Subject: the fastest way to find number of points in sphere(radius r)
Posted by snfinder@naver.com on Tue, 22 Nov 2005 08:16:35 GMT
View Forum Message <> Reply to Message

Dear all,

First of all, my data is three-dimensional set.
I have to use a loop because I have a lot of positions of centers of
spheres that I should examine.
I want to find the # of points inside each sphere.
Now, I use a where function in order to find points inside the cube,
then I compute distances of all. Next, I use where function again to
examine # of distances less than radius of sphere.
I think it is fairly slow when large data is considered.

Is there any faster way?
The fastest way to find the number of points in sphere(radius r)

---
data:
positions of points: X, Y, Z
centers of spheres: XC, YC, ZC
radius of spheres: r

Help me~

Park

Subject: Re: the fastest way to find number of points in sphere(radius r)
Posted by JD Smith on Tue, 29 Nov 2005 02:26:04 GMT
View Forum Message <> Reply to Message

On Tue, 22 Nov 2005 00:16:35 -0800, PYJ wrote:

> Dear all,
>
> First of all, my data is three-dimensional set.
> I have to use a loop because I have a lot of positions of centers of
> spheres that I should examine.
> I want to find the # of points inside each sphere.
> Now, I use a where function in order to find points inside the cube,
> then I compute distances of all. Next, I use where function again to
> examine # of distances less than radius of sphere.
> I think it is fairly slow when large data is considered.
>
> Is there any faster way?

> The fastest way to find the number of points in sphere(radius r)

There is (of course) a way to "brute force" this type of problem, but it will not scale to the number of points and searches you mention, since the memory requirements go as the product of the two. I'll mention it to illustrate yet another example where the typical IDL approach of "just throw more memory at it" doesn't always succeed:

```
np=5000L ;; number of points to search
ncen=500L ;; number of search centers

xyz=randomu(sd,3,np)*10000.
rad2=(20.+randomu(sd,ncen)*500.)^2
cen=randomu(sd,3,ncen)*10000.
t=systime(1)

n_inside=total(total((rebin(xyz,3,np,ncen) - $
                rebin(reform(cen,3,1,ncen),3,np,ncen))^2,1) le $
          rebin(transpose(rad2),np,ncen),1)

print,systime(1)-t
```

This works well enough for this number of points and sphere centers, and finds the total counts in about 1 second, for 5000 points and 500 search spheres. Note that if the sphere always has the same radius, you can just substitute rad2=rad^2, and take out the REBIN/REFORM manipulation of rad2, to save a tiny bit more time (not much). Now let's try scaling it up:

```
Points              Time (s)
================================
np=5000L & ncen=1000L:  1.9876420
np=10000L & ncen=500L:  1.9669490
np=10000L & ncen=1000L: 4.0297060
np=50000L & ncen=500L: 66.951994
```

Uh oh, what happened there at the last one? When the memory size grew to be larger than around 1GB, it exceeded my system's physical memory, and starting paging out to disk, which is very, very slow (maybe 1000x slower than direct memory access). It just doesn't scale. Going bigger will get much worse, much faster.

Is this the only problem here? If we had an inifinite amount of memory, should we just code everything in this brute force fashion? For this simple problem, we can explore this issue using a method which combines the brute force technique with a loop designed to keep the problem from over-filling the memory:

```
np=500000L
ncen=300L
xyz=randomu(sd,3,np)*10000.
cen=randomu(sd,3,ncen)*10000.

rad2=800.^2

n_inside=lonarr(ncen)
nchunk=20L

t=systime(1)

bigxyz=rebin(xyz,3,np,nchunk)
for i=0L,ncen-1L,nchunk do begin
    search=rebin(reform(cen[*,i:i+nchunk-1L],3,1,nchunk),3,np,nc hunk)
   n_inside[i]=total(total((bigxyz - search)^2,1) le rad2,1)
endfor

print,systime(1)-t
```

Tune the chunk size so that things fit in memory on your system (remember, each float takes ~4bytes). Since the amount of work being done per loop is quite large, you won't feel the looping penalty. If your number of points is not a multiple of the chunk size, you'll have to treat the last bin specially; I haven't dealt with that.

This "semi-brute-force" method works fine, and completes in 48s, without ever hitting the disk. The straight brute force method would have fallen to its knees paging to disk on a problem of this size, so it's not too bad. Time to celebrate? Probably not. Using this semi-BF method with your stated sphere count of 3 million, and 1/2 million points would take upwards of 5 days on my system. Ouch. Is there a better way?

