOGY

# How small can a nature reserve be?

*Why do some species vanish from parks that look big enough? The answer fits in one equation.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `03_critical_patch.py` | Algebra II, exponentials, one guided substitution | A laptop and published field studies |

## The question

A population grows inside a habitat and is lost when it wanders out. For a fixed area, which rectangle keeps a species alive, and how does the minimum size depend on how far the species roams?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 03_critical_patch.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: For an a by b rectangle the decay rate must match $D\pi^2(1/a^2 + 1/b^2)$, which gives the classic critical patch size.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Write density as u = e^(rt) · v, so v obeys the heat equation. Measure how fast v decays in each shape: the species survives when r beats that decay rate. Then step away from the screen: put measured dispersal and growth rates from published field studies into the model, then compare with the size of a real reserve.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: growth is exponential with no crowding, so this predicts whether a species survives, not how large its population becomes.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

Animals in a rectangular reserve multiply at rate r and wander at rate D. Any that cross the edge are lost. Write the density as u = e^(r t) v. Then v obeys the plain heat equation dv/dt = D laplacian(v) with v = 0 on the edge, and v dies away at some rate lambda. The species survives exactly when r is bigger than lambda.

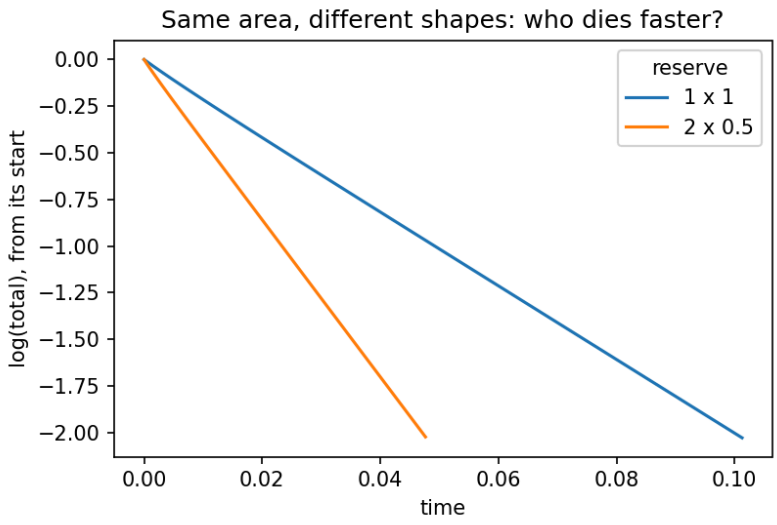
## What the starter does

- Stage 1: run this file. It measures lambda for a 1 x 1 and a 2 x 0.5 reserve (the same area) and checks both against the exact rate D pi^2 (1/a^2 + 1/b^2). The CHECK line should say PASS.
- Stage 2: use real dispersal and growth rates and ask which shapes of a real reserve would keep a species.

## What you should see

Run `python3 03_critical_patch.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
1 x 1 reserve: measured decay rate 19.749, exact 19.739, error 0.05%
   so the species needs a growth rate r > 19.75 to survive here
2 x 0.5 reserve: measured decay rate 42.084, exact 41.946, error 0.33%
   so the species needs a growth rate r > 42.08 to survive here
CHECK decay rates against D pi^2 (1/a^2 + 1/b^2) -> PASS (limit 1%)
  area 1, length/width 1: needs r > 19.74  (1.00 times the square)
  area 1, length/width 1.5: needs r > 21.38  (1.08 times the square)
  area 1, length/width 2: needs r > 24.67  (1.25 times the square)
  area 1, length/width 3: needs r > 32.90  (1.67 times the square)
  area 1, length/width 4: needs r > 41.95  (2.12 times the square)
  area 1, length/width 6: needs r > 60.86  (3.08 times the square)
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. The list above uses the formula. Check two of its shapes with decay_rate() before you trust it.
2. A real species: find published values of D (km^2 per year) and r (per year) for one animal or plant. What is the smallest square reserve that keeps it? Compare with the size of a real reserve.
3. Predict, then test: does an L-shaped reserve do better or worse than a square of the same area? (Hint: set v to zero on a corner block every step to cut the L out of a rectangle.)

4. Say what the model leaves out: here nothing limits growth, so it tells you "survives or not", not "how many".

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Reserve width | 0.5 to 4 length units | 1 |
| Reserve height | 0.5 to 4 length units | 1 |
| Roaming rate D | 0.2 to 4 area per time | 1 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Heat, 2D (with a substitution))** | 1 x 1: decay rate 19.762; 2 x 0.5: decay rate 41.717 |
| **Known answer** | 19.739; 41.946 |
| **Error** | 0.54% (limit 1.00%): passed |
| **A first finding** | same area, but the long thin reserve needs a growth rate 2.1 times higher to survive |

## The real world

Put measured dispersal and growth rates from published field studies into the model, then compare with the size of a real reserve.

## The honest limit

Growth is exponential with no crowding, so this predicts whether a species survives, not how large its population becomes.

How the full project checks itself: For an a by b rectangle the decay rate must match $D\pi^2(1/a^2 + 1/b^2)$, which gives the classic critical patch size. The online lab runs the same project on the research solver (Heat, 2D (with a substitution)).

