# Package 'acousticTS'

November 12, 2026

**Title** Physics-Based Models for Acoustic Target Strength

**Version** 2.1.0

**Copyright** See file inst/COPYRIGHTS.

**Description** Acoustic target strength (TS) represents the intensity of an echo returning from an individual scatterer such as bubbles, fish, or zooplankton. TS can be used to convert integrated or volumetric backscatter collected from fisheries acoustic surveys into units of number density (e.g. animals per m^3), abundance (e.g. number of animals), and biomass (e.g. kg). This parameter can also be used to aid in classifying backscatter, such as separating likely echoes of large predatory fish (e.g. adult cod) from smaller prey (e.g. shrimp). One way to estimate TS is to use physics-based models to calculate theoretical TS that comprise exact and approximate solutions as well as analytical approaches. The models provided can help provide TS estimates over broad statistical distributions of model parameters. Applications are described by Lucca et al. (2023) <doi:10.1121/10.0022459>, with fisheries-acoustics principles from Simmonds and MacLennan (2005) <doi:10.1002/9780470995303>.

**License** GPL-3

**URL** https://brandynlucca.github.io/acousticTS/, https://doi.org/10.5281/zenodo.7600659

**BugReports** https://github.com/brandynlucca/acousticTS/issues

**Depends** R (>= 4.0.0)

**Imports** grDevices, graphics, methods, parallel, pbapply, Rcpp, stats, tools, utils

**Suggests** knitr, rmarkdown, testthat (>= 3.0.0)

**LinkingTo** BH, Rcpp, RcppArmadillo, testthat

**VignetteBuilder** knitr, rmarkdown

**Config/testthat/edition** 3

**Encoding** UTF-8

**Language** en-US

**Collate** 'RcppExports.R' 'acoustics.R' 'bessel-cylindrical.R'
'bessel-spherical.R' 'constants.R' 'utilities-scatterer.R'
'create_scatterer.R' 'create_shape.R' 'data.R' 'harmonics.R'
'math.R' 'models.R' 'model-bcms.R' 'model-bbfm.R'
'model-dwba.R' 'model-pcdwba.R' 'model-ecms.R' 'model-essms.R'
'model-vesms.R' 'model-fcms.R' 'model-hpa.R' 'model-krm.R'
'model-psms.R' 'model-tmm-helpers.R' 'model-tmm-spherical.R'
'model-tmm-spheroidal.R' 'model-tmm.R' 'tmm-scattering.R'
'tmm-orientation.R' 'tmm-diagnostics.R' 'tmm-bistatic.R'
'tmm-plotting.R' 'model-sdwba.R' 'model-soems.R'
'model-sphms.R' 'model-trcm.R' 'validation-registry.R'
'pkgdown-helpers.R' 'scatterer.R' 'scatterer_plot.R'
'scatterer_reforge.R' 'shapes.R' 'show.R' 'simulation.R'
'spheroidal.R' 'utilities.R' 'utilities-init.R'
'utilities-modal.R' 'utilities-plotting.R'
'utilities-geometry.R' 'shape_manipulation.R'
'canonicalize_shape.R' 'validation.R'

**Config/roxygen2/version** 8.1.1

**NeedsCompilation** yes

**Author** Brandyn Lucca [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-3145-2969>),
Arnie Lee Van Buren [ctb, cph] (Author and copyright holder of upstream
prolate_swf code adapted for this package),
Jeffrey E. Boisvert [ctb] (Coauthor of upstream prolate_swf code
adapted for this package)

**Maintainer** Brandyn Lucca <brandyn.lucca@gmail.com>

**Repository** CRAN

**Date/Publication** 2026-11-12 22:40:02 UTC

# Contents

along_sum . . . . . 4
arbitrary . . . . . 5
available_models . . . . . 6
BBF . . . . . 6
bbf_generate . . . . . 7
benchmark_ts . . . . . 9
brake . . . . . 10
bulk . . . . . 11
CAL . . . . . 12
cal_generate . . . . . 12
canonicalize_shape . . . . . 14
cod . . . . . 15
compressibility . . . . . 17
create_shape . . . . . 18
CSC . . . . . 19

cylinder . . . 20
degrees . . . 21
ELA . . . 21
ela_generate . . . 22
ESS . . . 23
ess_generate . . . 24
extract . . . 26
flip_shape . . . 27
FLS . . . 28
fls_generate . . . 28
GAS . . . 31
gas_generate . . . 31
gauss_legendre . . . 33
hc . . . 34
hs . . . 36
inflate_shape . . . 37
jc . . . 39
js . . . 41
krill . . . 43
lame . . . 44
linear . . . 45
neumann . . . 46
oblate_spheroid . . . 47
offset_component . . . 48
plot.Scatterer . . . 49
Pn . . . 51
Pndk . . . 52
pois . . . 54
polynomial_cylinder . . . 55
prolate_spheroid . . . 56
Qn . . . 57
Qndk . . . 59
radians . . . 60
reanchor_shape . . . 61
reforge . . . 62
register_model . . . 63
resample_shape . . . 64
reset_model_registry . . . 65
rho . . . 65
Rmn . . . 66
sardine . . . 68
SBF . . . 70
sbf_generate . . . 70
Scatterer . . . 73
shear . . . 74
simulate_ts . . . 75
Smn . . . 78
smooth_shape . . . 80

sphere . . . . . 81
target_strength . . . . . 82
tmm_average_orientation . . . . . 84
tmm_bistatic_summary . . . . . 86
tmm_diagnostics . . . . . 87
tmm_orientation_distribution . . . . . 89
tmm_products . . . . . 90
tmm_scattering . . . . . 92
tmm_scattering_grid . . . . . 93
translate_shape . . . . . 95
transmission_coefficient . . . . . 96
unregister_model . . . . . 96
vecnorm . . . . . 97
wavenumber . . . . . 98
yc . . . . . 98
young . . . . . 100
ys . . . . . 101

**Index** **104**

---

`along_sum` *Along-matrix summing function*

---

## Description

Along-matrix summing function

## Usage

```
along_sum(rpos, iterations)
```

## Arguments

| | |
|---|---|
| `rpos` | Position vector |
| `iterations` | Number of iterations |

## Value

A numeric matrix containing the adjacent column sums of `rpos`.

## Examples

```
positions <- matrix(1:8, nrow = 2)
along_sum(positions, ncol(positions))
```

`arbitrary` *Creates arbitrary body shape from user inputs*

## Description

Creates arbitrary body shape from user inputs

## Usage

```
arbitrary(..., length_units = "m")
```

## Arguments

| | |
|---|---|
| `...` | Any coordinate or parameter vectors (e.g., x_body, y_body, z_body, radius_body, w_body, zU_body, zL_body, custom_*). Inputs are stored and padded to a common length; validation happens downstream in model-specific code. |
| `length_units` | Units for body length. Defaults to meters: "m" |

## Value

An `Arbitrary` shape object containing the padded position matrix and stored shape metadata.

## See Also

`Arbitrary`

## Examples

```
# Symmetric arbitrary shape using x/y/z + radius
arbitrary(
  x_body = c(0, 0.01), y_body = c(0, 0), z_body = c(0, 0),
  radius_body = c(0, 0.002)
)

# Dorsal/ventral style inputs
arbitrary(
  x_body = c(0, 0.015),
  w_body = c(0.005, 0.0075),
  zU_body = c(0.001, 0.002),
  zL_body = c(-0.001, -0.002)
)
```

## available_models *List available target-strength models*

### Description

List available target-strength models

### Usage

```
available_models()
```

### Value

A data frame describing currently available built-in and user-registered target-strength models.

### Examples

```
head(available_models())
```

## BBF *Backboned fish (BBF) object/class.*

### Description

A S4 class that stores a fluid-like flesh component together with an explicit elastic backbone component for swimbladder-less fish applications where those two structures should remain separate. The body and backbone each retain their own geometry, orientations, and material properties. This class is intended to support composite flesh-plus-backbone model development and future extensions where additional internal structures may need to be carried alongside the body.

### Value

An object of the BBF S4 class.

### See Also

Scatterer, CSC

### Examples

```
methods::getClass("BBF")
```

bbf_generate *Generate a BBF-class object.*

## Description

Generate a BBF-class object.

## Usage

```
bbf_generate(
  body_shape,
  backbone_shape,
  density_body = NULL,
  sound_speed_body = NULL,
  g_body = NULL,
  h_body = NULL,
  density_backbone,
  sound_speed_longitudinal_backbone,
  sound_speed_transversal_backbone,
  theta_body = pi/2,
  theta_backbone = pi/2,
  x_offset_backbone = 0,
  z_offset_backbone = 0,
  theta_units = "radians",
  length_units = "m",
  ID = NULL
)
```

## Arguments

| | |
|---|---|
| body_shape | Pre-built body shape. Must inherit from Shape. |
| backbone_shape | Pre-built backbone shape. Must be a cylindrical Shape. |
| density_body | Flesh density ($\rho_{body}$, kg $m^3$). |
| sound_speed_body | Flesh sound speed ($c_{body}$, m $s^{-1}$). |
| g_body | Body density contrast. |
| h_body | Body sound speed contrast. |
| density_backbone | Backbone density ($\rho_{bb}$, kg $m^3$). |
| sound_speed_longitudinal_backbone | Longitudinal wave speed in the backbone (m/s). |
| sound_speed_transversal_backbone | Transversal wave speed in the backbone (m/s). |
| theta_body | Body orientation relative to the incident wave (radians). |

`theta_backbone` Backbone orientation relative to the incident wave (radians).

`x_offset_backbone`
: Along-body translation applied to the backbone geometry (m).

`z_offset_backbone`
: Dorsoventral translation applied to the backbone geometry (m).

`theta_units` Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives.

`length_units` Compatibility argument. Scatterer constructors now assume meters and ignore non-SI alternatives.

`ID` Optional metadata identifier.

## Details

`bbf_generate()` is intended for swimbladder-less fish workflows where the flesh and backbone should remain explicit, separately parameterized components. The body is stored using the same segmented-body representation used by `FLS`, while the backbone is stored as an explicit cylindrical component carrying elastic material properties for use by cylinder-based modal models.

## Value

Generates a BBF-class object.

## See Also

`BBF`, `FLS`, `cylinder`

## Examples

```
body_shape <- arbitrary(
  x_body = c(0, 0.04, 0.08),
  zU_body = c(0.001, 0.004, 0.001),
  zL_body = c(-0.001, -0.004, -0.001)
)
backbone_shape <- cylinder(
  length_body = 0.06,
  radius_body = 0.0008,
  n_segments = 40
)
bbf_generate(
  body_shape = body_shape,
  backbone_shape = backbone_shape,
  density_body = 1070,
  sound_speed_body = 1570,
  density_backbone = 1900,
  sound_speed_longitudinal_backbone = 3500,
  sound_speed_transversal_backbone = 1700
)
```

benchmark_ts *Benchmark model outputs from Jech et al. (2015)*

## Description

A packaged list of benchmark target-strength spectra for the canonical sphere, shell-sphere, finite-cylinder, and prolate-spheroid comparison cases used throughout the package validation workflow. The stored values follow the benchmark definitions assembled by Jech et al. (2015) and distributed in the echoSMs reference resources.

