

# Package ‘tripack’

June 7, 2026

**Version** 1.3-9.4

**Title** Triangulation of Irregularly Spaced Data

**Maintainer** Albrecht Gebhardt <albrecht.gebhardt@aau.at>

**Description** A constrained two-dimensional Delaunay triangulation package providing both triangulation and generation of voronoi mosaics of irregular spaced data.  
Please note that most of the functions are now also covered in package `interp`, which is a re-implementation from scratch under a free license based on a different triangulation algorithm.

**License** file LICENSE

**Date** 2026-06-06

**NeedsCompilation** yes

**License\_restricts\_use** yes

**Repository** CRAN

**Date/Publication** 2026-06-07 05:50:02 UTC

**Author** Albrecht Gebhardt [aut, cre, cph] (R functions),  
R. J. Renka [aut] (author of underlying Fortran code),  
Stephen Eglen [aut, cph] (contributions),  
Sergej Zuyev [aut, cph] (contributions),  
Denis White [aut, cph] (contributions)

## Contents

add.constraint . . . . .	2
cells . . . . .	4
circles . . . . .	5
circtest . . . . .	6
circum . . . . .	6
circumcircle . . . . .	7
convex.hull . . . . .	9
identify.tri . . . . .	10
in.convex.hull . . . . .	11

left . . . . .	12
neighbours . . . . .	13
on.convex.hull . . . . .	14
outer.convhull . . . . .	15
plot.tri . . . . .	16
plot.voronoi . . . . .	17
plot.voronoi.polygons . . . . .	19
print.summary.tri . . . . .	20
print.summary.voronoi . . . . .	20
print.tri . . . . .	21
print.voronoi . . . . .	22
summary.tri . . . . .	23
summary.voronoi . . . . .	24
tri . . . . .	25
tri.dellens . . . . .	26
tri.find . . . . .	27
tri.mesh . . . . .	28
triangles . . . . .	30
tripack-internal . . . . .	31
tritest . . . . .	31
voronoi . . . . .	32
voronoi.area . . . . .	32
voronoi.findrejectsites . . . . .	33
voronoi.mosaic . . . . .	34
voronoi.polygons . . . . .	35

## Index 36

---

add.constraint	<i>Add a constraint to an triangulaion object</i>
----------------	---

---

### Description

This subroutine provides for creation of a constrained Delaunay triangulation which, in some sense, covers an arbitrary connected region  $R$  rather than the convex hull of the nodes. This is achieved simply by forcing the presence of certain adjacencies (triangulation arcs) corresponding to constraint curves. The union of triangles coincides with the convex hull of the nodes, but triangles in  $R$  can be distinguished from those outside of  $R$ . The only modification required to generalize the definition of the Delaunay triangulation is replacement of property 5 (refer to `tri.mesh` by the following:

5') If a node is contained in the interior of the circumcircle of a triangle, then every interior point of the triangle is separated from the node by a constraint arc.

In order to be explicit, we make the following definitions. A constraint region is the open interior of a simple closed positively oriented polygonal curve defined by an ordered sequence of three or more distinct nodes (constraint nodes)  $P(1), P(2), \dots, P(K)$ , such that  $P(I)$  is adjacent to  $P(I+1)$  for  $I = 1, \dots, K$  with  $P(K+1) = P(1)$ . Thus, the constraint region is on the left (and may have nonfinite area) as the sequence of constraint nodes is traversed in the specified order. The constraint regions must



---

cells *extract info about voronoi cells*

---

### Description

This function returns some info about the cells of a voronoi mosaic, including the coordinates of the vertices and the cell area.

### Usage

```
cells(voronoi.obj)
```

### Arguments

voronoi.obj     object of class voronoi

### Details

The function calculates the neighbourhood relations between the underlying triangulation and translates it into the neighbourhood relations between the voronoi cells.

### Value

retruns a list of lists, one entry for each voronoi cell which contains

cell	cell index
center	cell 'center'
neighbours	neighbour cell indices
nodes	2 times nnb matrix with vertice coordinates
area	cell area

### Note

outer cells have area=NA, currently also nodes=NA which is not really useful – to be done later

### Author(s)

A. Gebhardt

### See Also

[voronoi.mosaic](#), [voronoi.area](#)

**Examples**

```
data(tritest)
tritest.vm <- voronoi.mosaic(tritest$x, tritest$y)
tritest.cells <- cells(tritest.vm)
# highlight cell 12:
plot(tritest.vm)
polygon(t(tritest.cells[[12]]$nodes), col="green")
# put cell area into cell center:
text(tritest.cells[[12]]$center[1],
      tritest.cells[[12]]$center[2],
      tritest.cells[[12]]$area)
```

---

circles

*plot circles*

---

**Description**

This function plots circles at given locations with given radii.

**Usage**

```
circles(x, y, r, ...)
```

**Arguments**

x	vector of x coordinates
y	vector of y coordinates
r	vector of radii
...	additional graphic parameters will be passed through

**Note**

This function needs a previous plot where it adds the circles.

**Author(s)**

A. Gebhardt

**See Also**

[lines](#), [points](#)

**Examples**

```
x<-rnorm(10)
y<-rnorm(10)
r<-runif(10,0,0.5)
plot(x,y, xlim=c(-3,3), ylim=c(-3,3), pch="+")
circles(x,y,r)
```

---

circctest	<i>circctest / sample data</i>
-----------	--------------------------------

---

### Description

Sample data for the `link{circumcircle}` function.