PROJECT 4 · BIOPHYSICS AND MEDICINE

# How thick can lab-grown tissue be?

*Lab-grown organs have a size limit, and it comes from physics. Work out what it is.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `04_tissue_oxygen.py` | Algebra II, exponentials; cosh is introduced in the project | A laptop and published research |

## The question

Oxygen diffuses into tissue while cells consume it, which limits how large a tissue grown without blood vessels can be. What is the largest cube whose centre still gets enough oxygen, and do shape and surface area matter?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 04_tissue_oxygen.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: A slab has the exact answer c = cosh((x-a)/L) / cosh(a/L), with penetration length L = √(D/k).<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Find the side length at which the centre falls to a chosen oxygen threshold; repeat for flat slabs and long bars of the same volume. Then step away from the screen: compare the penetration length with published oxygen diffusion limits and with the sizes of real organoids and spheroids.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: consumption here is proportional to oxygen. Real cells keep consuming at a nearly fixed rate until oxygen is low, so say how that would change your estimate.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

Oxygen soaks into tissue from its surface while the cells use it up. With no blood vessels, the centre of a large piece starves. In scaled units the oxygen c obeys -laplacian(c) + c = 0 inside, with c = 1 on the surface. Lengths are measured in "penetration lengths" L = sqrt(D / k): D is how fast oxygen spreads, k how fast it is used.

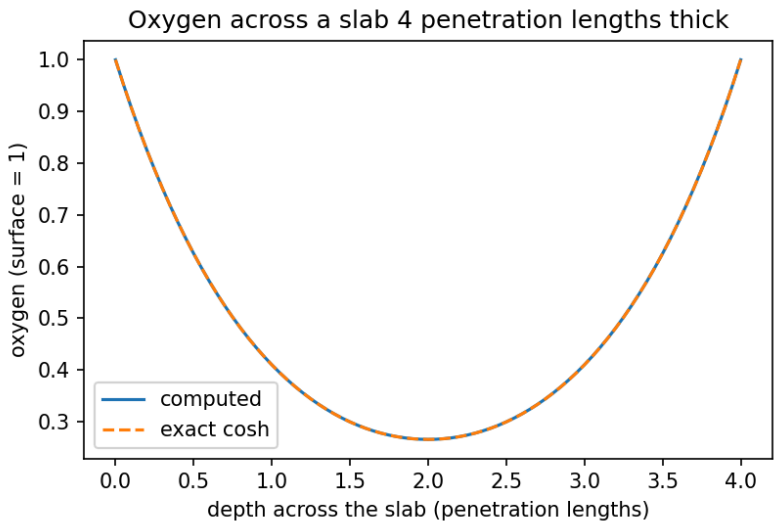
## What the starter does

- Stage 1: run this file. It solves a flat slab (exact answer: cosh) and a cube (exact answer: a series), and checks both. The CHECK line should say PASS.
- Stage 2: find the largest cube, slab and bar whose centre stays above an oxygen threshold you choose.

## What you should see

Run `python3 04_tissue_oxygen.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
slab 4 thick: centre 0.2659, exact 0.2658, worst error 5.3e-05
cube side 1: centre oxygen 0.9462, series 0.9459, error 0.03%
cube side 2: centre oxygen 0.8073, series 0.8061, error 0.15%
cube side 4: centre oxygen 0.4630, series 0.4596, error 0.73%
CHECK slab and cubes against exact answers -> PASS (limit 1%)
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Pick a threshold, say the centre must keep 20% of the surface oxygen. Predict the largest cube side, then find it by trying sides between 2 and 8 with box_centre_series (fast) and checking one with box().
2. Same volume, other shapes: a slab 0.5 thick and a long square bar (make one side 10). Which keeps its centre alive best? Is it the one with the most surface?
3. Turn units back into micrometres: look up D for oxygen in tissue and a cell consumption rate, compute L = sqrt(D / k), and compare your limit with published organoid sizes (often a few hundred micrometres).

4. Honest limit: real cells use oxygen at a nearly fixed rate until it runs low. Would that make your limit larger or smaller? Argue it before you test it.

## The data

*Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Cube side | 0.5 to 8 penetration lengths | 2 |

*Checked on the research solver*

| | |
|---|---|
| **Solver (Reaction-diffusion, 3D)** | side 1: centre 0.9462; side 2: centre 0.8072; side 4: centre 0.4632 |
| **Known answer** | 0.9459; 0.8061; 0.4596 |
| **Error** | 0.78% (limit 1.00%): passed |

## The real world

Compare the penetration length with published oxygen diffusion limits and with the sizes of real organoids and spheroids.

## The honest limit

Consumption here is proportional to oxygen. Real cells keep consuming at a nearly fixed rate until oxygen is low, so say how that would change your estimate.

How the full project checks itself: A slab has the exact answer c = cosh((x-a)/L) / cosh(a/L), with penetration length L = √(D/k). The online lab runs the same project on the research solver (Reaction-diffusion, 3D).

PROJECT 5 · ACOUSTICS AND MUSIC

# The missing notes of a closed pipe

*Tap a pipe and it plays a note. Predict that note, then find what the textbook gets wrong.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `05_pipe_resonance.py` | Algebra II, trigonometry | A phone and about $10 of PVC pipe |

## The question

A pipe closed at one end should play only odd harmonics. How closely does a real pipe follow the model, and what is the size of the 'end correction' the ideal model leaves out as the pipe gets wider?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 05_pipe_resonance.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: Pipe modes are $f_n = (2n - 1)c / 4L$, with no even harmonics.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Simulate a pulse in the pipe, extract its frequencies, and compare them with recordings of PVC pipes of several lengths and diameters. Then step away from the screen: tap or blow across PVC pipes and record them with a free phone spectrum app.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: the model has a perfect open end. The gap between the model and your recording is the discovery, not a failure.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

Sound in a pipe closed at one end and open at the other obeys the wave equation. The closed end reflects pressure as it is; the open end (ideally) forces pressure to zero. So the pipe only fits odd quarter-waves, and its notes are f_n = (2n - 1) c / 4L: 1, 3, 5, … times the lowest note, with no even harmonics.