## Usage

```
data(benchmark_ts)
```

## Format

An object of class `list` of length 2.

## Value

A list of benchmark target-strength spectra.

## Source

echoSMs benchmark resources, especially the target-definition and Jech benchmark tables archived at https://github.com/ices-tools-dev/echoSMs.

## References

Jech, J.M., Horne, J.K., Chu, D., Demer, D.A., Francis, D.T.I., Gorska, N., Jones, B., Lavery, A.C., Stanton, T.K., and Reeder, D.B. (2015). Comparisons among ten models of acoustic backscattering used in aquatic ecosystem research. *The Journal of the Acoustical Society of America*, 138, 3742-3764. doi:10.1121/1.4937607

Jech, M., and Macaulay, G. echoSMs: Acoustic backscattering models used in aquatic ecosystem research. GitHub repository: https://github.com/ices-tools-dev/echoSMs

## Examples

```
data("benchmark_ts", package = "acousticTS")
str(benchmark_ts, max.level = 1)
```

brake *Bend a scatterer body or body component*

## Description

Apply a smooth curvature transformation to an existing scatterer body or to a list-like body component containing an `rpos` matrix. This is useful when a target should keep the same broad identity while adopting a curved centerline for model comparisons or sensitivity studies.

## Usage

```
brake(object, radius_curvature, mode = "ratio")
```

## Arguments

| | |
|---|---|
| `object` | Dataframe or scatterer-class object |
| `radius_curvature` | Radius of curvature that can be parameterized either as a ratio relative to body length or actual measurement |
| `mode` | Either "ratio" or "measurement" |

## Value

A bent version of `object`, returned as the same broad object type with updated geometry and curvature metadata.

## See Also

`extract()`, `reforge()`, `translate_shape()`, `reanchor_shape()`, `inflate_shape()`, `smooth_shape()`, `resample_shape()`, `flip_shape()`, `offset_component()`

## Examples

```
shape_obj <- cylinder(
  length_body = 0.05,
  radius_body = 0.003,
  n_segments = 80
)
obj <- fls_generate(
  shape = shape_obj,
  density_body = 1045,
  sound_speed_body = 1520
)

bent_obj <- brake(obj, radius_curvature = 5)
head(extract(bent_obj, c("body", "rpos", "z")))
extract(bent_obj, c("shape_parameters", "radius_curvature_ratio"))
```

```
bent_body <- brake(
  extract(obj, "body"),
  radius_curvature = 0.35,
  mode = "measurement"
)
head(bent_body$rpos["z", ])
```

---

`bulk` *Calculate the bulk modulus (K).*

---

## Description

Calculates the bulk modulus (K) from two of the three other elastic moduli: Young's modulus (E), shear modulus (G), or Poisson's ratio ($\nu$). Assumes 3D material properties.

The relationships used are:

$$K = \frac{EG}{3(3G - E)}$$

$$K = \frac{2G(1 + \nu)}{3(1 - 2\nu)}$$

$$K = \frac{E}{3(1 - 2\nu)}$$

## Usage

```
bulk(E = NULL, G = NULL, nu = NULL)
```

## Arguments

| | |
|---|---|
| `E` | Young's modulus (Pa). |
| `G` | Shear modulus (Pa). |
| `nu` | Poisson's ratio (Dimensionless). |

## Value

Bulk modulus (K, Pa).

## Examples

```
bulk(E = 3e9, G = 1.1e9)
```

## CAL *Solid and calibration sphere (CAL) object/class.*

### Description

A S4 class that provides slots to contain relevant metadata for solid sphere objects belonging to the CAL-class scatterers. This object is created using parameters specific to the outer shell. The default behavior of this object is to only reference these outer elastic shell properties with few exceptions that are model-dependent. `CAL` inherits from the broader elastic-based ELA class rather than from ESS, because calibration targets are solid elastic bodies rather than elastic shells. See Scatterer for a more detailed description on how this S4 object is organized.

### Value

An object of the `CAL` S4 class.

### See Also

Scatterer

### Examples

```
methods::getClass("CAL")
```

## cal_generate *Generate a CAL-class object.*

### Description

Generate a CAL-class object.

### Usage

```
cal_generate(
  material = "WC",
  diameter = 0.0381,
  sound_speed_longitudinal = NULL,
  sound_speed_transversal = NULL,
  density_sphere = NULL,
  theta_sphere = pi,
  ID = NULL,
  diameter_units = "m",
  theta_units = "radians",
  n_segments = 100
)
```

## Arguments

| | |
|---|---|
| material | Material type for the solid sphere. See 'Details' for built-in material options. |
| diameter | Spherical diameter (m). |
| sound_speed_longitudinal | Longitudinal sound speed (m/s). |
| sound_speed_transversal | Transversal sound speed (m/s). |
| density_sphere | Density (kg/m^3). |
| theta_sphere | Backscattering direction (Default: pi radians). |
| ID | Optional metadata ID input. |
| diameter_units | Compatibility argument. cal_generate() now assumes meters and ignores non-SI alternatives. |
| theta_units | Compatibility argument. cal_generate() now assumes radians and ignores non-SI alternatives. |
| n_segments | Number of segments to discretize object shape. |

## Details

There are several options for the **material** argument:

| **Material** | **Argument** | **c1** | **c2** | $\rho 1$ |
|---|---|---|---|---|
| *Tungsten carbide* | "WC" | 6853 | 4171 | 14900 |
| *Stainless steel* | "steel" | 5980 | 3297 | 7970 |
| *Brass* | "brass" | 4372 | 2100 | 8360 |
| *Copper* | "Cu" | 4760 | 2288.5 | 8947 |
| *Aluminum* | "Al" | 6260 | 3080 | 2700 |

## Value

Generates a CAL-class object.

## See Also

CAL

## Examples

```
cal_generate(material = "WC", diameter = 38.1e-3, n_segments = 120)
```

canonicalize_shape *Canonicalize one shape into a canonical surrogate*

## Description

Build an explicit canonical surrogate shape from an existing Shape object. This is intended for workflows where a measured or segmented body should be approximated deliberately by one of the package's canonical geometries before being passed to a shape-specific model family such as SPHMS, PSMS, or FCMS.

## Usage

```
canonicalize_shape(
  shape,
  to = c("ProlateSpheroid", "Cylinder", "Sphere", "OblateSpheroid"),
  method = c("auto", "volume", "length_volume", "profile_l2"),
  n_segments = NULL,
  diagnostics = FALSE
)
```

## Arguments

| | |
|---|---|
| shape | A Shape object to approximate. |
| to | Canonical target shape. One of "Sphere", "Cylinder", "ProlateSpheroid", or "OblateSpheroid". |
| method | Canonicalization rule. "auto" selects a shape-specific default: "volume" for spheres, "length_volume" for cylinders, and "profile_l2" for spheroids. "volume" preserves enclosed volume only. "length_volume" preserves body length and enclosed volume. "profile_l2" fits the target canonical profile directly to the source equivalent-radius profile by least squares. |
| n_segments | Optional number of segments for the returned canonical shape. Defaults to the source shape segment count. |
| diagnostics | Logical. If FALSE (default), return only the canonical Shape. If TRUE, return a list containing the canonical shape and fit diagnostics. |

## Details

canonicalize_shape() is intentionally explicit. It does **not** run automatically inside model calls such as target_strength(..., model = "psms"), because there is no single defensible canonicalization rule for every biological or segmented target.

When the input shape contains both an explicit width profile and upper/lower height profiles, the canonicalization step first reduces those local cross-sections to an equal-area circular radius before fitting the canonical surrogate. That keeps the reduction axisymmetric while still honoring the original local cross-sectional area more closely than a height-only radius.

The returned diagnostics report how the source and target compare in length, enclosed volume, maximum equivalent radius, and equivalent-radius profile root-mean-square error on the source x-grid. Those diagnostics are the package's recommended way to judge whether a canonical surrogate is defensible for the intended model comparison.

## Value

If `diagnostics = FALSE`, a canonical `Shape` object. If `diagnostics = TRUE`, a list with elements:

- `shape`: the fitted canonical `Shape`
- `diagnostics`: a named list of source, target, and fit metrics

## See Also

`create_shape()`, `arbitrary()`, `sphere()`, `cylinder()`, `prolate_spheroid()`, `oblate_spheroid()`

## Examples

```
bladder_like <- arbitrary(
  x_body = c(0, 0.01, 0.02, 0.03, 0.04),
  radius_body = c(0, 0.004, 0.006, 0.004, 0)
)

psms_shape <- canonicalize_shape(
  bladder_like,
  to = "ProlateSpheroid"
)

cyl_fit <- canonicalize_shape(
  bladder_like,
  to = "Cylinder",
  diagnostics = TRUE
)
cyl_fit$diagnostics$fit$radius_nrmse
```

---

# cod *Sample cod shape with fully inflated swimbladder*

---

## Description

A pre-generated SBF scatterer containing all information required for target strength modeling. The packaged object corresponds to the historical Atlantic cod example used in the KRM literature and matches the Cod D case archived in the `echoSMs` KRM shape collection.

## Usage

```
data(cod)
```

## Format

A pre-generated SBF scatterer containing all information required for target strength modeling.

**metadata** Relevant and identifying metadata (`list`).

**model_parameters** Container for specified model parameters (`list`).

**model** Model outputs and results (`list`).

**body** List with:

- `rpos`: Position matrix (x, yw, zU, zL; m).
- `sound_speed`: Flesh sound speed ($c_{body}$, m/s).
- `density`: Flesh density ($\rho_{body}$, kg/m$^3$).
- `theta`: Orientation relative to transducer ($\theta_{body}$, radians).

**bladder** List with:

- `rpos`: Position matrix (x, yw, zU, zL; m).
- `sound_speed`: Flesh sound speed ($c_{bladder}$, m/s).
- `density`: Flesh density ($\rho_{bladder}$, kg/m$^3$).
- `theta`: Orientation relative to transducer ($\theta_{bladder}$, radians).

**shape_parameters** Named list with:

- `body`: List with length (m), ncyl (int), theta_units (str), length_units (str).
- `bladder`: List with length (m), ncyl (int), theta_units (str), length_units (str).

## Value

A pre-generated `SBF` scatterer object.