`circctest2` are points sampled from a circle with some jitter added, i.e. they represent the most complicated case for the `link{circumcircle}` function.

---

circum	<i>Determine the circumcircle of a triangle</i>
--------	---

---

### Description

This function returns the circumcircle of a triangle.

### Usage

```
circum(x, y)
```

### Arguments

x	Vector of three elements, giving the x coordinates of the triangle nodes.
y	Vector of three elements, giving the y coordinates of the triangle nodes.

### Details

This is an interface to the Fortran function CIRCUM found in TRIPACK.

### Value

x	'x' coordinate of center
y	'y' coordinate of center
radius	circumcircle radius
signed.area	signed area of triangle (positive iff nodes are numbered counter clock wise)
aspect.ratio	ratio "radius of inscribed circle"/"radius of circumcircle", varies between 0 and 0.5 0 means collinear points, 0.5 equilateral triangle.

### Note

This function is mainly intended to be used by `circumcircle`.

**Author(s)**

Fortran code: R. J. Renka, R code: A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

**See Also**

[circumcircle](#)

**Examples**

```
circum(c(0,1,0),c(0,0,1))
```

---

circumcircle	<i>Determine the circumcircle of a set of points</i>
--------------	--

---

**Description**

This function returns the (smallest) circumcircle of a set of n points

**Usage**

```
circumcircle(x, y = NULL, num.touch=2, plot = FALSE, debug = FALSE)
```

**Arguments**

x	vector containing x coordinates of the data. If y is missing x should contain two elements \$x and \$y.
y	vector containing y coordinates of the data.
num. touch	How often should the resulting circle touch the convex hull of the given points? default: 2 possible values: 2 or 3 Note: The circumcircle of a triangle is usually defined to touch at 3 points, this function searches by default the minimum circle, which may be only touching at 2 points. Set parameter num. touch accordingly if you dont want the default behaviour!
plot	Logical, produce a simple plot of the result. default: FALSE
debug	Logical, more plots, only needed for debugging. default: FALSE

**Details**

This is a (naive implemented) algorithm which determines the smallest circumcircle of n points:

First step: Take the convex hull.

Second step: Determine two points on the convex hull with maximum distance for the diameter of the set.

Third step: Check if the circumcircle of these two points already contains all other points (of the convex hull and hence all other points).

If not or if 3 or more touching points are desired (`num.touch=3`), search a point with minimum enclosing circumcircle among the remaining points of the convex hull.

If such a point cannot be found (e.g. for `data(circtest2)`), search the remaining triangle combinations of points from the convex hull until an enclosing circle with minimum radius is found.

The last search uses an upper and lower bound for the desired minimum radius:

Any enclosing rectangle and its circumcircle gives an upper bound (the axis-parallel rectangle is used).

Half the diameter of the set from step 1 is a lower bound.

**Value**

x	'x' coordinate of circumcircle center
y	'y' coordinate of circumcircle center
radius	radius of circumcircle

**Author(s)**

Albrecht Gebhardt

**See Also**

[convex.hull](#)

**Examples**

```
data(circtest)
# smallest circle:
circumcircle(circtest,num.touch=2,plot=TRUE)

# smallest circle with maximum touching points (3):
circumcircle(circtest,num.touch=3,plot=TRUE)

# some stress test for this function,
data(circtest2)
# circtest2 was generated by:
# 100 random points almost one a circle:
# alpha <- runif(100,0,2*pi)
# x <- cos(alpha)
# y <- sin(alpha)
# circtest2<-list(x=cos(alpha)+runif(100,0,0.1),
```

```
#           y=sin(alpha)+runif(100,0,0.1))
#
circumcircle(circtest2,plot=TRUE)
```

---

convex.hull                      *Return the convex hull of a triangulation object*

---

### Description

Given a triangulation `tri.obj` of  $n$  points in the plane, this subroutine returns two vectors containing the coordinates of the nodes on the boundary of the convex hull.

### Usage

```
convex.hull(tri.obj, plot.it=FALSE, add=FALSE,...)
```

### Arguments

<code>tri.obj</code>	object of class "tri"
<code>plot.it</code>	logical, if TRUE the convex hull of <code>tri.obj</code> will be plotted.
<code>add</code>	logical. if TRUE (and <code>plot.it=TRUE</code> ), add to a current plot.
<code>...</code>	additional plot arguments

### Value

<code>x</code>	x coordinates of boundary nodes.
<code>y</code>	y coordinates of boundary nodes.

### Author(s)

A. Gebhardt

### References

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

### See Also

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#), [add.constraint](#).

**Examples**

```
# rather simple example from TRIPACK:
data(tritest)
tr<-tri.mesh(tritest$x,tritest$y)
convex.hull(tr,plot.it=TRUE)
# random points:
rand.tr<-tri.mesh(runif(10),runif(10))
plot(rand.tr)
rand.ch<-convex.hull(rand.tr, plot.it=TRUE, add=TRUE, col="red")
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-17 & quakes[,1]>=-19.0 &
                    quakes[,2]<=182.0 & quakes[,2]>=180.0),]
quakes.tri<-tri.mesh(quakes.part$lon, quakes.part$lat, duplicate="remove")
plot(quakes.tri)
convex.hull(quakes.tri, plot.it=TRUE, add=TRUE, col="red")
```

---

identify.tri

*Identify points in a triangulation plot*


---

**Description**

Identify points in a plot of "x" with its coordinates. The plot of "x" must be generated with plot.tri.

