MAT 305: Mathematical Computing

Cython

John Perry

University of Southern Mississippi

Fall 2011

Outline

Outline

Mandelbrot Numbers

- Let
  - $c \in \mathbb{C}$
  - $f : \mathbb{C} \longrightarrow \mathbb{C}$ by $f_c(z) = z^2 + c$
- Let $\mu : \mathbb{C} \longrightarrow \mathbb{N}^+$ by

$$\mu(c) = n \iff n \text{ smallest such that } \left| f_c^n(0) \right| > 4$$

In other words, we count how many times we apply $f_{a,b}$ before result has size greater than 4.

Definition

- $\mu(c)$ is the **Mandelbrot number** of c.
- If $\mu(c) = \infty$ then c is in the **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

$\mu(1) = 3$

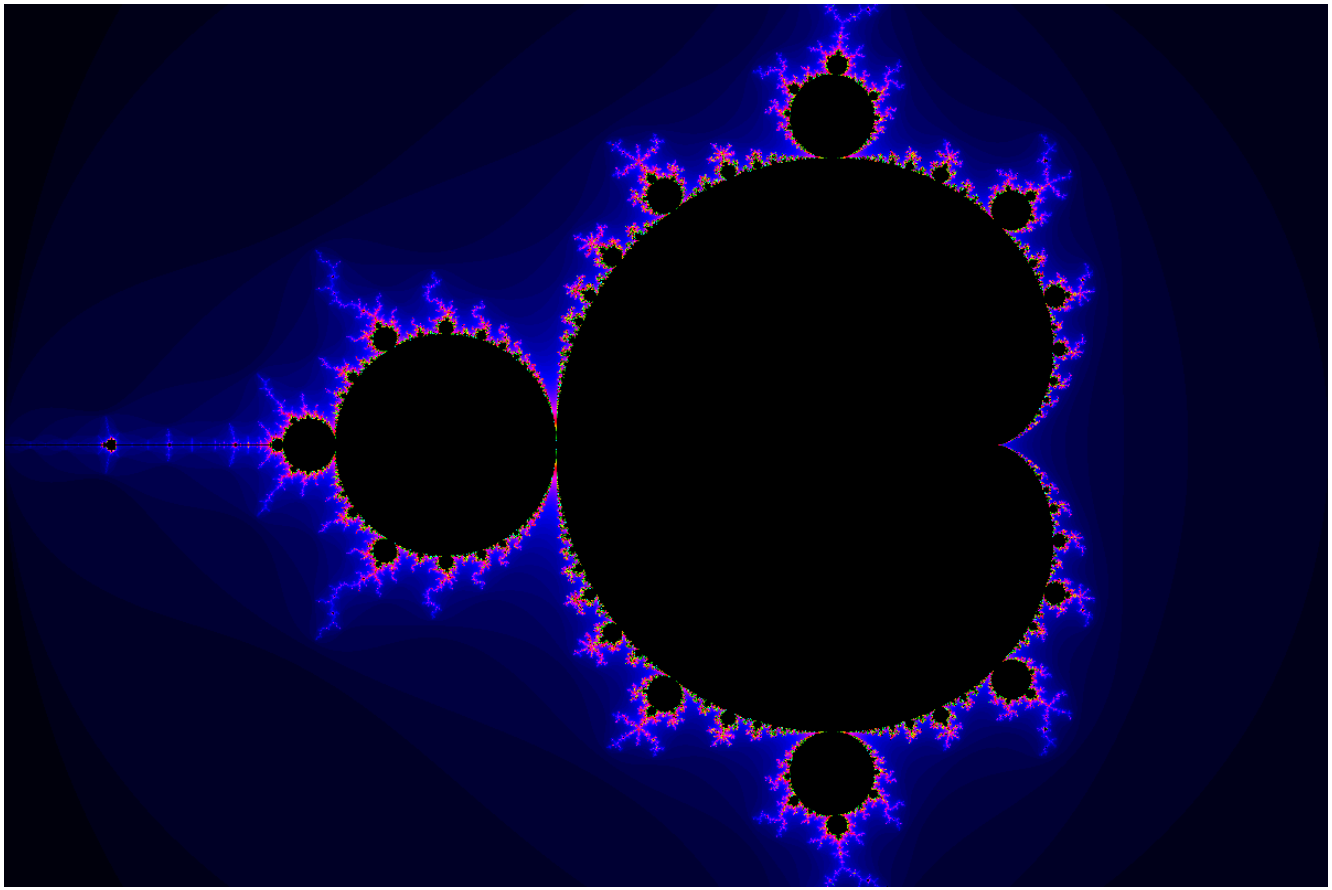
$$f_1(0) = 0^2 + 1 = 1$$
$$f_1^2(0) = f_1(1) = 1^2 + 1 = 2$$
$$f_1^3(0) = f_1(2) = 2^2 + 1 = 5.$$