## Source

Historical KRM cod shape collection archived in `echoSMs` (https://github.com/ices-tools-dev/echoSMs) and associated with the Clay and Horne Atlantic cod example.

## References

Clay, C.S., and Horne, J.K. (1994). Acoustic models of fish: The Atlantic cod (*Gadus morhua*). *The Journal of the Acoustical Society of America*, 96, 1661-1668. doi:10.1121/1.410245

## Examples

```
data("cod", package = "acousticTS")
cod
```

`compressibility` *Calculate the compressibility ($\kappa$) of a scattering boundary/interface.*

## Description

Calculates the compressibility contrast ($\kappa$) between a scattering interface and the surrounding medium. Compressibility is defined as:

$$K = \frac{1}{\rho c^2}$$

where $\rho$ is density ($kg\ m^{-3}$) and $c$ is sound speed ($m\ s^{-1}$).

The compressibility contrast is then:

$$\kappa = \frac{K_2 - K_1}{K_1}$$

where $K_1$ is the compressibility of the medium and $K_2$ is that of the target interface.

## Usage

```
compressibility(medium, target)
```

## Arguments

| | |
|---|---|
| `medium` | Dataframe object containing density ($kgm^{-3}$) and sound speed ($ms^{-1}$) values for a fluid medium external to a scattering interface (e.g., seawater). |
| `target` | Dataframe object containing density ($kgm^{-3}$) and sound speed ($ms^{-1}$) values for a target boundary. |

## Value

Compressibility contrast ($\kappa$), dimensionless.

## Examples

```
seawater <- data.frame(density = 1026, sound_speed = 1480)
animal <- data.frame(density = 1050, sound_speed = 1530)
compressibility(seawater, animal)
```

`create_shape` *A wrapper function that automatically creates generalized and/or canonical shapes for TS modeling.*

## Description

A wrapper function that automatically creates generalized and/or canonical shapes for TS modeling.

## Usage

```
create_shape(shape, ...)
```

## Arguments

| | |
|---|---|
| `shape` | Shape. Details for shape specification are provided under 'Details', including mandatory arguments. |
| `...` | Additional input arguments for subsequent shape generation functions. |

## Details

The **shape** argument specifies what shape for the function to generate the desired shape for TS modeling. Options currently include:

| **Object shape** | **shape = ...** | **Parameters** | **Root function** |
|---|---|---|---|
| *Discrete/tapered cylinder* | "cylinder" | length, radius | `cylinder(...)` |
| *Polynomial cylinder* | "polynomial_cylinder" | length, radius, polynomial | `polynomial_cylinder(...)` |
| *Oblate spheroid* | "oblate_spheroid" | length, radius | `oblate_spheroid(...)` |
| *Prolate spheroid* | "prolate_spheroid" | length, radius | `prolate_spheroid(...)` |
| *Sphere* | "sphere" | radius | `sphere(...)` |

**Model Parameter Definitions:**

- **length**: the x-axis length of the shape.
- **radius**: the radius of the shape when applicable.
- **length_radius_ratio**: the length-to-radius ratio (L/A), which specifically refers to the radius at the mid-point of the cylinder and should be the maximum value. A typical L/A ratio in the literature is 16 for krill.
- **taper**: the taper order (n), which parameterizes the tapering function reported by Chu *et al.* (1993) to create a tapered cylinder. The tapering order will converge on a prolate and oblate spheroid when L > 2a and L < 2a, respectively, and n = 2. A typical taper order in the literature is 10.
- **polynomial**: the vector of arbitrary polynomial coefficients to generate a deformed cylinder as reported by Smith *et al.* (2013). Although listed as a mandatory argument for the polynomial cylinder function, it has a default setting that uses the sixth-degree polynomial coefficients reported by Smith *et al.* (2013).

## Value

A Shape object.

## Examples

```
create_shape("sphere", radius_body = 0.01)
create_shape(
  "prolate_spheroid",
  length_body = 0.04, radius_body = 0.004
)
create_shape(
  "oblate_spheroid",
  length_body = 0.012, radius_body = 0.01
)
create_shape(
  "cylinder",
  length_body = 0.05, radius_body = 0.003
)
```

---

# CSC *Composite scatterer (CSC) object/class.*

---

## Description

A shared parent S4 class for composite scatterers that retain a primary body component together with one or more additional anatomically distinct sub-components. The CSC-class is intended to provide modular scaffolding for multi-component biological targets, such as swimbladder-bearing fish and swimbladder-less fish represented as flesh plus backbone. It stores the primary body, a general component list, the shared model output slot, and shape metadata without implying any one specific resonant or elastic inclusion type.

## Value

An object of the CSC S4 class.

## See Also

Scatterer, SBF, BBF

## Examples

```
methods::getClass("CSC")
```

cylinder *Creates a cylinder.*

## Description

Creates a cylinder.

## Usage

```
cylinder(length_body, radius_body = NULL, length_radius_ratio = NULL,
  taper = NULL, radius_curvature_ratio = NULL, n_segments = 100,
  length_units = "m")
```

## Arguments

| | |
|---|---|
| length_body | Length (m). |
| radius_body | Maximum/uniform radius (m). |
| length_radius_ratio | Optional ratio input when radius is not explicitly known. |
| taper | Optional input that is the degree of taper to round ends of the cylinder. |
| radius_curvature_ratio | Optional curvature ratio metadata for rounded cylinder-end workflows. |
| n_segments | Number of segments to discretize object shape. Defaults to 1e2 segments. |
| length_units | Units (default is meters, "m"). |

## Value

Creates the position vector for a tapered or untapered cylinder.

## See Also

Cylinder

## Examples

```
cylinder(length_body = 0.05, radius_body = 0.003, n_segments = 80)
```

## degrees *Convert angular measurements from radians to degrees*

### Description

Convert angular measurements from radians to degrees

### Usage

```
degrees(x)
```

### Arguments

| | |
|---|---|
| x | A real value in radians |

### Value

Angle in degrees

### Examples

```
orientation <- pi / 2 # radians
degrees(orientation) # this should return a value equal to 90 degrees
```

## ELA *Elastic-based scatterer (ELA) object/class.*

### Description

A shared parent S4 class for elastic-based scatterers. The ELA-class collects the slots common to solid elastic targets such as calibration spheres (CAL) and elastic-shelled targets (ESS), without implying a particular internal structure such as a shell or homogeneous elastic solid. See Scatterer for a more detailed description of the shared object organization.

### Value

An object of the ELA S4 class.

### See Also

Scatterer, CAL, ESS

### Examples

```
methods::getClass("ELA")
```

ela_generate *Generate a ELA-class object.*

## Description

Generate a ELA-class object.

## Usage

```
ela_generate(
  shape,
  density_body = NULL,
  sound_speed_longitudinal_body = NULL,
  sound_speed_transversal_body = NULL,
  theta_body = pi/2,
  ID = NULL,
  length_units = "m",
  theta_units = "radians"
)
```

## Arguments

| | |
|---|---|
| shape | Pre-built Shape object describing the elastic target geometry. |
| density_body | Optional body density (kg/m^3). |
| sound_speed_longitudinal_body | Optional longitudinal wave speed (m/s). |
| sound_speed_transversal_body | Optional transversal wave speed (m/s). |
| theta_body | Body orientation relative to the incident wave (radians). |
| ID | Optional metadata identifier. |
| length_units | Compatibility argument. Scatterer constructors now assume meters and ignore non-SI alternatives. |
| theta_units | Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives. |

## Details

ela_generate() builds a generic solid-elastic scatterer under the shared ELA class. Unlike CAL, it is not restricted to spheres, so it can carry prolate, oblate, cylindrical, or arbitrary elastic shapes together with the body density and longitudinal/transversal wave speeds used by solid-elastic models such as TMM.

## Value

ELA-class object.

## See Also

ELA, CAL, ESS

## Examples

```
elastic_sphere <- sphere(radius_body = 0.01, n_segments = 40)
ela_generate(
  shape = elastic_sphere,
  density_body = 7800,
  sound_speed_longitudinal_body = 5900,
  sound_speed_transversal_body = 3200
)
```

---

# ESS *Elastic shelled scatterer (ESS) object/class.*

---

## Description

A S4 class that provides slots to contain relevant metadata for elastic shelled scatterers/objects belonging to the ESS-class. This object can be created using values for both an outer shell and internal tissues, if applicable. The default behavior for this type of this object is to only reference the outer shell with few exceptions that are model-dependent. ESS inherits from the broader elastic-based ELA class. See Scatterer for a more detailed description on how this S4 object is organized.

## Value

An object of the ESS S4 class.

## See Also

Scatterer

## Examples

```
methods::getClass("ESS")
```

ess_generate *Generate an ESS-class object*

## Description

Generate an ESS-class object

## Usage

```
ess_generate(
  shape = NULL,
  x_body = NULL,
  y_body = NULL,
  z_body = NULL,
  radius_shell = NULL,
  shell_thickness = NULL,
  g_fluid = NULL,
  density_fluid = NULL,
  h_fluid = NULL,
  sound_speed_fluid = NULL,
  g_shell = NULL,
  density_shell = NULL,
  h_shell = NULL,
  sound_speed_shell = NULL,
  E = NULL,
  G = NULL,
  K = NULL,
  nu = NULL,
  theta_shell = pi/2,
  ID = NULL,
  theta_units = "radians",
  length_units = "m"
)
```

## Arguments

| | |
|---|---|
| shape | Pre-built Shape object describing the outer shell geometry. If omitted, explicit shell profile coordinates such as x_body, y_body, and z_body are treated as the manual geometry pathway. Legacy character dispatch such as "sphere" is retained only for backward compatibility and is now deprecated. |
| x_body | Vector containing x-axis body (m) shape data. |
| y_body | Vector containing y-axis body (m) shape data. |
| z_body | Vector containing z-axis body (m) shape data. |
| radius_shell | Radius of shell (m). |
| shell_thickness | Optional shell thickness (m). |

| | |
|---|---|
| g_fluid | Optional density contrast for fluid-like body. |
| density_fluid | Optional density for fluid-like body (kg/m³). |
| h_fluid | Optional sound speed contrast for fluid-like body. |
| sound_speed_fluid | Optional sound speed for fluid-like body (m/s). |
| g_shell | Density contrast for the shell. |
| density_shell | Optional density for the shell (kg/m³). |
| h_shell | Sound speed contrast for the shell. |
| sound_speed_shell | Optional sound speed for the shell (m/s). |
| E | Young's modulus (Pa) of the shell material. |
| G | Shear modulus (Pa) of the shell material. |
| K | Bulk modulus (Pa) of the shell material. |
| nu | Poisson's ratio (Dimensionless) of the shell material. |
| theta_shell | Object orientation relative to incident sound wave. |
| ID | Optional metadata entry. |
| theta_units | Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives. |
| length_units | Compatibility argument. Scatterer constructors now assume meters and ignore non-SI alternatives. |

## Details

The preferred workflow is to build the shell geometry first as a Shape object and pass it through shape, or to supply explicit shell profile coordinates directly. Character-based shape dispatch remains available only as a compatibility pathway and is now deprecated. Material properties for both the shell and the inner fluid may be supplied either as contrasts or as absolute values, but each property pair must use only one representation.