**Usage**

```
## S3 method for class 'tri'
identify(x,...)
```

**Arguments**

x                    object of class "tri"  
...                    additional paramters for identify

**Value**

an integer vector containing the indexes of the identified points.

**Author(s)**

A. Gebhardt

**See Also**

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#)

**Examples**

```
data(tritest)
tritest.tr<-tri.mesh(tritest$x,tritest$y)
plot(tritest.tr)
identify.tri(tritest.tr)
```

---

in.convex.hull	<i>Determines if points are in the convex hull of a triangulation object</i>
----------------	--

---

**Description**

Given a triangulation `tri.obj` of  $n$  points in the plane, this subroutine returns a logical vector indicating if the points  $(x_i, y_i)$  are contained within the convex hull of `tri.obj`.

**Usage**

```
in.convex.hull(tri.obj, x, y)
```

**Arguments**

<code>tri.obj</code>	object of class "tri"
<code>x</code>	vector of x-coordinates of points to locate
<code>y</code>	vector of y-coordinates of points to locate

**Value**

Logical vector.

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#), [add.constraint](#), [convex.hull](#).

**Examples**

```
# example from TRIPACK:
data(tritest)
tr<-tri.mesh(tritest$x,tritest$y)
in.convex.hull(tr,0.5,0.5)
in.convex.hull(tr,c(0.5,-1,1),c(0.5,1,1))
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-10.78 & quakes[,1]>=-19.4 &
                    quakes[,2]<=182.29 & quakes[,2]>=165.77),]
q.tri<-tri.mesh(quakes.part$lon, quakes.part$lat, duplicate="remove")
in.convex.hull(q.tri,quakes$lon[990:1000],quakes$lat[990:1000])
```

left

*Determines whether given points are left of a directed edge.***Description**

This function returns a logical vector indicating which elements of the given points P0 are left of the directed edge P1->P2.

**Usage**

```
left(x0, y0, x1, y1, x2, y2)
```

**Arguments**

x0	Numeric vector, 'x' coordinates of points P0 to check
y0	Numeric vector, 'y' coordinates of points P0 to check, same length as 'x'.
x1	'x' coordinate of point P1
y1	'y' coordinate of point P1
x2	'x' coordinate of point P2
y2	'y' coordinate of point P2

**Value**

Logical vector.

**Note**

This is an interface to the Fortran function VLEFT, which is modeled after TRIPACK's LEFT function but accepts more than one point P0.

**Author(s)**

A. Gebhardt

**See Also**

[in.convex.hull](#)

**Examples**

```
left(c(0,0,1,1),c(0,1,0,1),0,0,1,1)
```

---

neighbours	<i>List of neighbours from a triangulation object</i>
------------	---

---

**Description**

Extract a list of neighbours from a triangulation object

**Usage**

```
neighbours(tri.obj)
```

**Arguments**

tri.obj            object of class "tri"

**Value**

nested list of neighbours per point

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#)

**Examples**

```
data(tritest)
tritest.tr<-tri.mesh(tritest$x,tritest$y)
tritest.nb<-neighbours(tritest.tr)
```

---

on.convex.hull                      *Determines if points are on the convex hull of a triangulation object*

---

### Description

Given a triangulation `tri.obj` of  $n$  points in the plane, this subroutine returns a logical vector indicating if the points  $(x_i, y_i)$  lay on the convex hull of `tri.obj`.

### Usage

```
on.convex.hull(tri.obj, x, y)
```

### Arguments

<code>tri.obj</code>	object of class "tri"
<code>x</code>	vector of x-coordinates of points to locate
<code>y</code>	vector of y-coordinates of points to locate

### Value

Logical vector.

### Author(s)

A. Gebhardt

### References

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

### See Also

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#), [add.constraint](#), [convex.hull](#), [in.convex.hull](#).

### Examples

```
# example from TRIPACK:
data(tritest)
tr<-tri.mesh(tritest$x, tritest$y)
on.convex.hull(tr, 0.5, 0.5)
on.convex.hull(tr, c(0.5, -1, 1), c(0.5, 1, 1))
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-10.78 & quakes[,1]>=-19.4 &
                    quakes[,2]<=182.29 & quakes[,2]>=165.77),]
q.tri<-tri.mesh(quakes.part$lon, quakes.part$lat, duplicate="remove")
on.convex.hull(q.tri, quakes.part$lon[1:20], quakes.part$lat[1:20])
```

---

outer.convhull                    *Version of outer which operates only in a convex hull*

---

### Description

This version of outer evaluates FUN only on that part of the grid *cxscy* that is enclosed within the convex hull of the points (px,py).

This can be useful for spatial estimation if no extrapolation is wanted.

### Usage

```
outer.convhull(cx,cy,px,py,FUN,duplicate="remove",...)
```

### Arguments

cx	x coordinates of grid
cy	y coordinates of grid
px	vector of x coordinates of points
py	vector of y coordinates of points
FUN	function to be evaluated over the grid
duplicate	indicates what to do with duplicate ( $px_i, py_i$ ) points, default "remove".
...	additional arguments for FUN

### Value

Matrix with values of FUN (NAs if outside the convex hull).

