Cython Landau-Lifshitz Kernel

December 30, 2014

```
In [3]: %matplotlib inline
        from matplotlib import pyplot as plt
        import numpy as np
```

1 Cython Landau-Lifshitz Kernel

1.1 Python Kernel

Before developing the Cython Kernel, the Landau-Lifshitz equation will be solved using Python. This provides a benchmark to iterate from. The `scipy.integrate.ode` function is used to interate the moment based on its derivative.

```
In [13]: from scipy.integrate import ode

         # Simulation parameters
         alpha = 0.01
         gamma = 2.1*9.2740e-21/(6.6261e-27 /(2.0*np.pi))
         Ms = 140 # emu/cc
         h_ext = np.array([0, 1000, 0], dtype=np.float32) / Ms
         dt = 5e-13*gamma*Ms/(1 + alpha**2.)

         # Initial conditions
         t0, m0 = 0., np.array([-0.999, 0.001, 0.001], dtype=np.float32)

         def evolve(t, m, h_ext):
             h_eff = h_ext
             hxm = np.cross(h_eff, m)
             mxhxm = np.cross(m, hxm)
             return [hxm + alpha*mxhxm]
```

```
In [5]: %timeit evolve(t0, m0, h_ext)
```

```
10000 loops, best of 3: 18.6 µs per loop
```

This gives a rate of 50,000 evolutions per second, considering only the execution of the Landau-Liftshitz function.

The `dopri5` technique is a Runge-Kutta method of order 4(5) due to Dormand & Prince.

```
In [14]: def loop(m, h_ext, n, w):

             r = ode(evolve).set_integrator('dopri5')
             r.set_initial_value(m0, t0)
             r.set_f_params(h_ext)
```

```
    moments = np.zeros((w,3), dtype=np.float32)
    times = np.zeros((w,1), dtype=np.float32)

    i = 0

    while r.successful() and i < w*n:
        r.integrate(r.t + dt)
        if i % n == 0:
            #r.y = r.y/np.linalg.norm(r.y)
            moments[i/n], times[i/n] = r.y, r.t
        i += 1

    return times/(gamma*Ms/(1 + alpha**2.)), moments
```

```
In [15]: %timeit loop(m0, h_ext, 250, 1)
```

```
10 loops, best of 3: 67.3 ms per loop
```

This implies a rate of 3,570 evolutions per second, or 14 loops per second.

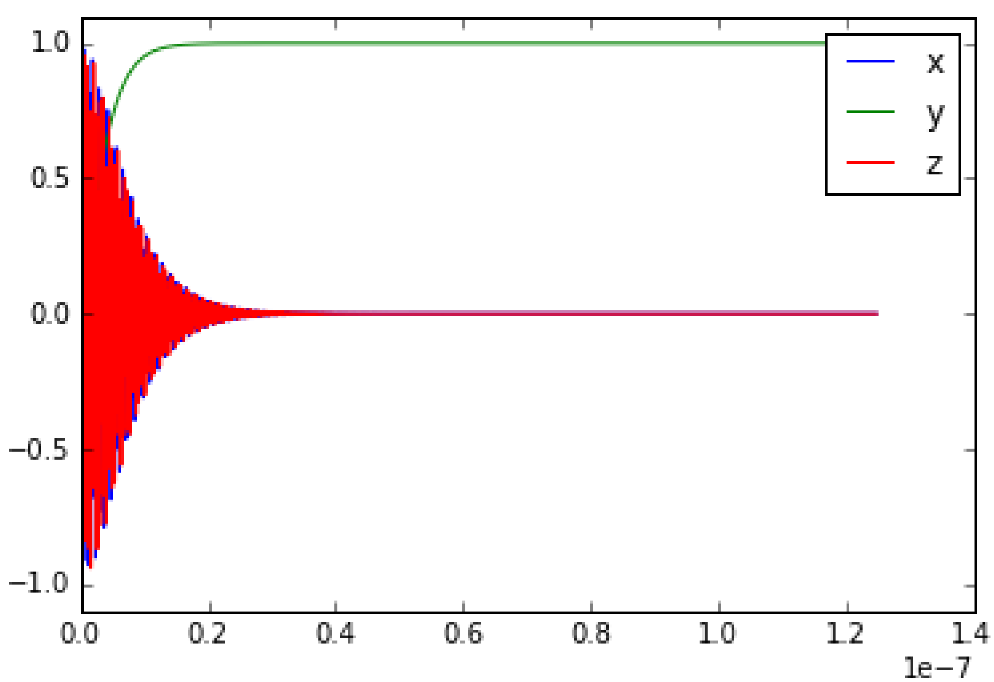
```
In [17]: %timeit loop(m0, h_ext, 250, 10)
```

```
1 loops, best of 3: 705 ms per loop
```

