Speedup scipy griddata for multiple interpolations between two irregular grids

Asked 8 years, 8 months ago Modified 2 years, 5 months ago Viewed 12k times

29

22

I have several values that are defined on the same irregular grid `(x, y, z)` that I want to interpolate onto a new grid `(x1, y1, z1)` . i.e., I have `f(x, y, z), g(x, y, z), h(x, y, z)` and I want to calculate `f(x1, y1, z1), g(x1, y1, z1), h(x1, y1, z1)` .

At the moment I am doing this using `scipy.interpolate.griddata` and it works well. However, because I have to perform each interpolation separately and there are many points, it is quite slow, with a great deal of duplication in the calculation (i.e finding which points are closest, setting up the grids etc...).

Is there a way to speedup the calculation and reduce the duplicated calculations? i.e something along the lines of defining the two grids, then changing the values for the interpolation?

python numpy scipy interpolation qhull

Share Improve this question Follow

edited Jun 21, 2017 at 20:53
user812786
4,102 3 40 50

asked Jan 4, 2014 at 1:02
s_haskey
479 1 5 8

What interpolation method are you using, i.e. `nearest` , `linear` ...? Also, how many points do you have in your irregular grid? – Jaime Jan 4, 2014 at 4:10

I'm using linear interpolation (nearest would not be good enough). The original grid (x,y,z) consists of 3.5million points. The new grid (x1,y1,z1) consists of about 300,000 points. The linear interpolation takes ~30s on a laptop with an i7 processor with a healthy amount of RAM. I have 6 sets of values to interpolate, so this is a major bottleneck for me. – s_haskey Jan 4, 2014 at 7:14

4 Answers

Sorted by: Highest score (default)

53

There are several things going on every time you make a call to `scipy.interpolate.griddata` :

1. First, a call to `sp.spatial.qhull.Delaunay` is made to triangulate the irregular grid coordinates.
2. Then, for each point in the new grid, the triangulation is searched to find in which triangle (actually, in which simplex, which in your 3D case will be in which tetrahedron) does it lay.

3. The barycentric coordinates of each new grid point with respect to the vertices of the enclosing simplex are computed.
4. An interpolated values is computed for that grid point, using the barycentric coordinates, and the values of the function at the vertices of the enclosing simplex.

The first three steps are identical for all your interpolations, so if you could store, for each new grid point, the indices of the vertices of the enclosing simplex and the weights for the interpolation, you would minimize the amount of computations by a lot. This is unfortunately not easy to do directly with the functionality available, although it is indeed possible:

```
import scipy.interpolate as spint
import scipy.spatial.qhull as qhull
import itertools

def interp_weights(xyz, uvw):
    tri = qhull.Delaunay(xyz)
    simplex = tri.find_simplex(uvw)
    vertices = np.take(tri.simplices, simplex, axis=0)
    temp = np.take(tri.transform, simplex, axis=0)
    delta = uvw - temp[:, d]
    bary = np.einsum('njk,nk->nj', temp[:, :d, :], delta)
    return vertices, np.hstack((bary, 1 - bary.sum(axis=1, keepdims=True)))

def interpolate(values, vtx, wts):
    return np.einsum('nj,nj->n', np.take(values, vtx), wts)
```

The function `interp_weights` does the calculations for the first three steps I listed above. Then the function `interpolate` uses those calcualted values to do step 4 very fast:

```
m, n, d = 3.5e4, 3e3, 3
# make sure no new grid point is extrapolated
bounding_cube = np.array(list(itertools.product([0, 1], repeat=d)))
xyz = np.vstack((bounding_cube,
                 np.random.rand(m - len(bounding_cube), d)))
f = np.random.rand(m)
g = np.random.rand(m)
uvw = np.random.rand(n, d)

In [2]: vtx, wts = interp_weights(xyz, uvw)

In [3]: np.allclose(interpolate(f, vtx, wts), spint.griddata(xyz, f, uvw))
Out[3]: True

In [4]: %timeit spint.griddata(xyz, f, uvw)
1 loops, best of 3: 2.81 s per loop

In [5]: %timeit interp_weights(xyz, uvw)
1 loops, best of 3: 2.79 s per loop

In [6]: %timeit interpolate(f, vtx, wts)
10000 loops, best of 3: 66.4 us per loop

In [7]: %timeit interpolate(g, vtx, wts)
10000 loops, best of 3: 67 us per loop
```

So first, it does the same as `griddata`, which is good. Second, setting up the interpolation, i.e. computing `vtx` and `wts` takes roughly the same as a call to `griddata`. But third, you can now interpolate for different values on the same grid in virtually no time.

