Package 'polyCub'

April 24, 2026

Title Cubature over Polygonal Domains

Version 0.9.4

Date 2026-04-24

Description Numerical integration of continuously differentiable functions f(x,y) over simple closed polygonal domains. The following cubature methods are implemented: product Gauss cubature (Sommariva and Vianello, 2007, <doi:10.1007/s10543-007-0131-2>), the simple two-dimensional midpoint rule (wrapping 'spatstat.geom' functions), and adaptive cubature for radially symmetric functions via line integrate() along the polygon boundary (Meyer and Held, 2014, <doi:10.1214/14-AOAS743>, Supplement B). For simple integration along the axes, the 'cubature' package is more appropriate.

License GPL-2

URL https://github.com/bastistician/polyCub

BugReports https://github.com/bastistician/polyCub/issues

Note Building the package requires R >= 4.6.0 for \bibshow{} et al.

Depends R (>= 3.4.0), methods

Imports grDevices, graphics, stats, sp (>= 1.0-11)

Suggests spatstat.geom, lattice, mvtnorm, statmod, sf, cubature, litedown (>= 0.9), microbenchmark

VignetteBuilder litedown

RoxygenNote 7.3.3

NeedsCompilation yes

Author Sebastian Meyer [aut, cre, trl] (ORCID: <https://orcid.org/0000-0002-1791-9449>), Leonhard Held [ths], Michael Hoehle [ths]

Maintainer Sebastian Meyer <seb.meyer@fau.de>

Repository CRAN

Date/Publication 2026-04-24 10:00:02 UTC

Contents

`polyCub-package` *Cubature over Polygonal Domains*

Description

The R package **polyCub** implements *cubature* (numerical integration) over *polygonal* domains. It solves the problem of integrating a continuously differentiable function $f(x, y)$ over simple closed polygons.

Details

polyCub provides the following cubature methods:

`polyCub.SV`**:** General-purpose *product Gauss cubature* (Sommariva and Vianello 2007)

`polyCub.midpoint`**:** Simple *two-dimensional midpoint rule* based on `as.im.function` from **spatstat.geom** (Baddeley, Rubak, and Turner 2015)

`polyCub.iso`**:** Adaptive cubature for *radially symmetric functions* via line `integrate()` along the polygon boundary (Meyer and Held 2014, Supplement B, Section 2.4)

A brief description and benchmark experiment of the above cubature methods can be found in the `vignette("polyCub")`.

There is also `polyCub.exact.Gauss`, intended to accurately (but slowly) integrate the *bivariate Gaussian density*; however, this implementation is disabled as of **polyCub** 0.9.0: it needs a reliable implementation of polygon triangulation.

Meyer (2010, Section 3.2) discusses and compares some of these methods.

Note

To cite package **polyCub** in publications, please use `citation("polyCub")`:

Meyer S (2019). "polyCub: An R package for Integration over Polygons." *Journal of Open Source Software*, **4**(34), 1056. ISSN 2475-9066. doi:10.21105/joss.01056.

Author(s)

Sebastian Meyer

References

Baddeley A, Rubak E, Turner R (2015). *Spatial Point Patterns: Methodology and Applications with R*. Chapman and Hall/CRC Press, London. ISBN 9781482210200.

Meyer S (2010). *Spatio-Temporal Infectious Disease Epidemiology based on Point Processes*. Master's thesis, Department of Statistics, LMU, Munich, Germany. https://epub.ub.uni-muenchen.de/11703/.

Meyer S, Held L (2014). "Power-law models for infectious disease spread." *Annals of Applied Statistics*, **8**(3), 1612–1639. doi:10.1214/14AOAS743.

Sommariva A, Vianello M (2007). "Product Gauss cubature over polygons based on Green's integration formula." *BIT Numerical Mathematics*, **47**(2), 441–453. doi:10.1007/s1054300701312.

See Also

`vignette("polyCub")`

For the special case of a rectangular domain along the axes (e.g., a bounding box), the **cubature** package is more appropriate.

`checkintrfr` *Check the Integral of $rf\_r(r)$*

Description

This function is auxiliary to `polyCub.iso` for the cubature of a radially symmetric function $f(x, y) = f_r(||(x, y) - \boldsymbol{\mu}||)$, with μ being the center of isotropy, over a polygonal domain. The (analytical) integral of $rf_r(r)$ from 0 to R, `intrfr`, is checked against an `integrate`-based approximation for various values (`rs`) of the upper bound R. A warning is issued if inconsistencies are found.

