Subject: TRIANGULATE. Finding contiguous cells efficiently?
Posted by Libertan on Sat, 24 Feb 2007 02:39:39 GMT
View Forum Message <> Reply to Message

Re: finding a cell's three contiguous neighbours in a Delaunay
Triangulation.

Dear IDL users,

I have searched this group, but cannot seem to find the solution to my
problem.

TRIANGULATE is a wonderful function, and is very fast indeed. I would
like to write a comparably expeditious routine for finding each
triangular cell's three contiguous (edge-sharing neighbour) cells. I
would imagine that this is done frequently, and is conceptually rather
staright forwards.

TRIANGULATE,x,y,TR

I understand that TR is a [3,N] array which essentially tells me the
three vertices of each of the N triangular Delaunay cells. It
certainly contains sufficient information to find an interior cell's 3
contiguous neighbouring cells. Clearly, contiguous cells have two
vertices in common.

My question: For a given row in TR, let us say row n, how best to find
all the other rows in TR which share two elements with row n? (The
algorithm must of course reject row n in its resulting list). I have
a rather inefficient solution, by far the slowest part of my code, and
I'm desperate to increase its speed (hopefully ten fold). My routine
uses one FOR loop (over cells n) within which are many WHERE commands
(which search for neighbour candidates).

My thoughts: Is there a clever way of solving the problem using subtle
IDL techniques/routines? Speed is key. Can it be entirely
vectorized? I even thought VORONOI (being so fast and triangulate's so-
called 'dual') might yield some helpful clues (apparently not).
TRIANGULATE's connectivity list seems to be more of a distraction than
a help, since I'm after neighbouring cells not neighbouring generator
points.

A replacement solution would be a serious contribution to my research
(and of course would be acknowledged upon publication of a paper), and
I would be happy to do speed comparisons if desired.

Yours.

Subject: Re: TRIANGULATE. Finding contiguous cells efficiently?
Posted by JD Smith on Thu, 01 Mar 2007 22:42:29 GMT
View Forum Message <> Reply to Message

On Fri, 23 Feb 2007 18:39:39 -0800, Libertan wrote:

> Re: finding a cell's three contiguous neighbours in a Delaunay
> Triangulation.
>
> Dear IDL users,
>
> I have searched this group, but cannot seem to find the solution to my
> problem.
>
> TRIANGULATE is a wonderful function, and is very fast indeed. I would
> like to write a comparably expeditious routine for finding each triangular
> cell's three contiguous (edge-sharing neighbour) cells. I would imagine
> that this is done frequently, and is conceptually rather staright
> forwards.
>
> TRIANGULATE,x,y,TR
>
> I understand that TR is a [3,N] array which essentially tells me the three
> vertices of each of the N triangular Delaunay cells. It certainly
> contains sufficient information to find an interior cell's 3 contiguous
> neighbouring cells. Clearly, contiguous cells have two vertices in
> common.
>
> My question: For a given row in TR, let us say row n, how best to find all
> the other rows in TR which share two elements with row n? (The algorithm
> must of course reject row n in its resulting list). I have a rather
> inefficient solution, by far the slowest part of my code, and I'm
> desperate to increase its speed (hopefully ten fold). My routine uses one
> FOR loop (over cells n) within which are many WHERE commands (which search
> for neighbour candidates).
>
> My thoughts: Is there a clever way of solving the problem using subtle IDL
> techniques/routines? Speed is key. Can it be entirely vectorized? I even
> thought VORONOI (being so fast and triangulate's so- called 'dual') might
> yield some helpful clues (apparently not). TRIANGULATE's connectivity list
> seems to be more of a distraction than a help, since I'm after
> neighbouring cells not neighbouring generator points.
>
> A replacement solution would be a serious contribution to my research (and
> of course would be acknowledged upon publication of a paper), and I would
> be happy to do speed comparisons if desired.

To such problems there are generally two types of solution: sort-based
and array-based. The latter is a brute force solution which usually

involve a large amount of memory, setting IDL loose en masse on that block of memory (something it's very good at). Such a method usually scales as N^2, where N is the number of elements. If you can fit N^2 elements into memory, brute force is usually faster. However, it suffers quite horribly when it hits your memory limits. SORT based methods are not as fast for small N, but scale well with N (usually as N log(N)), so are more appropriate when you don't know in advance the size of your problem (e.g. sometimes you have 100 triangles, and sometimes a million).

