Melbourne Bioinformatics Seminar

Why functional programming matters (Python edition)

Bernie Pope
Lead Bioinformatician, Melbourne Bioinformatics
Associate Director (Human Genome Informatics), Australian BioCommons
The University of Melbourne, Australia

code available:
https://github.com/bjpop/python_image_scalar_field

Functional programming

Key defining characteristic of FP:

Functions are *first class citizens*

- They can be stored in data structures
- They can be passed as arguments and returned as results from other functions

Higher-order functions

A higher-order function may:

- Receive function value(s) as arguments
- Return function value(s) as results

```
>>> list(map(lambda x: x+1, [1,2,3]))
[2, 3, 4]
```

Why functional programming matters?

J. Hughes, Why Functional Programming Matters, *The Computer Journal*, Volume 32, Issue 2, 1989, Pages 98–107, https://doi.org/10.1093/comjnl/32.2.98

Higher-order functions provide great opportunities for abstraction and composition

It encourages new ways of thinking about programming, problem solving, and computing

Demonstration by example

In this talk I will demonstrate functional programming by way of an example.

I'm using Python as the implementation language, but any language with higher-order functions will suffice.

A nice side-benefit is that it allows exploration of some advanced features of Python.

Digital images (2D graphics)

We are accustomed to digital images being represented by rectangular grids of pixels.

Most graphics programming techniques adopt this representation.

This *rasterisation* of images reflects a fairly low-level hardware oriented perspective, e.g. digital cameras, displays *etcetera*.

Digital images (2D graphics)

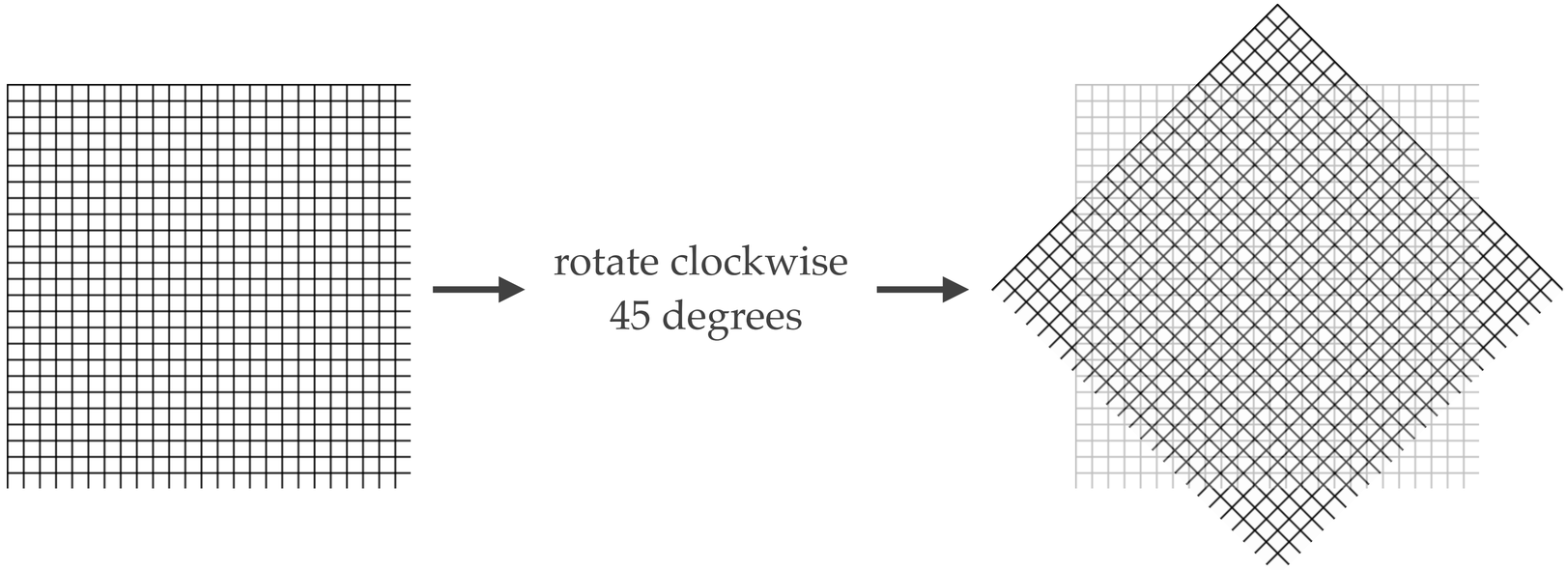
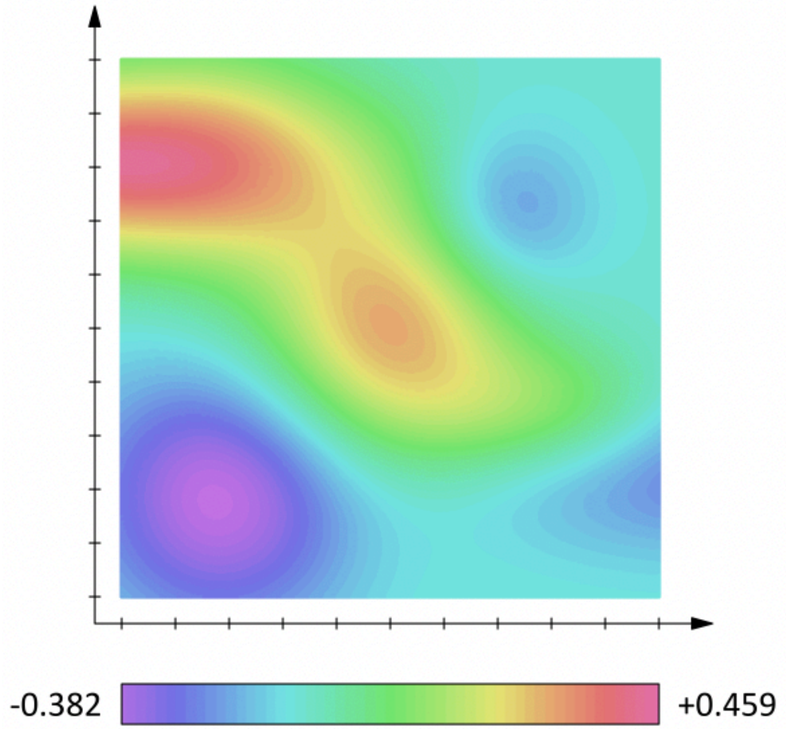
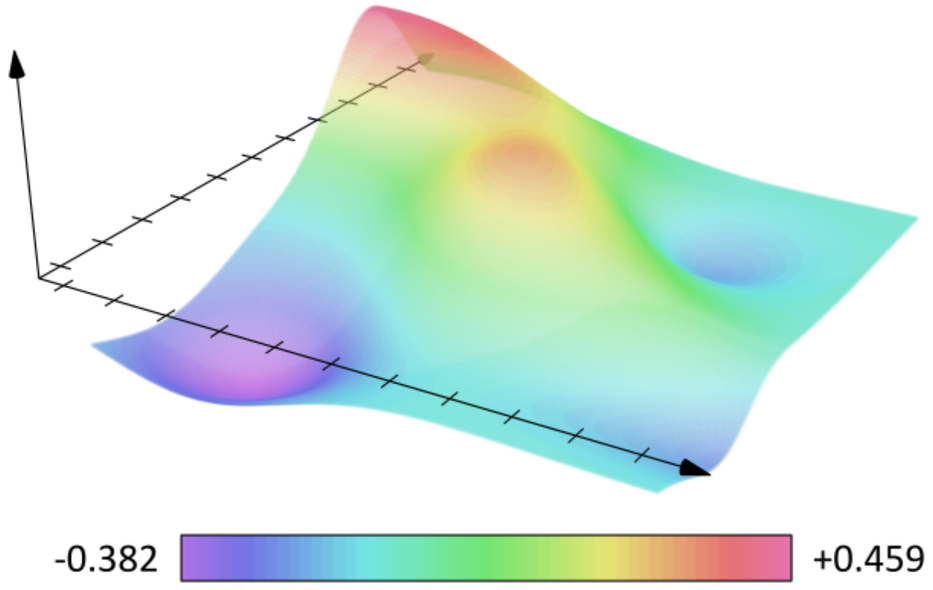


