MAT 305: Mathematical Computing

Cython

John Perry

University of Southern Mississippi

Spring 2016

Outline

Outline

An iterative function

Let $c \in \mathbb{C}$ and $f(z) = z^2 + c$.

What happens to $|f(0)|$, $|f(f(0))|$, $|f(f(f(0)))|$, ...?

An iterative function

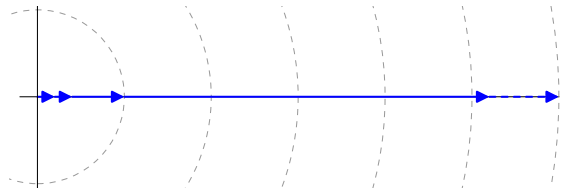
Let $c \in \mathbb{C}$ and $f(z) = z^2 + c$.

What happens to $|f(0)|, |f(f(0))|, |f(f(f(0)))|, \ldots$?

Example (can grow without bound...)

Suppose $c = 1$:

```
sage:  f(z) = z^2 + 1
sage:  a = 0
sage:  for each in xrange(10):
           a = f(a)
           print abs(a),
1 2 5 26 677 458330 210066388901
44127887745906175987802
1947270476915296449559703445493848930452791205
37918623102659260828682350280278932773702331522473885847617341507
```



An iterative function

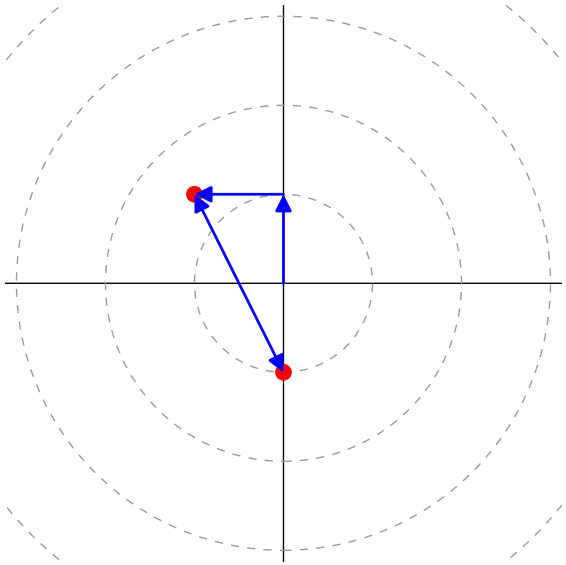
Let $c \in \mathbb{C}$ and $f(z) = z^2 + c$.

What happens to $|f(0)|, |f(f(0))|, |f(f(f(0)))|, \ldots$?

Example (. . . but not always)

Suppose $c = i$:

```
sage:  f(z) = z^2 + I
sage:  a = 0
sage:  for each in xrange(10):
           a = f(a)
           print abs(a),
1 sqrt(2) 1 sqrt(2) 1 sqrt(2) 1
sqrt(2) 1 sqrt(2)
```



Mandelbrot Numbers

Let $c \in \mathbb{C}$ and $f(z) = z^2 + c$.

Definition

The **Mandelbrot number** of c is smallest integer such that

$$\left| \underbrace{f(f(\cdots(f(0))))}_{\text{applied } n \text{ times}} \right| > 4.$$

- Notation: $\mu(c)$
- $\mu(c) = \infty$? c in **Mandelbrot set**

Examples

Example

$\mu(1) = 3$

$$f_1(0) = 0^2 + 1 = 1$$
$$f_1^2(0) = f_1(1) = 1^2 + 1 = 2$$
$$f_1^3(0) = f_1(2) = 2^2 + 1 = 5.$$

Examples

Example

$\mu(1) = 3$

$$f_1(0) = 0^2 + 1 = 1$$
$$f_1^2(0) = f_1(1) = 1^2 + 1 = 2$$
$$f_1^3(0) = f_1(2) = 2^2 + 1 = 5.$$