Ten loops requires 725 ms, which agrees with the rate expected from 1 loop.

```
In [18]: times, moments = loop(m0, h_ext, 250, 1000)
```

```
In [19]: plt.plot(times, moments[:,0], label='x')
         plt.plot(times, moments[:,1], label='y')
         plt.plot(times, moments[:,2], label='z')
         plt.ylim(-1.1, 1.1)
         plt.legend()
         plt.show()
```



The above behavior of the macrospin shows that the moment aligns to the external field as expected.

1.2 Cython Kernel

```
In [9]: %load_ext cythonmagic
```

```
In [43]: %%cython -a

         import numpy as np
         cimport numpy as np

         cdef:
             float CGS_GAMMA = 2.1*9.2740e-21/(6.6261e-27/(2.0*np.pi))
             float MS = 140 # emu/cc
             float ALPHA = 0.01
             float DT = 5e-13*CGS_GAMMA*MS/(1 + ALPHA**2.)

         def evolve(m, h_ext):
             h_eff = h_ext
             hxm = np.cross(h_eff, m)
             mxhxm = np.cross(m, hxm)
             m += DT*(hxm + ALPHA*mxhxm)

         def loop(m, h_ext, n, w):

             moments = np.zeros((w,3), dtype=np.float32)
             times = np.zeros((w,1), dtype=np.float32)

             for i in range(w):
                 for j in range(n):
                     evolve(m, h_ext)
                 m = m/np.linalg.norm(m)
                 moments[i] = m.copy()
                 times[i] = DT*n + times[i-1]

             return times/(CGS_GAMMA*MS/(1 + ALPHA**2.)), moments
```

```
Out[43]: <IPython.core.display.HTML at 0x7f1b6115f3d0>
```

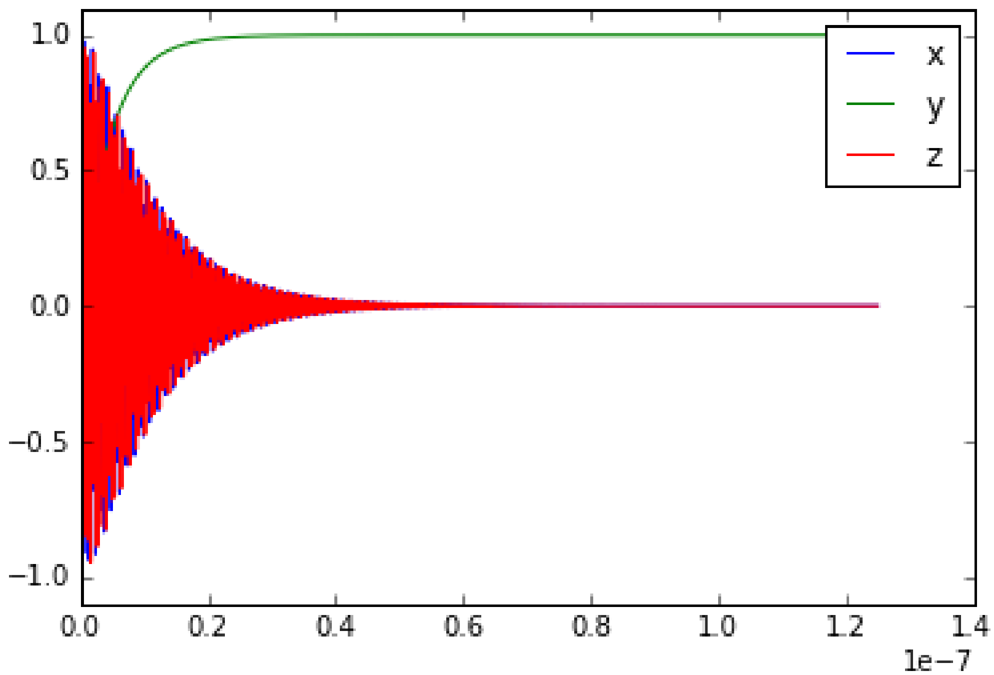
```
In [45]: Ms = 140
         m = np.array([-0.999, 0.001, 0.001], dtype=np.float32)
         h_ext = np.array([0, 1000, 0], dtype=np.float32) / Ms
```

```
In [50]: %timeit loop(m, h_ext, 250, 1)
```

```
100 loops, best of 3: 5.57 ms per loop
```

```
In [46]: times, moments = loop(m, h_ext, 250, 1000)
```

```
In [49]: plt.plot(times, moments[:,0], label='x')
         plt.plot(times, moments[:,1], label='y')
         plt.plot(times, moments[:,2], label='z')
         plt.ylim(-1.1, 1.1)
         plt.legend()
         plt.show()
```