### Author(s)

A. Gebhardt

### See Also

[in.convex.hull](#)

### Examples

```
x<-runif(20)
y<-runif(20)
z<-runif(20)
z.lm<-lm(z~x+y)
f.pred<-function(x,y)
  {predict(z.lm,data.frame(x=as.vector(x),y=as.vector(y)))}
xg<-seq(0,1,0.05)
yg<-seq(0,1,0.05)
image(xg,yg,outer.convhull(xg,yg,x,y,f.pred))
points(x,y)
```

---

`plot.tri`*Plot a triangulation object*

---

**Description**

plots the triangulation "x"

**Usage**

```
## S3 method for class 'tri'  
plot(x, add=FALSE, xlim=range(x$x), ylim=range(x$y),  
      do.points=TRUE, do.labels = FALSE, isometric=FALSE,...)
```

**Arguments**

<code>x</code>	object of class "tri"
<code>add</code>	logical, if TRUE, add to a current plot.
<code>do.points</code>	logical, indicates if points should be plotted.
<code>do.labels</code>	logical, indicates if points should be labelled
<code>xlim, ylim</code>	x/y ranges for plot
<code>isometric</code>	generate an isometric plot (default FALSE)
<code>...</code>	additional plot parameters

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[tri](#), [print.tri](#), [summary.tri](#)

**Examples**

```
# random points
plot(tri.mesh(rpois(100,lambda=20),rpois(100,lambda=20),duplicate="remove"))
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-10.78 & quakes[,1]>=-19.4 &
                    quakes[,2]<=182.29 & quakes[,2]>=165.77),]
quakes.tri<-tri.mesh(quakes.part$lon, quakes.part$lat, duplicate="remove")
plot(quakes.tri)
# use the whole quakes data set
# (will not work with standard memory settings, hence commented out)
#plot(tri.mesh(quakes$lon, quakes$lat, duplicate="remove"), do.points=F)
```

---

plot.voronoi	<i>Plot a voronoi object</i>
--------------	------------------------------

---

**Description**

Plots the mosaic "x".

Dashed lines are used for outer tiles of the mosaic.

**Usage**

```
## S3 method for class 'voronoi'
plot(x,add=FALSE,
      xlim=c(min(x$tri$x)-
             0.1*diff(range(x$tri$x)),
             max(x$tri$x)+
             0.1*diff(range(x$tri$x))),
      ylim=c(min(x$tri$y)-
             0.1*diff(range(x$tri$y)),
             max(x$tri$y)+
             0.1*diff(range(x$tri$y))),
      all=FALSE,
      do.points=TRUE,
      main="Voronoi mosaic",
      sub=deparse(substitute(x)),
      isometric=FALSE,
      ...)
```

**Arguments**

x	object of class "voronoi"
add	logical, if TRUE, add to a current plot.
xlim	x plot ranges, by default modified to hide dummy points outside of the plot
ylim	y plot ranges, by default modified to hide dummy points outside of the plot

<code>all</code>	show all (including dummy points in the plot)
<code>do.points</code>	logical, indicates if points should be plotted.
<code>main</code>	plot title
<code>sub</code>	plot subtitle
<code>isometric</code>	generate an isometric plot (default FALSE)
<code>...</code>	additional plot parameters

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[voronoi](#), [print.voronoi](#), [summary.voronoi](#)

**Examples**

```
# plot a random mosaic
plot(voronoi.mosaic(runif(100),runif(100),duplicate="remove"))
# use isometric=TRUE and all=TRUE to see the complete mosaic
# including extreme outlier points:
plot(voronoi.mosaic(runif(100),runif(100),duplicate="remove"),
      all=TRUE, isometric=TRUE)
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-17 & quakes[,1]>=-19.0 &
                    quakes[,2]<=182.0 & quakes[,2]>=180.0),]
quakes.vm<-voronoi.mosaic(quakes.part$lon, quakes.part$lat,
                         duplicate="remove")
plot(quakes.vm, isometric=TRUE)
# use the whole quakes data set
# (will not work with standard memory settings, hence commented out here)
#plot(voronoi.mosaic(quakes$lon, quakes$lat, duplicate="remove"), isometric=TRUE)
```

---

plot.voronoi.polygons *plots an voronoi.polygons object*

---

### Description

plots an voronoi.polygons object

### Usage

```
## S3 method for class 'voronoi.polygons'  
plot(x, which, color=TRUE, ...)
```

### Arguments

x	object of class voronoi.polygons
which	index vector selecting which polygons to plot
color	logical, determines if plot should be colored, default: TRUE
...	additional plot arguments

### Author(s)

A. Gebhardt

### See Also

[voronoi.polygons](#)

### Examples

```
##---- Should be DIRECTLY executable !! ----  
##-- ==> Define data, use random,  
##-- or do help(data=index) for the standard data sets.  
data(tritest)  
tritest.vm <- voronoi.mosaic(tritest$x, tritest$y)  
tritest.vp <- voronoi.polygons(tritest.vm)  
plot(tritest.vp)  
plot(tritest.vp, which=c(1,3,5))
```

---

print.summary.tri      *Print a summary of a triangulation object*

---

**Description**

Prints some information about tri.obj

**Usage**

```
## S3 method for class 'summary.tri'  
print(x, ...)
```

**Arguments**

x                    object of class "summary.tri", generated by [summary.tri](#).  
...                   additional parameters for print

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

**See Also**

[tri](#), [tri.mesh](#), [print.tri](#), [plot.tri](#), [summary.tri](#).