Usage

```
checkintrfr(intrfr, f, ..., center, control = list(), rs = numeric(0L),
  tolerance = control$rel.tol)
```

Arguments

intrfr	a function(R, ...), which implements the (analytical) antiderivative of $r f_r(r)$ from 0 to R. The first argument must be vectorized but not necessarily named R. If intrfr is missing, it will be approximated numerically via `integrate(function(r, ...) r * f(cbind(x0 + r, y0), ...), 0, R, ..., control = control)` where c(x0, y0) is the center of isotropy. Note that f will *not* be checked for isotropy.
f	a two-dimensional real-valued function. As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates.
...	further arguments for f or intrfr.
center	numeric vector of length 2, the center of isotropy.
control	list of arguments passed to integrate, the quadrature rule used for the line integral along the polygon boundary.
rs	numeric vector of upper bounds for which to check the validity of intrfr. If it has length 0 (default), no checks are performed.
tolerance	of all.equal.numeric when comparing intrfr results with numerical integration. Defaults to the relative tolerance used for integrate.

Value

The intrfr function, invisibly. If only f was given, an integrate-based approximation of intrfr is returned.

Examples

```
f_const <- function (coords) rep(1, nrow(coords))
intrfr_const <- function (R) R^2/2  # = \int_0^R r f_r(r) dr
checkintrfr(intrfr_const,  f = f_const, center = c(0,0), rs = 1:10) # OK
checkintrfr(function(R) R, f = f_const, center = c(0,0), rs = 1:10) # warns
```

---

circleCub.Gauss *Integration of the Isotropic Gaussian Density over Circular Domains*

Description

This function calculates the integral of the bivariate, isotropic Gaussian density (i.e., Σ = sd^2*diag(2)) over a circular domain via the cumulative distribution function pchisq of the (non-central) Chi-Squared distribution (Abramowitz and Stegun 1972, Formula 26.3.24).

Usage

```
circleCub.Gauss(center, r, mean, sd)
```

Arguments

center	numeric vector of length 2 (center of the circle).
r	numeric (radius of the circle). Several radii may be supplied.
mean	numeric vector of length 2 (mean of the bivariate Gaussian density).
sd	numeric (common standard deviation of the isotropic Gaussian density in both dimensions).

Value

The integral value (one for each supplied radius).

Note

The non-centrality parameter of the evaluated chi-squared distribution equals the squared distance between the mean and the center. If this becomes too large, the result becomes inaccurate, see pchisq.

References

Abramowitz M, Stegun IA (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, New York. ISBN 0486612724.

Examples

```
circleCub.Gauss(center=c(1,2), r=3, mean=c(4,5), sd=6)

## compare with cubature over a polygonal approximation of a circle
d2norm <- function (s, mean, sd)
  dnorm(s[,1], mean=mean[1], sd=sd) * dnorm(s[,2], mean=mean[2], sd=sd)
if (requireNamespace("spatstat.geom")) { # for the disc()
  npoly <- 32  # increase this for a closer match
  disc.poly <- spatstat.geom::disc(radius=3, centre=c(1,2), npoly=npoly)
  polyCub.iso(disc.poly, d2norm, mean=c(4,5), sd=6, center=c(4,5))
}
```

---

coerce-gpc-methods *Conversion between polygonal* "owin" *and* "gpc.poly"

Description

Package **polyCub** implements converters between the classes "owin" of package **spatstat.geom** and "gpc.poly" of package **gpclib**.

Usage

```
owin2gpc(object)

gpc2owin(object, ...)

as.owin.gpc.poly(W, ...)
```

Arguments

`object`	an object of class `"gpc.poly"` or `"owin"`, respectively.
`...`	further arguments passed to `owin`.
`W`	an object of class `"gpc.poly"`.

Value

The converted polygon of class `"gpc.poly"` or `"owin"`, respectively. If package **gpclib** is not available, `owin2gpc` will just return the `pts` slot of the `"gpc.poly"` (no formal class) with a warning.

Note

The converter `owin2gpc` requires the package **gpclib** for the formal class definition of a `"gpc.poly"`. It will produce vertices ordered according to the **sp** convention, i.e. clockwise for normal boundaries and anticlockwise for holes, where, however, the first vertex is *not* repeated!

