Fast interpolation of grid data

Asked 11 years, 6 months ago Modified 11 years, 3 months ago

Viewed 24k times

21

I have a large 3d np.ndarray of data that represents a physical variable sampled over a volume in a regular grid fashion (as in the value in array[0,0,0] represents the value at physical coords (0,0,0)).

I would like to go to a finer grid spacing by interpolating the data in the rough grid. At the moment I'm using scipy griddata linear interpolation but it's pretty slow (~90secs for 20x20x20 array). It's a bit overengineered for my purposes, allowing random sampling of the volume data. Is there anything out there that can take advantage of my regularly spaced data and the fact that there is only a limited set of specific points I want to interpolate to?

python numpy scipy interpolation

Share

Improve this question

Follow

edited Jul 1, 2013 at 15:00

denis

21.9k ● 12 ● 67 ● 90

asked Jun 7, 2013 at 12:12

Rowan

277 ● 1 ● 2 ● 6

fast-interpolation-of-regularly-sampled-3d-data-with-different-intervals-in-x-y also uses map_coordinates – denis Jul 1, 2013 at 14:57

2 Answers

Sorted by: Highest score (default)

39

Sure! There are two options that do different things but both exploit the regularly-gridded nature of the original data.

The first is `scipy.ndimage.zoom`. If you just want to produce a denser regular grid based on interpolating the original data, this is the way to go.

The second is `scipy.ndimage.map_coordinates`. If you'd like to interpolate a few (or many) arbitrary points in your data, but still exploit the regularly-gridded nature of the original data (e.g. no quadtree required), it's the way to go.

"Zooming" an array (`scipy.ndimage.zoom`)

As a quick example (This will use cubic interpolation. Use `order=1` for bilinear, `order=0` for nearest, etc.):

```
import numpy as np
import scipy.ndimage as ndimage

data = np.arange(9).reshape(3,3)
```

```
print 'Original:\n', data
print 'Zoomed by 2x:\n', ndimage.zoom(data, 2)
```

This yields:

```
Original:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Zoomed by 2x:
[[0 0 1 1 2 2]
 [1 1 1 2 2 3]
 [2 2 3 3 4 4]
 [4 4 5 5 6 6]
 [5 6 6 7 7 7]
 [6 6 7 7 8 8]]
```

This also works for 3D (and nD) arrays. However, be aware that if you zoom by 2x, for example, you'll zoom along *all* axes.

```
data = np.arange(27).reshape(3,3,3)
print 'Original:\n', data
print 'Zoomed by 2x gives an array of shape:', ndimage
```

This yields:

```
Original:
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
```

```
  [21 22 23]
  [24 25 26]]]
Zoomed by 2x gives an array of shape: (6, 6, 6)
```

If you have something like a 3-band, RGB image that you'd like to zoom, you can do this by specifying a sequence of tuples as the zoom factor:

```
print 'Zoomed by 2x along the last two axes:'
print ndimage.zoom(data, (1, 2, 2))
```

This yields:

```
Zoomed by 2x along the last two axes:
[[[ 0  0  1  1  2  2]
  [ 1  1  1  2  2  3]
  [ 2  2  3  3  4  4]
  [ 4  4  5  5  6  6]
  [ 5  6  6  7  7  7]
  [ 6  6  7  7  8  8]]

 [[ 9  9 10 10 11 11]
  [10 10 10 11 11 12]
  [11 11 12 12 13 13]
  [13 13 14 14 15 15]
  [14 15 15 16 16 16]
  [15 15 16 16 17 17]]

 [[18 18 19 19 20 20]
  [19 19 19 20 20 21]
  [20 20 21 21 22 22]
  [22 22 23 23 24 24]
  [23 24 24 25 25 25]
  [24 24 25 25 26 26]]]
```

Arbitrary interpolation of regularly-gridded data using `map_coordinates`

The first thing to undersand about `map_coordinates` is that it operates in *pixel* coordinates (e.g. just like you'd index the array, but the values can be floats). From your description, this is exactly what you want, but if often confuses people. For example, if you have x, y, z "real-world" coordinates, you'll need to transform them to index-based "pixel" coordinates.

At any rate, let's say we wanted to interpolate the value in the original array at position 1.2, 0.3, 1.4.