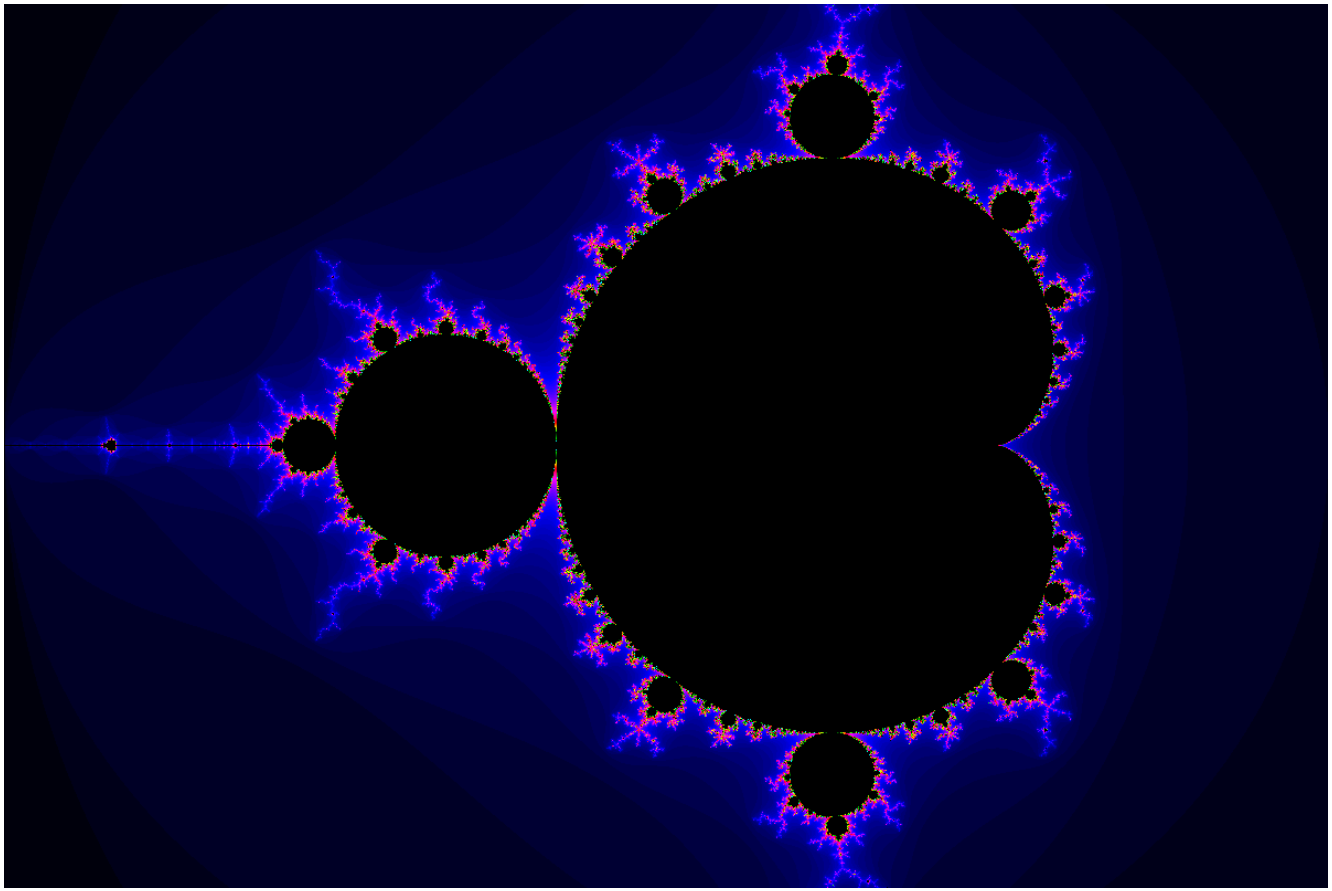
Example

$\mu(i) = \infty$

$$f_i(0) = 0^2 + i$$
$$f_i^2(0) = f_i(i) = i^2 + i = -1 + i$$
$$f_i^3(0) = f_i(i-1) = (i-1)^2 + i = -i$$
$$f_i^4(0) = f_i(-i) = (-i)^2 + i = -1 + i$$

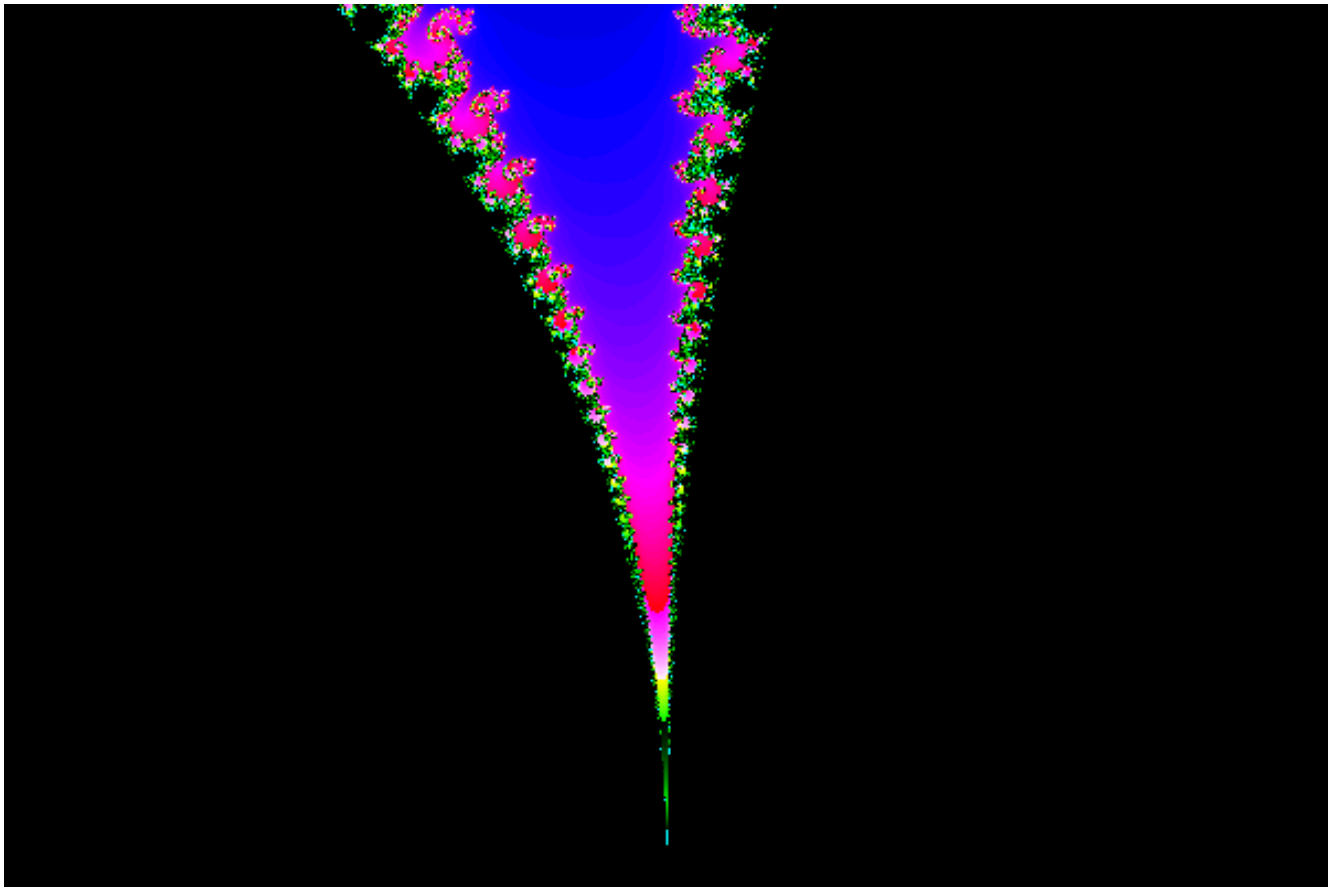
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