Author(s)

Sebastian Meyer

See Also

`xylist`

Examples

```
## use example polygons from
example(plotpolyf, ask = FALSE)
letterR  # a simple "xylist"

letterR.owin <- spatstat.geom::owin(poly = letterR)
letterR.gpc_from_owin <- owin2gpc(letterR.owin)
## warns if "gpclib" is unavailable

if (is(letterR.gpc_from_owin, "gpc.poly")) {
  letterR.xylist_from_gpc <- xylist(letterR.gpc_from_owin)
  stopifnot(all.equal(letterR, lapply(letterR.xylist_from_gpc, `[`, 1:2)))
  letterR.owin_from_gpc <- gpc2owin(letterR.gpc_from_owin)
  stopifnot(all.equal(letterR.owin, letterR.owin_from_gpc))
}
```

coerce-sp-methods *Coerce* "SpatialPolygons" *to* "owin"

Description

Package **polyCub** implements coerce-methods (`as(object, Class)`) to convert "SpatialPolygons" (or "Polygons" or "Polygon") of package **sp** to "owin" of package **spatstat.geom**. They are also available as `as.owin.*` functions to support `polyCub.midpoint`.

Usage

```
as.owin.SpatialPolygons(W, ...)

as.owin.Polygons(W, ...)

as.owin.Polygon(W, ...)
```

Arguments

W	an object of class "SpatialPolygons", "Polygons", or "Polygon".
...	further arguments passed to owin.

Value

The polygon(s) as an "owin" object.

Author(s)

Sebastian Meyer

See Also

xylist

Examples

```
diamond <- list(x = c(1,2,1,0), y = c(1,2,3,2))  # anti-clockwise
diamond.sp <- sp::Polygon(lapply(diamond, rev))  # clockwise

diamond.Ps <- sp::Polygons(list(diamond.sp), ID = "my diamond")
diamond.SpPs <- sp::SpatialPolygons(list(diamond.Ps))

if (require("spatstat.geom")) {
    diamond.owin <- owin(poly = diamond)
    diamond.owin_from_Polygon <- as.owin(diamond.sp)
    stopifnot(all.equal(diamond.owin, diamond.owin_from_Polygon))
    ## also for "Polygons" and "SpatialPolygons", using S3 or S4 methods:
    stopifnot(identical(diamond.owin, as.owin(diamond.Ps)))
    stopifnot(identical(diamond.owin, as(diamond.SpPs, "owin")))
```

```
}
```

---

plotpolyf *Plot Polygonal Domain on Image of Bivariate Function*

Description

Produces a combined plot of a polygonal domain and an image of a bivariate function, using either `lattice::levelplot` or `image`.

Usage

```
plotpolyf(polyregion, f, ..., npixel = 100, cuts = 15,
  col = rev(heat.colors(cuts + 1)), lwd = 3, xlim = NULL, ylim = NULL,
  use.lattice = TRUE, print.args = list())
```

Arguments

`polyregion`	a polygonal domain. The following classes are supported: `"owin"` from package **spatstat.geom**, `"gpc.poly"` from **gpclib**, `"SpatialPolygons"`, `"Polygons"`, and `"Polygon"` from package **sp**, as well as `"(MULTI)POLYGON"` from package **sf**. (For these classes, **polyCub** knows how to get an `xylist`.)
`f`	a two-dimensional real-valued function. As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates.
`...`	further arguments for `f`.
`npixel`	numeric vector of length 1 or 2 setting the number of pixels in each dimension.
`cuts`	number of cut points in the z dimension. The range of function values will be divided into `cuts+1` levels.
`col`	color vector used for the function levels.
`lwd`	line width of the polygon edges.
`xlim`, `ylim`	numeric vectors of length 2 setting the axis limits. `NULL` means using the bounding box of `polyregion`.
`use.lattice`	logical indicating if **lattice** graphics (`levelplot`) should be used.
`print.args`	a list of arguments passed to `print.trellis` for plotting the produced `"trellis"` object (if `use.lattice = TRUE`). The `print` step is omitted if `print.args` is not a list.

Author(s)

Sebastian Meyer

Examples

```
### a polygonal domain (a simplified version of spatstat.data::letterR$bdry)
letterR <- list(
    list(x = c(2.7, 3, 3.3, 3.9, 3.7, 3.4, 3.8, 3.7, 3.4, 2, 2, 2.7),
         y = c(1.7, 1.6, 0.7, 0.7, 1.3, 1.8, 2.2, 2.9, 3.3, 3.3, 0.7, 0.7)),
    list(x = c(2.6, 2.6, 3, 3.2, 3),
         y = c(2.2, 2.7, 2.7, 2.5, 2.2))
)

### f: isotropic exponential decay
fr <- function(r, rate = 1) dexp(r, rate = rate)
fcenter <- c(2,3)
f <- function (s, rate = 1) fr(sqrt(rowSums(t(t(s)-fcenter)^2)), rate = rate)

### plot
plotpolyf(letterR, f, use.lattice = FALSE)
plotpolyf(letterR, f, use.lattice = TRUE)
```

---

polyCub *Wrapper Function for the Various Cubature Methods*

Description

The wrapper function polyCub can be used to call specific cubature methods via its method argument. It calls the polyCub.SV function by default, which implements general-purpose product Gauss cubature. The desired cubature function should usually be called directly.