If you're thinking of this in terms of the earlier RGB image case, the first coordinate corresponds to the "band", the second to the "row" and the last to the "column". What order corresponds to what depends entirely on how you decide to structure your data, but I'm going to use these as "z, y, x" coordinates, as it makes the comparison to the printed array easier to visualize.

```
import numpy as np
import scipy.ndimage as ndimage

data = np.arange(27).reshape(3,3,3)

print 'Original:\n', data
print 'Sampled at 1.2, 0.3, 1.4:'
print ndimage.map_coordinates(data, [[1.2], [0.3], [1.
```

This yields:

```
Original:
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
Sampled at 1.2, 0.3, 1.4:
[14]
```

Once again, this is cubic interpolation by default. Use the `order` kwarg to control the type of interpolation.

It's worth noting here that all of `scipy.ndimage`'s operations preserve the dtype of the original array. If you want floating point results, you'll need to cast the original array as a float:

```
In [74]: ndimage.map_coordinates(data.astype(float), [
Out[74]: array([ 13.5965])
```

Another thing you may notice is that the interpolated coordinates format is rather cumbersome for a single point (e.g. it expects a 3xN array instead of an Nx3 array). However, it's arguably nicer when you have sequences of coordinate. For example, consider the case of sampling along a line that passes through the "cube" of data:

```
xi = np.linspace(0, 2, 10)
yi = 0.8 * xi
zi = 1.2 * xi
print ndimage.map_coordinates(data, [zi, yi, xi])
```

This yields:

```
[ 0  1  4  8 12 17 21 24  0  0]
```

This is also a good place to mention how boundary conditions are handled. By default, anything outside of the array is set to 0. Thus the last two values in the sequence are `0`. (i.e. `zi` is > 2 for the last two elements).

If we wanted the points outside the array to be, say `-999` (We can't use `nan` as this is an integer array. If you want `nan`, you'll need to cast to floats.):

```
In [75]: ndimage.map_coordinates(data, [zi, yi, xi], c
Out[75]: array([   0,    1,    4,    8,   12,   17,
```

If we wanted it to return the nearest value for points outside the array, we'd do:

```
In [76]: ndimage.map_coordinates(data, [zi, yi, xi], m
Out[76]: array([ 0,  1,  4,  8, 12, 17, 21, 24, 25, 25
```

You can also use `"reflect"` and `"wrap"` as boundary modes, in addition to `"nearest"` and the default

`"constant"`. These are fairly self-explanatory, but try experimenting a bit if you're confused.

For example, let's interpolate a line along the first row of the first band in the array that extends for twice the distance of the array:

```
xi = np.linspace(0, 5, 10)
yi, zi = np.zeros_like(xi), np.zeros_like(xi)
```

The default give:

```
In [77]: ndimage.map_coordinates(data, [zi, yi, xi])
Out[77]: array([0, 0, 1, 2, 0, 0, 0, 0, 0, 0])
```

Compare this to:

```
In [78]: ndimage.map_coordinates(data, [zi, yi, xi], m
Out[78]: array([0, 0, 1, 2, 2, 1, 2, 1, 0, 0])

In [78]: ndimage.map_coordinates(data, [zi, yi, xi], m
Out[78]: array([0, 0, 1, 2, 0, 1, 1, 2, 0, 1])
```

Hopefully that clarifies things a bit!

Share Improve this answer Follow

edited Jun 7, 2013 at 15:13

answered Jun 7, 2013 at 12:24

Joe Kington
284k ● 72 ● 617 ● 472

1 Thorough answer! Thanks for the tip on the map_coordinates function coordinates, had to look twice to figure that one out. – Rowan Jun 7, 2013 at 15:51

7

Great answer by Joe. Based on his suggestion, I created the **regulargrid** package (https://pypi.python.org/pypi/regulargrid/, source at https://github.com/JohannesBuchner/regulargrid)

It provides support for n-dimensional Cartesian grids (as needed here) via the very fast scipy.ndimage.map_coordinates for arbitrary coordinate scales.

Share Improve this answer Follow

answered Sep 20, 2013 at 9:59

j13r

2,661 ● 2 ● 22 ● 29