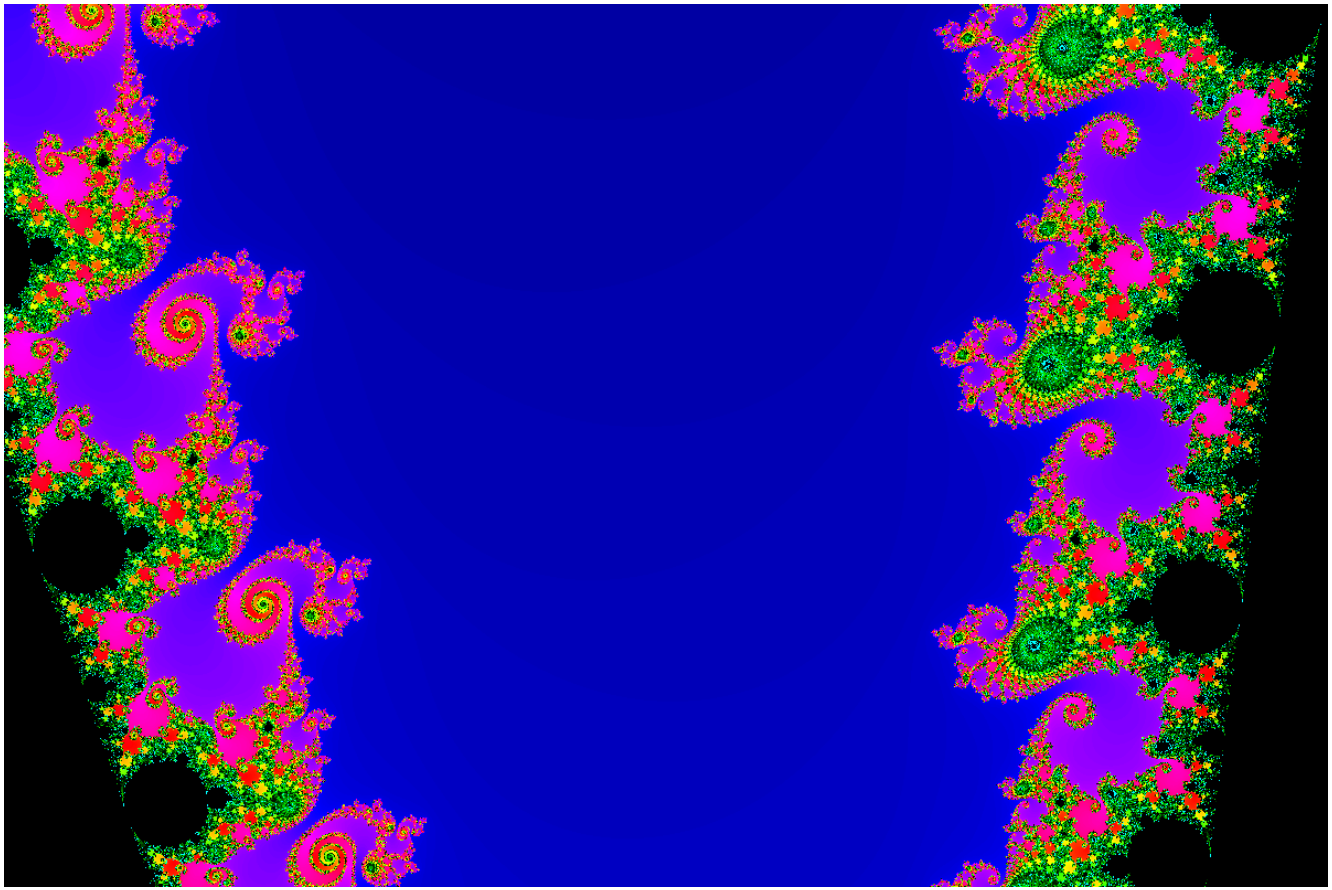
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



