Real-Time Image Processing in Python

Gabriel J. Ferrer
Hendrix College

The Situation

- Using netbook to control a robot
  - Supports research and teaching in computer vision
- Why Python?
  - Rapid prototyping of algorithms
  - Easy frame-grabbing with the pygame library
- Why not Python?
  - Notoriously bad performance in tight loops
- Solution
  - Use Cython and Numpy

Just how slow is Python, really?

- Benchmarking web site
  - http://shootout.alioth.debian.org
- Median Python vs. C
  - C is about 50 times faster than Python

Robot Vision Processing

- Goal
  - Use image sequences to guide robot behavior
- Process
  - Acquire an image
  - Transform it into a data structure
  - Select a robot action based on data structure
- Performance consideration
  - Rate of action selection is bounded by rate of image acquisition and transformation

Real-Time Image Processing

- Real-time systems
  - Correctness of code depends on whether deadlines are met
  - Efficiency is helpful
  - Only necessary for meeting a deadline
- Need for prompt action selection by the robot
  - Implies a *soft deadline* for the image computations
  - Ideal is 10 frames/second
  - Performance degrades below this point

Performance issues in image processing

- Images are arrays
  - Must visit every array element
  - Need fast array access
  - Need fast looping
- Typical operations
  - Image subtraction
  - Edge detection
  - Color matching
  - Connected components

Pygame Library

- As of version 1.9, includes frame grabbing
- Includes several important modules:
  - pygame.display
    - Handles rendering to a window
  - pygame.surface
    - Represents an image
  - pygame.camera
    - Grabs images from a camera

Initialization

```
import pygame
import pygame.camera
from pygame.locals import *

pygame.init()
pygame.camera.init()
size = (640, 480)
```

Setting up

```
d = pygame.display.set_mode(size, 0)
s = pygame.surface.Surface(size, 0, d)
c = pygame.camera.list_cameras()

cam = pygame.camera.Camera(c[0], size)
cam.start()

going = True
```

Main Loop

```
while going:
  if cam.query_image():
    s = cam.get_image(s)
  d.blit(s, (0, 0))
  pygame.display.flip()
  for e in pygame.event.get():
    if e.type == QUIT:
      cam.stop()
      going = False
```

Processing Images

- Pygame surfarray library
  - Converts pygame surfaces to numpy arrays
- Numpy (1.3)
  - High-speed n-dimensional arrays (ndarray)
  - All elements have the same data type
- Why is the same data type important?
  - Tight two-dimensional loop
  - Each inner loop iteration involves a type check!

Detecting Moving Objects

- For each frame:
  - Convert image to an array
  - Subtract the previous array
  - Find the non-zero regions of nontrivial size

Applying numpy (1)

- Add before the start of the loop:

```
last_array = None
diffs = None
s = pygame.surface.Surface(size)
```

Applying numpy (2)

- Inner loop, `if` statement body:

```
s = cam.get_image(s)
s2d = pygame.surfarray.array2d(s)
diffs = s2d
if last_array != None:
  diffs = s2d - last_array
last_array = s2d
pygame.surfarray.blit_array(s, diffs)
```

Problem

- Excessive background noise
- Solution: Hue, Saturation, Value (HSV)
  - Hue: The “type” of a color
  - Saturation: The “strength” of a color
  - Value: The “whiteness” of a color
- Disregard H and S; just use V

Extracting the Value

- Each color is 24 bits:
  - Bits 23-16 are Red (RGB) or Hue (HSV)
  - Bits 15-8 are Green (RGB) or Saturation (HSV)
  - Bits 7-0 are Blue (RGB) or Value (HSV)
- Mask all but the lower 8 bits to get V
- NB: Display is still RGB
  - The V will look blue

Code Alterations

- When initializing `cam`:

```
cam = pygame.camera.Camera(c[0],
size, “HSV”)
```

- Immediately after creating `s2d`:

```
s2d = numpy.bitwise_and(s2d, 0xFF)
```

Finding the Blobs

- Connected components ("blobs")
  - "Islands" in an image that share a characteristic
- Blob finding:
  - First, threshold the image
    - "Useful" pixels will be high values
  - Second, find the blobs
    - Returns a list of the blobs