Image transformations, such as rotation and scaling are complicated in the rasterised view of graphics.

Scalar fields





Images from Wikipedia

Scalar fields

We can represent a scalar field as a function with type

$$(\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$$

where

$\mathbb{R}$ is the set of *real numbers*

$\mathbb{R} \times \mathbb{R}$ is the cartesian plane, sometimes denoted $\mathbb{R}^2$

Images as scalar fields

Greyscale images can be represented as scalar fields, where the output is the intensity of the image at some point.

Colour images follow the same idea, but the output would have more dimensions, e.g. hue, saturation, value.

If colour is a vector, then the concept of an image can be generalised to a *vector field*.

For simplicity, we'll stick to greyscale images in this talk.

Greyscale images as scalar fields

For greyscale images we define the output of the function be the intensity of the image.

Without loss of generality we will restrict intensities to be within the range [0,1].

Thus:

$$\text{image} : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}_{[0,1]}$$

NB: when converting back to an output image format we will scale the intensity to the maximum pixel intensity. For 8-bit intensity images, this is 255.

Greyscale images as scalar fields

Some things to note about this representation:

- The cartesian plane is *infinite* and *continuous*
- Real numbers are *uncountably* infinite and cannot be represented precisely on digital computers
- We will use double-precision floating point numbers as an approximation to the reals

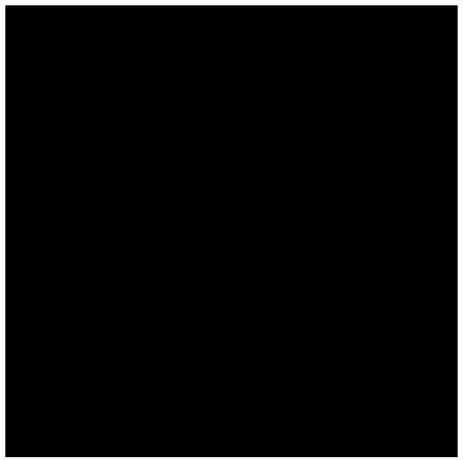
Thus, in our implementation:

```
image : (float, float) → float
```

