Subject: position matching
Posted by cmancone on Tue, 15 May 2007 12:55:24 GMT
View Forum Message <> Reply to Message

Hi everyone,
A common task I have to do is take two lists of stars with x & y positions and match up the closest stars within a certain radius (so that each star has at most one match, that one being the best match). A long time ago I wrote some code to do this that gets the job done, but probably not in the fastest way. It just uses a for loop over one of the lists and uses a where to search for the closest star to each star on the other list. Most of the time this is more than adequate, but anytime my star lists get around 10000-20000 stars each (which happens on a not-so irregular basis) the program turns into quite a beast and takes its sweet time (i.e. a minute or two). Granted, this isn't exactly research-stopping time delays, but I'm sure that with a well thought-out algorithm, the execution time could be pulled down to a handful of seconds. The problem is, I have yet to come up with a well thought-out algorithm. I'm sure I'm not the only one who has run into this, so I was hoping there might be someone else out there that has dealt with the same thing, and knows a better way.
-Conor

Subject: Re: position matching
Posted by cmancone on Thu, 17 May 2007 19:10:40 GMT
View Forum Message <> Reply to Message

Hmm.. very intriguing. I will most certainly have to try this out!
Thanks!
-Conor

On May 16, 4:47 pm, JD Smith <jdsm...@as.arizona.edu> wrote:
> On Tue, 15 May 2007 07:36:13 -0700, cmancone wrote:
>> Yes, I read that article. However, it doesn't quite translate well into
>> what I need. It presents two methods, one using arrays, the other using a
>> Delaunay triangulation (DT). For my purposes (20,000 stars) the array
>> method won't work - it requires way too much memory (I pondered a similar
>> solution myself). That leaves the DT method. There's two problems with
>> this. First, I don't just need the closest neighbor, I need the closest
>> neighbor within a certain distance. Persumably, this is easily solved with
>> a properly placed WHERE or IF statement. The bigger problem, however, is
>> that I am matching up two separate lists, and I can't have stars on one
>> list matching up stars on the same list. The DT doesn't make any
>> distinction between stars as far as I can tell. You give it one combined
>> list, and it finds the closest stars no matter where they come from, which
>> is a problem. I've been trying to figure out just how the DT works, so I
>> can determine if it is possible to disentangle the two star lists or not.

