The Fast Inverse Square Root method in Python

The inverse square root of a number x is x-1/2. For example, put in 25, you'll get back 0.2: the square root of 25 is 5, the inverse of 5 is 1/5, or 0.2 in decimal notation. It's a very common calculation in computer graphics, for example, where you need to normalise a lot of vectors.

If you've marvelled at the Fast Inverse Square Root method and want to use this piece of witchcraft to speed up your Python code, **stop**! Just stick with writing `x ** -0.5`. It will be *much* faster than any custom function and probably a lot more accurate too. Besides, if you're trying to optimise your number-crunching Python code to this level of hackery, you should probably choose another language for your project.

With that said, it's still fun to think up ways to implement the method in Python. Below are three ways it could be done. I maintain that you should always stick to `x ** -0.5`, but feel free to try them for speed.

Implementing the method in Python

The Fast Inverse Square Root method hinges on quickly reinterpreting the bits of a float as an integer, doing simple arithmetic on that integer, and then reinterpreting the bits of that integer as a float. This reinterpretating of memory is very easy in a language like C. We can point to a memory location (using a pointer) and say “read those bits as an integer” or “read those bits as a float” and carry on working.

Python doesn't use plain machine types like 32 bit integers and floats; it uses its own boxed-type objects instead. A Python float is more than simply 32 or 64 bits of memory: it has attributes and methods attached, as well as meta-data such reference counts (for memory management). Moreover, Python doesn't provide any straightforward way to locate and access the right part of your program's memory to get at the raw bits of a number. Nor does it let you say anything like “the bytes of this float are to be read as an integer”. This is the obstacle we must overcome.

I'll base the Python functions on the following C implementation of the Fast Inverse Square Root method taken from Wikipedia:

```c
float Q_rsqrt( float number )
{
    int i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;

    i  = * ( long * ) &y;                       /* float to int */
    i  = 0x5f3759df - ( i >> 1 );               /* int arithmetic */
    y  = * ( float * ) &i;                      /* int back to float */

    y  = y * ( threehalfs - ( x2 * y * y ) );   /* Newton's method */
    return y;
}
```

Compiling this function and using the float 16.0 as my input, I get back a value of 0.24957679212, close to the expected value 0.25.

This function uses one iteration of Newton's method to improve the accuracy of the returned float value. I'll do the same in the Python functions below. The other difference is that my functions will take Python **`float`** objects as input and also return a Python **`float`** object. Other than that, I'll stick as close as possible to the lines of the C function for the purpose of exposition.

Using ctypes

The ctypes library lets you create native C data type values in Python (int32, float16, pointers) so you can delgate hard work to C code. This makes it useful for implementing the Fast Inverse Square Root method where we really need to work with data types which Python lacks. However, we still need to convert between C data types and Python data types as we can't do arithmetic on C types from within Python itself. This means a lot of function calls and attribute lookups, slowing the code down:

```
from ctypes import c_float, c_int32, cast, byref, POINTER

def ctypes_isqrt(number):
    threehalfs = 1.5
    x2 = number * 0.5
    y = c_float(number)

    i = cast(byref(y), POINTER(c_int32)).contents.value
    i = c_int32(0x5f3759df - (i >> 1))
    y = cast(byref(i), POINTER(c_float)).contents.value

    y = y * (1.5 - (x2 * y * y))
    return y
```

This function calculates the inverse square root as well as can be expected after one iteration of Newton's method:

```
>>> ctypes_isqrt(16.0)
0.24957678739619552
```

Using struct

The struct module in Python's standard library provides a set of functions for turning a number into a string of bytes, or turning a string of bytes back into a number. This makes it an obvious choice for implementing the method. The **`pack`** and **`unpack`** methods are less concise than the pointer/reference approach that's possible in ctypes:

```
def struct_isqrt(number):
    threehalfs = 1.5
    x2 = number * 0.5
    y = number

    packed_y = struct.pack('f', y)
    i = struct.unpack('i', packed_y)[0]  # treat float's bytes as int
    i = 0x5f3759df - (i >> 1)            # arithmetic with magic number
    packed_i = struct.pack('i', i)
    y = struct.unpack('f', packed_i)[0]  # treat int's bytes as float
    y = y * (threehalfs - (x2 * y * y))   # Newton's method
    return y
```

Testing the function, we see that it works as expected:

```
>>> struct_isqrt(16.0)
0.24957678739619552
```

Using NumPy

Although not part of the standard library, NumPy is widely-used and provides a convenient API for working with machine integers and floats. The **`view`** method allows the bytes of an array or value to be reinterpreted as any other type (of the same width). That's all we need here but we must ensure that NumPy does not promote our integers during the magic number arithmetic:

```
def numpy_isqrt(number):
    threehalfs = 1.5
    x2 = number * 0.5
    y = np.float32(number)

    i = y.view(np.int32)
    i = np.int32(0x5f3759df) - np.int32(i >> 1)
    y = i.view(np.float32)

    y = y * (threehalfs - (x2 * y * y))
    return float(y)
```

Using **`view`** looks a lot neater than **`struct.pack`** and **`struct.upack`**. The function gives the same result as the previous two:

```
>>> numpy_isqrt(16.0)
0.24957678739619552
```

Timings

All of these functions are too slow and too inaccurate to be of any practical use in normal code. For comparison, here’s a function for the inverse square root as it should be implemented in Python:

```
def normal_isqrt(number):
    return number ** -0.5
```

Here are the timings of the functions above, from fastest to slowest, taking the best of 3 over 100000 loops:

```
normal_isqrt:   252 ns per loop
struct_isqrt:   1.65 µs per loop
ctypes_isqrt:   6.88 µs per loop
numpy_isqrt:    21.2 µs per loop
```

The additional method calls and arithmetic slow down all of the implementations of the Fast Inverse Square Root. I was a little surprised that the “simplest-looking” implementation, **`numpy_isqrt`**, came last by a significant margin. Perhaps this is because we had to do a few more type conversions and turning Python types into NumPy dtypes is not quick.

Written on April 1, 2016

ALSO ON **AJCR.NET**

A basic introduction to NumPy's einsum – …
9 years ago · 23 comments
Haphazard investigations

Python power towers – ajcr – Haphazard …
9 years ago · 1 comment
Haphazard investigations

Building a multivariate hypergeometric …
7 months ago · 2 comments
Haphazard investigations

An Illustra
Shape an
4 years ago ·
Haphazard

3 Comments 1 Login ▾

G

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Share Best Newest Oldest

L **Leonardo Paffi**
2 years ago
I noticed a typo in the C example (line 12) where the cast to (int *) should be replaced with (float*)
0 0 Reply

A **Alex** Mod ➦ Leonardo Paffi
a year ago
Thank you, I've corrected the typo.
0 0 Reply

B **best essay service**
8 years ago
Amazing that even in using this different activity can be done using Phyton. That is why there are many students who wanted to enhance their skills in programming that will probably help them to succeed in the future.
0 0 Reply

Subscribe Privacy Do Not Sell My Data