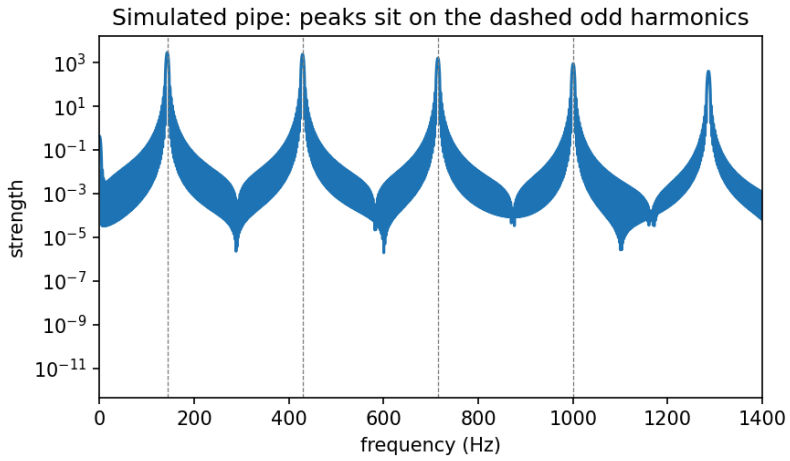
## What the starter does

- Stage 1: run this file. It starts a pressure pulse in a 0.6 m pipe, listens at the closed end, finds the notes with a Fourier transform, and checks the first two against the formula. The CHECK line should say PASS.
- Stage 2: record real PVC pipes, find their notes with the same code, and measure what the ideal model leaves out.

## What you should see

Run `python3 05_pipe_resonance.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
note 1: simulated  142.92 Hz, formula  142.92 Hz, error 0.000%
note 2: simulated  428.75 Hz, formula  428.75 Hz, error 0.001%
ratio of note 2 to note 1: 3.000 (the ideal pipe says 3: no even harmonic)
CHECK first two notes against (2n - 1) c / 4L -> PASS (limit 1%)
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Predict, then record: a 0.6 m PVC pipe closed at one end (a palm or a cap). Is its lowest note above or below the formula? Run: python 05_pipe_resonance.py my_pipe.wav
2. End correction: a real open end acts as if the pipe were longer, by about delta. From each recording, delta = c / (4 f_1) - L. Measure it for pipes of three or more widths. How does delta depend on the radius? (Textbooks quote about 0.6 times the radius. Do your pipes agree?)
3. Look for even harmonics in your recordings. If any appear, what could make them?
4. The speed of sound changes with temperature: c is about 331 + 0.6 T(C) m/s. Note the room temperature.

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Pipe length | 0.2 to 2 m | 0.6 |
| Speed of sound | 320 to 360 m/s | 343 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Wave, 1D)** | mode 1: 142.90 Hz; mode 2: 428.43 Hz |
| **Known answer** | 142.92 Hz; 428.75 Hz |
| **Error** | 0.07% (limit 1.00%): passed |
| **A first finding** | the second mode sits at 2.998 times the first: 3, with no even harmonic |

## The real world

Tap or blow across PVC pipes and record them with a free phone spectrum app.

## The honest limit

The model has a perfect open end. The gap between the model and your recording is the discovery, not a failure.

How the full project checks itself: Pipe modes are $f_n = (2n - 1)c / 4L$, with no even harmonics. The online lab runs the same project on the research solver (Wave, 1D).

PROJECT 6 · EARTHQUAKE SCIENCE

# Why earthquakes pick certain buildings

*In 1985 an earthquake spared some Mexico City buildings and destroyed others. Height was part of why.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `06_basin_resonance.py` | Algebra II, trigonometry | A laptop and published seismic data |

## The question

Soft sediment over rock traps seismic waves at a natural period. How do depth, width and wave speed set that period, and when does the quick estimate T = 4H/c stop being good enough?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 06_basin_resonance.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: Modes satisfy $\omega^2 = c^2\pi^2(m^2/W^2 + (n - ½)^2/H^2)$.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Model a basin cross-section with rock on three sides and a free surface on top, measure its period over many shapes, then map the error of the quick estimate. Then step away from the screen: compare with the well-documented 1985 Mexico City lakebed, where shaking near a 2-second period damaged mid-height buildings; compare with published data for another basin.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: wave speed is the same everywhere and the basin is a rectangle, so this is a first-order explanation and not a hazard estimate. Do not present it as a prediction for a real place.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

A valley of soft sediment sits in rock. Shear waves bounce between the rock walls and floor and the free ground surface, so the valley rings at a natural period. Buildings whose own sway period is close to it shake hardest.

This model is a rectangle W wide and H deep. Ground motion u obeys the wave equation with u = 0 at the rock (the two sides and the floor) and no stress at the surface. Its slowest ring has period T = 2 pi / omega, omega = c pi sqrt(1/W^2 + 1/(2H)^2). Engineers' quick estimate ignores the width: T = 4H / c.