Or in Python's **ugly** type hinting notation:

```
image = Callable[[Tuple[float, float]], float]
```

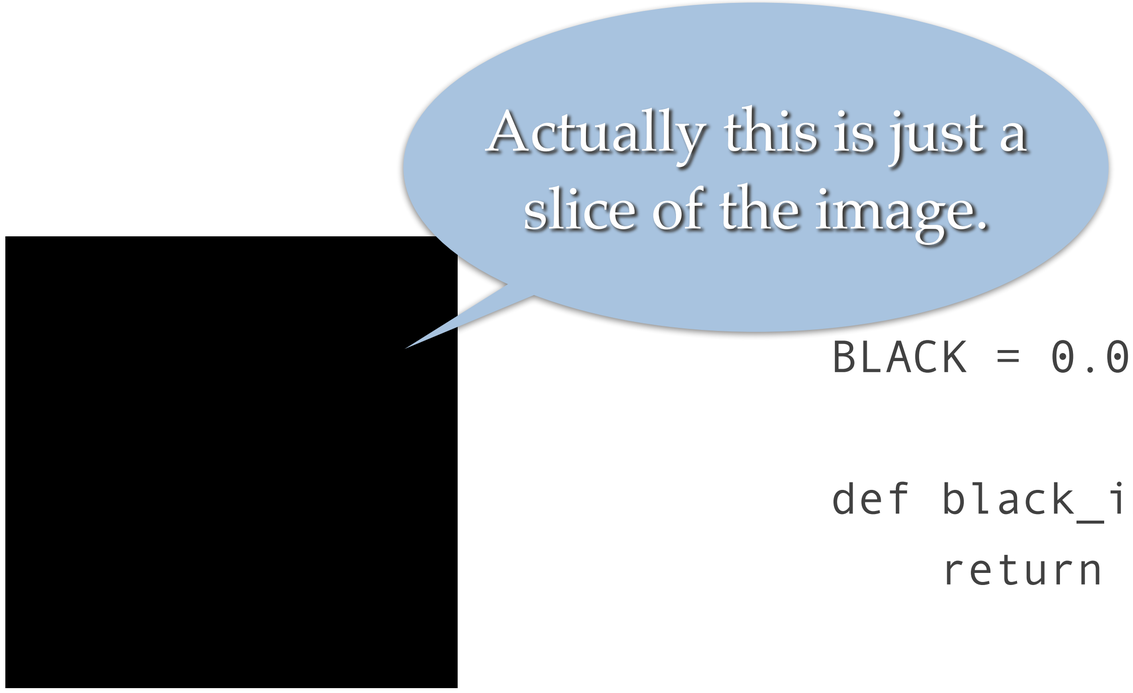
A constant image



```
BLACK = 0.0

def black_image(pos):
    return BLACK
```

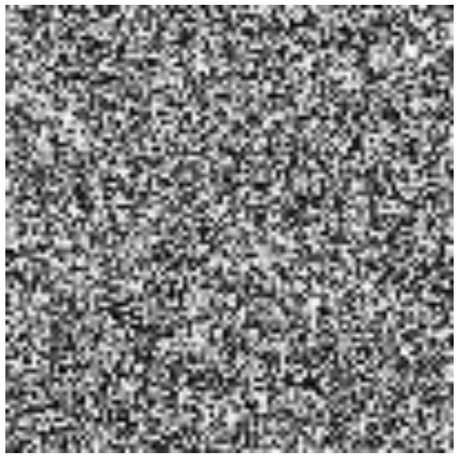
A constant image




```
BLACK = 0.0

def black_image(pos):
    return BLACK
```

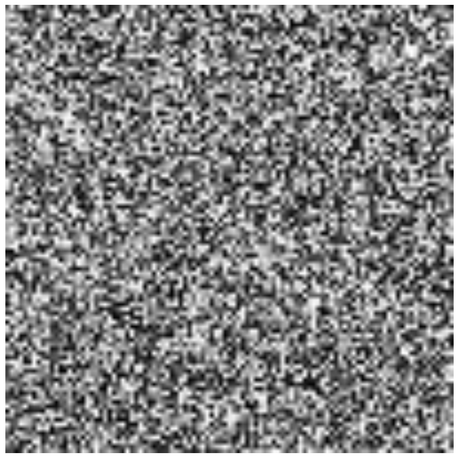
A random image



```
import random

def random_image(pos):
    return random.uniform(0, 1)
```

A random image

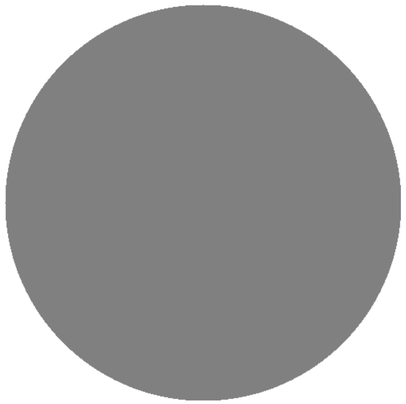


```
import random

def random_image(pos):
    return random.uniform(0, 1)
```

not a pure function!