One time-honored method to speed things up is to cut down on the total number of compares you have to do. What's the point of doing and re-doing all that complicated arithmetic on points all the way across the universe which are not going to come close to fitting in a given search sphere? In the 2D nearest neighbor problem of last year, we used DeLauney triangulation to cut down the search space (in that case, by a large margin). IDL doesn't have a 3D triangulator, and although they do exist, they begin to suffer in higher-dimensions anyway, so we'll try a simpler approach:

1. Pre-bin all points using HIST_ND, with a fixed bin size of the typical sphere search radius (you can tune this for speed).
2. For each sphere center, locate the individual 3D bins which bound

the sphere, compute the indices of the points included in those
bins (see REVERSE_INDICES), and then find the total number of these
points which are inside the sphere.

The speedup will depend on how densely and uniformly your points are distributed in the 3D space, and how big your search radii are. One wrinkle: HIST_ND returns a real n-dimensional array for the histogram, but a 1D reverse index vector. Converting between these is equivalent to transforming coordinates from 1D to ND and back for regular array indexing. E.g. to find those points which fell in bin [x_bin,y_bin,z_bin], just look up:

```
 bin=x_bin+n_xbins*(y_bin+n_ybins*z_bin)
 wh= ri[ri[bin]:ri[bin+1]-1]
```

Putting this altogether, it looks something like:

```
np=500000L
ncen=30000L
xyz=randomu(sd,3,np)*10000.
cen=randomu(sd,3,ncen)*10000.

rad=100.
rad2=rad^2

n_inside=lonarr(ncen)

h=hist_nd(xyz,rad,REVERSE_INDICES=ri,MIN=mn,MAX=mx)
nh=n_elements(h)
sh=size(h,/DIMENSIONS)

ncube=3
nall=27
all=lindgen(nall)
offs=[transpose(all mod ncube - 1), $
      transpose(all/ncube mod ncube - 1), $
      transpose(all/ncube/ncube mod 3 - 1)]

t=systime(1)
for i=0L,ncen-1L do begin
   center_bin=long((cen[*,i]-mn)/rad)
   bins=rebin(center_bin,3,nall,/SAMPLE)+offs
   bins=bins[0,*]+sh[0]*(bins[1,*]+sh[1]*bins[2,*])
   wh=where(bins ge 0 AND bins le nh-1L,keep_cnt,NCOMPLEMENT=bad_cnt)
   if keep_cnt eq 0 then continue
   if bad_cnt gt 0 then bins=bins[wh]
```

```
   for j=0L,keep_cnt-1L do begin
      if ri[bins[j]] eq ri[bins[j]+1] then continue
       n_inside[i]+=total(total((xyz[*,ri[ri[bins[j]]:ri[bins[j]+1] -1]]- $
                           rebin(cen[*,i],3,n_elements(wh)))^2,1) le $
                     rad2)
   endfor
endfor

print,systime(1)-t
```

Consult the HISTOGRAM tutorial to see how this reverse indices manipulation works. Using this method, I can do your 3 million searches on 1/2 million points in about 650s, assuming the average occupation count in each sphere is about 2. The larger the average occupation is, the slower it becomes (since you must search a much larger number of potential matches). If your mean count per bin is low, less than 10 say, you could see some additional improvement (factor of 2, say) using a thinned WHERE or histogram of histogram method to avoid having to explicitly loop over all 27 search bins for each sphere; see David' rehash of our discussion on drizzling, here: http://www.dfanning.com/code_tips/drizzling.html

Note that I've encoded a fixed sphere radius here. If the radius is variable, you'll need to quantify the number of bins to search sphere by sphere (the `offs' variable above), to ensure you bound the sphere. You'll also have to select some representative radius for the n-dimensional histogram bin size. The tradeoffs will be: smaller bin size => more bins to search per sphere, but more accurate pre-trimming for smaller spheres; larger binsize => fewer bins to search, but more "wasted" points searched which wouldn't have been had the binsize been smaller. Somewhere about the median radius is probably optimal, but it's easy to verify this. If your range of search radii is large, making the inner loop smarter than a simple loop over bins will probably pay off even more (since it can be much more than 27 bins).

One other option I have to mention: just write it in C. The loop logic is trivial (just translate from your current IDL version), and it will run at least 100x faster than your original IDL loop based version.

JD