Scatterer constructors store geometry in meters and orientations in radians. length_units and theta_units are retained as compatibility arguments, but non-SI values are normalized to the package-standard representation.

## Value

ESS-class object

## See Also

ESS

## Examples

```
shell <- sphere(radius_body = 0.03, n_segments = 80)
ess_generate(
  shape = shell,
  shell_thickness = 0.001,
  density_shell = 1050,
  sound_speed_shell = 2350,
  density_fluid = 1030,
  sound_speed_fluid = 1500,
  E = 3.5e9,
  nu = 0.34
)
```

---

# extract *Extract nested components, slots, or matrix/vector fields from objects*

---

## Description

Convenience accessor for reaching into `Scatterer` and `Shape` objects without spelling out direct slot access repeatedly. `extract()` can also walk through nested lists, matrices, and named vectors, which makes it useful for pulling model outputs, component properties, or position-matrix fields from a common interface. This is especially helpful after geometry manipulations such as `brake()` and `reforge()`, where inspecting the stored body profile is often the fastest way to confirm what changed.

## Usage

```
extract(object, feature)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer-class object. |
| `feature` | Feature(s) of interest (e.g. body). This can either be a scalar string, or a vector of names. When a vector is supplied, the function recursively accesses the Scatterer object using the 'feature' vector as a directory. For example, `feature = c("body", "rpos", "x")` would extract the 'x' coordinate of the position matrix ('rpos') from the 'body' scattering parameters. |

## Value

The extracted object, slot, list element, matrix row/column, or vector element identified by `feature`.

## See Also

`brake()`, `reforge()`, `translate_shape()`, `reanchor_shape()`, `inflate_shape()`, `smooth_shape()`, `resample_shape()`, `flip_shape()`, `offset_component()`

## Examples

```
obj <- fls_generate(
  shape = sphere(radius_body = 0.01, n_segments = 40),
  density_body = 1045,
  sound_speed_body = 1520
)

extract(obj, "body")
extract(obj, c("body", "density"))
extract(obj, c("shape_parameters", "shape"))

bent_obj <- brake(obj, radius_curvature = 5)
head(extract(bent_obj, c("body", "rpos", "z")))
extract(bent_obj, c("shape_parameters", "radius_curvature_ratio"))
```

---

# flip_shape *Flip a stored shape or scatterer profile across the x or z axis*

---

## Description

Reverse axial orientation (`axis = "x"`) or mirror the geometry dorsoventrally (`axis = "z"`) without manually editing the position matrix.

## Usage

```
flip_shape(
  object,
  axis = c("x", "z"),
  component = NULL,
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| `object` | Shape or Scatterer object. |
| `axis` | Flip axis. Use `"x"` to reverse nose/tail orientation while keeping the existing x grid, or `"z"` to mirror the profile vertically. |
| `component` | Optional component name for scatterers. Defaults to the primary geometry (`"body"` for most scatterers and `"shell"` for `ESS`). |
| `containment` | Containment policy used when a moved swimbladder or backbone is checked against its body: `"warn"`, `"error"`, or `"ignore"`. |

## Value

The modified object, returned as the same broad object type.

## See Also

translate_shape(), reanchor_shape(), inflate_shape(), smooth_shape()

## Examples

```
shape_obj <- arbitrary(
  x_body = c(0, 0.01, 0.02, 0.03),
  radius_body = c(0, 0.004, 0.002, 0)
)
flipped_shape <- flip_shape(shape_obj, axis = "x")
head(extract(flipped_shape, c("position_matrix", "zU")))
```

---

# FLS *Fluid-like scatterer (FLS) object/class.*

---

## Description

A S4 class that provides slots to contain relevant metadata for scatterers similar to the surrounding fluid medium (i.e fluid-like) belonging to FLS-class scatterers. See Scatterer for a more detailed description on how this S4 object is organized.

## Value

An object of the FLS S4 class.

## See Also

Scatterer

## Examples

```
methods::getClass("FLS")
```

---

# fls_generate *Generate a FLS-class object.*

---

## Description

Generate a FLS-class object.

## Usage

```
fls_generate(
  shape = NULL,
  x_body = NULL,
  y_body = NULL,
  z_body = NULL,
  length_body = NULL,
  radius_body = NULL,
  radius_curvature_ratio = NULL,
  n_segments = 18,
  g_body = NULL,
  h_body = NULL,
  density_body = NULL,
  sound_speed_body = NULL,
  theta_body = pi/2,
  ID = NULL,
  length_units = "m",
  theta_units = "radians",
  ...
)
```

## Arguments

| | |
|---|---|
| shape | Pre-built Shape object describing the target geometry. If omitted, explicit profile coordinates such as x_body, y_body, and z_body are treated as the manual geometry pathway. Legacy character dispatch such as "sphere" or "arbitrary" is retained only for backward compatibility and is now deprecated. |
| x_body | Vector containing x-axis body (m) shape data. |
| y_body | Vector containing y-axis body (m) shape data. |
| z_body | Vector containing z-axis body (m) shape data. |
| length_body | Optional input for a generic length value input. |
| radius_body | Vector containing radii (m). |
| radius_curvature_ratio | Length-to-curvature ratio (pc/L). |
| n_segments | Number of body segments. |
| g_body | Density contrast. This can either be a single value (i.e. homogenous) or a vector of values (i.e. inhomogenous). |
| h_body | Sound-speed contrast. This can either be a single value (i.e. homogenous) or a vector of values (i.e. inhomogenous). |
| density_body | Absolute density (kg/m^3) if contrasts are not supplied. |
| sound_speed_body | Absolute sound speed (m/s) if contrasts are not supplied. |
| theta_body | Orientation of the target relative to the transmit source ($\theta$). Broadside incidence is considered 90 degrees, or pi/2. Default value is pi/2; input should be in radians. |

| | |
|---|---|
| ID | Optional metadata entry. |
| length_units | Compatibility argument. Scatterer constructors now assume meters and ignore non-SI alternatives. |
| theta_units | Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives. |
| ... | Additional parameters. |

## Details

The preferred workflow is to build a geometry first with sphere(), cylinder(), prolate_spheroid(), or arbitrary(), then pass that Shape object to fls_generate(). Material properties can be supplied either as contrasts (g_body/h_body) or as absolute density/sound-speed values (density_body/sound_speed_body), but not both for the same property pair. Downstream models derive contrasts automatically when only absolute values are supplied.

The only supported public geometry paths are now:

1. supply a pre-built Shape object, or
2. supply explicit profile coordinates directly to the constructor.

Character-based shape dispatch is retained only as a compatibility pathway and is now deprecated. Internally, every pathway is resolved to the same Shape-first geometry contract before the FLS object is built.

Scatterer constructors store geometry in meters and orientations in radians. length_units and theta_units are retained as compatibility arguments, but non-SI values are normalized to the package-standard representation.

## Value

FLS-class object

## See Also

FLS

## Examples

```
shape <- prolate_spheroid(
  length_body = 0.04, radius_body = 0.004, n_segments = 50
)
fls_generate(
  shape = shape, density_body = 1045, sound_speed_body = 1520
)
```

## GAS *Generic gas-filled scatterer (GAS) object/class.*

### Description

A S4 class that provides slots to contain relevant metadata for gas-bearing scatterers belonging to the GAS-class. This object can include simple gas-filled bubbles to other scatterers with gas occlusions, swimbladders, and other internal features, if applicable. The default behavior for this type of object is to only reference the gaseous/fluid feature with exceptions that are model-dependent. See Scatterer for a more detailed description on how this S4 object is organized.

### Value

An object of the GAS S4 class.

### See Also

Scatterer

### Examples

```
methods::getClass("GAS")
```

## gas_generate *Generate a GAS-class object*

### Description

Generate a GAS-class object

### Usage

```
gas_generate(shape = NULL, radius_body = NULL, h_fluid = 0.22,
  g_fluid = 0.0012, sound_speed_fluid = NULL, density_fluid = NULL,
  theta_body = pi/2, ID = NULL, radius_units = "m",
  theta_units = "radians", n_segments = 100, ...)
```

### Arguments

| | |
|---|---|
| shape | Pre-built Shape object describing the gas-filled geometry. If omitted, explicit profile coordinates such as x_body, y_body, and z_body are treated as the manual geometry pathway. Legacy character dispatch such as "sphere" is retained only for backward compatibility and is now deprecated. |
| radius_body | Radius (m). For non-canonical shapes, this would be the maximum or mean radius at the scatterer midsection. |

| | |
|---|---|
| h_fluid | Sound speed contrast of fluid relative to surrounding medium (h). |
| g_fluid | Density contrast of fluid relative to surrounding density (g). |
| sound_speed_fluid | Optional fluid sound speed (m/s). |
| density_fluid | Optional fluid density (m/s). |
| theta_body | Orientation of the target relative to the incident wave (radians). |
| ID | Optional metadata identifier. |
| radius_units | Compatibility argument. gas_generate() now assumes meters and ignores non-SI alternatives. |
| theta_units | Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives. |
| n_segments | Number of body segments. |
| ... | Additional manual profile arguments or legacy canonical shape arguments used by the compatibility geometry pathway, such as x_body, y_body, z_body, radius_body, length_body, or taper. |

## Details

The preferred workflow is to supply a pre-built Shape object or explicit profile coordinates and then describe the internal gas by either contrasts (g_fluid, h_fluid) or absolute density/sound-speed values (density_fluid, sound_speed_fluid). Character-based shape dispatch remains available only as a compatibility pathway and is now deprecated.

Scatterer constructors store geometry in meters and orientations in radians. radius_units and theta_units are retained as compatibility arguments, but non-SI values are normalized to the package-standard representation.

## Value

GAS-class object

## See Also

GAS

## Examples

```
shape_gas <- sphere(radius_body = 0.01, n_segments = 60)
gas_generate(shape = shape_gas, g_fluid = 0.0012, h_fluid = 0.22)
```

`gauss_legendre` *Gauss-Legendre nodes and weights*

## Description

Compute Gauss-Legendre quadrature nodes and weights on an interval $[a,\ b]$.