>> It's a bit confusing though, and I have yet to determine if it will work
>> for my purposes.
>
> The DT is just a cheeky way to organize points in 2D (and higher
> dimension, but less efficiently). That algorithm uses the fact that
> the DT graph has as a sub-graph the nearest neighbors. Then you can
> start with your star of interest, and work your way out to nearby
> stars along the DT lines, to find the Nth nearest neighbor, by
> comparing a small number of stars. For matching two lists, this, as
> you pointed out, is awkward.
>
> As Paolo noted, the array method can be made to work by dividing it
> into "fits in memory" sized chunks. As also mentioned on the page you
> read, such a method doesn't necessarily mean you're doing it the most
> efficient way (just maximizing the brute force throughput). For
> searching 20,000 stars, however, the segmented brute force approach
> with arrays will probably work fine. I could do 20000x20000 in under
> a minute on my (slowish) machine with 2GB. I suspect if you get just
> a few min for similar sizes with a purely loop solution, your machine
> is much faster than mine. Here's an implementation. Tune 'chunk',
> which limits the size of arrays to compare, to optimize speed.
>
> ;==========
==========
> ; Match stars in one list to another, with brute force array techniques
> n1=20000
> x1=randomu(sd,n1)
> y1=randomu(sd,n1)
>
> n2=n1
> x2=randomu(sd,n2)
> y2=randomu(sd,n2)
>
> t=systime(1)
>
> ;; Divide the problem into manageable chunks: use [x2,y2] in full
> chunk=1.e6 ;largest number of elements to check at once
> nchunk=ceil(n1/(float(chunk)/n2))>2
> n1piece=ceil(float(n1)/nchunk)
>
> print,nchunk,' Chunks of size ',n1piece,'x',n2
>
> max_r=.001 ;maximum allowed radius
>
> mpos=lonarr(n1)
> for i=0L,nchunk-1 do begin
> low=n1piece*i
> high=(n1piece*(i+1))<(n1-1)

```
>    cnt=high-low+1
>    d=(rebin(x2,n2,cnt,/SAMPLE)- $
>       rebin(transpose(x1[low:high]),n2,cnt,/SAMPLE))^2+ $
>      (rebin(y2,n2,cnt,/SAMPLE)- $
>       rebin(transpose(y1[low:high]),n2,cnt,/SAMPLE))^2
>    void=min(d,DIMENSION=1,p)
>    mpos[low]=p mod n2
>    wh=where(sqrt(d[p]) gt max_r,cnt)
>    if cnt gt 0 then mpos[wh]=-1L
>  endfor
>
>  print,systime(1)-t
>   ;============================================================
==============
>
>  That works well enough, but is certainly not optimal.  It uses the
>  full set of [x2,y2] stars, comparing them against chunks of stars from
>  the list [x1,y1] at a time.  All stars on the target list are compared
>  to all stars on the search list.
>
>  In all cases like this, the best approach to speed up the calculation
>  is to think to yourself "how can I reduce the number of possible
>  points which must be matched, *before* I commence the matching".  For
>  closest match in a single set of stars, this led to the DT method.  In
>  this case, you have set a natural scale to the problem, max_r, which
>  will be *very* useful, allowing you to subdivide and conquer.  The
>  argument is as follows.  If you bin the search stars into bins of size
>  2*max_r, the closest point to a given target star [x,y], which is at
>  least as close as max_r in radius, *must* fall into one of 4 bins (the
>  bin which [x,y] is in, and the three bins to the upper-left,
>  upper-right, lower-left, or lower-right of it, depending on where it
>  falls in its bin).  If there is no star in any of those bins, then
>  there is no star within max_r.
>
>  I'll use HIST_ND to bin the search stars into a large grid.  Then,
>  instead of searching *all* points for the closest, I'll only search
>  ones which fell in that bin (conveniently indexed using
>  REVERSE_INDICES), and the relevant 3 adjacent bins (depending on
>  location within the bin).  You can use the same "brute-force" array
>  tricks here *within* the bin, but of course they are infinitely
>  faster, as you've pre-trimmed out the vast majority of possible
>  matches.  Sprinkle in a few more vectorizing HISTOGRAM tricks (in
>  particular the DUAL HISTOGRAM method, as described in the DRIZZLE
>  discussion), and you get the code below.
>
>  With this code, matching 20000x20000 points takes almost no time at
>  all, 0.1s.  I can match 1,000,000 vs. 1,000,000 stars in roughly 4.5
>  seconds, with a strong dependence on the initial binning size (too
```