---

print.summary.voronoi      *Print a summary of a voronoi object*

---

**Description**

Prints some information about x

**Usage**

```
## S3 method for class 'summary.voronoi'  
print(x, ...)
```

**Arguments**

x                    object of class "summary.voronoi", generated by [summary.voronoi](#).  
...                   additional paramters for print

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

**See Also**

[voronoi](#), [voronoi.mosaic](#), [print.voronoi](#), [plot.voronoi](#), [summary.voronoi](#).

---

*print.tri*                    *Print a triangulation object*

---

**Description**

prints a adjacency list of "x"

**Usage**

```
## S3 method for class 'tri'  
print(x,...)
```

**Arguments**

x                    object of class "tri"  
...                   additional paramters for print

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

**See Also**

[tri](#), [plot.tri](#), [summary.tri](#)

---

print.voronoi	<i>Print a voronoi object</i>
---------------	-------------------------------

---

**Description**

prints a summary of "x"

**Usage**

```
## S3 method for class 'voronoi'  
print(x,...)
```

**Arguments**

x	object of class "voronoi"
...	additional paramters for print

**Value**

None

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

**See Also**

[voronoi](#), [plot.voronoi](#), [summary.voronoi](#)

---

summary.tri	<i>Return a summary of a triangulation object</i>
-------------	---

---

### Description

Returns some information (number of nodes, triangles, arcs, boundary nodes and constraints) about object.

### Usage

```
## S3 method for class 'tri'  
summary(object,...)
```

### Arguments

object	object of class "tri"
...	additional paramters for summary

### Value

An objekt of class "summary.tri", to be printed by [print.summary.tri](#).

It contains the number of nodes (n), of arcs (na), of boundary nodes (nb), of triangles (nt) and constraints (nc).

### Author(s)

A. Gebhardt

### References

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. ACM Transactions on Mathematical Software. **22**, 1-8.

### See Also

[tri](#), [print.tri](#), [plot.tri](#), [print.summary.tri](#).

---

summary.voronoi	<i>Return a summary of a voronoi object</i>
-----------------	---

---

**Description**

Returns some information about object

**Usage**

```
## S3 method for class 'voronoi'  
summary(object,...)
```

**Arguments**

object	object of class "voronoi"
...	additional parameters for summary

**Value**

Object of class "summary.voronoi".

It contains the number of nodes (nn) and dummy nodes (nd).

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[voronoi](#), [voronoi.mosaic](#), [print.voronoi](#), [plot.voronoi](#), [print.summary.voronoi](#).

---

tri	<i>A triangulation object</i>
-----	-------------------------------

---

### Description

R object that represents the triangulation of a set of 2D points, generated by [tri.mesh](#) or [add.constraint](#).

### Arguments

n	Number of nodes
x	x coordinates of the triangulation nodes
y	y coordinates of the triangulation nodes
tlist	Set of nodal indexes which, along with tlptr, tlend, and tlnew, define the triangulation as a set of $n$ adjacency lists – counterclockwise-ordered sequences of neighboring nodes such that the first and last neighbors of a boundary node are boundary nodes (the first neighbor of an interior node is arbitrary). In order to distinguish between interior and boundary nodes, the last neighbor of each boundary node is represented by the negative of its index.
tlptr	Set of pointers in one-to-one correspondence with the elements of tlist. <code>tlist[tlptr[i]]</code> indexes the node which follows <code>tlist[i]</code> in cyclical counterclockwise order (the first neighbor follows the last neighbor).
tlend	Set of pointers to adjacency lists. <code>tlend[k]</code> points to the last neighbor of node $k$ for $k = 1, \dots, n$ . Thus, <code>tlist[tlend[k]] &lt; 0</code> if and only if $k$ is a boundary node.
tlnew	Pointer to the first empty location in tlist and tlptr (list length plus one).
nc	number of constraints
lc	starting indices of constraints in x and y
call	call, which generated this object

### Note

The elements `tlist`, `tlptr`, `tlend` and `tlnew` are mainly intended for internal use in the appropriate Fortran routines.

### Author(s)

A. Gebhardt

### References

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

### See Also

[tri.mesh](#), [print.tri](#), [plot.tri](#), [summary.tri](#)

---

tri.dellens                      *Compute the Delaunay segment lengths*

---

### Description

Return a vector of Delaunay segment lengths for the voronoi object. The Delaunay triangles connected to sites contained in exceptions vector are ignored (unless inverse is TRUE, when only those Delaunay triangles are accepted).

The exceptions vector is provided so that sites at the border of a region can be removed, as these tend to bias the distribution of Delaunay segment lengths. exceptions can be created by [voronoi.findrejectsites](#).

### Usage

```
tri.dellens(voronoi.obj, exceptions = NULL, inverse = FALSE)
```

### Arguments

voronoi.obj	object of class "voronoi"
exceptions	a numerical vector
inverse	Logical

### Value

A vector of Delaunay segment lengths.