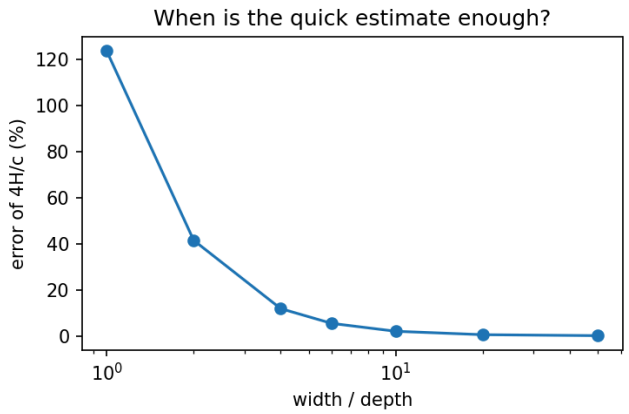
## What the starter does

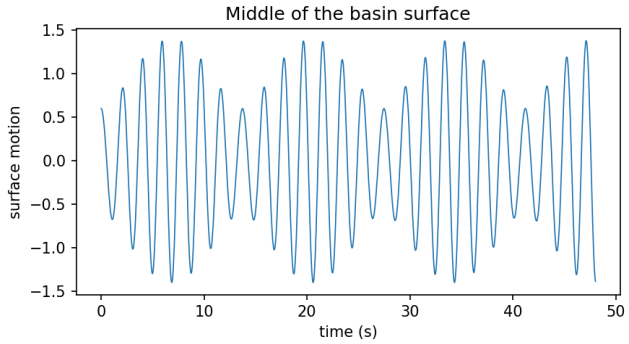
- Stage 1: run this file. It shakes the basin, listens at the middle of the surface, and checks the period it hears against the exact one. The CHECK line should say PASS.
- Stage 2: map where the quick estimate is good enough, and compare with a real, documented basin.

## What you should see

Run `python3 06_basin_resonance.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
basin 400 m wide, 40 m deep, c = 80 m/s
simulated period 1.961 s, exact 1.961 s, quick estimate 4H/c = 2.000 s
CHECK period against the exact formula: error 0.01% -> PASS (limit 1%)
  width/depth   1: 4H/c is off by 123.6%
  width/depth   2: 4H/c is off by  41.4%
  width/depth   4: 4H/c is off by  11.8%
  width/depth   6: 4H/c is off by   5.4%
  width/depth  10: 4H/c is off by   2.0%
  width/depth  20: 4H/c is off by   0.5%
  width/depth  50: 4H/c is off by   0.1%
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Before running: at what width/depth does 4H/c reach 10% error? Read it off the plot and check with simulate().
2. The formula assumes a rectangle. Make the floor slope (set u = 0 below a slanted line each step) and see how the period moves. Explain why.

3. Mexico City, 1985: look up published lakebed depth and shear wave speed. Does the model's period come near the roughly 2 s that damaged mid-height buildings? Say clearly that this explains, and does not predict.
4. Rule of thumb: a building's sway period is about 0.1 s per storey. Which heights match your basin?

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Basin width | 100 to 2000 m | 400 |
| Sediment depth | 20 to 100 m | 40 |
| Shear wave speed | 50 to 400 m/s | 80 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Wave, 2D)** | fundamental period 1.962 s |
| **Known answer** | exact 1.961 s; quick estimate 4H/c = 2.000 s |
| **Error** | 0.04% (limit 1.00%): passed |

## The real world

Compare with the well-documented 1985 Mexico City lakebed, where shaking near a 2-second period damaged mid-height buildings; compare with published data for another basin.

## The honest limit

Wave speed is the same everywhere and the basin is a rectangle, so this is a first-order explanation and not a hazard estimate. Do not present it as a prediction for a real place.

How the full project checks itself: Modes satisfy $\omega^2 = c^2\pi^2(m^2/W^2 + (n - ½)^2/H^2)$. The online lab runs the same project on the research solver (Wave, 2D).

PROJECT 7 · PHYSICS AND MUSIC

# Why a plucked note changes as it fades

*A guitar note changes as it fades. Capture that on your phone and explain what the simple model misses.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `07_damped_string.py` | Algebra II, trigonometry, logarithms | A guitar or a home-made string, and a phone |

## The question

This damping model makes every harmonic fade at one rate. In a real guitar the upper harmonics die sooner. By how much, and how does it depend on the string's material and gauge?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 07_damped_string.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: Each mode has the exact solution e^(-at/2) times an oscillation at $\sqrt{(\omega^2 - a^2/4)}$.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Fit the damping from a recording, simulate the same note, and measure where the single-rate model departs from the data for each harmonic. Then step away from the screen: record plucked strings on a phone and measure each harmonic's decay on a spectrogram.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: one constant damping term cannot describe a real string. Showing exactly how it fails, and proposing a fix, is the result.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

A string of length L vibrates as u_tt + a u_t = c^2 u_xx, fixed at both ends. The term a u_t is a simple drag. In this model every harmonic fades as e^(-a t / 2), all at the same rate. A real guitar loses its high harmonics sooner, so the sound gets mellower as it fades. Measuring by how much is the research.