A circle

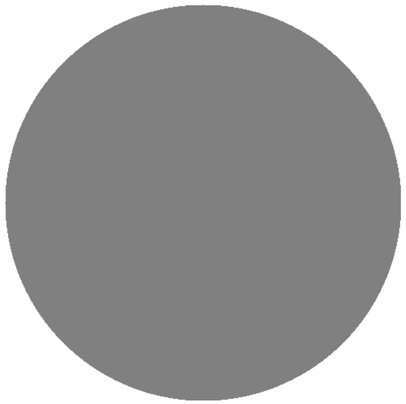


```
WHITE = 1.0

def circle(radius, centre, intensity):
    def image(pos):
        if distance(centre, pos) <= radius:
            return intensity
        else:
            return WHITE
    return image

circle_image = circle(1, (0, 0), 0.5)
```

A circle



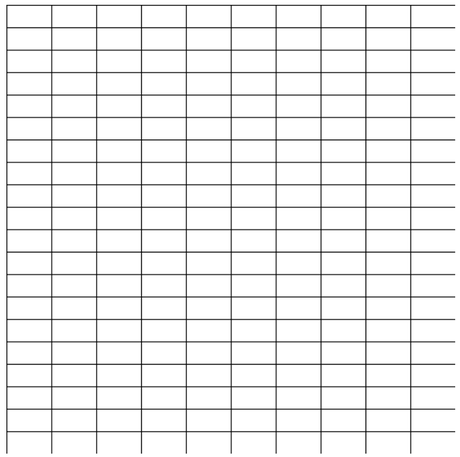
```
WHITE = 1.0

def circle(radius, centre, intensity):
    def image(pos):
        if distance(centre, pos) <= radius:
            return intensity
        else:
            return WHITE
    return image

circle_image = circle(1, (0, 0), 0.5)
```

Note the nested function definition. This is called a “closure”.

A grid



```
def grid(width, height):
    def image(pos):
        x, y = pos
        if x % width < 1 or y % height < 1:
            return BLACK
        else:
            return WHITE
    return image

grid_image = grid(50, 25)
```

Rasterisation

We need to turn an image into a rasterised rectangular pixel grid in order to save/display it.

This will require sampling the scalar field within a given sub-plane and resolution.

We may also want to read from a rasterised file and convert it into a scalar field image.

Convert image to raster

```
def to_raster(top_left, bot_right, x_res, y_res, image):
    grid = [[0 for _ in range(x_res)] for _ in range(y_res)]
    top_left_x, top_left_y = top_left
    bot_right_x, bot_right_y = bot_right
    y_step = (top_left_y - bot_right_y) / (y_res + 1)
    x_step = (bot_right_x - top_left_x) / (x_res + 1)
    for row in range(y_res):
        y = top_left_y - ((row + 1) * y_step)
        for col in range(x_res):
            x = top_left_x + ((col + 1) * x_step)
            pixel = round(image((x, y)) * MAX_INTENSITY)
            grid[row][col] = pixel
    return grid
```

Convert image to raster

```
def to_raster(top_left, bot_right, x_res, y_res, image):
    grid = [[0 for _ in range(x_res)] for _ in range(y_res)]
    top_left_x, top_left_y = top_left
    bot_right_x, bot_right_y = bot_right
    y_step = (top_left_y - bot_right_y) / (y_res
    x_step = (bot_right_x - top_left_x) / (x_res
    for row in range(y_res):
        y = top_left_y - ((row + 1) * y_step)
        for col in range(x_res):
            x = top_left_x + ((col + 1) * x_step)
            pixel = round(image((x, y)) * MAX_INTENSITY)
            grid[row][col] = pixel
    return grid
```

image is a scalar field (a function). We sample it by calling the function at specific coordinates

Convert image to raster

```
def to_raster(top_left, bot_right, x_res, y_res, image):
    grid = [[0 for _ in range(x_res)] for _ in range(y_res)]
    top_left_x, top_left_y = top_left
    bot_right_x, bot_right_y = bot_right
    y_step = (top_left_y - bot_right_y) / (y_res +
    x_step = (bot_right_x - top_left_x) / (x_res
    for row in range(y_res):
        y = top_left_y - ((row + 1) * y_step)
        for col in range(x_res):
            x = top_left_x + ((col + 1) * x_step)
            pixel = round(image((x, y)) * MAX_INTENSITY)
            grid[row][col] = pixel
    return grid
```

this is the only place where we have a nested loop to walk over pixels

Convert image to raster

```
# assuming write_image takes a filename and a rasterised
# 2D list of intensities and writes it to an image file

circle_raster = to_raster((-1, 1), (1, -1), 500, 500, circle_image)
write_image("circle.png", circle_raster)

grid_raster = to_raster((-1, 1), (1, -1), 500, 500, grid_image)
write_image("grid.png", grid_raster)
```

Convert raster to image

```
def from_raster(width, height, raster):
    def image(pos):
        x, y = pos
        x = trunc(x)
        y = trunc(y)
        x = x % width
        y = y % height
        col = x
        row = (height - y) - 1
        return raster[row][col] / MAX_INTENSITY
    return image
```