## Usage

```
gauss_legendre(n, a = -1, b = 1)
```

## Arguments

| | |
|---|---|
| `n` | Number of quadrature nodes (n >= 1). |
| `a` | Left endpoint of the integration interval. |
| `b` | Right endpoint of the integration interval (b > a). |

## Details

Gauss-Legendre quadrature provides exact integration for polynomials of degree up to $2n-1$ using n nodes and weights chosen as the roots of the Legendre polynomial $P_n(x)$ on the canonical interval $[-1, 1]$. For a general interval $[a, b]$ the canonical nodes are shifted and rescaled onto $[a, b]$, and the weights are scaled by $(b-a)/2$.

This wrapper performs basic argument validation and calls the C++ routine to obtain nodes and weights with high accuracy for moderate `n`.

## Value

A list with components:

**nodes** Quadrature abscissae $x_i$ in $[a,\ b]$.

**weights** Quadrature weights $w_i$ such that $\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i\, f(x_i)$.

## References

Davis, P. J., & Rabinowitz, P. (2007). Methods of Numerical Integration (2nd ed.).

## Examples

```
rule <- gauss_legendre(n = 4, a = 0, b = 1)
sum(rule$weights * rule$nodes^2)
```

hc *Cylindrical Bessel function of the third kind (Hankel), H_$\nu(x)$, and its respective derivatives*

## Description

Computes the cylindrical Hankel function of the first kind ($H_\nu^{(1)}(z)$) and its derivatives through `hcdk()`.

## Usage

```
hc(l, n)

hcdk(l, n, k)
```

## Arguments

| | |
|---|---|
| `l` | Numeric. The order ($\nu$) of the Hankel function. Must be purely real; complex orders are not supported. |
| `n` | Numeric or complex. The argument ($z$) at which to evaluate the function. Supports purely real or purely imaginary values. General complex arguments are not supported. |
| `k` | Non-negative integer. The order of the derivative for `hcdk`. |

## Details

The Hankel function of the first kind is defined as:

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$

where $J_\nu(z)$ is the Bessel function of the first kind and $Y_\nu(z)$ is the Bessel function of the second kind.

**Supported argument types:** Since $H_\nu^{(1)}(z)$ is computed from $J_\nu(z)$ and $Y_\nu(z)$, the same restrictions apply:

- **Purely real arguments** ($z = x$): Fully supported.
- **Purely imaginary arguments** ($z = iy$): Supported.
- **General complex arguments** ($z = x + iy$): **Not supported**.

**Derivatives:**

- First derivative: $\frac{d}{dz}H_\nu^{(1)}(z) = \frac{\nu}{z}H_\nu^{(1)}(z) - H_{\nu+1}^{(1)}(z)$
- Second derivative: $\frac{d^2}{dz^2}H_\nu^{(1)}(z) = H_{\nu-2}^{(1)}(z) - \frac{2\nu-1}{z}H_{\nu-1}^{(1)}(z) + \frac{\nu^2+\nu}{z^2}H_\nu^{(1)}(z)$
- k-th derivative (DLMF 10.6.1): $\frac{d^k}{dz^k}H_\nu^{(1)}(z) = \frac{1}{2^k}\sum_{j=0}^{k}(-1)^j\binom{k}{j}H_{\nu-k+2j}^{(1)}(z)$

## Value

A complex vector containing:

- hc: $H_\nu^{(1)}(z)$
- hcdk(..., k = 1): $\frac{d}{dz} H_\nu^{(1)}(z)$ (first derivative)
- hcdk(..., k = 2): $\frac{d^2}{dz^2} H_\nu^{(1)}(z)$ (second derivative)
- hcdk: $\frac{d^k}{dz^k} H_\nu^{(1)}(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 9.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Hankel function definition: https://dlmf.nist.gov/10.2
- k-th derivative formula: Eq. 10.6.1 at https://dlmf.nist.gov/10.6

## See Also

jc for Bessel functions of the first kind, yc for Bessel functions of the second kind, hs for spherical Hankel functions.

## Examples

```
# Hankel function
hc(0, 1)
hc(1, 2.5)

# Fractional order
hc(0.5, 3)

# Purely imaginary argument
hc(1, 1i)

# First derivative
hcdk(1, 2, 1)

# Second derivative
hcdk(1, 2, 2)

# k-th derivative
hcdk(1, 2, 3) # Third derivative
```

hs *Spherical Bessel function of the third kind (Hankel), h_ν(x), and its respective derivatives*

## Description

Computes the spherical Hankel function of the first kind ($h_\nu^{(1)}(z)$) and its first-th derivative.

## Usage

```
hs(l, n)

hsdk(l, n, k)
```

## Arguments

| | |
|---|---|
| l | Numeric. The order of the spherical Hankel function. Can be integer or fractional. |
| n | Numeric. The argument ($z$) at which to evaluate the function. |
| k | Non-negative integer. The order of the derivative for hsdk. |

## Details

The spherical Hankel function of the first kind is defined as:

$$h_\nu^{(1)}(z) = j_\nu(z) + iy_\nu(z)$$

where $j_\nu(z)$ is the spherical Bessel function of the first kind and $y_\nu(z)$ is the spherical Bessel function of the second kind.

It is related to the cylindrical Hankel function by:

$$h_\nu^{(1)}(z) = \sqrt{\frac{\pi}{2z}} H_{\nu+1/2}^{(1)}(z)$$

The spherical Hankel functions are used extensively in scattering theory to represent outgoing spherical waves.

**Derivative:**

$$\frac{d}{dz} h_\nu^{(1)}(z) = \frac{\nu}{z} h_\nu^{(1)}(z) - h_{\nu+1}^{(1)}(z)$$

## Value

A complex vector containing:

- hs: $h_\nu^{(1)}(z)$
- hsdK: $\frac{d}{dz^k} h_l^{(1)}(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 10.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Spherical Bessel functions: https://dlmf.nist.gov/10.47
- Relation to cylindrical Hankel functions: Eq. 10.47.5 at https://dlmf.nist.gov/10.47

## See Also

hc for cylindrical Hankel functions, js for spherical Bessel functions of the first kind, ys for spherical Bessel functions of the second kind.

## Examples

```
# Spherical Hankel function
hs(0, 1)
hs(1, 2.5)

# Fractional order
hs(0.5, 3)

# Vector input
hs(0, c(1, 2, 3))

# First derivative
hsdk(1, 2, 1)
```

---

# inflate_shape *Locally widen, pinch, or taper a stored shape profile*

---

## Description

Apply a localized scaling window to a stored profile. Values of `scale > 1` inflate the selected region, while `scale < 1` pinch or taper it. For canonical `Shape` objects, the result is returned as an `Arbitrary` shape because the edited profile is no longer guaranteed to preserve the canonical class geometry.

## Usage

```
inflate_shape(
  object,
  x_range = NULL,
  scale = 1,
  component = NULL,
  axis = c("radius", "width", "height"),
```

```
  profile = c("cosine", "linear", "box"),
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| object | Shape or Scatterer object. |
| x_range | Optional axial interval over which the local manipulation is applied. |
| scale | Positive local scale factor. |
| component | Optional component name for scatterers. Defaults to the primary geometry ("body" for most scatterers and "shell" for ESS). |
| axis | Which profile dimension to scale. |
| profile | Local window profile. "cosine" gives a smooth bump centered inside x_range, "linear" gives a triangular bump, and "box" applies a uniform factor inside the interval. |
| containment | Containment policy used when a moved swimbladder or backbone is checked against its body: "warn", "error", or "ignore". |

## Value

The modified object. Shape inputs that are locally reshaped are returned as Arbitrary shapes.

## See Also

smooth_shape(), resample_shape(), flip_shape(), brake(), reforge()

## Examples

```
shape_obj <- cylinder(
  length_body = 0.05, radius_body = 0.003,
  n_segments = 80
)
pinched_shape <- inflate_shape(
  shape_obj,
  x_range = c(0.015, 0.035),
  scale = 0.7
)
max(extract(pinched_shape, c("shape_parameters", "max_radius")))
```

jc *Cylindrical Bessel function of the first kind, J_ν(z), and its respective derivatives*

## Description

Computes the cylindrical Bessel function of the first kind ($J_\nu(z)$) and its k-th derivatives.

## Usage

```
jc(l, n)

jcdk(l, n, k)
```

## Arguments

| | |
|---|---|
| l | Numeric. The order ($\nu$) of the Bessel function. Must be purely real; complex orders are not supported. |
| n | Numeric or complex. The argument ($z$) at which to evaluate the function. Supports purely real or purely imaginary values. General complex arguments ($x+iy$ with $x \neq 0$ and $y \neq 0$) are not supported. |
| k | Non-negative integer. The order of the derivative for `jcdk`. |

## Details

The cylindrical Bessel function of the first kind satisfies Bessel's differential equation:

$$z^2 \frac{d^2 J_\nu}{dz^2} + z \frac{dJ_\nu}{dz} + (z^2 - \nu^2) J_\nu = 0$$

**Supported argument types:**

- **Purely real arguments** ($z = x$, where $x \in \mathbb{R}$): Fully supported for both positive and negative values.
- **Purely imaginary arguments** ($z = iy$, where $y \in \mathbb{R}$): Computed using the identity $J_\nu(iy) = e^{i\pi\nu/2} I_\nu(y)$ for $y > 0$, where $I_\nu$ is the modified Bessel function of the first kind. For negative imaginary arguments and real order, $J_\nu(-iy) = \overline{J_\nu(iy)}$.
- **General complex arguments** ($z = x + iy$, where $x \neq 0$ and $y \neq 0$): **Not supported**.

**Special cases:**

- $J_\nu(0) = 1$ if $\nu = 0$, otherwise $J_\nu(0) = 0$.
- For negative real arguments with integer order $n$: $J_n(-x) = (-1)^n J_n(x)$.
- For negative real arguments with non-integer order $\nu$: $J_\nu(-x) = e^{i\pi\nu} J_\nu(x)$ (complex result).

**Derivatives:**

- First derivative: $J'_\nu(z) = J_{\nu-1}(z) - \frac{\nu}{z} J_\nu(z)$

- Second derivative: $J''_\nu(z) = \frac{1}{4}\left[J_{\nu-2}(z) - 2J_\nu(z) + J_{\nu+2}(z)\right]$
- k-th derivative (DLMF 10.6.1):

$$\frac{d^k}{dz^k} J_\nu(z) = \frac{1}{2^k} \sum_{j=0}^{k} (-1)^j \binom{k}{j} J_{\nu-k+2j}(z)$$

## Value

A complex vector containing:

- `jc`: $J_\nu(z)$
- `jcdk(..., k = 1)`: $J'_\nu(z)$ (first derivative)
- `jcdk(..., k = 2)`: $J''_\nu(z)$ (second derivative)
- `jcdk`: $J^{(k)}_\nu(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 9.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Bessel's equation: https://dlmf.nist.gov/10.2
- Negative argument identity ($J_\nu(-z)$): Eq. 10.4.1 at https://dlmf.nist.gov/10.4
- Imaginary argument identity ($J_\nu(iz)$): Eq. 10.27.6 at https://dlmf.nist.gov/10.27

## See Also

`yc` for Bessel functions of the second kind, `hc` for Hankel functions (third kind), `js` for spherical Bessel functions of the first kind.

## Examples