Note that the oscillations in $\hat{x}$ and $\hat{z}$ now take longer to subside. This is likely because the Euler method is used to step the moment. The Euler method is expected to have less accuracy for the same time step as the Runga-Kutta method, however these more accurate methods requires multiple calls to the derivative function. The computational expense of calling the derivative function multiple times must be weighed against the increased timestep that can be used to achieve the same accuracy.

1.3 Improved Cython take 1

Instead of using `numpy.ndarrays` and their `memoryviews` (Cython Typed Memoryviews), vectors will be stored in a custom `float3 struct`. Since C does not allow operators to be overloaded, these `structs` require additonal functions to preform the relevant functions of addition, multiplication, cross product, etc.

```
In [44]: %%cython -a

         cimport cython
         import numpy as np
         cimport numpy as np
         from libc.math cimport sqrt

         cdef:
             float CGS_GAMMA = 2.1*9.2740e-21/(6.6261e-27/(2.0*np.pi))
             float MS = 140 # emu/cc
             float ALPHA = 0.01
             float DT = 5e-13*CGS_GAMMA*MS/(1 + ALPHA**2.)

         ctypedef struct float3:
             float x, y, z

         @cython.boundscheck(False)
```

```
@cython.wraparound(False)
cpdef float3 make_float3(float[::1] m):
    cdef float3 r
    r.x, r.y, r.z = m[0], m[1], m[2]
    return r

cdef float3 add(float3 a, float3 b):
    cdef float3 c
    c.x = a.x + b.x
    c.y = a.y + b.y
    c.z = a.z + b.z
    return c

cdef float3 cross(float3 a, float3 b):
    cdef float3 c
    c.x = a.y*b.z - a.z*b.y
    c.y = a.z*b.x - a.x*b.z
    c.z = a.x*b.y - a.y*b.x
    return c

cdef float3 mult(float3 a, float3 b):
    cdef float3 c
    c.x = a.x * b.x
    c.y = a.y * b.y
    c.z = a.z * b.z
    return c

cdef float3 times(float a, float3 b):
    cdef float3 c
    c.x = a*b.x
    c.y = a*b.y
    c.z = a*b.z
    return c

@cython.cdivision(True)
cdef float3 unit(float3 a):
    cdef:
        float mag
        float3 c
    mag = sqrt(a.x*a.x + a.y*a.y + a.z*a.z)
    c.x = a.x/mag
    c.y = a.y/mag
    c.z = a.z/mag
    return c

cdef evolve(float3 m, float3 h_ext):
    h_eff = h_ext
    hxm = cross(h_eff, m)
    mxhxm = cross(m, hxm)
    return add(m, times(DT, add(hxm, times(ALPHA, mxhxm))))

def loop(m, h_ext, int n, int w):

    moments = np.zeros((w,3), dtype=np.float32)
```

```
    times = np.zeros((w,1), dtype=np.float32)

    cdef:
        int i, j
        float3 mf3 = make_float3(m)
        float3 hf3 = make_float3(h_ext)

    for i in range(w):
        for j in range(n):
            mf3 = evolve(mf3, hf3)
        mf3 = unit(mf3) # normalize
        moments[i] = [mf3.x, mf3.y, mf3.z]
        times[i] = DT*n + times[i-1]

    return times/(CGS_GAMMA*MS/(1 + ALPHA**2.)), moments
```

```
Out[44]: <IPython.core.display.HTML at 0x7f1b61234950>
```

Comparing the annotation with the initial Cython kernel illustrates that few of the lines take significant calls in the body of the loop.

```
In [45]: Ms = 140
         m = np.array([-0.999, 0.001, 0.001], dtype=np.float32)
         h_ext = np.array([0, 1000, 0], dtype=np.float32) / Ms
```