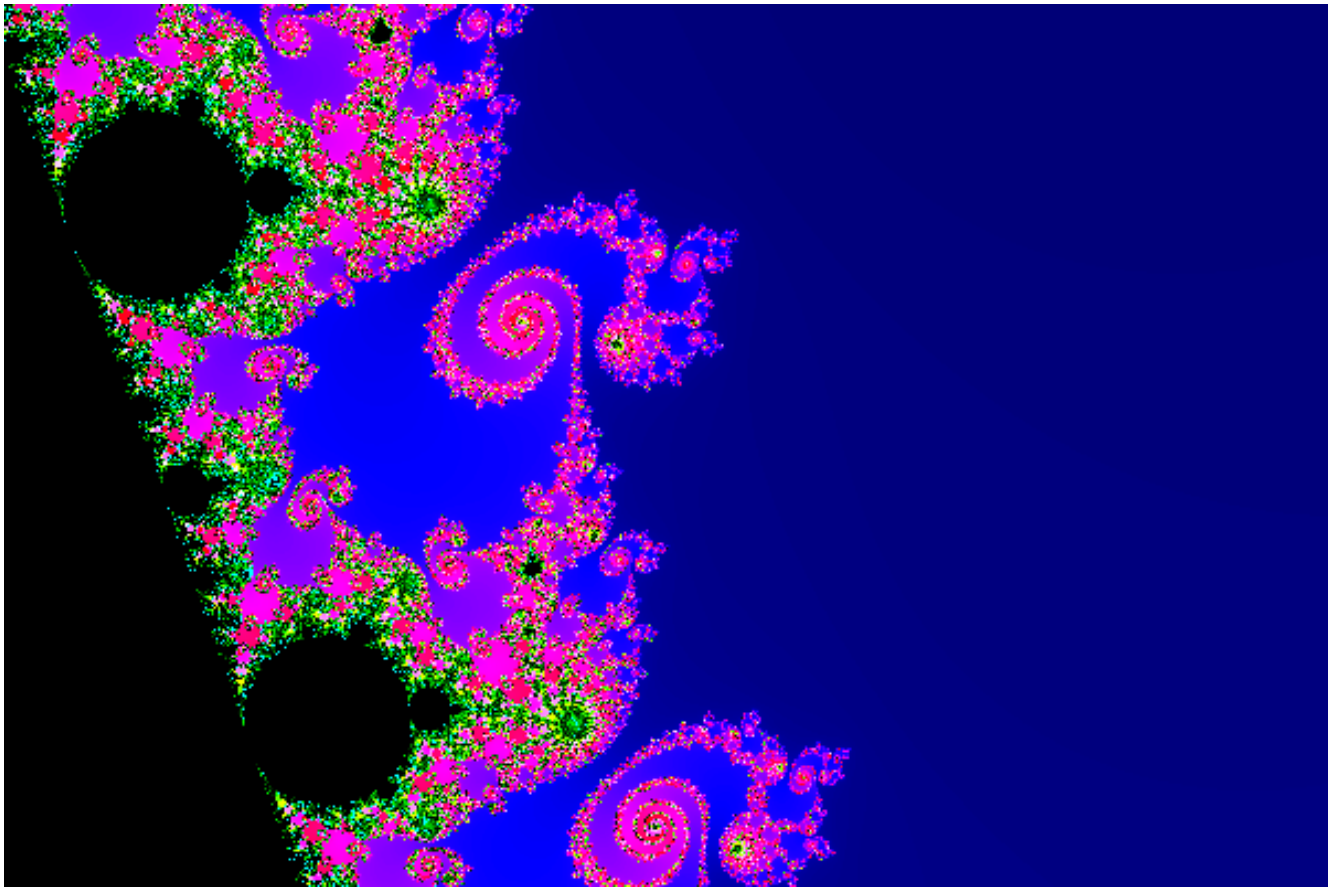
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



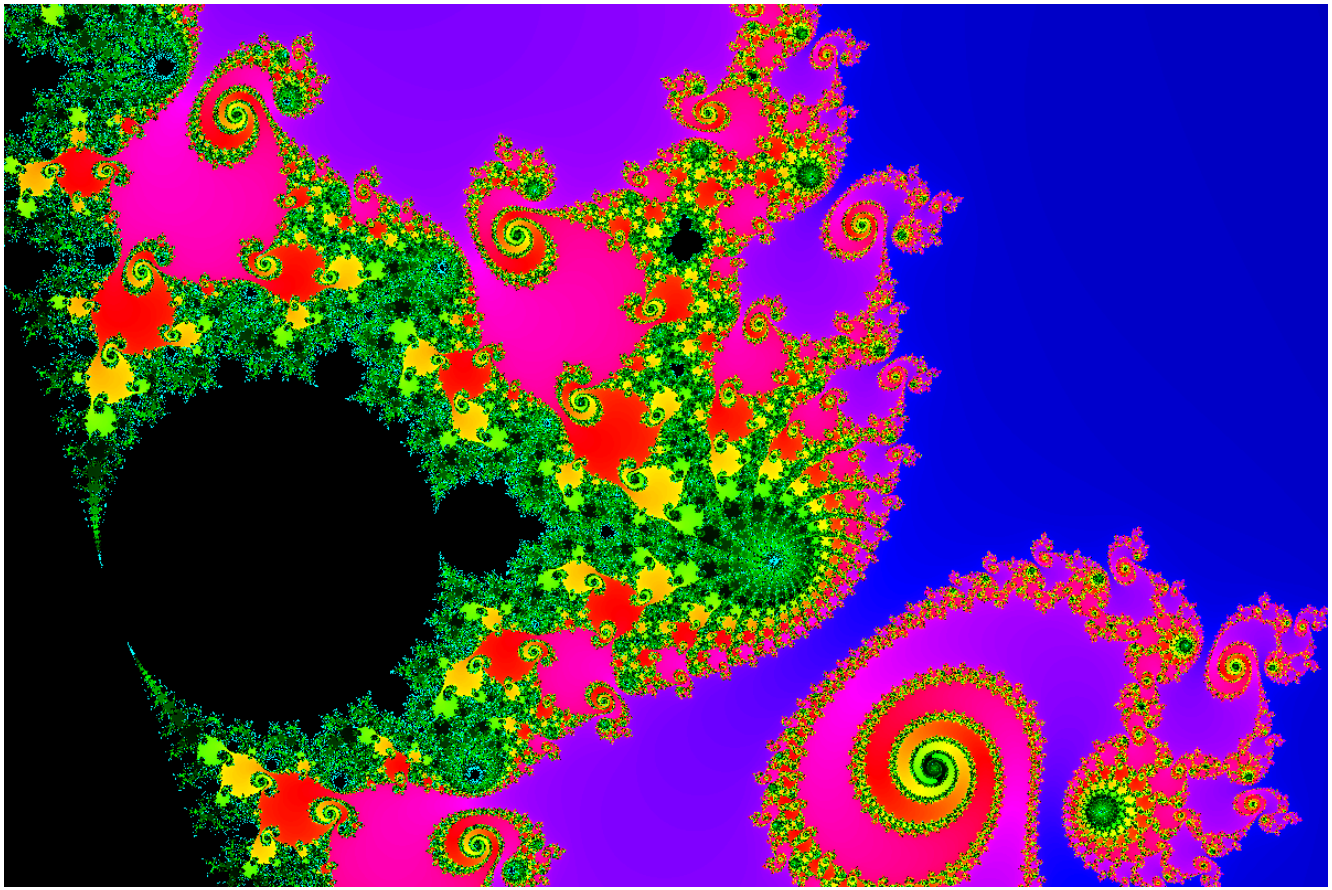
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