```
# Real argument, integer order
jc(0, 1)
jc(1, 2.5)

# Real argument, fractional order
jc(0.5, 3)

# Negative real argument (integer order gives real result)
jc(2, -1.5)

# Negative real argument (fractional order gives complex result)
jc(0.5, -2)

# Purely imaginary argument
jc(1, 1i)
jc(2, 3i)
```

```
# First derivative
jcdk(1, 2, 1)

# Second derivative
jcdk(1, 2, 2)
```

---

`js` *Spherical Bessel function of the first kind, j_ν(z), and its respective derivatives*

---

## Description

Computes the spherical Bessel function of the first kind ($j_\nu(z)$) and its k-th derivative.

## Usage

```
js(l, n)

jsdk(l, n, k)
```

## Arguments

| | |
|---|---|
| `l` | Numeric. The order of the spherical Bessel function. Can be integer or fractional. |
| `n` | Numeric or matrix. The argument ($z$) at which to evaluate the function. If a matrix is provided, the function is applied column-wise. |
| `k` | Non-negative integer. The order of the derivative for `jsdk`. |

## Details

The spherical Bessel function of the first kind is related to the cylindrical Bessel function by:

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+1/2}(z)$$

where $J_\nu(z)$ is the cylindrical Bessel function of the first kind.

The spherical Bessel functions satisfy the differential equation:

$$z^2 \frac{d^2 j_\nu}{dz^2} + 2z \frac{dj_\nu}{dz} + [z^2 - l(l+1)] j_\nu = 0$$

**Special cases:**

- $j_\nu(0) = 0$ for all $\nu$.
- $j_0(z) = \frac{\sin(z)}{z}$

- $j_1(z) = \frac{\sin(z)}{z^2} - \frac{\cos(z)}{z}$

**Derivatives:**

- First derivative: $j'_\nu(z) = j_{\nu-1}(z) - \frac{\nu+1}{z} j_\nu(z)$
- Second derivative: $j''_\nu(z) = \frac{(\nu+1)(\nu+2)-z^2}{z^2} j_\nu(z) - \frac{2}{z} j_{\nu-1}(z)$

## Value

A numeric vector or matrix (matching the input structure) containing:

- `js`: $j_\nu(z)$
- `jsdk`: $j_l^{(k)'}(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 10.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Spherical Bessel functions: https://dlmf.nist.gov/10.47
- Relation to cylindrical Bessel functions: Eq. 10.47.3 at https://dlmf.nist.gov/10.47

## See Also

`jc` for cylindrical Bessel functions of the first kind, `ys` for spherical Bessel functions of the second kind, `hs` for spherical Hankel functions.

## Examples

```
# Spherical Bessel function
js(0, 1)
js(1, 2.5)

# Fractional order
js(0.5, 3)

# Vector input
js(0, c(1, 2, 3))

# Matrix input (applied column-wise)
js(1, matrix(1:6, nrow = 2))

# First derivative
jsdk(1, 2, 1)

# Second derivative
jsdk(1, 2, 2)
```

krill *Sample krill (Euphausia superba) shape taken from McGehee et al. (1998)*

## Description

A dataset containing a sample krill (Euphausia superba) body shape proposed by McGehee et al. (1998).

## Usage

```
data(krill)
```

## Format

A pre-generated FLS scatterer containing all information required for target strength modeling.

**metadata** Relevant and identifying metadata (`list`).

**model_parameters** Container for specified model parameters (`list`).

**model** Model outputs and results (`list`).

**body** A list with elements:

- `rpos`: Position matrix (x, y, z; m).
- `radius`: Radius of each discrete cylinder (m).
- `g`: Body density contrast relative to medium.
- `h`: Sound speed contrast relative to medium.
- `theta`: Orientation angle ($\theta_{body}$, radians).

**shape_parameters** A list with:

- `length`: Body length (m).
- `ncyl`: Number of cylinders.
- `theta_units`: Unit of orientation.
- `length_units`: Unit of length.

## Value

A pre-generated `FLS` scatterer object.

## Examples

```
data("krill", package = "acousticTS")
krill
```

`lame` *Calculate Lamé's first parameter ($\lambda$)*

## Description

Calculates Lamé's first parameter ($\lambda$) from two of the four other elastic moduli: bulk modulus (K), Young's modulus (E), shear modulus (G), or Poisson's ratio ($\nu$). Assumes 3D material properties.

The relationships used are:

$$\lambda = K - \frac{2G}{3}$$

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

$$\lambda = \frac{2G\nu}{1-2\nu}$$

$$\lambda = \frac{3K\nu}{1+\nu}$$

$$\lambda = \frac{3K(3K-E)}{9K-E}$$

$$\lambda = \frac{G(E-2G)}{3G-E}$$

## Usage

```
lame(K = NULL, E = NULL, G = NULL, nu = NULL)
```

## Arguments

| | |
|---|---|
| `K` | Bulk modulus (Pa). |
| `E` | Young's modulus (Pa). |
| `G` | Shear modulus (Pa). |
| `nu` | Poisson's ratio (Dimensionless). |

## Value

Lamé's first parameter ($\lambda$, Pa).

## Examples

```
lame(K = 2.5e9, G = 1.1e9)
```

`linear` *Convert between logarithmic (dB) and linear domains for backscatter values.*

## Description

The `linear` function converts a value from the logarithmic (dB) domain to the linear domain, while the `db` function converts a value from the linear domain to the logarithmic (dB) domain. These are commonly used for target strength (TS) and backscattering coefficient ($\sigma_{bs}$) conversions.

The conversions are defined as:

$$\text{linear}(x) = c^{x/c}$$

$$\text{db}(x) = c \log_c(x)$$

where $c$ is the coefficient (default 10).

## Usage

```
linear(value, coefficient = 10)

db(value, coefficient = 10)
```

## Arguments

| | |
|---|---|
| `value` | Numeric value to convert. For `linear`, this is a logarithmic value (e.g., dB TS); for `db`, this is a linear value (e.g., $\sigma_{bs}$). |
| `coefficient` | Optional. Numeric coefficient (base) for the logarithm. Default is 10. |

## Value

For `linear`, returns the value converted to the linear domain. For `db`, returns the value converted to the logarithmic (dB) domain.

## Examples

```
sigma_bs <- linear(-40)
db(sigma_bs)
```

neumann *Compute the Neumann factor $\nu_n$*

## Description

The Neumann factor, denoted $\nu_n$, is a simple multiplicative constant commonly used in spherical or spheroidal function theory. It accounts for the symmetry of cosine terms or the duplication of even-order contributions in integrals.

## Usage

```
neumann(x)
```

## Arguments

| | |
|---|---|
| x | An integer iterator. |

## Details

Formally, the Neumann factor is defined as:

$$\eta_n = \begin{cases} 1, & n = 0, \\ 2, & n > 0. \end{cases}$$

This factor frequently appears in the normalization of spherical Bessel functions, Legendre expansions, and spheroidal wave functions. It is not related to the "von Neumann ordinals" used in set theory.

## Value

A numeric vector of the same length as x, containing values of $\eta_x$, each equal to 1 or 2.

## Examples

```
neumann(0) # should return 1
neumann(1) # should return 2
neumann(2) # should return 2

# Vectorized use:
neumann(0:4)
```

`oblate_spheroid` *Creates an oblate spheroid.*

## Description

Creates an oblate spheroid.

## Usage

```
oblate_spheroid(
  length_body = NULL,
  radius_body = NULL,
  length_radius_ratio = NULL,
  semimajor_length = NULL,
  semiminor_length = NULL,
  n_segments = 100,
  length_units = "m"
)
```

## Arguments

| | |
|---|---|
| `length_body` | Body-axis length (m). |
| `radius_body` | Maximum equatorial radius (m). |
| `length_radius_ratio` | Optional ratio input when radius is not explicitly known. |
| `semimajor_length` | Optional alias for maximum equatorial radius (m). |
| `semiminor_length` | Optional alias for body-axis semi-length (m). |
| `n_segments` | Number of segments to discretize object shape. Defaults to 100 segments. |
| `length_units` | Units for body matrix (defaults to m). |

## Value

Creates the position vector for an oblate spheroid object of defined body-axis length and maximum equatorial radius.

## See Also

`OblateSpheroid`

## Examples

```
oblate_spheroid(
  length_body = 0.012, radius_body = 0.01, n_segments = 60
)
oblate_spheroid(
  semiminor_length = 0.006, semimajor_length = 0.01
)
```

---

`offset_component` *Offset an internal scatterer component without rebuilding the object*

---

## Description

Translate an internal geometry-bearing component such as a swimbladder or backbone while leaving the outer body unchanged.

## Usage

```
offset_component(
  object,
  component = c("bladder", "backbone"),
  x_offset = 0,
  z_offset = 0,
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer object containing an internal component. |
| `component` | Internal component name. Currently most useful for `"bladder"` and `"backbone"`. |
| `x_offset` | Along-axis translation (m). |
| `z_offset` | Vertical translation (m). |
| `containment` | Containment policy used when the shifted component is checked against the body: `"warn"`, `"error"`, or `"ignore"`. |

## Value

The modified scatterer object.

## See Also

`translate_shape()`, `reanchor_shape()`, `reforge()`

## Examples

```
fish <- sbf_generate(
  x_body = c(0, 0.1),
  w_body = c(0.006, 0.008),
  zU_body = c(0.001, 0.002),
  zL_body = c(-0.001, -0.002),
  x_bladder = c(0.02, 0.08),
  w_bladder = c(0, 0),
  zU_bladder = c(0.0012, 0.0012),
  zL_bladder = c(-0.0012, -0.0012),
  density_body = 1040,
  density_bladder = 1.2,
  sound_speed_body = 1500,
  sound_speed_bladder = 340
)
shifted_fish <- offset_component(
  fish,
  component = "bladder",
  x_offset = 0.003
)
min(extract(shifted_fish, c("bladder", "rpos", "x_bladder")))
```

---

# plot.Scatterer *Plot scatterer geometry, stored model results, or stored TMM scattering views*

---

## Description

S3 plotting method for `Scatterer` objects. Depending on `type`, the method draws the target geometry, one or more stored model curves, or a stored single-target `TMM` scattering slice / map when the object was computed with retained T-matrix state.