Usage

```
polyCub(polyregion, f, method = c("SV", "midpoint", "iso", "exact.Gauss"),
  ..., plot = FALSE)
```

Arguments

polyregion	a polygonal domain. The following classes are supported: "owin" from package **spatstat.geom**, "gpc.poly" from **gpclib**, "SpatialPolygons", "Polygons", and "Polygon" from package **sp**, as well as "(MULTI)POLYGON" from package **sf**. (For these classes, **polyCub** knows how to get an xylist.)
f	a two-dimensional real-valued function to be integrated over polyregion. As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates. For the "exact.Gauss" method, f is ignored since it is specific to the bivariate normal density.
method	choose one of the implemented cubature methods (partial argument matching is applied), see help("polyCub-package") for an overview. Defaults to using product Gauss cubature implemented in polyCub.SV.

`...`	arguments of `f` or of the specific `method`.
`plot`	logical indicating if an illustrative plot of the numerical integration should be produced.

Value

The approximated integral of `f` over `polyregion`.

See Also

Details and examples in the `vignette("polyCub")` and on the method-specific help pages.

Other cubature methods: `polyCub.SV()`, `polyCub.iso()`, `polyCub.midpoint()`

---

polyCub.exact.Gauss *Quasi-Exact Cubature of the Bivariate Normal Density (DEFUNCT)*

Description

This cubature method is **defunct** as of **polyCub** version 0.9.0. It relied on `tristrip()` from package **gpclib** for polygon triangulation, but that package did not have a FOSS license and was no longer maintained on a mainstream repository.
Contributions to resurrect this cubature method are welcome: an alternative implementation for constrained polygon triangulation is needed, see https://github.com/bastistician/polyCub/issues/2.

Usage

```
polyCub.exact.Gauss(polyregion, mean = c(0, 0), Sigma = diag(2),
  plot = FALSE)
```

Arguments

`polyregion`	a `"gpc.poly"` polygon or something that can be coerced to this class, e.g., an `"owin"` polygon (via `owin2gpc`), or an `"sfg"` polygon (via `sfg2gpc`).
`mean, Sigma`	mean and covariance matrix of the bivariate normal density to be integrated.
`plot`	logical indicating if an illustrative plot of the numerical integration should be produced. Note that the `polyregion` will be transformed (shifted and scaled).

Details

The bivariate Gaussian density can be integrated based on a triangulation of the (transformed) polygonal domain, using formulae from the Abramowitz and Stegun (1972) handbook (Section 26.9, Example 9, pp. 956f.). This method is quite cumbersome because the A&S formula is only for triangles where one vertex is the origin (0,0). For each triangle we have to check in which of the 6 outer regions of the triangle the origin (0,0) lies and adapt the signs in the formula appropriately: $(AOB + BOC - AOC)$ or $(AOB - AOC - BOC)$ or $(AOB + AOC - BOC)$ or $(AOC + BOC - AOB)$ or .... However, the most time consuming step is the evaluation of `pmvnorm`.

Value

The integral of the bivariate normal density over `polyregion`. Two attributes are appended to the integral value:

`nEval`	number of triangles over which the standard bivariate normal density had to be integrated, i.e. number of calls to `pmvnorm` and `pnorm`, the former of which being the most time-consuming operation.
`error`	Approximate absolute integration error stemming from the error introduced by the `nEval` `pmvnorm` evaluations. For this reason, the cubature method is in fact only quasi-exact (as is the `pmvnorm` function).

References

Abramowitz M, Stegun IA (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, New York. ISBN 0486612724.

See Also

`circleCub.Gauss` for quasi-exact cubature of the isotropic Gaussian density over a circular domain.

`polyCub.iso` *Cubature of Isotropic Functions over Polygonal Domains*

Description

`polyCub.iso` numerically integrates a radially symmetric function $f(x,y) = f_r(||(x,y) - \boldsymbol{\mu}||)$, with μ being the center of isotropy, over a polygonal domain. It internally approximates a line integral along the polygon boundary using `integrate`. The integrand requires the antiderivative of $rf_r(r)$), which should be supplied as argument `intrfr` (`f` itself is only required if `check.intrfr=TRUE`). The two-dimensional integration problem thereby reduces to an efficient adaptive quadrature in one dimension. If `intrfr` is not available analytically, `polyCub.iso` can use a numerical approximation (meaning `integrate` within `integrate`), but the general-purpose cubature method `polyCub.SV` might be more efficient in this case. See Meyer and Held (2014, Supplement B, Section 2.4) for mathematical details.