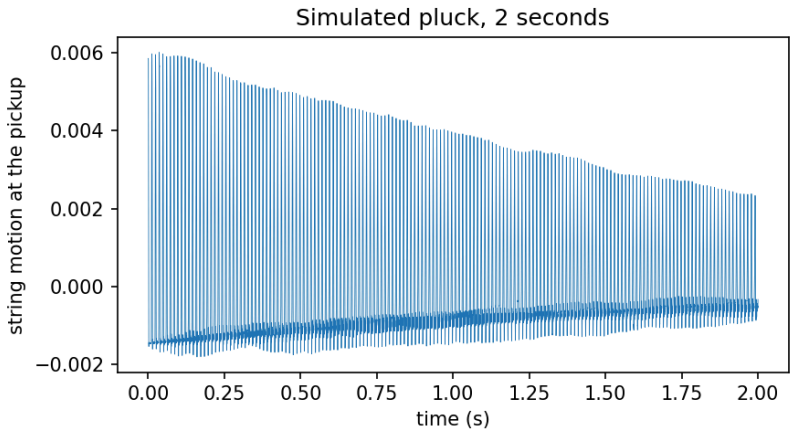
## What the starter does

- Stage 1: run this file. It plucks a low E string (82.4 Hz), steps it for four periods, and compares the whole string with the exact answer. Then it measures each harmonic's fade from the simulated sound. The CHECK line should say PASS.
- Stage 2: record real strings, measure each harmonic's fade the same way, and show where one damping rate fails.

## What you should see

Run `python3 07_damped_string.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
after four periods: largest error along the string 3.04e-06 (pluck size 1.3)
CHECK string against the exact damped answer -> PASS (limit 1% of the pluck)
each harmonic's fade in the model (strength ~ e^(-rate t)); the model says every rate is a/2
= 0.5
  harmonic 1 (  82.4 Hz): rate 0.500 per second
  harmonic 2 ( 164.8 Hz): rate 0.500 per second
  harmonic 3 ( 247.2 Hz): rate 0.500 per second
  harmonic 4 ( 329.6 Hz): rate 0.500 per second
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Record one plucked string on a phone (a quiet room, 3 seconds, saved as .wav). Set F to its note and run: python 07_damped_string.py my_pluck.wav
2. Plot rate against harmonic number. One a cannot fit it. Try rate = b0 + b2 * k^2. Does that fit better?
3. Compare a wound and a plain string, or nylon and steel. Which loses its high harmonics fastest?
4. Then fix the model: what term would make high harmonics fade faster? (Hint: a drag on u_xxt.)

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Note frequency | 40 to 440 Hz | 82.4 |
| Damping | 0.1 to 8 1/s | 1 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Telegraph, 1D)** | largest error against the exact solution over 8 frames: 7.05e-4 |
| **Known answer** | exact damped two-mode solution, amplitude 1.3 |
| **Error** | 0.05% (limit 1.00%): passed |

## The real world

Record plucked strings on a phone and measure each harmonic's decay on a spectrogram.

## The honest limit

One constant damping term cannot describe a real string. Showing exactly how it fails, and proposing a fix, is the result.

How the full project checks itself: Each mode has the exact solution e^(-at/2) times an oscillation at $\sqrt{(\omega^2 - a^2/4)}$. The online lab runs the same project on the research solver (Telegraph, 1D).

PROJECT 8 · MECHANICS AND STRUCTURES

# Why a ruler twists so easily

*Twist a ruler, then a pencil. Why such a difference? Work out the rule, then test it with your hands.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `08_beam_torsion.py` | Geometry, Algebra II; numerical integration | Foam or wood bars and some weights |

## The question

Twisting a bar is governed by a surface shaped like a soap film over its cross-section. How does stiffness depend on the cross-section's shape, and how well does the simple thin-strip rule work?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 08_beam_torsion.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: A square bar's torsion constant is 0.1406 times the side to the fourth power; a long thin strip approaches 1/3.<br>Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start.<br>Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Solve for rectangles of many aspect ratios, compute the twist stiffness from each solution, then twist real bars and compare. Then step away from the screen: hang known weights from an arm attached to foam or wooden bars and measure the twist angle.<br>Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: only rectangles are supported, and only small twists stay in the elastic range.<br>Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

When you twist a bar, its stiffness comes from a "stress function" phi over the cross-section, shaped like a soap film blown up over a frame the shape of the section: -laplacian(phi) = 2 inside, phi = 0 on the edge. The torsion constant is J = 2 x (volume under the film). For an a by b rectangle (b the short side) write J = k a b^3. A square has k = 0.1406; a long thin strip approaches k = 1/3. Twist per length = torque / (G J), with G the shear modulus.

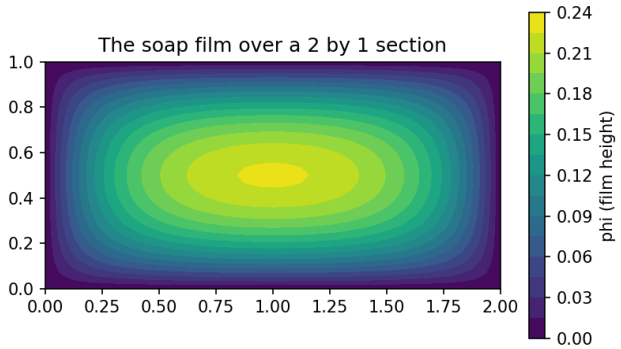
## What the starter does

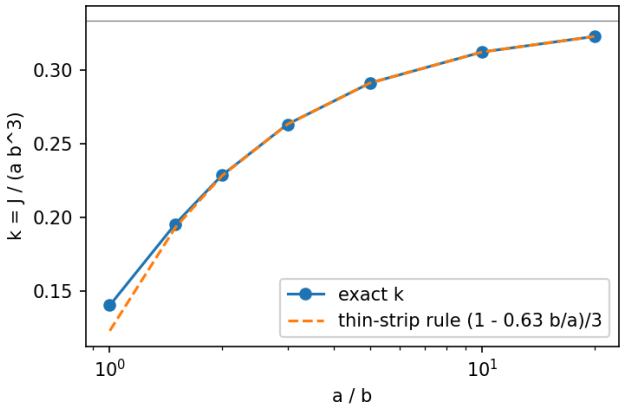
- Stage 1: run this file. It solves for phi on three rectangles and checks k against the exact series. The CHECK line should say PASS.
- Stage 2: twist real bars, measure their stiffness, and test the rule.