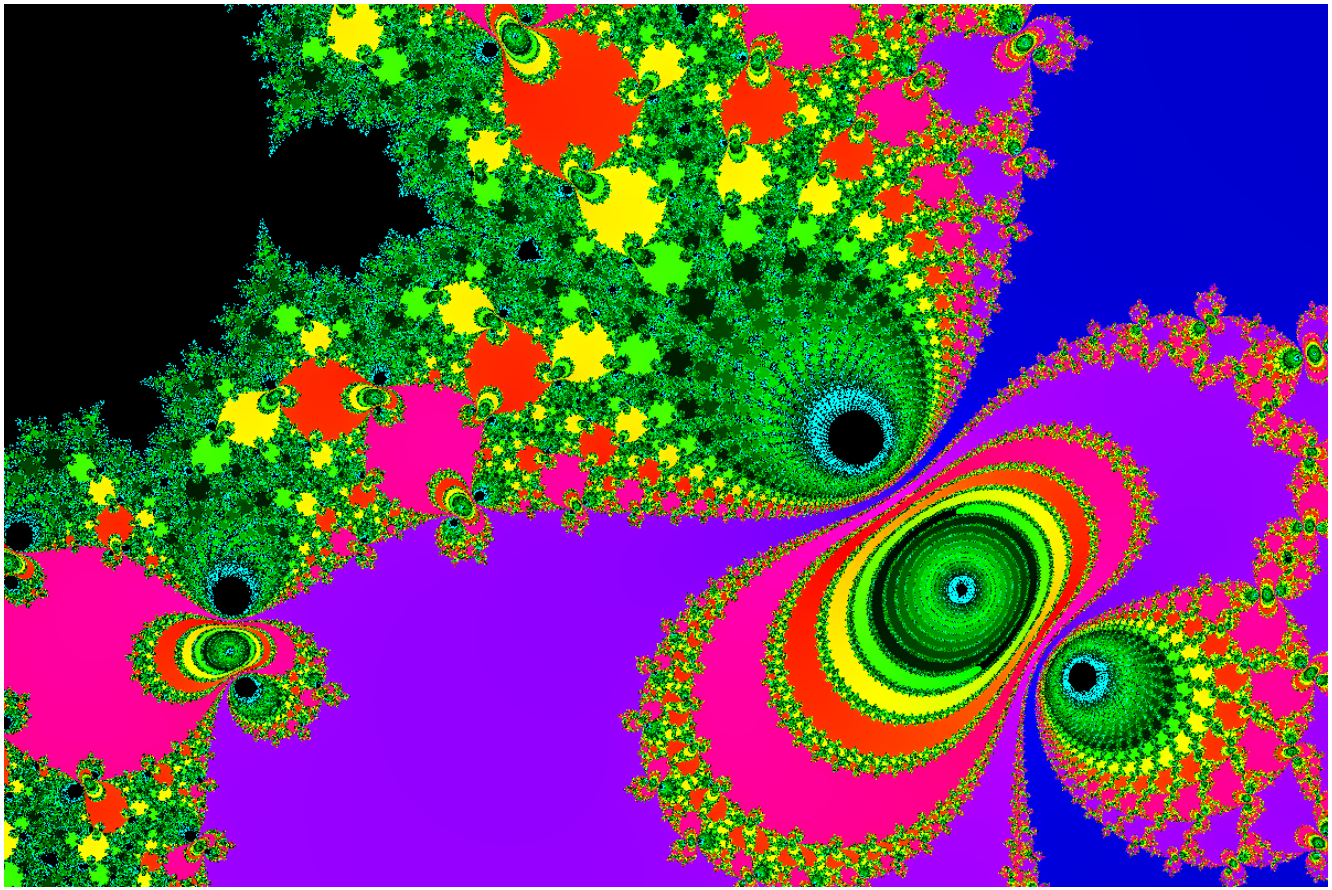
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