`.polyCub.iso` is a "bare-bone" version of `polyCub.iso`.

Usage

```
polyCub.iso(polyregion, f, intrfr, ..., center, control = list(),
  check.intrfr = FALSE, plot = FALSE)

.polyCub.iso(polys, intrfr, ..., center, control = list(),
  .witherror = FALSE)
```

Arguments

polyregion	a polygonal domain. The following classes are supported: "owin" from package **spatstat.geom**, "gpc.poly" from **gpclib**, "SpatialPolygons", "Polygons", and "Polygon" from package **sp**, as well as "(MULTI)POLYGON" from package **sf**. (For these classes, **polyCub** knows how to get an xylist.)
f	a two-dimensional real-valued function. As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates.
intrfr	a function(R, ...), which implements the (analytical) antiderivative of $r f_r(r)$ from 0 to R. The first argument must be vectorized but not necessarily named R. If intrfr is missing, it will be approximated numerically via `integrate(function(r, ...) r * f(cbind(x0 + r, y0), ...), 0, R, ..., control = control)` where c(x0, y0) is the center of isotropy. Note that f will *not* be checked for isotropy.
...	further arguments for f or intrfr.
center	numeric vector of length 2, the center of isotropy.
control	list of arguments passed to integrate, the quadrature rule used for the line integral along the polygon boundary.
check.intrfr	logical (or numeric vector) indicating if (for which r's) the supplied intrfr function should be checked against a numerical approximation. This check requires f to be specified. If TRUE, the set of test r's defaults to a seq of length 20 from 1 to the maximum absolute x or y coordinate of any edge of the polyregion.
plot	logical indicating if an image of the function should be plotted together with the polygonal domain, i.e., plotpolyf(polyregion, f, ...).
polys	something like owin$bdry, but see xylist.
.witherror	logical indicating if an upper bound for the absolute integration error should be attached as an attribute to the result?

Value

The approximate integral of the isotropic function f over polyregion.
If the intrfr function is provided (which is assumed to be exact), an upper bound for the absolute integration error is appended to the result as attribute "abs.error". It equals the sum of the absolute errors reported by all integrate calls (there is one for each edge of polyregion).

Author(s)

Sebastian Meyer

The basic mathematical formulation of this efficient integration for radially symmetric functions was ascertained with great support by Emil Hedevang (Dept. of Mathematics, Aarhus University, Denmark) during the Summer School on Topics in Space-Time Modeling and Inference (May 2013, Aalborg, Denmark).

References

Meyer S, Held L (2014). "Power-law models for infectious disease spread." *Annals of Applied Statistics*, **8**(3), 1612–1639. doi:10.1214/14AOAS743.

See Also

`system.file("include", "polyCubAPI.h", package = "polyCub")` for a full C-implementation of this cubature method (for a *single* polygon). The corresponding C-routine `polyCub_iso` can be used by other R packages, notably **surveillance**, via 'LinkingTo: polyCub' (in the 'DESCRIPTION') and '#include <polyCubAPI.h>' (in suitable '/src' files). Note that the `intrfr` function must then also be supplied as a C-routine. An example can be found in the package tests.

Other cubature methods: `polyCub()`, `polyCub.SV()`, `polyCub.midpoint()`

Examples

```
## we use the example polygon and f (exponential decay) from
example(plotpolyf)

## numerical approximation of 'intrfr' (not recommended)
(intISOnum <- polyCub.iso(letterR, f, center = fcenter))

## analytical 'intrfr'
## intrfr(R) = int_0^R r*f(r) dr, for f(r) = dexp(r), gives
intrfr <- function (R, rate = 1) pgamma(R, 2, rate) / rate
(intISOana <- polyCub.iso(letterR, f, intrfr = intrfr, center = fcenter,
                          check.intrfr = TRUE))
## f is only used to check 'intrfr' against a numerical approximation

stopifnot(all.equal(intISOana, intISOnum, check.attributes = FALSE))


### polygon area: f(r) = 1, f(x,y) = 1, center does not really matter

## intrfr(R) = int_0^R r*f(r) dr = int_0^R r dr = R^2/2
intrfr.const <- function (R) R^2/2
(area.ISO <- polyCub.iso(letterR, intrfr = intrfr.const, center = c(0,0)))

if (require("spatstat.geom")) { # check against area.owin()
    stopifnot(all.equal(area.owin(owin(poly = letterR)),
                        area.ISO, check.attributes = FALSE))
}
```

`polyCub.midpoint` *Two-Dimensional Midpoint Rule*

Description

The surface is converted to a binary pixel image using the `as.im.function` method from package **spatstat.geom**. The integral under the surface is then approximated as the sum over (pixel area * f(pixel midpoint)).