## Usage

```
## S3 method for class 'Scatterer'
plot(
  x,
  y = NULL,
  type = "shape",
  nudge_y = 1.1,
  nudge_x = 1.05,
  aspect_ratio = "manual",
  x_units = "frequency",
  y_units = "TS",
  ...
)
```

## Arguments

| | |
|---|---|
| x | Scatterer-class object. |
| y | Ignored (required for the base plot() method signature). |
| type | Plot mode. Use "shape" for the stored geometry, "model" for the currently stored model output, or "scattering" for stored TMM angular products. |
| nudge_y | Multiplicative vertical padding applied to automatically derived y-axis limits. |
| nudge_x | Multiplicative horizontal padding applied to automatically derived x-axis limits. |
| aspect_ratio | Aspect-ratio control for supported shape plots. Use "manual" to honor nudge_x / nudge_y, or "auto" to let the plotting helper derive the aspect ratio directly. |
| x_units | Horizontal-axis convention for type = "model". Supported values depend on the stored model family and typically include "frequency" and geometry-scaled wavenumber variants such as "ka". |
| y_units | Stored response quantity for type = "model" or type = "scattering". Typical values include "TS" and "sigma_bs". |
| ... | Additional arguments passed through to the class-specific plotting helpers. For stored TMM scattering plots, this includes controls such as frequency, vary, polar, and heatmap. |

## Details

Plotting behavior is selected from the object's class. The supported type values therefore depend on what has been stored on the object. For example, type = "model" requires the object to already contain model output, and type = "scattering" currently applies only to stored TMM results.

## Value

Called for its side effect of drawing a plot; returns the input invisibly.

## See Also

extract(), target_strength(), tmm_scattering(), tmm_scattering_grid()

## Examples

```
data("krill", package = "acousticTS")
plot(krill)
```

Pn *Legendre Polynomial of the First Kind, P_ν(x)*

## Description

Computes the Legendre polynomial of the first kind, $P_\nu(x)$, for real order $\nu$ (integer or fractional) and real argument $x$.

## Usage

```
Pn(n, x)
```

## Arguments

| | |
|---|---|
| n | Numeric vector. Degree (order) of the Legendre polynomial. Can be integer or fractional (e.g., 0, 1, 2.5, 3.7). Can also be negative. |
| x | Numeric vector. Real argument(s) at which to evaluate the polynomial. Valid for all real values including $\|x\| > 1$. |

## Details

The Legendre polynomial of the first kind satisfies the differential equation:

$$(1-x^2)\frac{d^2P_\nu}{dx^2} - 2x\frac{dP_\nu}{dx} + \nu(\nu+1)P_\nu = 0$$

For integer order $n$, the function uses the recurrence relation:

$$P_0(x) = 1$$

$$P_1(x) = x$$

$$P_n(x) = \frac{(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)}{n}$$

For fractional order $\nu$, the function uses:

- For $|x| \leq 1$: The Ferrers function via the hypergeometric series $P_\nu(x) = {}_2F_1(-\nu, \nu+1; 1; \frac{1-x}{2})$
- For $|x| > 1$: A numerical contour integral representation

## Value

A numeric matrix of dimension `length(n)` by `length(x)`, where element `[i, j]` contains $P_{n_i}(x_j)$.

## Note

This function calls underlying $C++$ code via `Rcpp` for computational efficiency and to support different cases for both order and argument that are not readily available in `R`.

## References

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications. Chapter 8: Legendre Functions.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/14

## See Also

Pndk for the kth derivative of the Legendre polynomial of the first kind, Qn for Legendre functions of the second kind.

## Examples

```
# Single values
Pn(1, 1)

# Multiple orders, single argument
Pn(c(1, 2, 3), 1)

# Single order, multiple arguments
Pn(1, c(1, 2, 3))

# Multiple orders and arguments (returns a matrix)
Pn(c(1, 2, 3), c(1, 2, 3))

# Fractional orders
Pn(c(0.5, 2.2, 3), c(1, 2, 3))

# Negative arguments
Pn(c(0.5, 2, -3), c(1, -2.5, 3))
```

---

# Pndk *Derivative of the Legendre Polynomial of the First Kind*

---

## Description

Computes the $k$-th derivative of the Legendre polynomial of the first kind, $\frac{d^k}{dx^k} P_\nu(x)$, with respect to the argument $x$.

## Usage

```
Pndk(n, x, k = 1L)
```

## Arguments

| | |
|---|---|
| n | Numeric vector. Degree (order) of the Legendre polynomial. Can be integer or fractional. |
| x | Numeric vector. Real argument(s) at which to evaluate the derivative. |
| k | Integer. Order of the derivative ($k \geq 0$). Default is 1 for the first derivative. |

## Details

**For integer order $n$:**

The derivative is computed using the relationship with associated Legendre polynomials:

$$\frac{d^m P_n(x)}{dx^m} = \frac{1}{(1-x^2)^{m/2}} P_n^m(x)$$

where $P_n^m(x)$ is the associated Legendre polynomial (the `Boost C++` uses Condon-Shortley phase convention).

At the endpoints $x = \pm 1$, the known closed-form expressions are used:

$$P_n^{(k)}(1) = \frac{1}{2^k k!} \prod_{j=0}^{k-1} (n-j)(n+1+j)$$

$$P_n^{(k)}(-1) = (-1)^{n+k} P_n^{(k)}(1)$$

If $k > n$ for integer $n$, the result is 0 (derivative of a polynomial of degree $n$ taken more than $n$ times).

**For fractional order $\nu$:**

Derivatives are computed using central finite differences:

$$\frac{dP_\nu}{dx} \approx \frac{P_\nu(x+h) - P_\nu(x-h)}{2h}$$

Higher-order derivatives use the generalized finite difference stencil. Note that accuracy may be limited for fractional orders due to the numerical integration underlying `Pn`.

## Value

A numeric matrix of dimension `length(n)` by `length(x)`, where element `[i, j]` contains $\frac{d^k}{dx^k} P_{n_i}(x_j)$.

## Note

For fractional orders, the finite difference approximation may have reduced accuracy (typically 4-6 significant digits) compared to integer orders. A warning is issued for higher-order derivatives ($k > 1$) of fractional orders.

This function calls underlying $C++$ code via `Rcpp` for computational efficiency and to support different cases for both order and argument that are not readily available in `R`.

## References

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions*. Dover Publications. Section 8.5: Associated Legendre Functions.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/14.10

## See Also

`Pn` for the Legendre polynomial of the first kind.

## Examples

```
# First derivative of P_2(x) at x = 0.5
# P_2(x) = (3x^2 - 1)/2, so P'_2(x) = 3x, P'_2(0.5) = 1.5
Pndk(2, 0.5, 1)

# Second derivative of P_2(x)
# P''_2(x) = 3
Pndk(2, 0.5, 2)

# Multiple orders
Pndk(c(1, 2, 3), 0.5, 1)

# First derivative at multiple points
Pndk(2, c(-0.5, 0, 0.5), 1)

# Fractional order (uses finite differences)
Pndk(0.5, 0.5, 1)

# Derivative at endpoint
# P'_n(1) = n(n+1)/2
Pndk(3, 1, 1)  # Should be 3*4/2 = 6
```

---

# pois *Calculate the Poisson's ratio ($\nu$)*

---

## Description

Calculates Poisson's ratio ($\nu$) from two of the three other elastic moduli: bulk modulus (K), Young's modulus (E), or shear modulus (G). Assumes 3D material properties.

The relationships used are:

$$\nu = \frac{E}{2G} - 1$$

$$\nu = \frac{3K - 2G}{2(3K + G)}$$

$$\nu = \frac{3K - E}{6K}$$

## Usage

```
pois(K = NULL, E = NULL, G = NULL)
```

## Arguments

| | |
|---|---|
| K | Bulk modulus (K, Pa). |
| E | Young's modulus (E, Pa). |
| G | Shear modulus (Pa). |

## Value

Poisson's ratio ($\nu$), dimensionless.

## Examples

```
pois(E = 3e9, G = 1.1e9)
```

---

`polynomial_cylinder` *Creates a polynomial deformed cylinder*

---

## Description

Creates a polynomial deformed cylinder

## Usage

```
polynomial_cylinder(
  length_body,
  radius_body,
  n_segments = 100,
  polynomial,
  length_units = "m"
)
```

## Arguments

| | |
|---|---|
| `length_body` | Length (m). |
| `radius_body` | Maximum/uniform radius (m). |
| `n_segments` | Number of segments to discretize object shape. Defaults to 1e2 segments. |
| `polynomial` | Polynomial coefficient vector. |
| `length_units` | Units (default is meters, "m"). |

## Value

Creates the position vector for a polynomial deformed cylinder.

## References

Smith, J.N., Ressler, P.H., and Warren, J.D. 2013. A distorted wave Born approximation target strength model for Bering Sea euphausiids. ICES Journal of Marine Science, 70(1): 204-214. https://doi.org/10.1093/icesjms/fss140

## See Also

`PolynomialCylinder`

## Examples

```
# We can use the polynomial coefficients defined in Smith et al. (2013) to
# define the position vector of a sub-Arctic krill.
poly_vec <- c(0.83, 0.36, -2.10, -1.20, 0.63, 0.82, 0.64)
# Create the position vector
# This outputs a list containing "rpos" and "radius"
pos <- polynomial_cylinder(
  length_body = 15e-3, radius_body = 2e-3,
  polynomial = poly_vec
)
str(pos)
```

---

# prolate_spheroid *Creates a prolate spheroid.*

---

## Description

Creates a prolate spheroid.

## Usage

```
prolate_spheroid(
  length_body = NULL,
  radius_body = NULL,
  length_radius_ratio = NULL,
  semimajor_length = NULL,
  semiminor_length = NULL,
  n_segments = 100,
  length_units = "m"
)
```

## Arguments

| | |
|---|---|
| length_body | Semi-major axis length (m). |
| radius_body | Semi-minor axis length (m). This can also be stylized as the "maximum radius" of the scattering object. |
| length_radius_ratio | Optional ratio input when radius is not explicitly known. |
| semimajor_length | Optional alias for semi-major axis length (m). |
| semiminor_length | Optional alias for semi-minor axis length (m). |
| n_segments | Number of segments to discretize object shape. Defaults to 18 segments. |
| length_units | Units for body matrix (defaults to m). |

## Value

Creates the position vector for a prolate spheroid object of defined semi-major and -minor axes.

## See Also

ProlateSpheroid

## Examples

```
prolate_spheroid(
  length_body = 0.04, radius_body = 0.004, n_segments = 60
)
prolate_spheroid(
  semimajor_length = 0.05, semiminor_length = 0.003
)
```

---

# Qn *Legendre Function of the Second Kind, Q_$\nu(x)$*

---

## Description

Computes the Legendre function of the second kind, $Q_\nu(x)$, for real order $\nu$ (integer or fractional) and real argument $x$. Returns complex values when $|x| > 1$.