### Author(s)

S. J. Eglen

### See Also

[voronoi.findrejectsites](#), [voronoi.mosaic](#),

### Examples

```
data(tritest)
tritest.vm <- voronoi.mosaic(tritest$x, tritest$y)

tritest.vm.rejects <- voronoi.findrejectsites(tritest.vm, 0, 1, 0, 1)
trilens.all <- tri.dellens(tritest.vm)
trilens.acc <- tri.dellens(tritest.vm, tritest.vm.rejects)
trilens.rej <- tri.dellens(tritest.vm, tritest.vm.rejects, inverse=TRUE)

par(mfrow=c(3,1))
dotchart(trilens.all, main="all Delaunay segment lengths")
dotchart(trilens.acc, main="excluding border sites")
dotchart(trilens.rej, main="only border sites")
```

---

tri.find	<i>Locate a point in a triangulation</i>
----------	--

---

**Description**

This subroutine locates a point  $P=(x,y)$  relative to a triangulation created by `tri.mesh`. If  $P$  is contained in a triangle, the three vertex indexes are returned. Otherwise, the indexes of the rightmost and leftmost visible boundary nodes are returned.

**Usage**

```
tri.find(tri.obj,x,y)
```

**Arguments**

<code>tri.obj</code>	an triangulation object
<code>x</code>	x-coordinate of the point
<code>y</code>	y-coordinate of the point

**Value**

A list with elements `i1,i2,i3` containing nodal indexes, in counterclockwise order, of the vertices of a triangle containing  $P=(x,y)$ , or, if  $P$  is not contained in the convex hull of the nodes, `i1` indexes the rightmost visible boundary node, `i2` indexes the leftmost visible boundary node, and `i3 = 0`. Rightmost and leftmost are defined from the perspective of  $P$ , and a pair of points are visible from each other if and only if the line segment joining them intersects no triangulation arc. If  $P$  and all of the nodes lie on a common line, then `i1=i2=i3 = 0` on output.

**Author(s)**

A. Gebhardt

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#), [convex.hull](#)

**Examples**

```

data(tritest)
tritest.tr<-tri.mesh(tritest$x,tritest$y)
plot(tritest.tr)
pnt<-list(x=0.3,y=0.4)
triangle.with.pnt<-tri.find(tritest.tr,pnt$x,pnt$y)
attach(triangle.with.pnt)
lines(tritest$x[c(i1,i2,i3,i1)],tritest$y[c(i1,i2,i3,i1)],col="red")
points(pnt$x,pnt$y)

```

---

tri.mesh

---

*Create a delaunay triangulation*


---

**Description**

This subroutine creates a Delaunay triangulation of a set of  $N$  arbitrarily distributed points in the plane referred to as nodes. The Delaunay triangulation is defined as a set of triangles with the following five properties:

- 1) The triangle vertices are nodes.
- 2) No triangle contains a node other than its vertices.
- 3) The interiors of the triangles are pairwise disjoint.
- 4) The union of triangles is the convex hull of the set of nodes (the smallest convex set which contains the nodes).
- 5) The interior of the circumcircle of each triangle contains no node.

The first four properties define a triangulation, and the last property results in a triangulation which is as close as possible to equiangular in a certain sense and which is uniquely defined unless four or more nodes lie on a common circle. This property makes the triangulation well-suited for solving closest point problems and for triangle-based interpolation.

The triangulation can be generalized to a constrained Delaunay triangulation by a call to `add.constraint`. This allows for user-specified boundaries defining a nonconvex and/or multiply connected region.

The operation count for constructing the triangulation is close to  $O(N)$  if the nodes are presorted on  $X$  or  $Y$  components. Also, since the algorithm proceeds by adding nodes incrementally, the triangulation may be updated with the addition (or deletion) of a node very efficiently. The adjacency information representing the triangulation is stored as a linked list requiring approximately  $13N$  storage locations.

**Usage**

```

tri.mesh(x, y = NULL, duplicate = "error",
        jitter = 10^-12, jitter.iter = 6, jitter.random = FALSE)

```

**Arguments**

x	vector containing x coordinates of the data. If y is missing x should contain two elements \$x and \$y.
y	vector containing y coordinates of the data.
duplicate	flag indicating how to handle duplicate elements. Possible values are: "error" – default, "strip" – remove all duplicate points, "remove" – leave one point of duplicate points.
jitter	Jitter of amount of $\text{diff}(\text{range}(XX)) * \text{jitter}$ ( $XX=x$ or $y$ ) will be added to coordinates if collinear points are detected. Afterwards interpolation will be tried once again.  Note that the jitter is not generated randomly unless <code>jitter.random</code> is set to TRUE. This ensures reproducible results. <code>interp</code> of package <code>akima</code> uses the same jitter mechanism. That means you can plot the triangulation on top of the interpolation and see the same triangulation as used for interpolation, see examples for <code>interp</code> .
jitter.iter	number of iterations to retry with jitter, amount will be increased in each iteration by $\text{iter}^{1.5}$
jitter.random	logical, see jitter, defaults to FALSE

**Value**

An object of class "tri"

**Note**

There is re-implementation of this function available in package `interp` under the same name with the same arguments. But the return value is of a different class. So returned objects from this function can not be used by functions of same name in package `interp`. But code written to use functions from this package can be reused with the new package unless a constrained triangulation is wanted. This is the only thing missing in the new implementation.

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

**See Also**

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#), [convex.hull](#), [neighbours](#), [add.constraint](#).