Usage

```
polyCub.midpoint(polyregion, f, ..., eps = NULL, dimyx = NULL,
  plot = FALSE)
```

Arguments

polyregion	a polygonal integration domain. It can be any object coercible to the **spatstat.geom** class "owin" via a corresponding as.owin-method. Note that this includes polygons of the classes "gpc.poly" and "SpatialPolygons", because **polyCub** defines methods as.owin.gpc.poly and as.owin.SpatialPolygons, respectively. **sf** also registers suitable as.owin methods for its "(MULTI)POLYGON" classes.
f	a two-dimensional real-valued function. As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates.
...	further arguments for f.
eps	width and height of the pixels (squares), see as.mask.
dimyx	number of subdivisions in each dimension, see as.mask.
plot	logical indicating if an illustrative plot of the numerical integration should be produced.

Value

The approximated value of the integral of f over polyregion.

See Also

Other cubature methods: polyCub(), polyCub.SV(), polyCub.iso()

Examples

```
## a function to integrate (here: isotropic zero-mean Gaussian density)
f <- function (s, sigma = 5)
    exp(-rowSums(s^2)/2/sigma^2) / (2*pi*sigma^2)

## a simple polygon as integration domain
hexagon <- list(
    list(x = c(7.33, 7.33, 3, -1.33, -1.33, 3),
         y = c(-0.5, 4.5, 7, 4.5, -0.5, -3))
)

if (require("spatstat.geom")) {
    hexagon.owin <- owin(poly = hexagon)

    show_midpoint <- function (eps)
    {
        plotpolyf(hexagon.owin, f, xlim = c(-8,8), ylim = c(-8,8),
                  use.lattice = FALSE)
        ## add evaluation points to plot
```

```
        with(as.mask(hexagon.owin, eps = eps),
             points(expand.grid(xcol, yrow), col = t(m), pch = 20))
        title(main = paste("2D midpoint rule with eps =", eps))
    }

    ## show nodes (eps = 0.5)
    show_midpoint(0.5)

    ## show pixel image (eps = 0.5)
    polyCub.midpoint(hexagon.owin, f, eps = 0.5, plot = TRUE)

    ## use a decreasing pixel size (increasing number of nodes)
    for (eps in c(5, 3, 1, 0.5, 0.3, 0.1))
        cat(sprintf("eps = %.1f: %.7f\n", eps,
                    polyCub.midpoint(hexagon.owin, f, eps = eps)))
}
```

`polyCub.SV` *Product Gauss Cubature over Polygonal Domains*

Description

Product Gauss cubature over polygons as proposed by Sommariva and Vianello (2007).

Usage

```
polyCub.SV(polyregion, f, ..., nGQ = 20, alpha = NULL, rotation = FALSE,
  engine = "C", plot = FALSE)
```

Arguments

`polyregion`	a polygonal domain. The following classes are supported: `"owin"` from package **spatstat.geom**, `"gpc.poly"` from **gpclib**, `"SpatialPolygons"`, `"Polygons"`, and `"Polygon"` from package **sp**, as well as `"(MULTI)POLYGON"` from package **sf**. (For these classes, **polyCub** knows how to get an `xylist`.)
`f`	a two-dimensional real-valued function to be integrated over `polyregion` (or `NULL` to only compute nodes and weights). As its first argument it must take a coordinate matrix, i.e., a numeric matrix with two columns, and it must return a numeric vector of length the number of coordinates.
`...`	further arguments for `f`.
`nGQ`	degree of the one-dimensional Gauss-Legendre quadrature rule (default: 20) as implemented in function `gauss.quad` of package **statmod**. Nodes and weights up to `nGQ=60` are cached in **polyCub**, for larger degrees **statmod** is required.
`alpha`	base-line of the (rotated) polygon at $x = \alpha$ (see Sommariva and Vianello 2007, for an explication). If `NULL` (default), the midpoint of the x-range of each polygon is chosen if no `rotation` is performed, and otherwise the x-coordinate of the rotated point `"P"` (see `rotation`). If `f` has its maximum value at the origin $(0, 0)$, e.g., the bivariate Gaussian density with zero mean, `alpha = 0` is a reasonable choice.