Example

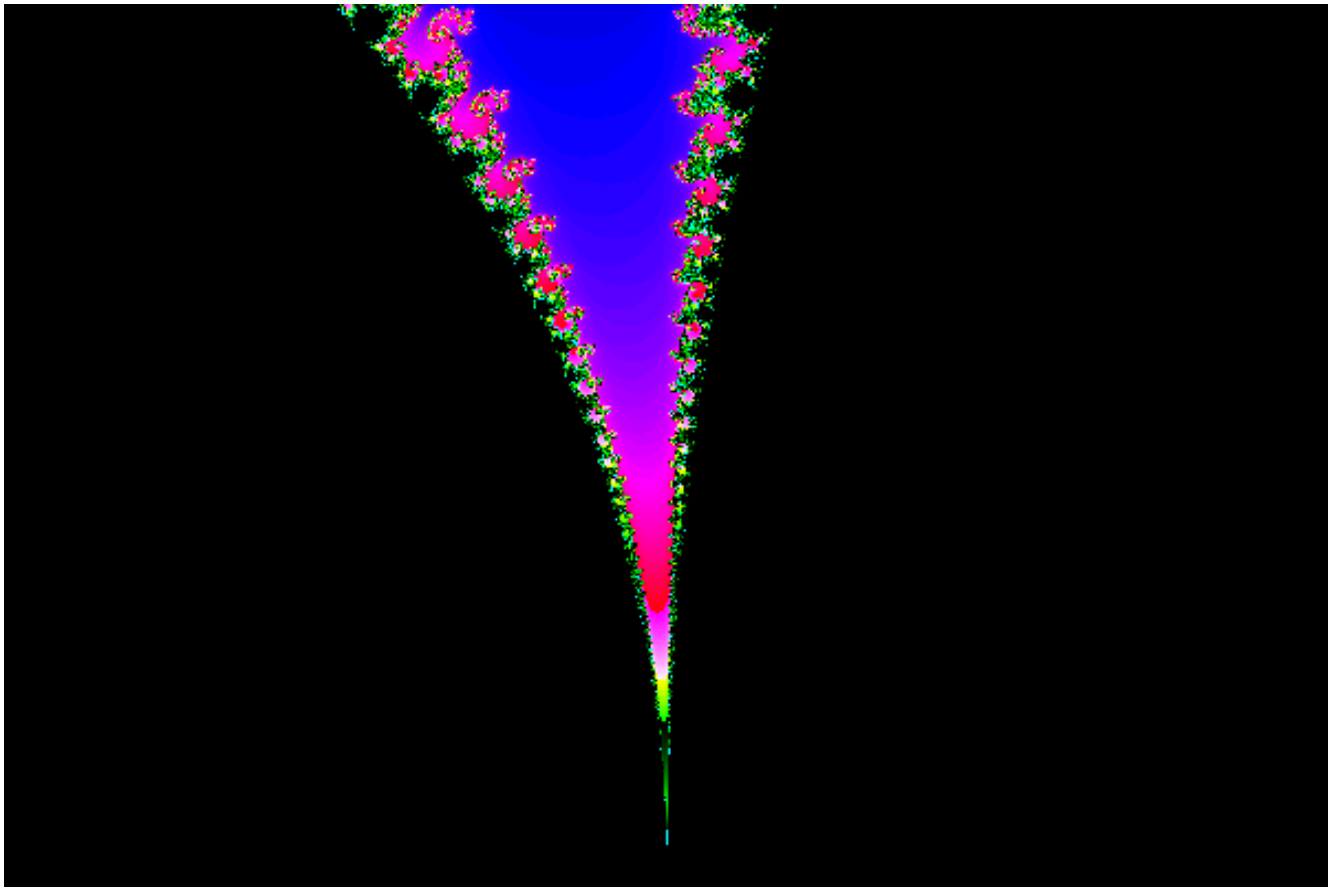
$\mu(i) = \infty$

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

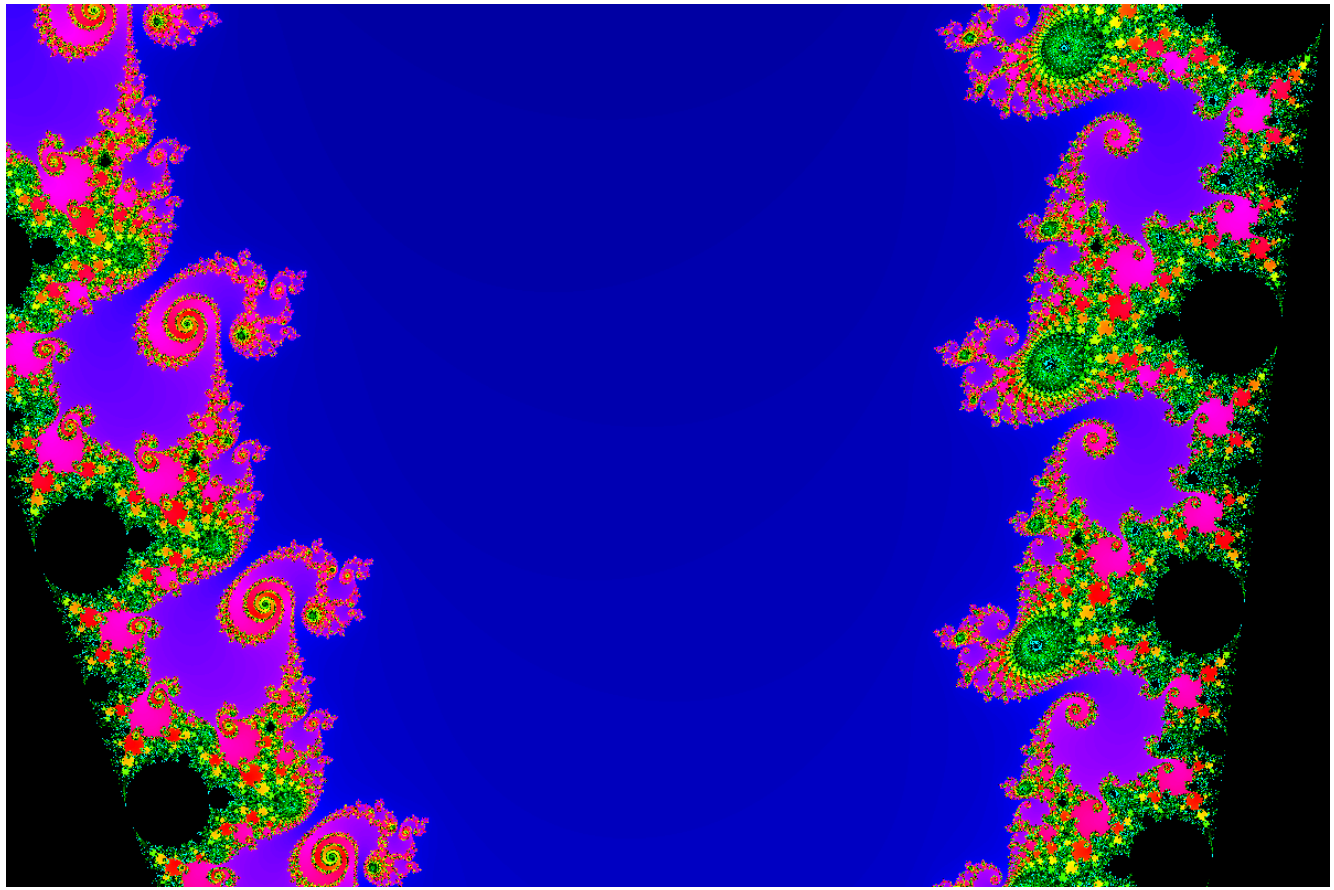
Cool pictures



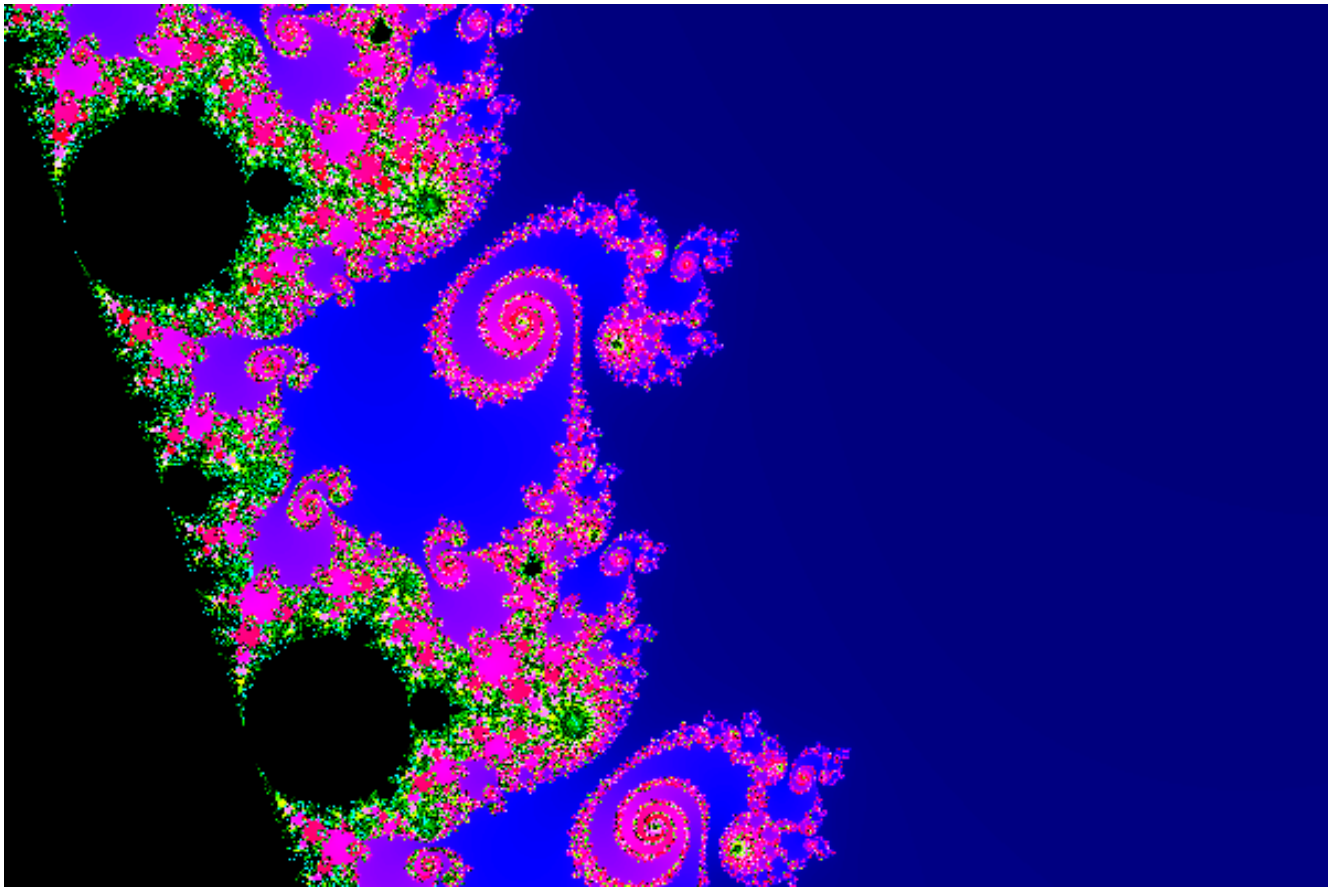
Cool pictures



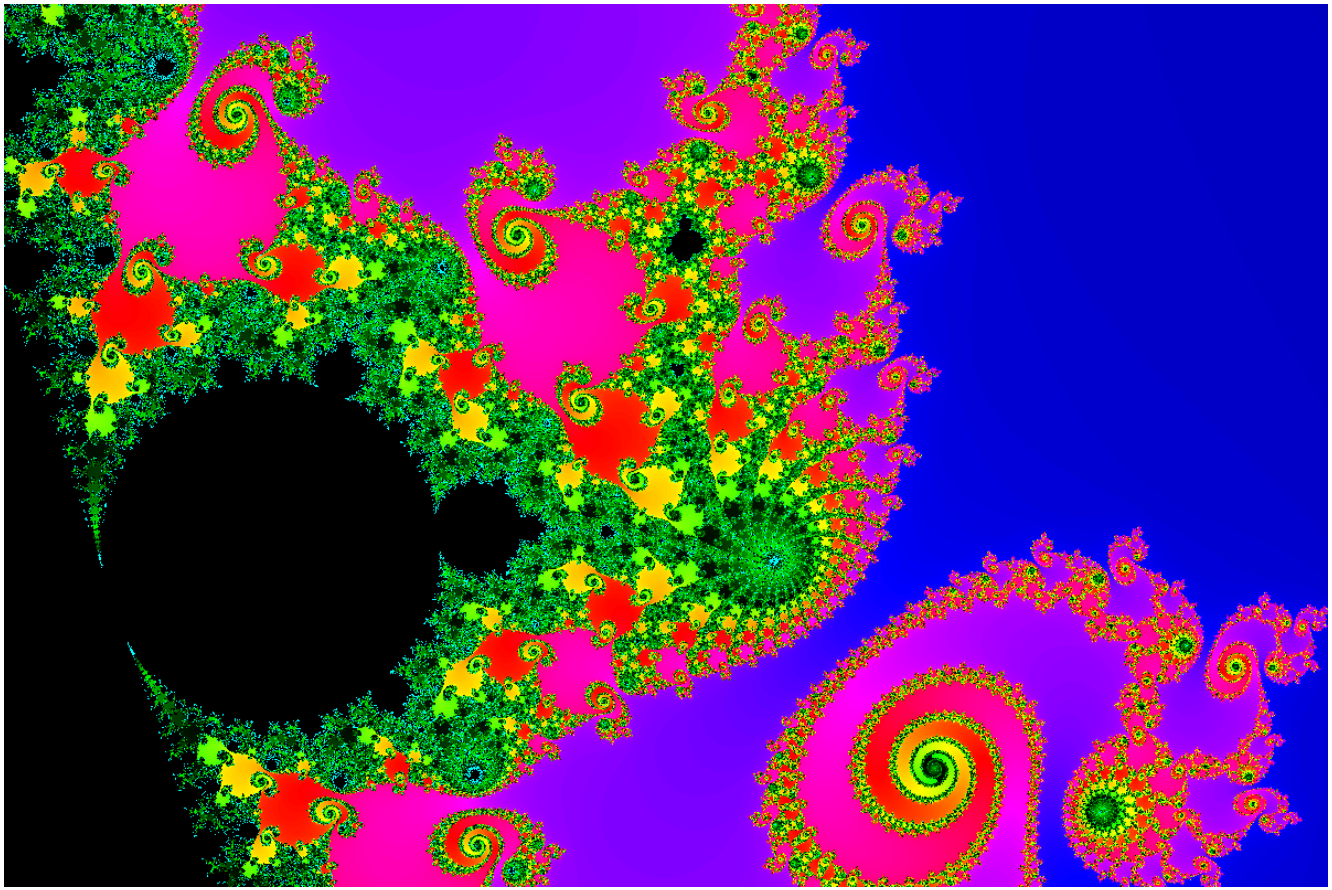
Cool pictures



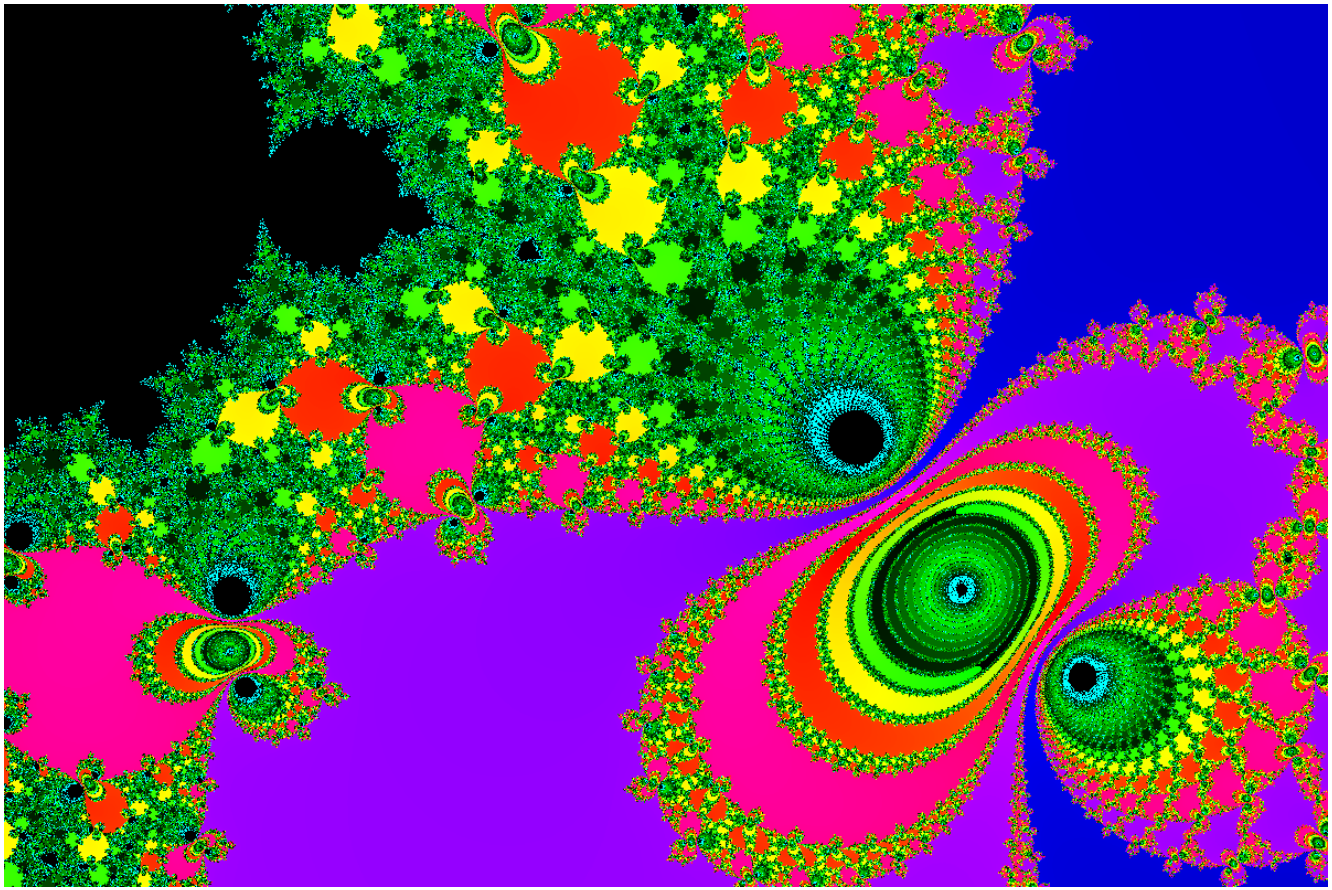
Cool pictures



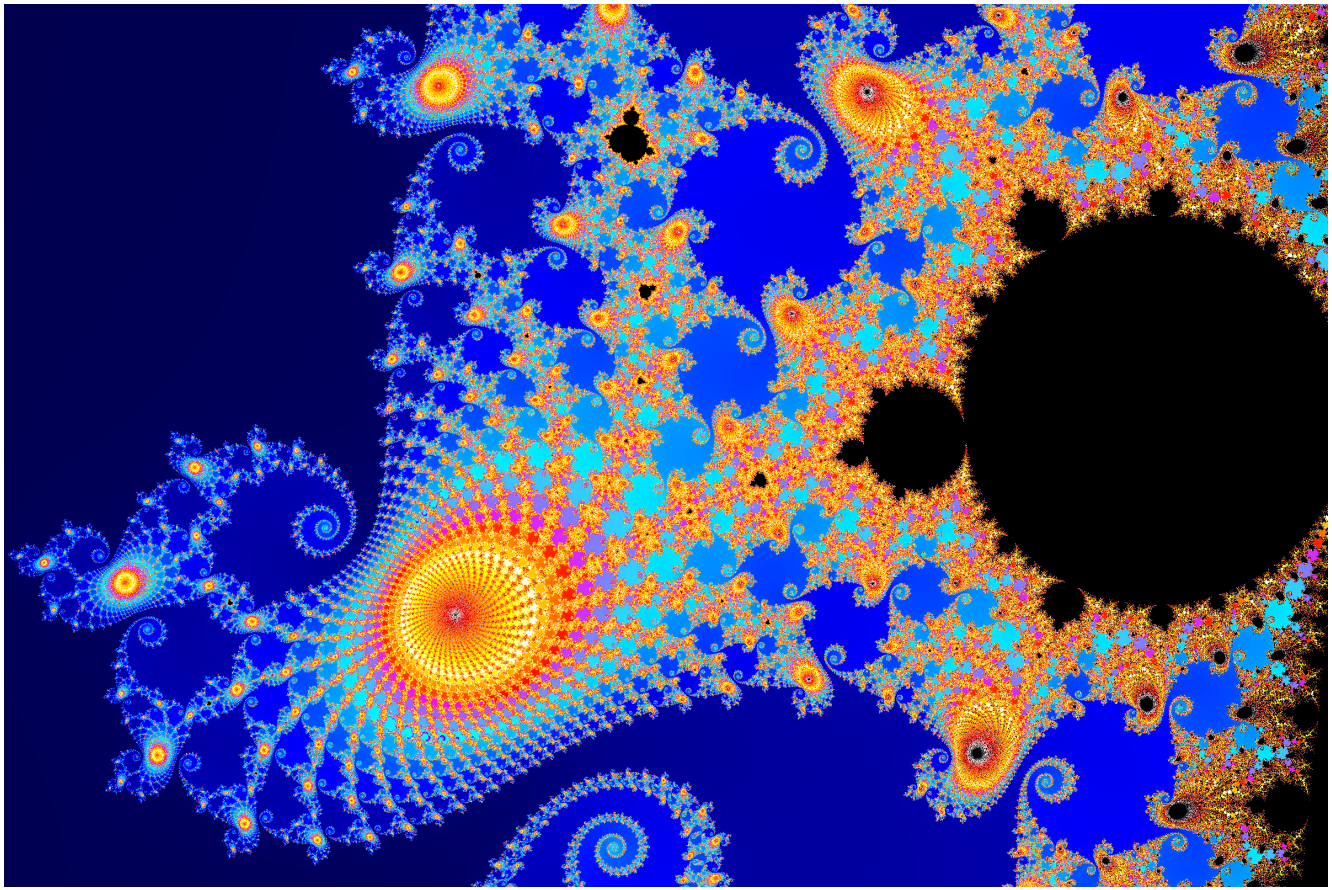
Cool pictures



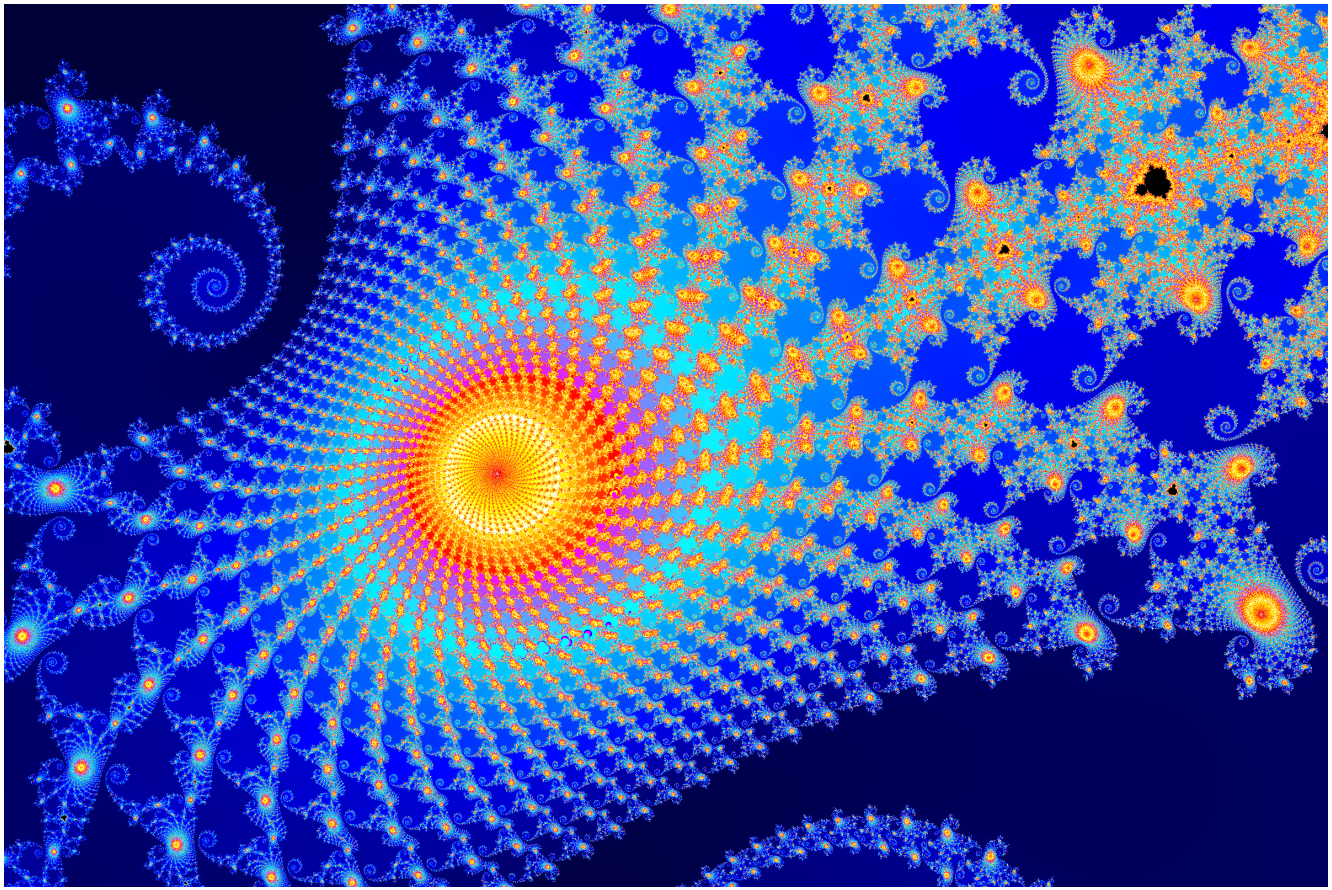
Cool pictures



Cool pictures



Cool pictures



How to do this?

Challenge!

- c not in set if $\lim_{n\to\infty} \left| f_c^n(0) \right| > 4$.
- We can't check $\lim_{n\to\infty} \left| f_c^n(0) \right|$ for most c.

How to do this?

Challenge!

- c not in set if $\lim_{n\to\infty} \left|f_c^n(0)\right| > 4$.