In both of these cases, however, the best approach, if at all possible, is to reduce N somehow to begin with, ideally to a small fixed number. Here, it seems that the connectivity array will allow you to do just this. It is a "reverse indices" style array, which you can read about in the HISTOGRAM tutorial. We used just this array in the "5th nearest neighbor" problem as well: http://www.dfanning.com/code_tips/slowloops.html. You can use it here as follows:

1. Compute the triangulation with connectivity vector C.

2. For a given triangle of interest T, for each edge of T, find the sets of points connected to both the points in the edge.

3. Find the point in common among these two sets: your adjacent triangle consists of the edge and this point.

Here's some test code which implements this. It uses the "brute force" array comparison method, which is not so brute given that points are connected directly only to a handful of other points.

```
 ============================================================
==============
;; Find triangles adjacent to an edge for a given triangulation
tvlct,[255,0,0],[0,255,0],[0,0,255],1
n=25
x=randomu(sd,n) & y=randomu(sd,n)
plot,x,y,PSYM=4,SYMSIZE=4

triangulate,x,y,t,CONNECTIVITY=c

p=n/2                       ; which triangle to consider

w=[0,1,2,0]
for i=0,(size(t,/DIMENSIONS))[1]-1 do plots,x[t[w,i]],y[t[w,i]]
plots,x[t[w,p]],y[t[w,p]],COLOR=1,PSYM=-4,THICK=2

adj=make_array(3,VALUE=-1L)
```

```
extra=adj
tp=t[*,p]                    ; the triangle to match

for i=0,2 do begin
  n=(i+1) mod 3

  if i gt 0 then begin
    c1=c2 & nc1=nc2
  endif else begin
    c1=c[c[tp[0]]:c[tp[0]+1]-1]
    nc1=n_elements(c1)
  endelse

  c2=c[c[tp[n]]:c[tp[n]+1]-1]  ;points connected to second point in edge
  nc2=n_elements(c2)

  ;; Find the adjacent point(s) in common, and not on this triangle
  w=where(rebin(c1,nc1,nc2,/SAMP) eq rebin(transpose(c2),nc1,nc2,/SAMP),cnt)
  if cnt eq 0 then continue
  cb=c2[w/nc1]

  ;; Find the points in common *not* on the triangle
  w=where(total(rebin(transpose(cb),3,cnt,/SAMP) ne $
            rebin(tp,3,cnt,/SAMP),1,/PRESERVE_TYPE) eq 3b,cnt)
  if cnt eq 0 then continue
  adj[i]=cb[w[0]]              ; note: there could be more than 1
  if cnt gt 1 then extra[i]=cb[w[1]]
endfor

for i=0,2 do begin
  if adj[i] eq -1 then continue
  ;; If an extra was stored, swap it in if this one is a repeat
  if extra[i] ge 0 && total(adj eq adj[i],/PRESERVE_TYPE) gt 1 then $
    adj[i]=extra[i]

  n=(i+1) mod 3
  plots,x[adj[i]],y[adj[i]],PSYM=4,COLOR=2,SYMSIZE=4,THICK=2
  plots,[x[tp[i]],x[adj[i]]],[y[tp[i]],y[adj[i]]],COLOR=2
  plots,[x[tp[n]],x[adj[i]]],[y[tp[n]],y[adj[i]]],COLOR=2
endfor
 ============================================================
==============
```

At the end, 'adj' contains the 3 points which, when combined with the three edges, produce the desired adjacent triangles.

Note that the problem is somewhat ill-defined. Some triangles have edges on the boundary, which don't have an adjacent triangle (in which

case 'adj' will contain -1 for that edge). Occasionally, a given edge may have more than 1 valid adjacent triangles (think of a diamond with a horizontal line across the middle, and another point above this bisector line). Here I keep an 'extra' point set vector, and at the end, swap in the extra one if this would result in less duplication. Maybe you don't care about this, and could speed it up by removing this test. Run this many times and you'll find a random case which illustrates the point. I think, but have not proved, that there will be at most 2 such adjacency triangles (hence I only store 1).

For large triangulations, this method should be much faster than any of the other methods mentioned, since it looks only at the ~5-10 connected points for each vertex, independent of the total number of points.

JD