## What you should see

Run `python3 08_beam_torsion.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
1:1 bar: k = 0.1403, series 0.1406, error 0.20%
2:1 bar: k = 0.2283, series 0.2287, error 0.18%
4:1 bar: k = 0.2802, series 0.2808, error 0.20%
CHECK torsion factor against the series -> PASS (limit 1%)
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. A ruler is about 30 by 3 mm; a pencil about 7 by 7 mm. Predict which is stiffer in twist, and by how much, using J = k a b^3. Is the answer what your hands tell you?
2. Where does the thin-strip rule (the dashed line) go wrong by more than 5%? Read it from the plot, then check.
3. Experiment: clamp one end of a foam or wood bar, hang a weight from an arm at the other end, and measure the twist angle. Stiffness = torque x length / angle should be G J. Do bars of different shapes agree on G?
4. Honest limit: only rectangles, and only small twists. Try a hollow square by setting phi to a constant on an inner hole, and say what changes.

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Long side | 1 to 10 cm | 2 |
| Short side | 1 to 4 cm | 1 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Poisson, 2D)** | 1:1 beta 0.1404; 2:1 beta 0.2284; 4:1 beta 0.2804 |
| **Known answer** | 0.1406; 0.2287; 0.2808 |
| **Error** | 0.16% (limit 1.00%): passed |
| **A first finding** | beta = 0.1406 for the square, as published |

## The real world

Hang known weights from an arm attached to foam or wooden bars and measure the twist angle.

## The honest limit

Only rectangles are supported, and only small twists stay in the elastic range.

How the full project checks itself: A square bar's torsion constant is 0.1406 times the side to the fourth power; a long thin strip approaches 1/3. The online lab runs the same project on the research solver (Poisson, 2D).

PROJECT 9 · CIVIL AND MECHANICAL ENGINEERING

# When the textbook beam formula breaks

*Every engineering student learns this formula. Find out exactly when it stops being true.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `09_cantilever_beam.py` | Algebra II | Rulers, coins and a clamp |

## The question

The classic beam formula is taught as fact, but it assumes a long, slender beam. How short can a beam be before the formula is wrong by 5%, and does the material's Poisson ratio change that?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 09_cantilever_beam.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: A slender beam's tip sag should approach $3\rho g L^4 / (2Eh^2)$. Our test runs give 0.996 of that at length/depth 14 and 1.073 at 3. Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start. Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Simulate a square-section beam fixed at one end, sagging under its own weight, while shortening it step by step, and map the error of the formula. Then step away from the screen: clamp rulers or strips at different lengths to a desk, load them with coins, and measure the sag. Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: the clamped end has a stress singularity, so results near it depend on the grid. Report grid sensitivity; this is a study and not a design tool. Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

A beam fixed at one end sags under its own weight. The textbook (Euler-Bernoulli) answer for the tip sag is w = q L^4 / (8 E I) with q = weight per length, E the stiffness of the material, I = h^4 / 12 for a square. It assumes a long, slender beam. A short beam also bends by shear, which the formula leaves out. Timoshenko's theory adds it: w = q L^4 / (8 E I) + q L^2 / (2 kappa G A), with G = E / (2 (1 + nu)) and kappa = 5/6.

The full 3D answer needs the online lab (elasticity in 3D is too heavy for a starter). This file does the parts you can do on a laptop:

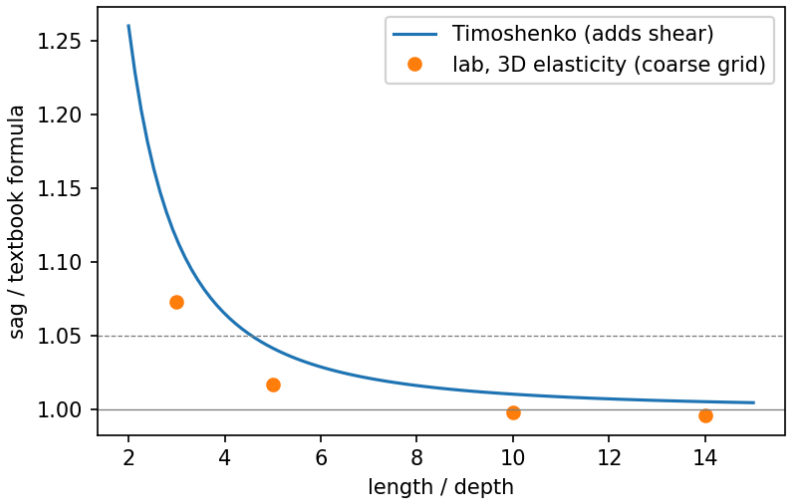
## What the starter does

- Stage 1: solve the beam equation E I w'''' = q numerically and check it against the formula. CHECK should say PASS.
- Then it compares both theories with the lab's 3D results that we measured when we tested this project.
- Stage 2: measure real rulers, fit the L^4 law, and find where the formula stops working.