Implementing Blob Finding

- Useful variables to initialize at the start:

```
b = (0, 0, 0xFF)

r = (0xFF, 0, 0)

t = (0x5A, 0xAA, 0xAA)
```

Additional Code

- Inside the `if` statement, after `blit_array`:

```
m = pygame.mask.from_threshold(s, b, t)
for blob in m.connected_components(10):
    coord = blob.centroid()
    pygame.draw.circle(s, r, coord, 50, 5)
```

Image Shrinking

- Often an effective technique to boost frame rate
- Into our original program, insert at the top:

```
shrunken = (320, 240)
```

- Then replace the `blit()` call with:

```
p = pygame.transform.scale(s, size)
d.blit(p, (0, 0))
```

Writing Custom Routines

- Numpy is very nice, but it doesn't do everything
- Basic threshold function:

```
def threshold(img, value, hi, lo):
  for i in range(img.shape[0])
    for j in range(img.shape[1]):
      if img[i,j] < value:
        img[i,j] = lo
      else:
        img[i,j] = hi
```

Cython

- Compiles Python programs to C
  - From C, compiles to binary object code
- Using version 0.11
  - Still very experimental
- Superset of Python
  - Will compile any Python program
  - For best results, augment with type declarations

Cython Initialization

```
cimport numpy
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
```

Cython Type Declarations

```
def threshold
(numpy.ndarray[numpy.int32_t,
ndim=2] img,
numpy.int32_t value,
numpy.int32_t hi,
numpy.int32_t lo):
    cdef Py_ssize_t i, j
```

Performance Difference

- Interpreted Python
  - 0.85 frames/second
- Cython with type declarations
  - 10.81 frames/second

Creating a `setup.py` script

- Program name: `filters.pyx`

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension(“filters”,
[“filters.pyx”])]
setup(name = 'Img proc filters',
      cmdclass = {'build_ext': build_ext},
      ext_modules = ext_modules)
```

Compiling the program

```
python setup.py build_ext --inplace
```

Generated C Code

- Lots of setup code at function start
  - Checks expected vs. actual arguments
  - Creates lots of temporary variables
- Inefficient function calls inside tight loops
  - Use `cdef` to minimize this
  - `cdef` functions are not callable from Python
- Code is otherwise a direct translation into C
- Numpy arrays are not C arrays
  - Array accesses use a macro for pointer arithmetic

Generated C Code

```
for (__pyx_t_1 = 0; __pyx_t_1 < (__pyx_v_img->dimensions[0]); __pyx_t_1+=1) {
    __pyx_v_i = __pyx_t_1;
  for (__pyx_t_2 = 0; __pyx_t_2 < (__pyx_v_img->dimensions[1]); __pyx_t_2+=1)
{
      __pyx_v_j = __pyx_t_2;
      __pyx_t_3 = __pyx_v_i;
      __pyx_t_4 = __pyx_v_j;
      if (((*__Pyx_BufPtrStrided2d(__pyx_t_5numpy_int32_t *, __pyx_bstruct_img.b
uf, __pyx_t_3, __pyx_bstride_0_img, __pyx_t_4, __pyx_bstride_1_img)) < __pyx_v_v
alue)) {
        __pyx_1 = __pyx_v_lo;
      } else {
        __pyx_1 = __pyx_v_hi;
      }
      __pyx_t_5 = __pyx_v_i;
      __pyx_t_6 = __pyx_v_j;
      *__Pyx_BufPtrStrided2d(__pyx_t_5numpy_int32_t *, __pyx_bstruct_img.buf, __
pyx_t_5, __pyx_bstride_0_img, __pyx_t_6, __pyx_bstride_1_img) = __pyx_1;
  }
}
```

Conclusion

- You can do robot vision in Python!
- Pygame
  - Frame grabbing
  - Some image processing
- numpy
  - High-performance arrays
  - Matrix arithmetic
- cython
  - Compilation; high-performance object code

Resources

- http://www.pygame.org
- http://www.cython.org
- http://wiki.cython.org/tutorials/numpy
- http://ozark.hendrix.edu/~ferrer/presentations/
  - These slides
  - Sample code