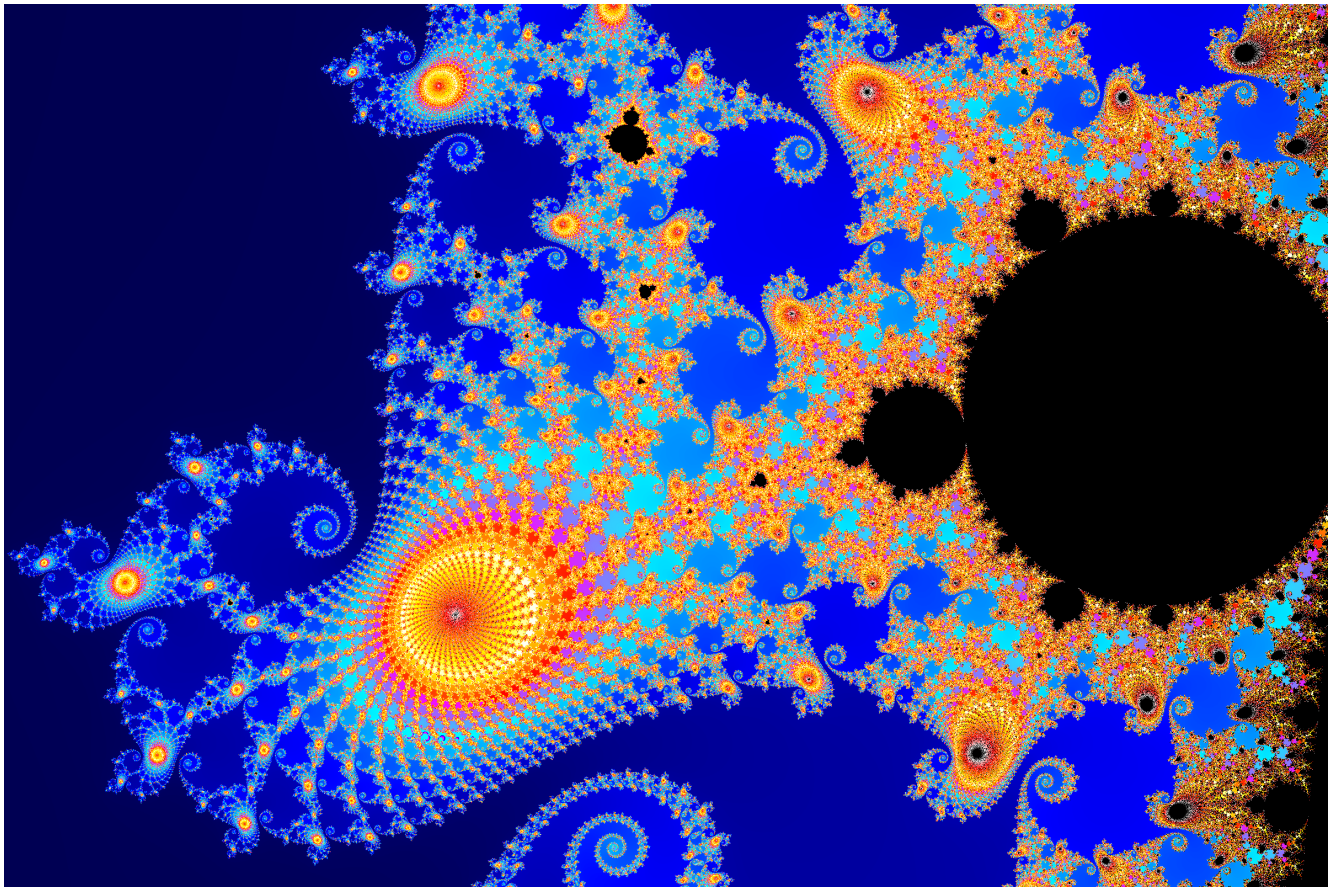
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



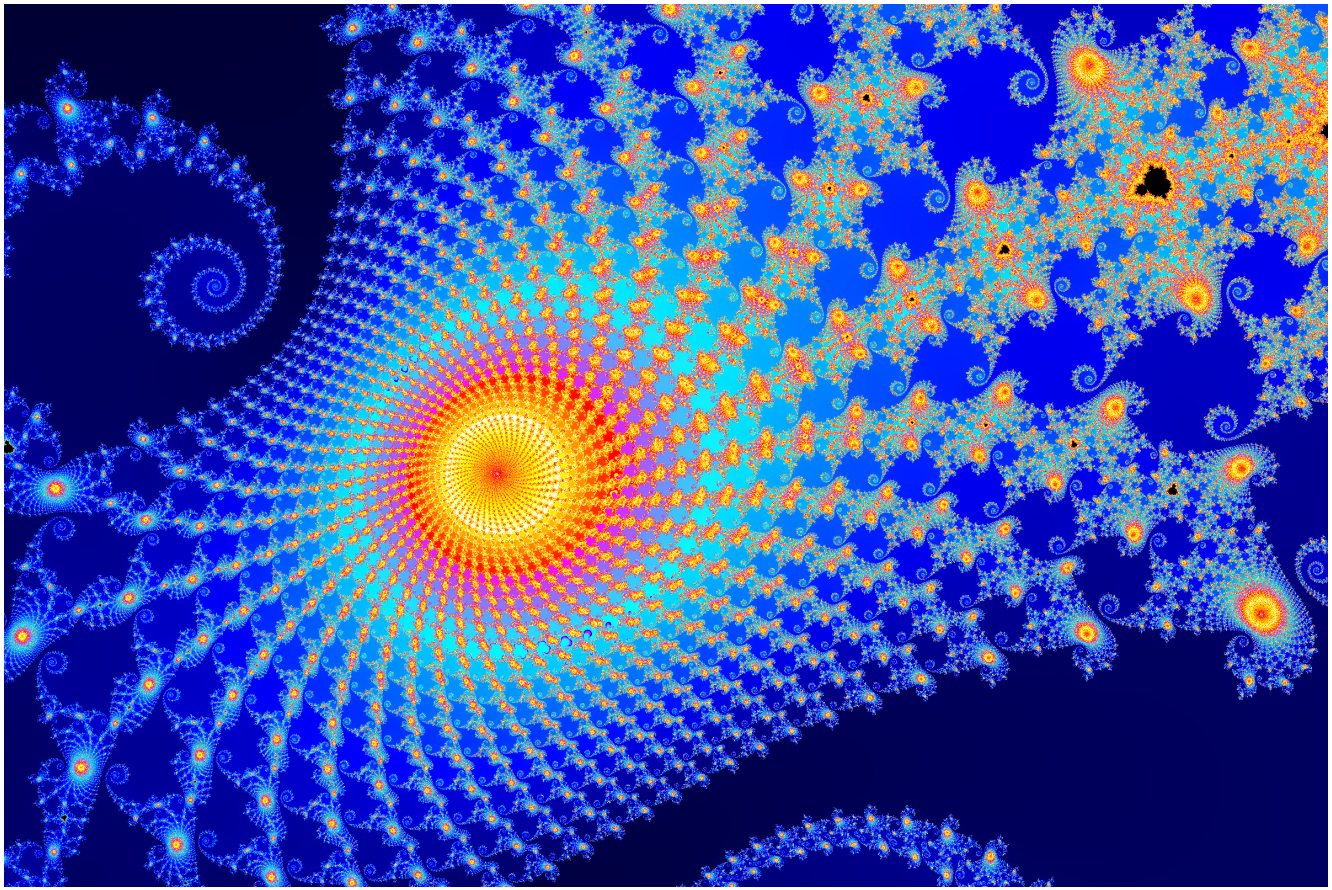
Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