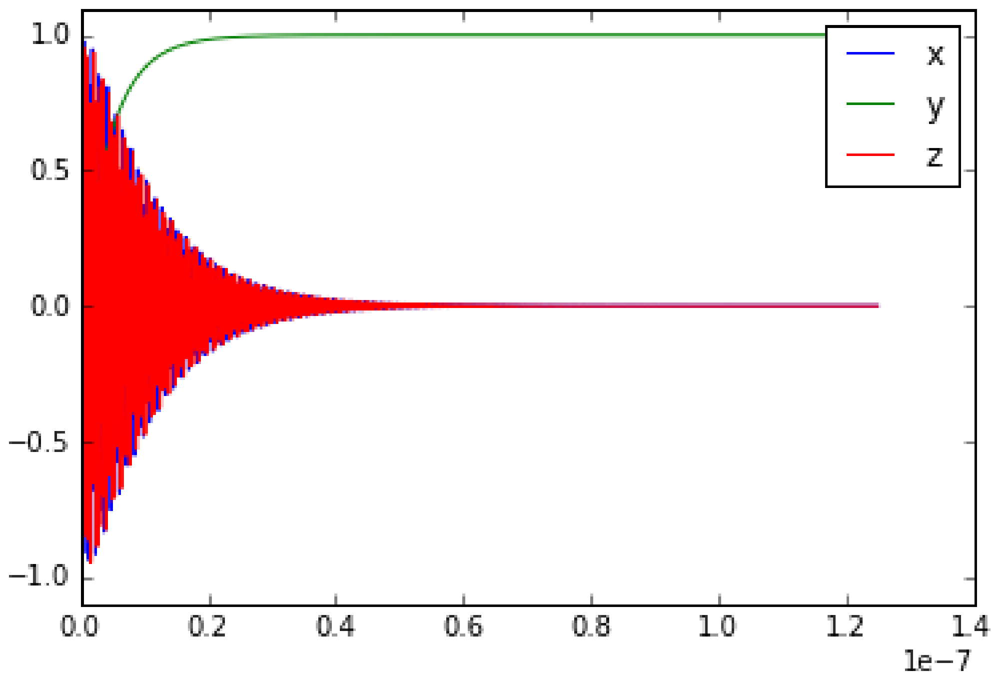
```
In [46]: %timeit loop(m, h_ext, 250, 1)
```

```
10000 loops, best of 3: 39.2 µs per loop
```

This yeilds a rate of 6.25 million evolutions per second, or 25,000 loops per second. Compared to the pure Python rate of 14 loops per second, this is a 1785x improvement. Now 1,000 loops take 40 ms.

```
In [41]: times, moments = loop(m, h_ext, 250, 1000)
```

```
In [42]: plt.plot(times, moments[:,0], label='x')
         plt.plot(times, moments[:,1], label='y')
         plt.plot(times, moments[:,2], label='z')
         plt.ylim(-1.1, 1.1)
         plt.legend()
         plt.show()
```



The expected behavior of the oscillations is reproduced, that matches the previous Cython example using the Euler method.

1.3.1 Disadvantages of this approach

Despite the significant speed increase, this kernel is verbose from the vector functions needed to preform operations on the `float3 struct`. This is because C does not support vector operations intrinsicly and does not allow overloading of the operators.

One solution is to use a C++ struct or class to define the vector, or load one from an existing C++ library like ROOT (CERN).

1.4 Using a C++ Struct

The C++ `struct` can overload its operation methods (+,-,/,etc.). The `float3 struct` is extended in C++ to allow easy vector operations inside the `evolve` function.

The `cythonmagic` does not use the `g++` compiler when using C++. Its not clear at this point if that is an error in usage or an issue with `cythonmagic`.

To work around this point, the first option is to manually set the C compiler to `g++`. This method may disturb the rest of the C Cython.

import os os.environ["CC"] = "g++" os.environ["CXX"] = "g++"

1.5 Method for compiling C++ in IPython

Note: This is a work-around given the state of `cythonmagic`.