> coarse, and bins will have too many points to fit in memory, too
> sparse, and you'll have too many empty bins). If your maximum radius
> is tiny (compared to the maximum distance between stars), it probably
> pays just to make larger bin sizes, and then weed out the ones which
> are "too far" post-facto (I've left that undone -- a simple WHERE will
> suffice). If your maximum radius is large, the bin size will be too
> coarse, and you won't have removed many for a given target
> search.... you'll be searching many tens or hundreds of thousands of
> stars per bin, and be right back in the same sort of memory trouble
> you had originally.
>
> I should emphasize that this code does *not* guarantee that the
> closest match itself is returned, only making the guarantee that *if*
> the closest match is within 1/2 of the bin size, then it is correctly
> returned. For this problem, this sets a minimum bin size: 2 * the max
> search radius. You can of course go to larger bin sizes (and you may
> want to if your stars are sprinkled very sparsely over the grid, or
> you require a very precise match, such that the histogram could grow
> excessively large). If you go smaller you risk missing the correct
> star.
>
> JD
>
> ;===
==========
> ; Match stars in one list to another, within some tolerance.
> ; Pre-bin into a 2D histogram, and use DUAL HISTOGRAM matching to select
>
> n1=1000000 ;number of stars
> x1=randomu(sd,n1) ;points to find matches near
> y1=randomu(sd,n1)
>
> n2=n1
> x2=randomu(sd,n2) ;points to search in
> y2=randomu(sd,n2)
>
> t1=systime(1)
>
> max_r=.0005 ;maximum allowed radius for a match
> bs=2*max_r ;this is the smallest binsize allowed
> h=hist_nd([1#x2,1#y2],bs,MIN=0,MAX=1,REVERSE_INDICES=ri)
> bs=bs[0]
> d=size(h,/DIMENSIONS)
>
> ;; Bin location of X1,Y1 in the X2,Y2 grid
> xoff=x1/bs & yoff=y1/bs
> xbin=floor(xoff) & ybin=floor(yoff)
> bin=(xbin + d[0]*ybin)<(d[0]*d[1]-1L) ;The bin it's in

```
>
>  ;; We must search 4 bins worth for closest match, depending on
>  ;; location within bin (towards any of the 4 quadrants).
>  xoff=1-2*((xoff-xbin) lt 0.5)        ;add bin left or right
>  yoff=1-2*((yoff-ybin) lt 0.5)        ;add bin down or up
>
>  min_pos=make_array(n1,VALUE=-1L)
>  min_dist=fltarr(n1,/NOZERO)
>
>  for i=0,1 do begin ;; Loop over 4 bins in the correct quadrant direction
>    for j=0,1 do begin
>      b=0L>(bin+i*xoff+j*yoff*d[0])<(d[0]*d[1]-1) ;current bins (offset)
>
>      ;; Dual HISTOGRAM method, loop by repeat count in bins
>      h2=histogram(h[b],MIN=1,REVERSE_INDICES=ri2)
>
>      ;; Process all bins with the same number of repeats >= 1
>      for k=0L,n_elements(h2)-1 do begin
>        if h2[k] eq 0 then continue
>        these_bins=ri2[ri2[k]:ri2[k+1]-1] ;the points with k+1 repeats in bin
>
>        if k eq 0 then begin   ; single point (n)
>          these_points=ri[ri[b[these_bins]]]
>        endif else begin       ; range over k+1 points, (n x k+1)
>          these_points=ri[ri[rebin(b[these_bins],h2[k],k+1,/SAMPLE)]+ $
>                          rebin(lindgen(1,k+1),h2[k],k+1,/SAMPLE)]
>          these_bins=rebin(temporary(these_bins),h2[k],k+1,/SAMPLE)
>        endelse
>
>        dist=(x2[these_points]-x1[these_bins])^2 + $
>            (y2[these_points]-y1[these_bins])^2
>
>        if k gt 0 then begin           ;multiple point in bin: find closest
>          dist=min(dist,DIMENSION=2,p)
>          these_points=these_points[p] ;index of closest point in bin
>          these_bins=ri2[ri2[k]:ri2[k+1]-1] ;original bin list
>        endif
>
>        ;; See if a minimum is already set
>        set=where(min_pos[these_bins] ge 0, nset, $
>                  COMPLEMENT=unset,NCOMPLEMENT=nunset)
>
>        if nset gt 0 then begin
>          ;; Only update those where the new distance is less
>          closer=where(dist[set] lt min_dist[these_bins[set]], cnt)
>          if cnt gt 0 then begin
>            set=set[closer]
>            min_pos[these_bins[set]]=these_points[set]
```

```
>              min_dist[these_bins[set]]=dist[set]
>            endif
>          endif
>
>          if nunset gt 0 then begin ;; Nothing set, closest by default
>            min_pos[these_bins[unset]]=these_points[unset]
>            min_dist[these_bins[unset]]=dist[unset]
>          endif
>       endfor
>    endfor
> endfor
>
> print,systime(1)-t1
```