Cool pictures

Black: Mandelbrot set. Color: iterations before too large.



How to do this?

Challenge!

c in Mandelbrot set if

$$\lim_{n\to\infty} |f_c^n(0)| \leq 4,$$

usually impractical/impossible to check

How to do this?

Challenge!

c in Mandelbrot set if

$$\lim_{n\to\infty} |f_c^n(0)| \leq 4,$$

usually impractical/impossible to check

How we cope

- Choose "big" $N \in \mathbb{N}^+$
- $\mu(c) \geq N$? "pretend" c in Mandelbrot set
- color c according to $\mu(c)$

Pseudocode

algorithm *Mandelbrot Number*
inputs
$c \in \mathbb{C}$
$N \in \mathbb{N}^+$
outputs
$\min(\mu(c), N)$
do
let $z = 0$
let $n = 1$
while $|z| \leq 4$ **and** $n \leq N$
let $z = z^2 + c$
increment n
return n

Pseudocode

algorithm *Mandelbrot Number*
inputs
$c \in \mathbb{C}$
$N \in \mathbb{N}^+$
outputs
$\min(\mu(c), N)$
do
let $z = 0$
let $n = 1$
while $|z| \leq 4$ **and** $n \leq N$
let $z = z^2 + c$
increment n
return n

. . . of course, a lot more is needed to make a picture

First sage implementation

1. Download Python implementation of Mandelbrot
2. Attach (from **sage in terminal**, not worksheet)
   ```
   sage: %attach mandelbrot_mat305.py
   ```
3. Run
   ```
   sage: M, im = mandelbrot(optional_ymin=-1.0)
   ```
   (This will be a little slow)
4. Save image:
   ```
   sage: im.save('python_mandelbrot.png')
   ```
5. Switch to File view, click on `python_mandelbrot.png`

Problem

It's too slow.

We can fix this.

We have the technology.

Better, stronger, faster…

Second sage implementation

1. Download Cython implementation of Mandelbrot
2. Attach (command-line, not worksheet)
   ```
   sage:  %attach mandelbrot_mat305.pyx
   ```
3. Run
   ```
   sage:  M, im = mandelbrot(optional_ymin=-1.0)
   ```
   (This step should be *quite* fast now)
4. See
   ```
   sage:  im.save('cython_mandelbrot.png')
   ```
5. Switch to File view, click on `cython_mandelbrot.png`

Outline

Compiled v. Interpreted programming

Recall:

- **interpreted software**
  - computer reads one line of program
  - translates to machine code
  - *executes*, forgets translation
- **compiled software**
  - computer reads entire program
  - translates to machine code
  - *saves* translation, does not execute

Sage v. Python v. Cython

- Sage built w/Python and (some) Cython

Sage v. Python v. Cython

- Sage built w/Python and (some) Cython
- Python: interpreted
  - facilities for easy, efficient, elegant programming
  - many operations still slow
  - variable's type can change

Sage v. Python v. Cython

- Sage built w/Python and (some) Cython
- Python: interpreted
  - facilities for easy, efficient, elegant programming
  - many operations still slow
  - variable's type can change
- Cython: compiled
  - works with most Python
  - not standalone (runs w/Python interpreter)
  - variable's type can be unchangeable

"Type"?

Definition

kind of data variable contains

Example

integers, rounded numbers, strings, ...

"Type"?

Definition

kind of data variable contains

Example

integers, rounded numbers, strings, ...

Python variables can contain "any data"

Example

```
sage:  a = 2
sage:  a = 'hello'
sage:  a = 3.0**5
```

"Type"?

Undeclared variables prohibited in "strongly typed" languages

compiler error in C:

```
void main() {
  a = 2;
}
```

```
test.c:2: error: 'a' undeclared (first use in this function)
```

"Type"?

Abusing type a bad idea in "strongly typed" languages

complier warning/error in C:

```
void main() {
  int a = 2;
  a = ''hello'';
}
```

```
test.c:3: warning: assignment makes integer from
pointer without a cast
```

"Type"?

Changing type prohibited in "strongly typed" languages

Compiler error in C:

```
void main() {
  int a = 2;
  char *a = ''hello'';
}
```

```
test.c:3: error: conflicting types for ‘a’
test.c:2: error: previous definition of ‘a’ was
here
```

"Type"?

Declaration has advantages and disadvantages

Disadvantages

- Can be harder to read or work
  - C++ templates border on incomprehensible
- Type often inferred or known from context
  - `x = 2.0`

Advantages

- Type known $\Longrightarrow$ no guessing
  - $2 \in \mathbb{Z}$, $2 \in \mathbb{Q}$, $2 \in \mathbb{R}$, $2 \in \mathbb{C}$, ...?
- No guessing $\Longrightarrow$ no checking $\Longrightarrow$ more efficient
  - a *lot* of behind-the-scenes work in interpreted languages

Cython's approach

declare types of functions, variables *when you want to*

- can leave some undeclared
- declare variables w/`cdef <type> <name>`
- declare functions w/`cpdef <type> <name>(...)`
  or `cdef <type> <name>(...)`
  (if you don't want to call from Python)

Types available?