Instead of doing this, the Cython code is written to a `.pyx` file. Cython is called from bash (in the notebook). This creates the `.cpp` file, which is compiled with `g++` into a shared object file (`.so`). A shared object file can be imported directly into Python. To ensure that the `.pyx` file does not interfere with this import, it is removed. The code still exists in the notebook.

```
cython --cplus example.pyx
g++ -shared -I/usr/include/python2.7 -fPIC -o example.so example.cpp
rm example.pyx
```

```
In [8]: %%writefile float3.h

        #include <math.h>

        struct float3 {
            float x, y, z;

            float3 operator+(const float3 o) {
                float3 r;
                r.x = x+o.x;
                r.y = y+o.y;
                r.z = z+o.z;
                return r;
            }
        };
```

```
Overwriting float3.h
```

```
In [8]: %%writefile hello_float3.pyx
        # distutils: language = c++

        cdef extern from "float3.h":
            cdef cppclass float3:
                float x, y, z
                float3 operator+(float3)

        def hello_float3():
            cdef float3 v, u
            v.x, v.y, v.z = 1.0, 2.0, 3.0
            u.x, u.y, u.z = 1.0, 1.0, 1.0

            print "u:", u.x, u.y, u.z
            print "v:", v.x, v.y, v.z

            cdef float3 r = u + v
            print "u + v:", r.x, r.y, r.z
```

```
Writing hello_float3.pyx
```

```
In [9]: %%bash
        cython --cplus hello_float3.pyx
        g++ -shared -I/usr/include/python2.7 -fPIC -o hello_float3.so hello_float3.cpp
        rm hello_float3.pyx
```

```
In [1]: from hello_float3 import hello_float3
        hello_float3()
```

```
u: 1.0 1.0 1.0
v: 1.0 2.0 3.0
u + v: 2.0 3.0 4.0
```

The vectors are observed to add correctly.

1.6 Overloading the C++ struct operators

Given the operations can be overloaded correctly, vector operations are defined for `float3`.

In [9]:
```
%%writefile float3.h

#include <math.h>

struct float3 {
    float x, y, z;

    float3(float x, float y, float z)
        : x(x), y(y), z(z)
    {}

    float3()
        : x(0.0), y(0.0), z(0.0)
    {}

    float3 operator+(const float3 o) {
        return float3(x+o.x, y+o.y, z+o.z);
    }

    float3 operator-(const float3 o) {
        return float3(x-o.x, y-o.y, z-o.z);
    }

    float3 operator*(const float3 o) {
        return float3(x*o.x, y*o.y, z*o.z);
    }

    float3 operator*(const float o) {
        return float3(x*o, y*o, z*o);
    }

    float3 operator/(const float o) {
        return float3(x/o, y/o, z/o);
    }

    float3 cross(const float3& o) {
        return float3(
            y*o.z - z*o.y,
            z*o.x - x*o.z,
            x*o.y - y*o.x
        );
    }

    float dot(const float3& o) {
        return x*o.x + y*o.y + z*o.z;
    }

    float mag() {
        return sqrt(x*x + y*y + z*z);
    }
```

```
    void normalize() {
        float mag = this->mag();
        x = x/mag;
        y = y/mag;
        z = z/mag;
    }

};
```