Convert raster to image

```
def from_raster(width, height, raster):
    def image(pos):
        x, y = pos
        x = trunc(x)
        y = trunc(y)
        x = x % width
        y = y % height
        col = x
        row = (height - y) - 1
        return raster[row][col] / MAX_INTENSITY
    return image
```

modulus means that the image is infinitely tiled

Convert raster to image

```
# assuming read_image takes a filename and returns a
# 2D list of intensities from the rasterised file

in_raster = read_image("images/coffee.jpg")
width = get_width(in_raster)
height = get_height(in_raster)

image = from_raster(width, height, in_raster)

out_raster = to_raster((0, height * 4), (width * 4, 0), width * 2, height * 2, image)
write_image("tiled_coffee.png", out_raster)
```

Convert raster to image

input raster

output raster

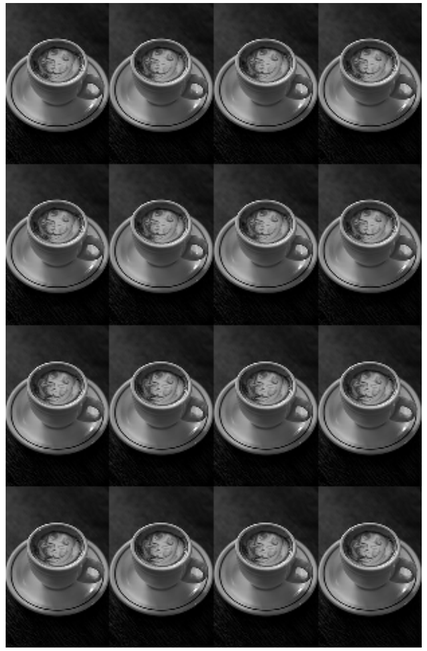
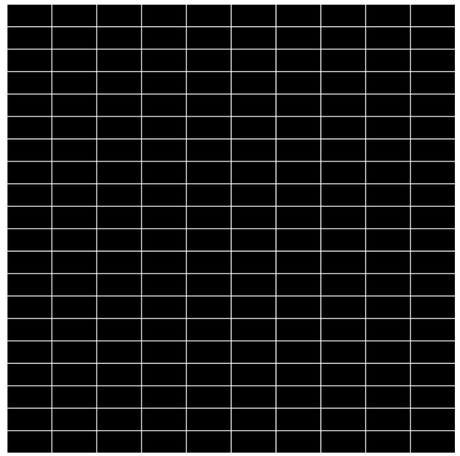


Image transformation

An image transformation is a function that maps an input image to an output image.

```
transform : image → image
```

Invert



```
def invert(in_image):
    def out_image(pos):
        return WHITE - image(pos)
    return out_image

inverted_grid = invert(grid_image)
```

Translate, scale (at origin)

```
def translate(x_delta, y_delta, in_image):
    def out_image(pos):
        x, y = pos
        return in_image((x - x_delta, y - y_delta))
    return out_image

def scale_at_origin(x_scale, y_scale, in_image):
    def out_image(pos):
        x, y = pos
        return in_image((x / x_scale, y / y_scale))
    return out_image
```

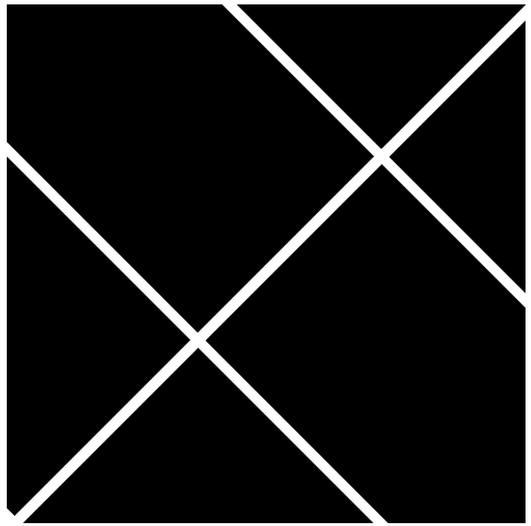
Rotate (at origin)

```
def rotate_at_origin(angle_degrees, in_image):
    angle_radians = radians(angle_degrees)
    def out_image(pos):
        x, y = pos
        rot_x = x * cos(angle_radians) - y * sin(angle_radians)
        rot_y = x * sin(angle_radians) + y * cos(angle_radians)
        return in_image((rot_x, rot_y))
    return out_image
```

Homework

Generalise the scale and rotate transformations to work at arbitrary centre points, rather than at the origin.