**Examples**

```
data(tritest)
tritest.tr<-tri.mesh(tritest$x,tritest$y)
tritest.tr
```

---

`triangles`*Extract a list of triangles from a triangulation object*

---

### Description

This function extracts a triangulation data structure from an triangulation object created by `tri.mesh`.

The vertices in the returned matrix (let's denote it with `retval`) are ordered counterclockwise with the first vertex taken to be the one with smallest index. Thus, `retval[i,"node2"]` and `retval[i,"node3"]` are larger than `retval[i,"node1"]` and index adjacent neighbors of node `retval[i,"node1"]`. The columns `trx` and `arcx`,  $x=1,2,3$  index the triangle and arc, respectively, which are opposite (not shared by) node `nodex`, with `trix=0` if `arcx` indexes a boundary arc. Vertex indexes range from 1 to `N`, triangle indexes from 0 to `NT`, and, if included, arc indexes from 1 to `NA = NT+N-1`. The triangles are ordered on first (smallest) vertex indexes, except that the sets of constraint triangles (triangles contained in the closure of a constraint region) follow the non-constraint triangles.

### Usage

```
triangles(tri.obj)
```

### Arguments

`tri.obj`            object of class "tri"

### Value

A matrix with columns `node1,node2,node3`, representing the vertex nodal indexes, `tr1,tr2,tr3`, representing neighboring triangle indexes and `arc1,arc2,arc3` representing arc indexes.

Each row represents one triangle.

### Author(s)

A. Gebhardt

### References

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

### See Also

[tri](#), [print.tri](#), [plot.tri](#), [summary.tri](#), [triangles](#)

**Examples**

```
# use a slightly modified version of data(tritest)
data(tritest2)
tritest2.tr<-tri.mesh(tritest2$x, tritest2$y)
triangles(tritest2.tr)
```

---

tripack-internal	<i>Internal functions</i>
------------------	---------------------------

---

**Description**

Internal tripack functions

**Details**

These functions are not intended to be called by the user.

---

tritest	<i>tritest / sample data</i>
---------	------------------------------

---

**Description**

A very simply set set of points to test the tripack functions, taken from the FORTRAN original. `tritest2` is a slight modification by adding `runif(, -0.1, 0.1)` random numbers to the coordinates.

**References**

R. J. Renka (1996). Algorithm 751: TRIPACK: a constrained two-dimensional Delaunay triangulation package. *ACM Transactions on Mathematical Software*. **22**, 1-8.

---

voronoi	<i>Voronoi object</i>
---------	-----------------------

---

**Description**

An voronoi object is created with [voronoi.mosaic](#)

**Arguments**

x, y	x and y coordinates of nodes of the voronoi mosaic. Each node is a circumcircle center of some triangle from the Delaunay triangulation.
node	logical vector, indicating real nodes of the voronoi mosaic. These nodes are the centers of circumcircles of triangles with positive area of the delaunay triangulation. If node[i]=FALSE, (c[i],x[i]) belongs to a triangle with area 0.
n1, n2, n3	indices of neighbour nodes. Negative indices indicate dummy points as neighbours.
tri	triangulation object, see <a href="#">tri</a> .
area	area of triangle i. area[i]=-1 indicates a removed triangle with area 0 at the border of the triangulation.
ratio	aspect ratio (inscribed radius/circumradius) of triangle i.
radius	circumradius of triangle i.
dummy.x, dummy.y	x and y coordinates of dummy points. They are used for plotting of unbounded tiles.

**Author(s)**

A. Gebhardt

**See Also**

[voronoi.mosaic](#), [plot.voronoi](#)

---

voronoi.area	<i>Calculate area of Voronoi polygons</i>
--------------	---

---

**Description**

Computes the area of each Voronoi polygon. For some sites at the edge of the region, the Voronoi polygon is not bounded, and so the area of those sites cannot be calculated, and hence will be NA.

**Usage**

```
voronoi.area(voronoi.obj)
```

**Arguments**

voronoi.obj     object of class "voronoi"

**Value**

A vector of polygon areas.

**Author(s)**

S. J. Eglen

**See Also**

[voronoi](#),

**Examples**

```
data(tritest)
tritest.vm <- voronoi.mosaic(tritest$x, tritest$y)
tritest.vm.areas <- voronoi.area(tritest.vm)
plot(tritest.vm)
text(tritest$x, tritest$y, tritest.vm.areas)
```

---

```
voronoi.findrejectsites
```

*Find the Voronoi sites at the border of the region (to be rejected).*

---

**Description**

Find the sites in the Voronoi tessellation that lie at the edge of the region. A site is at the edge if any of the vertices of its Voronoi polygon lie outside the rectangle with corners (xmin,ymin) and (xmax,ymax).

**Usage**

```
voronoi.findrejectsites(voronoi.obj, xmin, xmax, ymin, ymax)
```

**Arguments**

voronoi.obj     object of class "voronoi"  
xmin            minimum x-coordinate of sites in the region  
xmax            maximum x-coordinate of sites in the region  
ymin            minimum y-coordinate of sites in the region  
ymax            maximum y-coordinate of sites in the region

**Value**

A logical vector of the same length as the number of sites. If the site is a reject, the corresponding element of the vector is set to TRUE.