## What you should see

Run `python3 09_cantilever_beam.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
beam equation: tip sag 0.12500, formula 0.12500, error 0.003%
CHECK numerical beam against q L^4 / (8 E I) -> PASS (limit 1%)
tip sag / textbook formula, by length / depth:
  L/h   Timoshenko   lab 3D (our test)
   14   1.005        0.996
   10   1.010        0.998
    5   1.042        1.017
    3   1.116        1.073
Timoshenko says the textbook formula is 5% low once L/h < 4.6; does the 3D solver agree?
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Rulers: clamp a ruler to a desk with 10, 15, 20, 25 and 30 cm hanging over, measure the tip sag, and save a CSV with columns length_cm,sag_mm. Run: python 09_cantilever_beam.py my_rulers.csv Predict the power first. Coins at the tip change it: why?
2. From your fit, work out E for the ruler (weigh it to get q). Compare with a published value for the material.

3. In the online lab, rerun the 3D beam at L/h = 3 and 5 with a finer grid. How much do the numbers above move? The clamped end has a stress singularity, so report how your answer depends on the grid.
4. Change nu to 0.05 and 0.4. Does the 5% threshold move? Check one value in the lab.

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Length / depth | 2 to 14 ratio | 10 |
| Poisson ratio | 0.05 to 0.4 | 0.3 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Linear elasticity, 3D)** | L/h 14: sag/formula 0.996 (1066 iterations, 47.2 s); L/h 10: sag/formula 0.998 (737 iterations, 22.7 s); L/h 5: sag/formula 1.017 (433 iterations, 8.8 s); L/h 3: sag/formula 1.073 (313 iterations, 5.5 s) |
| **Known answer** | with shear 1.005; with shear 1.010; with shear 1.042; with shear 1.116 |
| **Error** | 3.77% (limit 8.00%): passed |
| **A first finding** | the formula is close for long beams and increasingly wrong for short ones, which is the project's question |

## The real world

Clamp rulers or strips at different lengths to a desk, load them with coins, and measure the sag.

## The honest limit

The clamped end has a stress singularity, so results near it depend on the grid. Report grid sensitivity; this is a study and not a design tool.

How the full project checks itself: A slender beam's tip sag should approach $3\rho g L^4 / (2Eh^2)$. Our test runs give 0.996 of that at length/depth 14 and 1.073 at 3. The online lab runs the same project on the research solver (Linear elasticity, 3D).

PROJECT 10 · NUMERICAL ANALYSIS AND COMPUTING

# The race between solvers

*Make a simulation twice as sharp and it can take many times longer. Race the methods and find out which wins, and when.*

| FILE | MATH | YOU NEED |
|---|---|---|
| `10_solver_race.py` | Algebra II, logarithms; some programming helps | A laptop; some programming helps |

## The question

Doubling a simulation's resolution makes a basic solver far slower. How do accuracy, iteration count and run time scale, and at what size do higher-order methods, multigrid and GPUs actually pay off?

## Objectives

| | |
|---|---|
| **Weeks 1 to 3** | **Learn to trust the model.** Run 10_solver_race.py until its CHECK line says PASS, and explain every line of it to your mentor. Then reproduce a known answer: The solution has a known exact form, so you can measure the real error. Our test run gives rates of 2.00, 3.00 and 4.95 for orders 2, 4 and 6. Part of your project is explaining why the labelled order and the measured rate differ. Checkpoint: you reproduce a known answer and report how close you got. |
| **Weeks 4 and 5** | **Ask your own question.** Narrow the question above to one sentence that could turn out to be wrong, and write your prediction before you run anything. The first experiments below are a place to start. Checkpoint: a one-page plan with your question and prediction. |
| **Weeks 6 to 9** | **Run it, then measure the world.** Measure error and iterations as the grid grows for orders 2, 4 and 6, then race conjugate gradients against multigrid, and CPU against GPU, with the sbp-sat-suite tools. Then step away from the screen: time everything on real hardware and fit the power laws yourself. In our test, CG needed 127 to 416 iterations from 65 to 513 points while multigrid stayed near 15, and multigrid won on time only past about 257 points. Your machine's break-even point is your own result. Checkpoint: one chart comparing your model with reality. |
| **Weeks 10 to 12** | **Explain the gap, and share it.** Work out why your model and your measurements disagree, test at least one explanation, and write it up yourself. Say which numbers came from the starter and which from the lab. Keep the honest limit in view: the online lab covers conjugate gradients only. Multigrid and GPU runs use the sbp-sat-suite tools with a mentor. Checkpoint: a final paper, your code and a ten-minute talk. |

## The model in the starter

The test problem: -laplacian(u) = f on the unit square, u = 0 on the edge, with f chosen so the exact answer is u = g(x) g(y), g(s) = s (1 - s) e^s. Knowing the answer lets you measure the real error of every method. (The lab uses u = sin(pi x) sin(pi y). Try it here, in problem(): CG then finishes in one iteration.

Why?)

This file races three ways of solving the same grid equations (plain second-order finite differences): Jacobi: each point becomes the average the equation wants. Simple, and very slow on big grids. Conjugate gradients: the standard method for this kind of system; its iterations grow with the grid. Multigrid (V-cycle): fixes smooth errors on coarser grids; its iterations should stay flat.