The only thing that `griddata` does that is not contemplated here is assigning `fill_value` to points that have to be extrapolated. You could do that by checking for points for which at least one of the weights is negative, e.g.:

```
def interpolate(values, vtx, wts, fill_value=np.nan):
    ret = np.einsum('nj,nj->n', np.take(values, vtx), wts)
    ret[np.any(wts < 0, axis=1)] = fill_value
    return ret
```

Share Improve this answer Follow

edited Mar 23, 2018 at 19:05
unutbu
796k 170 1720 1622

answered Jan 5, 2014 at 6:41
Jaime
63.6k 17 119 158

3 Perfect, exactly what I was after! Thanks very much. It would be nice if this sort of functionality was included in scipy for future versions of griddata. – s_haskey Jan 6, 2014 at 23:59

works very well for me! It also uses much less memory than scipy.itnerpolate.griddata when run several times on my machine. – Matthias123 Jul 4, 2014 at 9:34

1 Also, `griddata` accommodates missing values/holes in the function - `nan`, which does not work with this solution? – FooBar Mar 26, 2017 at 9:59

@Jaime if I would like to update the data with additional points, I can use `tri = qhull.Delaunay(xy, incremental=True)` and alter `tri.add_points(xy2)` to speed up the delaunay part, do you have any Idea on how to speed up the `find_simplex` to only cover the updated indices? – Merlin Jul 13, 2017 at 10:52

2 how would one use a cubic interpolation (which for `griddata` is just a keyword)? – John Smith Sep 9, 2019 at 18:34

6

Great thanks to Jaime for his solution (even if I don't really understand how the barycentric computation is done ...)

Here you will find an example adapted from his case in 2D :

```
import scipy.interpolate as spint
import scipy.spatial.qhull as qhull
import numpy as np

def interp_weights(xy, uv,d=2):
    tri = qhull.Delaunay(xy)
    simplex = tri.find_simplex(uv)
    vertices = np.take(tri.simplices, simplex, axis=0)
    temp = np.take(tri.transform, simplex, axis=0)
    delta = uv - temp[:, d]
```

```
    bary = np.einsum('njk,nk->nj', temp[:, :d, :], delta)
    return vertices, np.hstack((bary, 1 - bary.sum(axis=1, keepdims=True)))

def interpolate(values, vtx, wts):
    return np.einsum('nj,nj->n', np.take(values, vtx), wts)

m, n = 101,201
mi, ni = 1001,2001

[Y,X]=np.meshgrid(np.linspace(0,1,n),np.linspace(0,2,m))
[Yi,Xi]=np.meshgrid(np.linspace(0,1,ni),np.linspace(0,2,mi))

xy=np.zeros([X.shape[0]*X.shape[1],2])
xy[:,0]=Y.flatten()
xy[:,1]=X.flatten()
uv=np.zeros([Xi.shape[0]*Xi.shape[1],2])
uv[:,0]=Yi.flatten()
uv[:,1]=Xi.flatten()

values=np.cos(2*X)*np.cos(2*Y)

#Computed once and for all !
vtx, wts = interp_weights(xy, uv)
valuesi=interpolate(values.flatten(), vtx, wts)
valuesi=valuesi.reshape(Xi.shape[0],Xi.shape[1])
print "interpolation error: ",np.mean(valuesi-np.cos(2*Xi)*np.cos(2*Yi))
print "interpolation uncertainty: ",np.std(valuesi-np.cos(2*Xi)*np.cos(2*Yi))
```

It is possible to applied image transformation such as image mapping with a udge speed-up