**Author(s)**

S. J. Eglen

**See Also**

[tri.dellens](#)

---

voronoi.mosaic	<i>Create a Voronoi mosaic</i>
----------------	--------------------------------

---

**Description**

This function creates a Voronoi mosaic.

It creates first a Delaunay triangulation, determines the circumcircle centers of its triangles, and connects these points according to the neighbourhood relations between the triangles.

**Usage**

```
voronoi.mosaic(x,y=NULL,duplicate="error")
```

**Arguments**

x	vector containing x coordinates of the data. If y is missing x should contain two elements \$x and \$y.
y	vector containing y coordinates of the data.
duplicate	flag indicating how to handle duplicate elements. Possible values are: "error" – default, "strip" – remove all duplicate points, "remove" – leave one point of duplicate points.

**Value**

An object of class [voronoi](#).

**Author(s)**

A. Gebhardt

**See Also**

[voronoi](#), [voronoi.mosaic](#), [print.voronoi](#), [plot.voronoi](#)

**Examples**

```
# example from TRIPACK:
data(tritest)
tritest.vm<-voronoi.mosaic(tritest$x,tritest$y)
tritest.vm
# use a part of the quakes data set:
data(quakes)
quakes.part<-quakes[(quakes[,1]<=-17 & quakes[,1]>=-19.0 &
                    quakes[,2]<=182.0 & quakes[,2]>=180.0),]
quakes.vm<-voronoi.mosaic(quakes.part$lon, quakes.part$lat, duplicate="remove")
quakes.vm
```

---

voronoi.polygons	<i>extract polygons from a voronoi mosaic</i>
------------------	---

---

**Description**

This functions extracts polygons from a voronoi.mosaic object.

**Usage**

```
voronoi.polygons(voronoi.obj)
```

**Arguments**

voronoi.obj     object of class voronoi.mosaic

**Value**

Returns an object of class voronoi.polygons with unnamed list elements for each polygon. These list elements are matrices with columns x and y.

**Author(s)**

Denis White

**See Also**

[plot.voronoi.polygons](#), [voronoi.mosaic](#)

**Examples**

```
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##--     or do help(data=index) for the standard data sets.

data(tritest)
tritest.vm <- voronoi.mosaic(tritest$x, tritest$y)
tritest.vp <- voronoi.polygons(tritest.vm)
tritest.vp
```

# Index

- \* **aplot**
  - circles, 5
- \* **datasets**
  - circtest, 6
  - tritest, 31
- \* **spatial**
  - add.constraint, 2
  - cells, 4
  - circum, 6
  - circumcircle, 7
  - convex.hull, 9
  - identify.tri, 10
  - in.convex.hull, 11
  - left, 12
  - neighbours, 13
  - on.convex.hull, 14
  - outer.convhull, 15
  - plot.tri, 16
  - plot.voronoi, 17
  - plot.voronoi.polygons, 19
  - print.summary.tri, 20
  - print.summary.voronoi, 20
  - print.tri, 21
  - print.voronoi, 22
  - summary.tri, 23
  - summary.voronoi, 24
  - tri, 25
  - tri.dellens, 26
  - tri.find, 27
  - tri.mesh, 28
  - triangles, 30
  - tripack-internal, 31
  - voronoi, 32
  - voronoi.area, 32
  - voronoi.findrejectsites, 33
  - voronoi.mosaic, 34
  - voronoi.polygons, 35
- add.constraint, 2, 9, 11, 14, 25, 29
- cells, 4
- circles, 5
- circtest, 6
- circtest2(circtest), 6
- circum, 6
- circumcircle, 6, 7, 7
- convex.hull, 3, 8, 9, 11, 14, 27, 29
- identify.tri, 10
- in.convex.hull, 11, 13–15
- interp, 29
- is.left.of(left), 12
- left, 12
- lines, 5
- neighbours, 13, 29
- on.convex.hull, 14
- outer.convhull, 15
- plot.tri, 3, 9–11, 13, 14, 16, 20, 22, 23, 25, 27, 29, 30
- plot.voronoi, 17, 21, 22, 24, 32, 34
- plot.voronoi.polygons, 19, 35
- points, 5
- print.summary.tri, 20, 23
- print.summary.voronoi, 20, 24
- print.tri, 3, 9–11, 13, 14, 16, 20, 21, 23, 25, 27, 29, 30
- print.voronoi, 18, 21, 22, 24, 34
- summary.tri, 3, 9–11, 13, 14, 16, 20, 22, 23, 25, 27, 29, 30
- summary.voronoi, 18, 21, 22, 24
- tri, 3, 9–11, 13, 14, 16, 20, 22, 23, 25, 27, 29, 30, 32
- tri.dellens, 26, 34
- tri.find, 27
- tri.mesh, 2, 20, 25, 28

`tri.swap` (tripack-internal), 31  
`tri.vordist` (tripack-internal), 31  
triangles, 3, 9, 11, 13, 14, 27, 29, 30, 30  
tripack-internal, 31  
tritest, 31  
`tritest2` (tritest), 31

voronoi, 18, 21, 22, 24, 32, 33, 34  
`voronoi.area`, 4, 32  
`voronoi.findrejectsites`, 26, 33  
`voronoi.findvertices`  
    (tripack-internal), 31  
`voronoi.mosaic`, 4, 21, 24, 26, 32, 34, 34, 35  
`voronoi.polyarea` (tripack-internal), 31  
`voronoi.polygons`, 19, 35