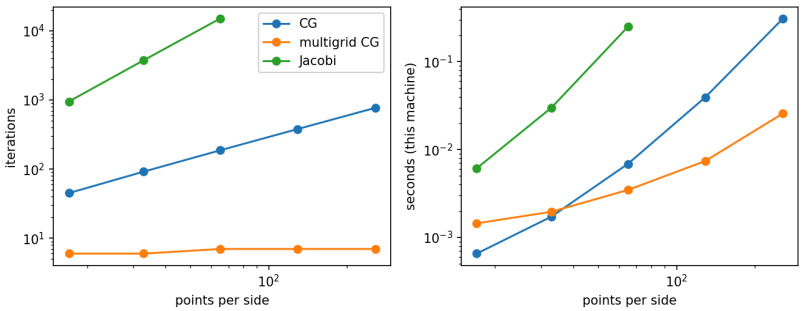
## What the starter does

- Stage 1: run it. It checks that the error falls like h^2 (rate near 2). The CHECK line should say PASS.
- Stage 2: time everything as the grid grows, fit the power laws, and find your own machine's break-even point. The online lab uses higher-order SBP-SAT methods; sbp-sat-suite has a mentor-run multigrid and GPU version.

## What you should see

Run `python3 10_solver_race.py`. This is the output from the run that built this guide. The numbers that depend on your computer's speed may differ; the CHECK line must say PASS.

```
points   method          iterations   seconds   max error
    17   CG                    45     0.001   3.62e-04
    17   multigrid CG           6     0.001   3.62e-04
    17   Jacobi               950     0.006   3.62e-04
    33   CG                    92     0.002   9.07e-05
    33   multigrid CG           6     0.002   9.07e-05
    33   Jacobi              3750     0.030   9.07e-05
    65   CG                   187     0.007   2.27e-05
    65   multigrid CG           7     0.003   2.27e-05
    65   Jacobi             15050     0.250   2.27e-05
   129   CG                   379     0.039   5.68e-06
   129   multigrid CG           7     0.007   5.68e-06
   257   CG                   768     0.310   1.42e-06
   257   multigrid CG           7     0.026   1.42e-06
measured convergence rate: 2.00 (second-order differences should give 2)
CHECK convergence rate near 2 -> PASS (limit 2 +/- 0.1)
```




## Your first experiments

From the YOUR TURN list at the bottom of the file. Write your prediction in your log *before* you run each one.

1. Predict first: if CG's iterations grow like n^p, what is p? Fit it from the table (polyfit on the logs). Do the same for time. Why does time grow faster than iterations?
2. Add 513 points. From what size does multigrid win on time here? Compiled code (sbp-sat-suite) shifts that point, because each method's cost per iteration changes. Your machine's break-even point is your own result.
3. In the online lab, run orders 2, 4 and 6 at 17, 33, 65 and 129 points and measure each order's rate. Order 4 measures about 3 in our tests. Why might the boundary be the reason?
4. With a mentor: race CPU against GPU in sbp-sat-suite. Report where your numbers differ from its README.

## The data

### *Settings in the online lab*

| Setting | Range | Starts at |
|---|---|---|
| Grid points per side | 9 to 129 | 33 |
| Accuracy order | 2 to 6 | 4 |

### *Checked on the research solver*

| | |
|---|---|
| **Solver (Poisson, 2D; sbp-sat-suite for multigrid and GPU)** | order 2: rate 2.00, iterations 29 > 52 > 91 > 154; order 4: rate 3.00, iterations 31 > 61 > 113 > 200; order 6: rate 4.95, iterations 36 > 77 > 145 > 265 |
| **Known answer** | error falls at least as fast as the order's boundary rate (order 2 gives about 2) |
| **Error** | 0.03% (limit 10.00%): passed |

## The real world

Time everything on real hardware and fit the power laws yourself. In our test, CG needed 127 to 416 iterations from 65 to 513 points while multigrid stayed near 15, and multigrid won on time only past about 257 points. Your machine's break-even point is your own result.

## The honest limit

The online lab covers conjugate gradients only. Multigrid and GPU runs use the sbp-sat-suite tools with a mentor.

How the full project checks itself: The solution has a known exact form, so you can measure the real error. Our test run gives rates of 2.00, 3.00 and 4.95 for orders 2, 4 and 6. Part of your project is explaining why the labelled order and the measured rate differ. The online lab runs the same project on the research solver (Poisson, 2D; sbp-sat-suite for multigrid and GPU).

REFERENCE

# When something goes wrong

| You see | Try this |
|---|---|
| `No module named numpy` | Run the install line again with the same Python you use to run the file: `python3 -m pip install -r requirements.txt` |
| Numbers become nan or huge | A time step is past its speed limit, usually after you made the grid finer. Each solver works out its step from the grid; check you did not fix it by hand. |
| `did not settle` | An iterative solver needs more sweeps for a finer grid. Raise its limit, or keep the grid smaller while you explore. |
| A CHECK line says FAIL after your change | Good: your change moved the model away from the case with a known answer. Undo it, confirm PASS, then change one thing at a time. |
| No window opens | Add `--show`, or open the PNG files in the `output` folder. |
| A recording gives strange notes | Record in a quiet room, save as .wav (not .m4a), trim silence at the start, and set the note you expect in the file. |

## How the starters relate to the online lab

The online lab, for members, runs each project on the research solver (AutoSBPSAT): high-order SBP-SAT methods, the full 3D elasticity for project 9, and frames of a time series. The starters use plain second-order methods instead, so every line is short enough to read. Both are checked against the same kind of exact answers. Use the starter to understand and to explore quickly; use the lab for results you will quote, and say in your paper which one produced each number.

## How these starters were checked

Every number in "What you should see" came from running the starter while this guide was built, on Python 3.9.23, numpy 1.24.4, matplotlib 3.5.2. `check_all.py` also feeds each optional data reader (the soil-station CSV, the ruler CSV and two .wav recordings) a made-up file with a known answer, and confirms the answer comes back. They have not yet been used by a student. The twelve-week plan and the time estimates in it are a design, to be revised after the first group: tell your mentor what was unclear.