- We can't check $\lim_{n\to\infty} \left|f_c^n(0)\right|$ for most c.

Workaround

- Let $N \in \mathbb{N}^+$ be "big".
  - here, "big" can be as small as 100 or even 10.
- If $\mu(c) \geq N$, we "pretend" c in Mandelbrot set.
- Idea of precision: 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 (command-line, not worksheet)

   ```
   sage:  attach mandelbrot_mat305.py
   ```

3. Run

   ```
   sage:  M, im = mandelbrot(optional_ymin=-1.0)
   ```

   (This step might be a little slow)

4. See

   ```
   sage:  im.show()
   ```

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.show()
   ```

Outline

Compiled v. Interpreted programming

Recall from textbook:

- in **interpreted software**:
  - computer reads one line of program
  - translates it to machine code
  - executes it, forgets translation
  - repeat as necessary
- in **compiled software**:
  - computer reads entire program
  - translates it to machine code once
  - saves translation to memory or file
  - executes *many* times

Sage v. Python v. Cython

- Sage is built using Python

Sage v. Python v. Cython

- Sage is built using Python
- Python is interpreted
  - facilities for fast, efficient, elegant programming
  - many operations still slow
  - variable's type can change

Sage v. Python v. Cython

- Sage is built using Python
- Python is interpreted
  - facilities for fast, efficient, elegant programming
  - many operations still slow
  - variable's type can change