Overwriting float3.h

In [2]:
```
%%writefile hello_float3.pyx
# distutils: language = c++

import numpy as np
cimport numpy as np

cdef extern from "float3.h":
    cdef cppclass float3:
        float3(x, y, z)
        float3()
        float x, y, z
        float3 operator+(float3)
        float3 operator-(float3)
        float3 operator*(float3)
        float3 operator*(float)
        float3 operator/(float)
        float3 cross(float3)
        float dot(float3)
        float mag()
        void normalize()

cdef float3 make_float3(float[::1] l):
    cdef float3 r
    r.x, r.y, r.z = l[0], l[1], l[2]
    return r

def hello_float3():
    u_np = np.array([1, 2, 3], dtype=np.float32)
    v_np = np.array([1, 1, 1], dtype=np.float32)

    cdef:
        float3 v = make_float3(v_np)
        float3 u = make_float3(u_np)
        float3 r

    print "u:", u.x, u.y, u.z
    print "v:", v.x, v.y, v.z

    r = u + v
    print "u+v:", r.x, r.y, r.z
    print ">>", u_np+v_np, "\n"

    r = u - v
    print "u-v:", r.x, r.y, r.z
```

```
        print ">>", u_np-v_np, "\n"

        f = u.dot(v)
        print "u dot v:", f
        print ">>", u_np.dot(v_np), "\n"

        r = u.cross(v)
        print "u cross v:", r.x, r.y, r.z
        print ">>", np.cross(u_np, v_np), "\n"

        r = u/2.
        print "u/2:", r.x, r.y, r.z
        print ">>", u_np/2., "\n"

        r = u*2.
        print "u*2:", r.x, r.y, r.z
        print ">>", u_np*2., "\n"

        r = u*v
        print "u*v:", r.x, r.y, r.z
        print ">>", np.multiply(u_np, v_np), "\n"

        u.normalize()
        print "normal u:", u.x, u.y, u.z
        print ">>", u_np/np.linalg.norm(u_np), "\n"
```

```
Writing hello_float3.pyx
```

```
In [3]: %%bash
        cython --cplus hello_float3.pyx
        g++ -shared -I/usr/include/python2.7 -fPIC -o hello_float3.so hello_float3.cpp
        rm hello_float3.pyx
```

```
In file included from /usr/include/python2.7/numpy/ndarraytypes.h:1761:0,
                 from /usr/include/python2.7/numpy/ndarrayobject.h:17,
                 from /usr/include/python2.7/numpy/arrayobject.h:4,
                 from hello_float3.cpp:239:
/usr/include/python2.7/numpy/npy_1_7_deprecated_api.h:15:2: warning: #warning "Using deprecated NumPy API
 #warning "Using deprecated NumPy API, disable it by " \
  ^
```

```
In [1]: from hello_float3 import hello_float3
        hello_float3()
```

```
u: 1.0 2.0 3.0
v: 1.0 1.0 1.0
u+v: 2.0 3.0 4.0
>> [ 2.  3.  4.]

u-v: 0.0 1.0 2.0
>> [ 0.  1.  2.]

u dot v: 6.0
>> 6.0
```

```
u cross v: -1.0 2.0 -1.0
>> [-1.  2. -1.]

u/2: 0.5 1.0 1.5
>> [ 0.5  1.   1.5]

u*2: 2.0 4.0 6.0
>> [ 2.  4.  6.]

u*v: 1.0 2.0 3.0
>> [ 1.  2.  3.]

normal u: 0.267261236906 0.534522473812 0.801783680916
>> [ 0.26726124  0.53452247  0.80178368]
```

This demonstrates that `float3` operations are working in accordance to expectations of `numpy.ndarrays`. Note that the multiplication and division with `floats` have only been defined for the case when the `float` is on the left hand side of the `float3`.

1.7 Cython Kernel 2

The Landau-Liftshitz kernel is re-written with the use of the C++ `float3 struct`.

In [8]:
```
%%writefile llg_float3.pyx
# distutils: language = c++

cimport cython
import numpy as np
cimport numpy as np

cdef:
    float CGS_GAMMA = 2.1*9.2740e-21/(6.6261e-27/(2.0*np.pi))
    float MS = 140 # emu/cc
    float ALPHA = 0.01
    float DT = 5e-13*CGS_GAMMA*MS/(1 + ALPHA**2.)

cdef extern from "float3.h":
    cdef cppclass float3:
        float3(x, y, z)
        float3()
        float x, y, z
        float3 operator+(float3)
        float3 operator-(float3)
        float3 operator*(float3)
        float3 operator*(float)
        float3 operator/(float)
        float3 cross(float3)
        float dot(float3)
        float mag()
        void normalize()

@cython.boundscheck(False)
@cython.wraparound(False)
cdef float3 make_float3(float[::1] l):
    cdef float3 r
```

```
        r.x, r.y, r.z = l[0], l[1], l[2]
        return r

    cdef float3 evolve(float3 m, float3 h_ext):
        h_eff = h_ext
        hxm = h_eff.cross(m)
        mxhxm = m.cross(hxm)
        return m + (hxm + mxhxm*ALPHA)*DT

    def loop(m_np, h_ext_np, int n, int w):

        moments = np.zeros((w,3), dtype=np.float32)
        times = np.zeros((w,1), dtype=np.float32)

        cdef:
            int i, j
            float3 m = make_float3(m_np)
            float3 h_ext = make_float3(h_ext_np)

        for i in range(w):
            for j in range(n):
                m = evolve(m, h_ext)
            m.normalize()
            moments[i] = [m.x, m.y, m.z]
            times[i] = DT*n + times[i-1]

        return times/(CGS_GAMMA*MS/(1 + ALPHA**2.)), moments
```

```
Writing llg_float3.pyx
```