Image transformation



```
image = scale_at_origin(10, 10,
                        rotate_at_origin(45,
                        invert(grid(50, 25))))

raster = to_raster((0, 500), (500, 0), 500, 500, image)
write_image("grid.png", raster)
```

Ripple

```
def ripple(in_image):
    def out_image(pos):
        x, y = pos
        distance = sqrt(x * x + y * y)
        scale = cos(radians(distance)) * 20
        if x <= 0:
            new_x = x - scale
        else:
            new_x = x + scale
        if y <= 0:
            new_y = y - scale
        else:
            new_y = y + scale
        return image((new_x, new_y))
    return out_image
```

Ripple

```
in_raster = read_image("images/coffee.jpg")
width = get_width(in_raster)
height = get_height(in_raster)

image = ripple(from_raster(width, height, in_raster))

out_raster = to_raster((0, height * 4), (width * 4, 0), width * 2, height * 2, image)
write_image("rippled_coffee.png", out_raster)
```

Ripple

Fractal: Mandelbrot set

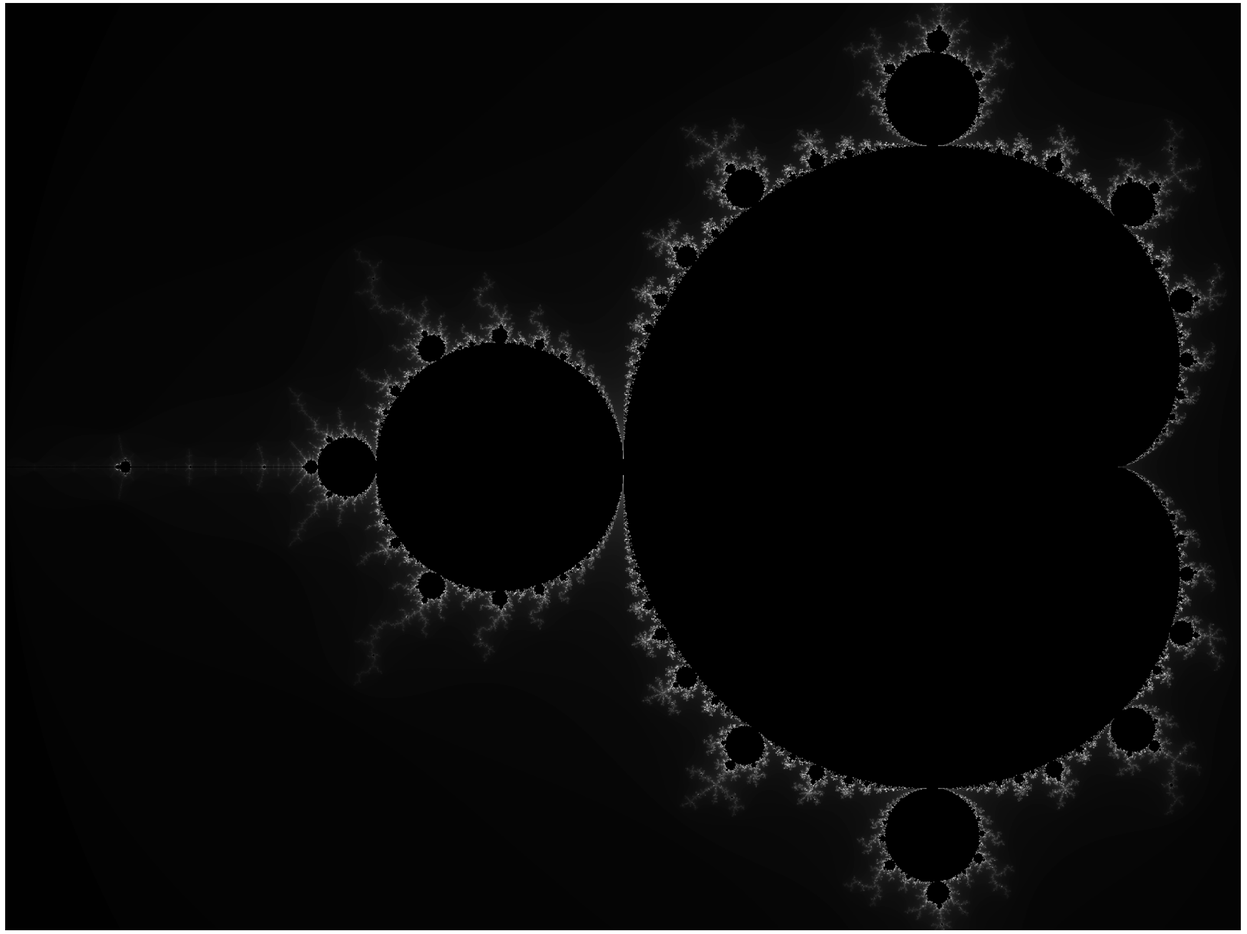
```
def mandelbrot(pos):
    MAX_ITER = 200
    x, y = pos
    z = complex(0.0, 0.0)
    complex_point = complex(x, y)
    for iteration_number in range(MAX_ITER):
        if abs(z) >= 2.0:
            return iteration_number / MAX_ITER
        else:
            z = z * z + complex_point
    return BLACK
```

Fractal: Mandelbrot set

```
# centre the image on the “seahorse valley” of Mandelbrot

raster = to_raster((-2.0, 0.9375), (0.5, -0.9375), 1600, 1200, mandelbrot)
write_image("mandel.png", raster)
```