- C types
  - `int`, `float`, `struct`
  - pointers: `T*`, `T**`, etc.
    - need to manage memory!
- Python types
  - `list`, `set`, `tuple`, `string`, `dict`, ...
- Sage objects
  - complicated

Compare

mandelbrot.py:

```
def compute_mandelbrot_iterates(xmin, ymin, \
    xsteps, ysteps, max_n, dx, dy):
  M = [[-1 for j in xrange(xsteps)]
  for i in xrange(ysteps)]:  ...
```

Compare

mandelbrot.pyx:

```
cdef list compute_mandelbrot_iterates(float xmin, \
    float ymin, int xsteps, int ysteps, \
    int max_n, float dx, float dy):
  cdef int i, j, n
  cdef float x, y, x0, y0, xtemp
  cdef list M = [[-1 for j in xrange(xsteps)] \
    for i in xrange(ysteps)]
  for i in xrange(ysteps):  ...
```

Do I need to write a script file?

No! Can compile Sage code in any cell by placing

```
%cython
```

before Cython function

Example of Cython in worksheet

```
sage:  %cython
       cpdef float c_square(float x):
           return x**2
       cpdef float cRiemann_left(float a, float b,
                int n):
           cdef int i
           cdef float Delta_x, xi, S
           Delta_x = (b-a)/n
           S = 0.0
           for i in xrange(n):
               xi = a + i*Delta_x
               S += c_square(xi)*Delta_x
           return S
```

Let's try it

```
sage:  %time cRiemann_left(0.0, 10.0, 100000)
333.3279113769531
CPU time:  0.00 s, Wall time:  0.00 s
```

Let's try it

```
sage:  %time cRiemann_left(0.0, 10.0, 100000)
333.3279113769531
CPU time:  0.00 s, Wall time:  0.00 s
```

Compare to a "Pythonic" version:

```
sage:  %time pRiemann_left(0.0, 10.0, 100000)
333.328333350000
CPU time:  0.23 s, Wall time:  0.23 s
```

Let's try it

```
sage:  %time cRiemann_left(0.0, 10.0, 100000)
333.3279113769531
CPU time:  0.00 s, Wall time:  0.00 s
```

Compare to a "Pythonic" version:

```
sage:  %time pRiemann_left(0.0, 10.0, 100000)
333.328333350000
CPU time:  0.23 s, Wall time:  0.23 s
```

The Cython version is appreciably slower.

Visualizing the Python load

You can see the C source code produced, along with an indication of Python-intensive lines

- command line: `sage -cython -a filename`
- worksheet: after entering cell, click on link labeled, `Auto-generated code...`

Example

```
 01:
+02: include "interrupt.pxi"  # ctrl-c interrupt block support
 03: include "stdsage.pxi"
 04:
 05: include "cdefs.pxi"
 06: from libc.math cimport sin
+07: cpdef float c_square(float x):
+08:     return x**2
+09: cpdef float c_Riemann_left(float a, float b, int n):
 10:     cdef int i
 11:     cdef float Delta_x, xi, S
+12:     Delta_x = (b - a)/n
+13:     S = 0.0
+14:     for i in xrange(n):
+15:         xi = a + i*Delta_x
+16:         S += c_square(xi)*Delta_x
+17:     return S
```

Example

```
 01:
+02: include "interrupt.pxi"  # ctrl-c interrupt block support
 03: include "stdsage.pxi"
 04:
 05: include "cdefs.pxi"
 06: from libc.math cimport sin
+07: cpdef float c_square(float x):
+08:     return x**2
+09: cpdef float c_Riemann_left(float a, float b, int n):
 10:     cdef int i
 11:     cdef float Delta_x, xi, S
+12:     Delta_x = (b - a)/n
+13:     S = 0.0
+14:     for i in xrange(n):
+15:         xi = a + i*Delta_x
+16:         S += c_square(xi)*Delta_x
+17:     return S
```

Why is there Python code in line 12? Click on it to expand!

Example

```
 01:
+02: include "interrupt.pxi"  # ctrl-c interrupt block support
 03: include "stdsage.pxi"
 04:
 05: include "cdefs.pxi"
 06: from libc.math cimport sin
+07: cpdef float c_square(float x):
+08:     return x**2
+09: cpdef float c_Riemann_left(float a, float b, int n):
 10:     cdef int i
 11:     cdef float Delta_x, xi, S
+12:     Delta_x = (b - a)/n
   __pyx_t_1 = (__pyx_v_b - __pyx_v_a);
   if (unlikely(__pyx_v_n == 0)) {
     PyErr_SetString(PyExc_ZeroDivisionError, "float division");
     {__pyx_filename = __pyx_f[0]; __pyx_lineno = 12; __pyx_cline
   }
   __pyx_v_Delta_x = (__pyx_t_1 / __pyx_v_n);
+13:     S = 0.0
+14:     for i in xrange(n):
+15:         xi = a + i*Delta_x
+16:         S += c_square(xi)*Delta_x
+17:     return S
```

Why is there Python code in line 12? To avoid division by 0!

. . . and a lot more, too!

- linking to code written in C, C++, other languages
- extending Python, Sage w/efficient data types, routines
- & more!

Outline

Summary

- Compilation can improve performance of code
- Sage uses Cython to compile code
- Cython can use data types to improve performance