You can't use the same function definition as the new coordinates will change at every iteration but you can compute triangulation Once for all.

```
import scipy.interpolate as spint
import scipy.spatial.qhull as qhull
import numpy as np
import time

# Definition of the fast  interpolation process. May be the Tirangulation process can
be removed !!
def interp_tri(xy):
    tri = qhull.Delaunay(xy)
    return tri


def interpolate(values, tri,uv,d=2):
    simplex = tri.find_simplex(uv)
    vertices = np.take(tri.simplices, simplex, axis=0)
    temp = np.take(tri.transform, simplex, axis=0)
    delta = uv- temp[:, d]
    bary = np.einsum('njk,nk->nj', temp[:, :d, :], delta)
    return np.einsum('nj,nj->n', np.take(values, vertices),  np.hstack((bary, 1.0 -
bary.sum(axis=1, keepdims=True))))

m, n = 101,201
mi, ni = 101,201

[Y,X]=np.meshgrid(np.linspace(0,1,n),np.linspace(0,2,m))
[Yi,Xi]=np.meshgrid(np.linspace(0,1,ni),np.linspace(0,2,mi))
```

```
xy=np.zeros([X.shape[0]*X.shape[1],2])
xy[:,1]=Y.flatten()
xy[:,0]=X.flatten()
uv=np.zeros([Xi.shape[0]*Xi.shape[1],2])
# creation of a displacement field
uv[:,1]=0.5*Yi.flatten()+0.4
uv[:,0]=1.5*Xi.flatten()-0.7
values=np.zeros_like(X)
values[50:70,90:150]=100.

#Computed once and for all !
tri = interp_tri(xy)
t0=time.time()
for i in range(0,100):

values_interp_Qhull=interpolate(values.flatten(),tri,uv,2).reshape(Xi.shape[0],Xi.shape[1

t_q=(time.time()-t0)/100

t0=time.time()
values_interp_griddata=spint.griddata(xy,values.flatten(),uv,fill_value=0).reshape(values

t_g=time.time()-t0

print "Speed-up:", t_g/t_q
print "Mean error: ",(values_interp_Qhull-values_interp_griddata).mean()
print "Standard deviation: ",(values_interp_Qhull-values_interp_griddata).std()
```

On my laptop the speed-up is between 20 and 40x !

Hope that can help someone

Share Improve this answer Follow

edited Nov 6, 2015 at 14:00
christopherlovell
3,402 3 16 24

answered Aug 11, 2015 at 12:55
Jeff Witz
61 1 3

`interp_weights` function fails here, `delta = uv - temp[:, d]`, since `d` is out of bounds on `temp`
– christopherlovell Nov 6, 2015 at 13:15

3

I had the same problem (griddata extremely slow, grid stays the same for many interpolations) and I liked the solution described here the best, mainly because it is very easy to understand and apply.

It is using the `LinearNDInterpolator`, where one can pass the Delaunay triangulation that needs to be computed only once. Copy & paste from that post (all credits to xdze2):

```
from scipy.spatial import Delaunay
from scipy.interpolate import LinearNDInterpolator

tri = Delaunay(mesh1)  # Compute the triangulation
```

```
# Perform the interpolation with the given values:
interpolator = LinearNDInterpolator(tri, values_mesh1)
values_mesh2 = interpolator(mesh2)
```

That speeds up my computations by a factor of approximately 2.

Share Improve this answer Follow edited Jun 17, 2019 at 10:04 answered Jun 17, 2019 at 9:55

Waterkant
253 2 9

0

You can try to use Pandas, as it provides high-performance data structures.

It is true that the interpolation method is a **wrapper of the scipy interpolation** BUT maybe with the improved structures you obtain better speed.

```
import pandas as pd;
wp = pd.Panel(randn(2, 5, 4));
wp.interpolate();
```

`interpolate()` fills the NaN values in the Panel dataset using different methods. Hope it is faster than Scipy.

If it doesn't work, there is one way to improve the performance (instead of using a parallelized version of your code): use Cython and implement small routine in C to use inside your Python code. Here you have an example about this.

Share Improve this answer Follow edited Jan 4, 2014 at 10:52 answered Jan 4, 2014 at 10:45

phyrox
2,403 14 23