Fractal: Mandelbrot set



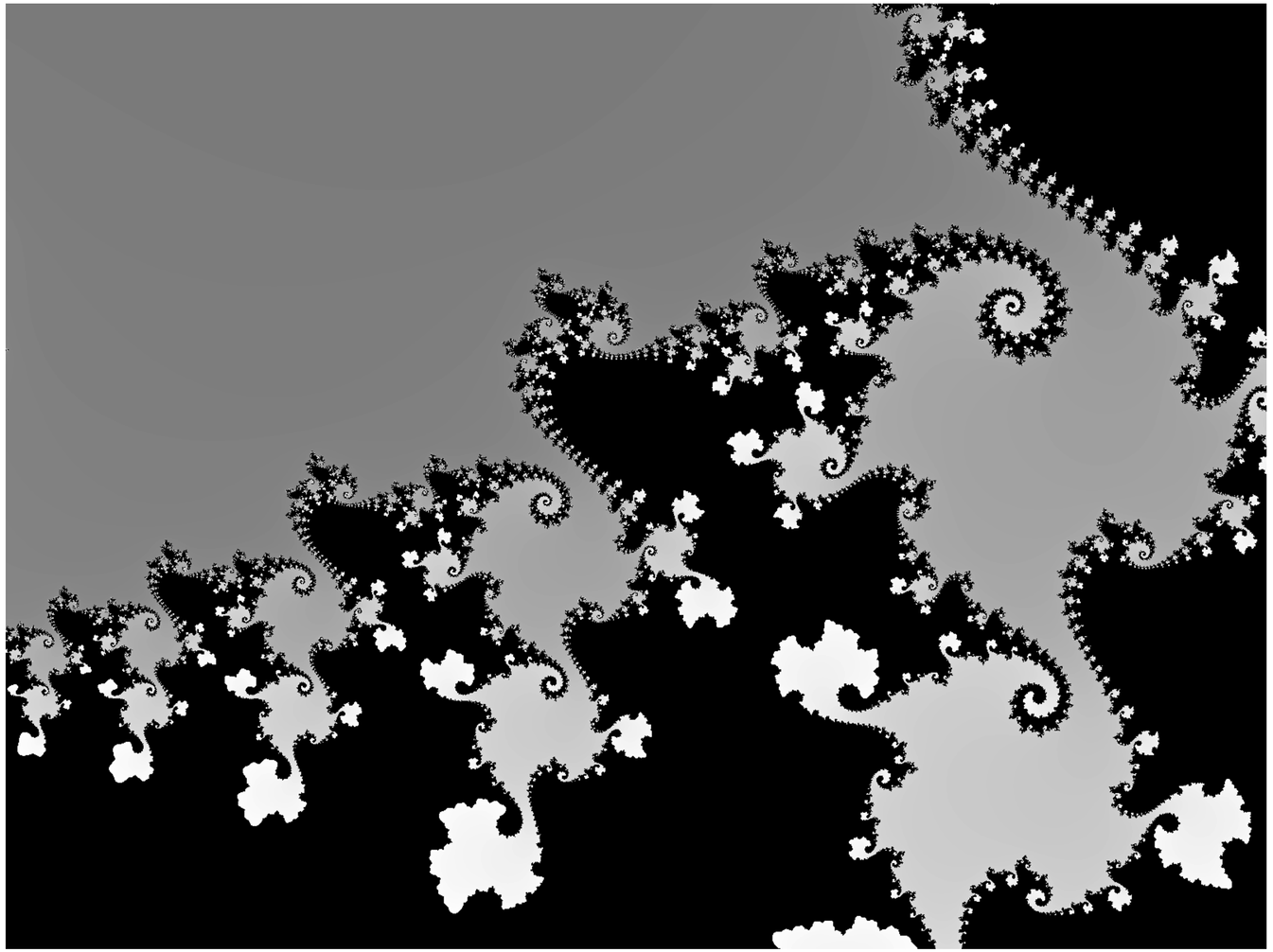
Fractal: Mandelbrot set

```
# zoom in 10,000 times to the “seahorse valley”

image = scale_at_point((-0.74519683,  0.10186988), 10000, 10000, mandelbrot)

raster = to_raster((-2.0, 0.9375), (0.5, -0.9375), 1600, 1200, image)
write_image("mandel_zoom.png", raster)
```

Fractal: Mandelbrot set



Parallelism

Every point in the scalar field can be evaluated independently from the rest.

This lends itself to straightforward parallelisation of the `to_raster` function, where all the work happens.

Since October 2024 Python has supported “free threading”, which avoids the infamous Global Interpreter Lock (GIL).