`rotation` logical (default: `FALSE`) or a list of points `"P"` and `"Q"` describing the preferred direction. If `TRUE`, the polygon is rotated according to the vertices `"P"` and `"Q"`, which are farthest apart (see Sommariva and Vianello 2007). For convex polygons, this rotation guarantees that all nodes fall inside the polygon.

`engine` character string specifying the implementation to use. Up to **polyCub** version 0.4-3, the two-dimensional nodes and weights were computed by R functions and these are still available by setting `engine = "R"`. The new C-implementation is now the default (`engine = "C"`) and requires approximately 30% less computation time.
The special setting `engine = "C+reduce"` will discard redundant nodes at (0,0) with zero weight resulting from edges on the base-line $x = \alpha$ or orthogonal to it. This extra cleaning is only worth its cost for computationally intensive functions `f` over polygons which really have some edges on the baseline or parallel to the x-axis. Note that the old R implementation does not have such unset zero nodes and weights.

`plot` logical indicating if an illustrative plot of the numerical integration should be produced.

Value

The approximated value of the integral of `f` over `polyregion`.
In the case `f = NULL`, only the computed nodes and weights are returned in a list of length the number of polygons of `polyregion`, where each component is a list with `nodes` (a numeric matrix with two columns), `weights` (a numeric vector of length `nrow(nodes)`), the rotation `angle`, and `alpha`.

Author(s)

Sebastian Meyer

References

Sommariva A, Vianello M (2007). "Product Gauss cubature over polygons based on Green's integration formula." *BIT Numerical Mathematics*, **47**(2), 441–453. doi:10.1007/s1054300701312.
Their MATLAB implementation 'polygauss', on which this R implementation was based, is available (in revised versions) at https://sites.google.com/view/alvisesommarivaunipd/home-page/software/software_matlab under the GNU GPL (>=2) license.

See Also

Other cubature methods: `polyCub()`, `polyCub.iso()`, `polyCub.midpoint()`

Examples

```
## a function to integrate (here: isotropic zero-mean Gaussian density)
f <- function (s, sigma = 5)
    exp(-rowSums(s^2)/2/sigma^2) / (2*pi*sigma^2)

## a simple polygon as integration domain
hexagon <- list(
    list(x = c(7.33, 7.33, 3, -1.33, -1.33, 3),
```

```
         y = c(-0.5, 4.5, 7, 4.5, -0.5, -3))
)

## image of the function and integration domain
plotpolyf(hexagon, f)

## use a degree of nGQ = 3 and show the corresponding nodes
polyCub.SV(hexagon, f, nGQ = 3, plot = TRUE)

## extract nodes and weights
nw <- polyCub.SV(hexagon, f = NULL, nGQ = 3)[[1]]
nrow(nw$nodes)

## manually apply the cubature rule
sum(nw$weights * f(nw$nodes))

## use an increasing number of nodes
for (nGQ in c(1:5, 10, 20, 60))
    cat(sprintf("nGQ = %2i: %.16f\n", nGQ,
                polyCub.SV(hexagon, f, nGQ = nGQ)))

## polyCub.SV() is the default method used by the polyCub() wrapper
polyCub(hexagon, f, nGQ = 3)  # calls polyCub.SV()


### now using a simple *rectangular* integration domain

rectangle <- list(list(x = c(-1, 7, 7, -1), y = c(-3, -3, 7, 7)))
polyCub.SV(rectangle, f, plot = TRUE)

## effect of rotation given a very low nGQ
opar <- par(mfrow = c(1,3))
polyCub.SV(rectangle, f, nGQ = 4, rotation = FALSE, plot = TRUE)
           title(main = "without rotation (default)")
polyCub.SV(rectangle, f, nGQ = 4, rotation = TRUE,  plot = TRUE)
           title(main = "standard rotation")
polyCub.SV(rectangle, f, nGQ = 4,
           rotation = list(P = c(0,0), Q = c(2,-3)), plot = TRUE)
           title(main = "custom rotation")
par(opar)

## comparison with the "cubature" package
if (requireNamespace("cubature")) {
    fc <- function (s, sigma = 5)  # non-vectorized version of f
        exp(-sum(s^2)/2/sigma^2) / (2*pi*sigma^2)
    cubature::hcubature(fc, lowerLimit = c(-1, -3), upperLimit = c(7, 7))
}
```

---

sfg2gpc *Convert polygonal* "sfg" *to* "gpc.poly"

Description

Package **polyCub** implements a converter from class "(MULTI)POLYGON" of package **sf** to "gpc.poly" of package **gpclib** such that polyCub.exact.Gauss can be used with simple feature polygons.