In [9]:
```
%%bash
cython --cplus llg_float3.pyx
g++ -shared -I/usr/include/python2.7 -fPIC -o llg_float3.so llg_float3.cpp
rm llg_float3.pyx
```

```
In file included from /usr/include/python2.7/numpy/ndarraytypes.h:1761:0,
                 from /usr/include/python2.7/numpy/ndarrayobject.h:17,
                 from /usr/include/python2.7/numpy/arrayobject.h:4,
                 from llg_float3.cpp:239:
/usr/include/python2.7/numpy/npy_1_7_deprecated_api.h:15:2: warning: #warning "Using deprecated NumPy API
 #warning "Using deprecated NumPy API, disable it by " \
  ^
```

In [1]:
```
from llg_float3 import loop
```

In [4]:
```
Ms = 140
m = np.array([-0.999, 0.001, 0.001], dtype=np.float32)
h_ext = np.array([0, 1000, 0], dtype=np.float32) / Ms
```

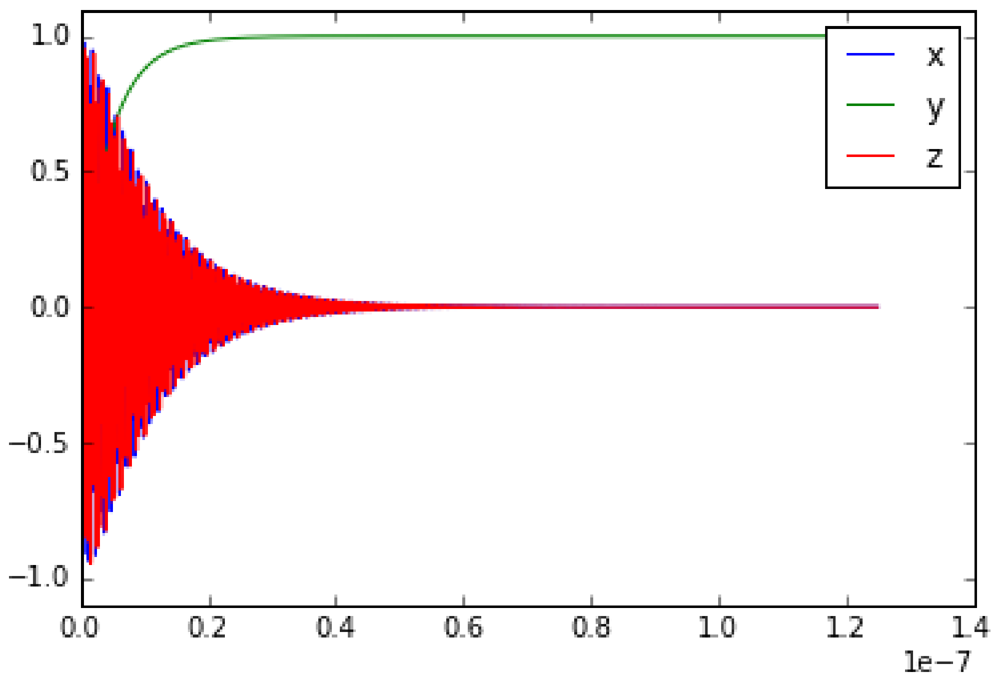
In [5]:
```
%timeit loop(m, h_ext, 250, 1)
```

```
10000 loops, best of 3: 46.5 µs per loop
```

This is 13% slower than the pure C method of using the float3 struct. The ease of working with the LL equation in this form is certainly worth that price.

In [6]:
```
times, moments = loop(m, h_ext, 250, 1000)
```

```
In [7]: plt.plot(times, moments[:,0], label='x')
        plt.plot(times, moments[:,1], label='y')
        plt.plot(times, moments[:,2], label='z')
        plt.ylim(-1.1, 1.1)
        plt.legend()
        plt.show()
```



2 Conclusions

- The Euler method yeilds a different settle time than the Runga-Kutta Dormand-Prince method
- The Cython annotate tool is extremely useful for examining efficiency
- The use of Cython improves the speed of the LL algorithm by over 1000x
- The use of an overloaded C++ struct provides a means for vector operations to be easily used in Cython
- There are issues using C++ with `cythonmagic`

3 Next Steps

- The Landau-Lifshitz kernel will be written in Cython to include demagnetization, anisotropy, and spin transfer torque