- Cython is compiled
  - project in development
  - works with most Python constructs
  - not standalone (runs w/in Python interpreter)
  - variable's type can be unchangeable

Type?

A variable's "type" indicates the kind of data it contains

- integers, rounded numbers, strings, ...

Python variables can contain "any data"

Example

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

Type?

You cannot use untyped variables in "strongly typed" languages

In C, for example,

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

. . . generates a compiler error:

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

Type?

You cannot abuse typed variables in "strongly typed" languages

In C, for example,

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

. . . generates a compiler warning:

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

Type?

You cannot redefine typed variables in "strongly typed" languages

In C, for example,

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

. . . generates a compiler error:

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

Type?

Declaring a variable's type has advantages and disadvantages

Disadvantages

- Can be harder to read or work in *interpreted* languages
- Type often inferred easily or known from context
  - `x = 2.0` seems relatively clear

Advantages

- Type known $\Longrightarrow$ compiler doesn't have to guess
  - Do you mean $2 \in \mathbb{Z}$, $2 \in \mathbb{Q}$, $2 \in \mathbb{R}$, $2 \in \mathbb{C}$, ...?
- Identifying type at compile time? faster run time!

Cython's approach

- end sage file with `.pyx` (not `.py`)
- declare types of functions, variables *when you want to*
  - can leave some undeclared
  - declare functions w/`cpdef <type> <name>(...)` or `cdef <type> <name>(...)` (if you don't want to call from Python)
  - declare variables w/`cdef <type> <name>`

Types available?

- C types
  - int, float, struct
  - pointers: T*, T**, etc.
    - manage your own memory!
- Python types
  - list, set, tuple, string, dict, ...
- Sage objects
  - somewhat complicated, see me if you need it

Compare

In `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

In `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):  ...
```