Usage

```
sfg2gpc(object)
```

Arguments

object	a "POLYGON" or "MULTIPOLYGON" "sfg" object.

Value

The converted polygon of class "gpc.poly". If package **gpclib** is not available, sfg2gpc will just return the pts slot of the "gpc.poly" (no formal class) with a warning.

Note

Package **gpclib** is required for the formal class definition of a "gpc.poly".

Author(s)

Sebastian Meyer

See Also

xylist

Examples

```
## use example polygons from
example(plotpolyf, ask = FALSE)
letterR  # a simple "xylist"

letterR.sfg <- sf::st_polygon(lapply(letterR, function(xy)
    rbind(cbind(xy$x, xy$y), c(xy$x[1], xy$y[1]))))
letterR.sfg
stopifnot(identical(letterR, xylist(letterR.sfg)))

## convert sf "POLYGON" to a "gpc.poly"
letterR.gpc_from_sfg <- sfg2gpc(letterR.sfg)
letterR.gpc_from_sfg
```

xylist *Convert Various Polygon Classes to a Simple List of Vertices*

Description

Different packages concerned with spatial data use different polygon specifications, which sometimes becomes very confusing (see Details below). To be compatible with the various polygon classes, package **polyCub** uses an S3 class `"xylist"`, which represents a polygonal domain (of potentially multiple polygons) by its core feature only: a list of lists of vertex coordinates (see the "Value" section below). The generic function `xylist` can deal with the following polygon classes:

- `"owin"` from package **spatstat.geom**
- `"gpc.poly"` from package **gpclib**
- `"Polygons"` from package **sp** (as well as `"Polygon"` and `"SpatialPolygons"`)
- `"(MULTI)POLYGON"` from package **sf**

The (somehow useless) default `xylist`-method does not perform any transformation but only ensures that the polygons are not closed (first vertex not repeated).

Usage

```
xylist(object, ...)

## S3 method for class 'owin'
xylist(object, ...)

## S3 method for class 'sfg'
xylist(object, ...)

## S3 method for class 'gpc.poly'
xylist(object, ...)

## S3 method for class 'SpatialPolygons'
xylist(object, reverse = TRUE, ...)

## S3 method for class 'Polygons'
xylist(object, reverse = TRUE, ...)

## S3 method for class 'Polygon'
xylist(object, reverse = TRUE, ...)

## Default S3 method:
xylist(object, ...)
```

Arguments

`object`	an object of one of the supported spatial classes.
`...`	(unused) argument of the generic.
`reverse`	logical (`TRUE`) indicating if the vertex order of the **sp** classes should be reversed to get the `xylist`/`owin` convention.

Details

Polygon specifications differ with respect to:

- is the first vertex repeated?
- which ring direction represents holes?

Package overview:

spatstat.geom: `"owin"` does *not repeat* the first vertex, and anticlockwise = normal boundary, clockwise = hole. This convention is also used for the return value of `xylist`.

sp: *Repeat* first vertex at the end (closed), anticlockwise = hole, clockwise = normal boundary

sf: *Repeat* first vertex at the end (closed), clockwise = hole, anticlockwise = normal boundary; *however*, **sf** does not check the ring direction by default, so it cannot be relied upon.

gpclib: There seem to be no such conventions for polygons of class `"gpc.poly"`.

Thus, for polygons from **sf** and **gpclib**, `xylist` needs to check the ring direction, which makes these two formats the least efficient for integration domains in **polyCub**.

Value

Applying `xylist` to a polygon object, one gets a simple list, where each component (polygon) is a list of `"x"` and `"y"` coordinates. These represent vertex coordinates following **spatstat.geom**'s `"owin"` convention (anticlockwise order for exterior boundaries, without repeating any vertex).

Author(s)

Sebastian Meyer

Examples

```
diamond <- list(x = c(1,2,1,0), y = c(1,2,3,2))  # anti-clockwise
diamond.sp <- sp::Polygon(lapply(diamond, rev))  # clockwise

diamond.Ps <- sp::Polygons(list(diamond.sp), ID = "my diamond")
diamond.SpPs <- sp::SpatialPolygons(list(diamond.Ps))

stopifnot(identical(xylist(diamond.sp), list(diamond)))
stopifnot(identical(xylist(diamond.Ps), list(diamond)))
stopifnot(identical(xylist(diamond.SpPs), list(diamond)))
```

Index