Lock free multithreading

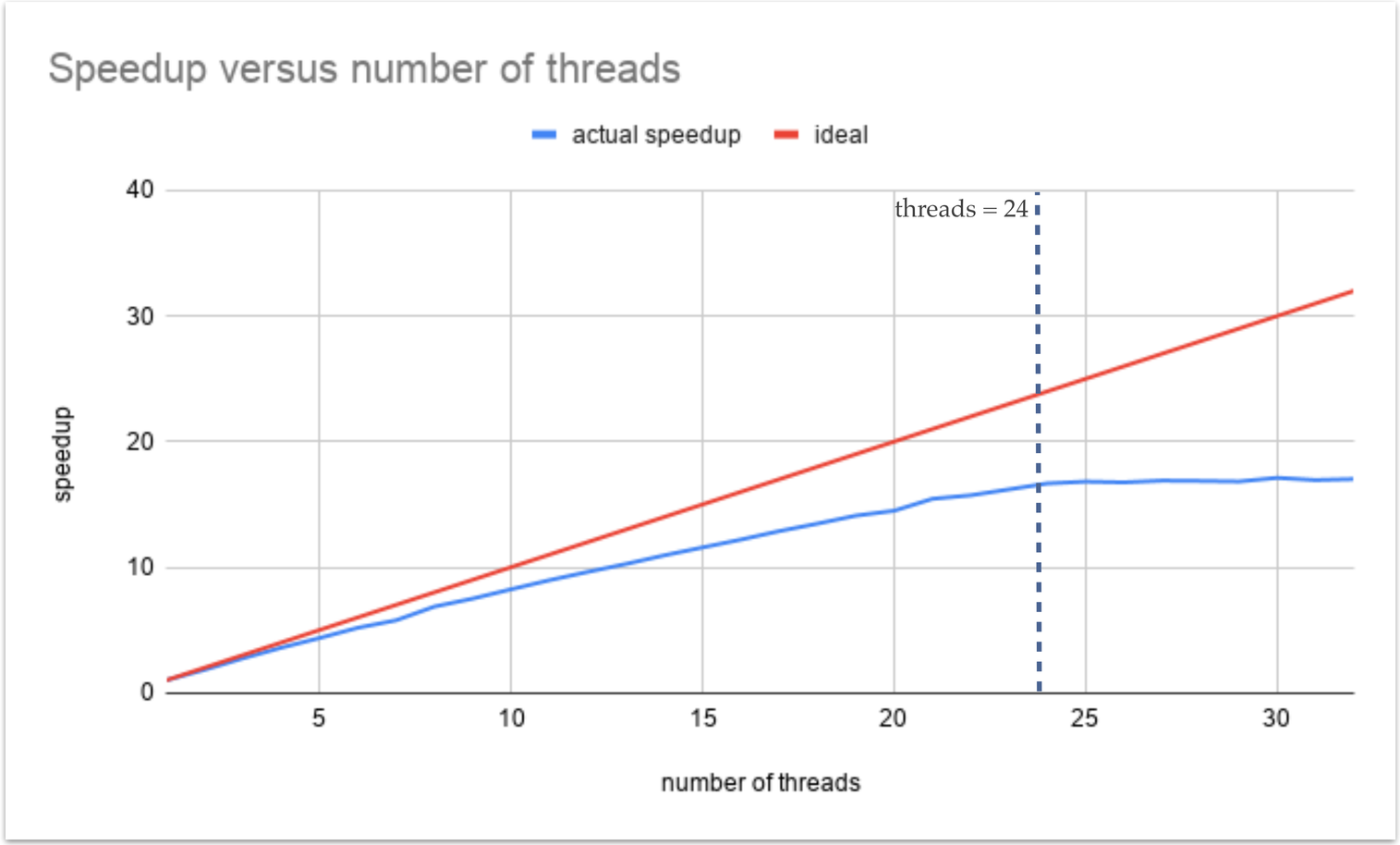
```
from concurrent.futures import ThreadPoolExecutor

def to_raster(top_left, bot_right, x_res, y_res, threads, image):
    grid = [[0 for _ in range(x_res)] for _ in range(y_res)]
    top_left_x, top_left_y = top_left
    bot_right_x, bot_right_y = bot_right
    y_step = (top_left_y - bot_right_y) / (y_res + 1)
    x_step = (bot_right_x - top_left_x) / (x_res + 1)
    def to_raster_row(row):
        y = top_left_y - ((row + 1) * y_step)
        for col in range(x_res):
            x = top_left_x + ((col + 1) * x_step)
            pixel = round(image((x, y)) * MAX_INTENSITY)
            grid[row][col] = pixel
    with ThreadPoolExecutor(max_workers=threads) as executor:
        executor.map(to_raster_row, range(y_res))
    return grid
```

Each thread rasterises a separate row of the image in parallel.

We only have to change the highlighted parts to get good parallelisation!

Speed up on 24 core CPU



Test environment:

CPU: AMD Ryzen Threadripper 3960X 24-Core

RAM: 128GB

O/S: Pop!_OS 22.04 LTS. Linux Kernel: 7.0.11

Python: 3.14.6 (free-threading build), installed via conda 23.11.0.

```
image = scale_at_point((-0.74519683,  0.10186988), 10000, 10000, mandelbrot)
raster = to_raster((-2.0, 0.9375), (0.5, -0.9375), 1600, 1200, THREADS, image)
write_image("mandel_zoom.png", raster)
```

Conclusion

Is this a practical way to program graphics?

Or, how does this differ from state of the art graphics programming approaches?

We've only scratched the surface of FP in this talk.

Missing things:

- Type inference / type checking
- Pure functions, first class effects
- Currying (implemented in my code, not shown here)
- Algebraic data types and pattern matching
- Non-strict evaluation
- Fancy abstractions, such as Functors and Monads