In the worksheet

You can compile the Sage code in any cell by starting with %cython

Visualizing the improvement

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, `...spyx.html`

Example: Python

```
20: def compute_mandelbrot_iterates(xmin, ymin, \
21:     xsteps, ysteps, max_n, dx, dy):
22:   r"""
23:     Computes an array ``M`` of integers constituting
24:     the number of iterations before the complex number
25:     at the corresponding location was determined not
26:     to be in the Mandelbrot set.
27:
28:     INPUT::
29:       * ``xmin``, ``ymin`` --
30:         The lower left corner of the part of the complex plane
31:         that we want to graph. The horizontal numbers are real parts;
32:         the verical numbers are complex parts.
33:       * ``xsteps``, ``ysteps`` --
34:         How many steps to travel the horizontal and vertical directions.
35:       * ``max_n`` --
36:         How many times to apply the iteration before assuming the number
37:         is in the Mandelbrot set.
38:       * ``dx``, ``dy`` --
39:         How far to travel in the horizontal and vertical directions on each step.
40:     OUTPUT::
41:       * ``M`` --
42:         ``M[i][j]`` corresponds to the number of times the iteration was applied
43:         to location ``xmin + i*dx``, ``ymin + j*dy``.
44:   """
45:   M = [[-1 for j in xrange(xsteps)] for i in xrange(ysteps)]
46:   for i in xrange(ysteps):
47:     if i % 50 == 0: print "row", i, "out of", ysteps
48:     for j in xrange(xsteps):
49:       n = 0
50:       x0 = xmin + j * dx
51:       y0 = ymin + i * dy
52:       x = 0.0
53:       y = 0.0
54:       while (x*x + y*y <= 4.0) and n <= max_n:
55:         xtemp = x*x - y*y + x0
56:         y = 2.0*x*y + y0
57:         x = xtemp
58:         n += 1
59:       M[i][j] = n
60:   return M
```

Example: Cython

```
25: cdef list compute_mandelbrot_iterates(float xmin, float ymin, \
26:     int xsteps, int ysteps, int max_n, float dx, float dy):
27:   r"""
28:     Computes an array ``M`` of integers constituting
29:     the number of iterations before the complex number
30:     at the corresponding location was determined not
31:     to be in the Mandelbrot set.
32:
33:     INPUT::
34:       * ``xmin``, ``ymin`` --
35:         The lower left corner of the part of the complex plane
36:         that we want to graph. The horizontal numbers are real parts;
37:         the verical numbers are complex parts.
38:       * ``xsteps``, ``ysteps`` --
39:         How many steps to travel the horizontal and vertical directions.
40:       * ``max_n`` --
41:         How many times to apply the iteration before assuming the number
42:         is in the Mandelbrot set.
43:       * ``dx``, ``dy`` --
44:         How far to travel in the horizontal and vertical directions on each step.
45:     OUTPUT::
46:       * ``M`` --
47:         ``M[i][j]`` corresponds to the number of times the iteration was applied
48:         to location ``xmin + i*dx``, ``ymin + j*dy``.
49:   """
50:   cdef int i, j, n
51:   cdef float x, y, x0, y0, xtemp
52:   cdef list M = [[-1 for j in xrange(xsteps)] for i in xrange(ysteps)]
53:   for i in xrange(ysteps):
54:     if i % 50 == 0: print "row", i, "out of", ysteps
55:     for j in xrange(xsteps):
56:       n = 0
57:       x0 = xmin + j * dx
58:       y0 = ymin + i * dy
59:       x = 0.0
60:       y = 0.0
61:       while (x*x + y*y <= 4.0) and n <= max_n:
62:         xtemp = x*x - y*y + x0
63:         y = 2.0*x*y + y0
64:         x = xtemp
65:         n += 1
66:       M[i][j] = n
67:   return M
```

. . . 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