## Usage

```
Qn(n, x)
```

## Arguments

| | |
|---|---|
| n | Numeric vector. Degree (order) of the Legendre function. Can be integer or fractional (e.g., 0, 1, 2.5, 3.7). |
| x | Numeric vector. Real argument(s) at which to evaluate the function. Valid for all real values. Note that $x = \pm 1$ are singularities. |

## Details

The Legendre function of the second kind satisfies the same differential equation as $P_\nu(x)$:

$$(1 - x^2)\frac{d^2Q_\nu}{dx^2} - 2x\frac{dQ_\nu}{dx} + \nu(\nu + 1)Q_\nu = 0$$

but represents the linearly independent second solution.

**For $|x| < 1$ (real result):**

- For integer order $n$: Uses the C++ Boost library implementation

- For fractional order $\nu$: Uses the Ferrers identity

$$Q_\nu(x) = \frac{\pi}{2\sin(\pi\nu)}\left[\cos(\pi\nu)P_\nu(x) - P_\nu(-x)\right]$$

**For $x = \pm 1$:** Returns infinity (singularity).

**For $|x| > 1$ (complex result):**

- Real part: Computed via the integral representation

$$\mathrm{Re}[Q_\nu(x)] = \int_0^\infty \frac{dt}{(x + \sqrt{x^2-1}\cosh t)^{\nu+1}}$$

- Imaginary part:

$$\mathrm{Im}[Q_\nu(x)] = -\frac{\pi}{2}P_\nu(x)$$

## Value

A complex matrix of dimension `length(n)` by `length(x)`, where element `[i, j]` contains $Q_{n_i}(x_j)$. For $|x| < 1$, the imaginary part is zero.

## Note

This function calls underlying $C++$ code via `Rcpp` for computational efficiency and to support different cases for both order and argument that are not readily available in `R`.

## References

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications. Chapter 8: Legendre Functions.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/14

## See Also

Qndk for the kth derivative of the Legendre polynomial of the second kind, Pn for Legendre functions of the first kind.

## Examples

```
# Single values
Qn(1, 0.5)

# Multiple orders, single argument
Qn(c(1, 2, 3), 0.5)

# Single order, multiple arguments
Qn(1, c(-0.5, 0, 0.5))

# Multiple orders and arguments (returns a matrix)
Qn(c(1, 2, 3), c(0.25, 0.5, 0.75))
```

```
# Fractional orders
Qn(c(0.5, 1.5, 2.7), c(0.5, 0.9))

# Arguments |x| > 1 return complex values
Qn(c(1, 2, 3.5), c(2, 3.5))

# Singularity at x = 1
Qn(1, 1)
```

---

Qndk *Derivative of the Legendre function of the second kind*

---

## Description

Derivative of the Legendre function of the second kind

## Usage

```
Qndk(n, x, k = 1L)
```

## Arguments

| | |
|---|---|
| n | Numeric vector. Degree (order) of the Legendre function. |
| x | Numeric vector. Real argument(s) at which to evaluate the derivative. Avoid $x = \pm 1$. |
| k | Integer. Order of the derivative ($k \geq 0$). Default is 1. |

## Value

A complex matrix of dimension `length(n)` by `length(x)`, where element `[i, j]` contains $\frac{d^k}{dx^k} Q_{n_i}(x_j)$.

## Note

Derivatives are computed via finite differences with step size $h = 10^{-6}$. Accuracy is typically 4-6 significant digits for first derivatives, less for higher orders.

This function calls underlying $C++$ code via `Rcpp` for computational efficiency and to support different cases for both order and argument that are not readily available in `R`.

## References

Abramowitz, M. and Stegun, I. A. (1972). *Handbook of Mathematical Functions*. Dover Publications. Chapter 8: Legendre Functions.

## See Also

`Qn` for Legendre functions of the second kind.

## Examples

```
# First derivative of Q_1(x) at x = 0.5
Qndk(1, 0.5, 1)

# Compare with numerical derivative
h <- 1e-6
(Qn(1, 0.5 + h) - Qn(1, 0.5 - h)) / (2 * h)

# Second derivative
Qndk(2, 0.5, 2)

# Multiple orders
Qndk(c(1, 2, 3), 0.5, 1)

# Complex result for |x| > 1
Qndk(1, 2.0, 1)
```

---

`radians` *Convert angular measurements from degrees to radians.*

---

## Description

Convert angular measurements from degrees to radians.

## Usage

```
radians(x)
```

## Arguments

| | |
|---|---|
| `x` | A real value in degrees |

## Value

Angle in radians.

## Examples

```
orientation <- 90 # degrees
radians(orientation) # this should return a value equal to pi / 2 radians
```

reanchor_shape *Re-anchor a stored shape or scatterer component along the x axis*

## Description

Translate a shape or scatterer component so that its nose, center, or tail lies at a specified x position.

## Usage

```
reanchor_shape(
  object,
  anchor = c("nose", "center", "tail", "max_x", "min_x"),
  at = 0,
  component = NULL,
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| object | Shape or Scatterer object. |
| anchor | Anchor location used to define the translation target. `nose` is treated as the maximum stored x position and `tail` as the minimum. |
| at | Target x location (m). |
| component | Optional component name for scatterers. Defaults to the primary geometry (`"body"` for most scatterers and `"shell"` for `ESS`). |
| containment | Containment policy used when a moved swimbladder or backbone is checked against its body: `"warn"`, `"error"`, or `"ignore"`. |

## Value

The modified object, returned as the same broad object type.

## See Also

`translate_shape()`, `brake()`, `reforge()`, `extract()`

## Examples

```
shape_obj <- prolate_spheroid(length_body = 0.04, radius_body = 0.004)
centered_shape <- reanchor_shape(shape_obj, anchor = "center", at = 0)
range(extract(centered_shape, c("position_matrix", "x")))
```

`reforge` *Resize or reparameterize a scatterer object*

## Description

Generic function for rescaling or otherwise reparameterizing an existing scatterer while preserving its class semantics. The class-specific methods handle the component bookkeeping needed to keep the resulting object structurally valid after lengths, widths, heights, shell thicknesses, or related geometry descriptors are changed.

## Usage

```
reforge(object, ...)
```

## Arguments

| | |
|---|---|
| `object` | A scatterer object. |
| `...` | Additional arguments passed to specific methods. |

## Details

`reforge()` is the package's main post-construction geometry-adjustment tool. It is useful when a target should be modified in place rather than rebuilt from scratch. The available method-specific arguments depend on the scatterer class and typically include direct scale factors and/or target dimensions for length, width, or height.

## Value

A scatterer of the same broad class as `object`, rebuilt with the requested geometric changes.

## See Also

`extract()`, `plot.Scatterer()`, `brake()`

## Examples

```
obj <- fls_generate(
  shape = sphere(radius_body = 0.01, n_segments = 40),
  density_body = 1045,
  sound_speed_body = 1520
)

bigger_obj <- reforge(obj, body_target = c(length = 0.03))
extract(bigger_obj, c("shape_parameters", "length"))
```

register_model *Register a user-defined target-strength model*

## Description

Register a user-defined target-strength model

## Usage

```
register_model(
  name,
  initialize,
  solver,
  slot = NULL,
  aliases = character(),
  persist = FALSE,
  overwrite = FALSE
)
```

## Arguments

| | |
|---|---|
| name | Canonical model name passed to target_strength(). |
| initialize | Initializer function or function reference. |
| solver | Solver function or function reference. |
| slot | Result-slot name stored on the scatterer. Defaults to the usual upper-case model label derived from name. |
| aliases | Optional alternative model names. |
| persist | Logical scalar. When TRUE, the model registration is written to the user's R_user_dir() config path and reloaded in later sessions. Persistent registrations require package-qualified function references such as "mypkg::tsl_initialize" and "mypkg::TSL". |
| overwrite | Logical scalar. When TRUE, an existing user registration with the same canonical name can be replaced. Built-in models cannot be overwritten. |

## Value

Invisibly returns the normalized registry entry.

## Examples

```
if (interactive()) {
  initialize_demo <- function(object, frequency, ...) object
  solve_demo <- function(object) object
  register_model(
    "demo",
    initialize = initialize_demo,
    solver = solve_demo
```

```
  )
  unregister_model("demo")
}
```

---

`resample_shape` *Resample a stored shape or scatterer profile to a new segment count*

---

## Description

Re-discretize a shape or scatterer component along its x axis without rebuilding it from scratch.

## Usage

```
resample_shape(
  object,
  n_segments,
  component = NULL,
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| `object` | Shape or Scatterer object. |
| `n_segments` | New number of intervals used to discretize the profile. |
| `component` | Optional component name for scatterers. Defaults to the primary geometry (`"body"` for most scatterers and `"shell"` for `ESS`). |
| `containment` | Containment policy used when a moved swimbladder or backbone is checked against its body: `"warn"`, `"error"`, or `"ignore"`. |

## Value

The modified object, returned as the same broad object type.

## See Also

`smooth_shape()`, `inflate_shape()`, `reforge()`, `extract()`

## Examples

```
shape_obj <- sphere(radius_body = 0.01, n_segments = 20)
shape_fine <- resample_shape(shape_obj, n_segments = 80)
extract(shape_fine, c("shape_parameters", "n_segments"))
```

## reset_model_registry *Clear user-defined model registrations*

### Description

Clear user-defined model registrations

### Usage

```
reset_model_registry(remove_persisted = FALSE)
```

### Arguments

| | |
|---|---|
| remove_persisted | Logical scalar. When TRUE, persisted user model registrations are also deleted from the user's config file. |

### Value

Invisibly returns NULL.

### Examples

```
if (interactive()) {
  reset_model_registry()
}
```

## rho *Calculate the density contrast ($\rho$) of a scattering boundary*

### Description

Calculate the density contrast ($\rho$) of a scattering boundary

### Usage

```
rho(medium, target)
```

### Arguments

| | |
|---|---|
| medium | Dataframe object containing density ($kgm^{-3}$) and sound speed ($ms^{-1}$) values for the external medium. |
| target | Dataframe object containing density ($kgm^{-3}$) and sound speed ($ms^{-1}$) values for the target boundary. |

## Value

Density contrast, defined as $(\rho_{target} - \rho_{medium})/\rho_{target}$.

## Examples

```
seawater <- data.frame(density = 1026, sound_speed = 1480)
animal <- data.frame(density = 1050, sound_speed = 1530)
rho(seawater, animal)
```

---

Rmn *Prolate Spheroidal Radial Functions*

---

## Description

Computes the prolate spheroidal radial functions of the first ($R^{(1)}_{mn}$), second ($R^{(2)}_{mn}$), third ($R^{(3)}_{mn}$), and fourth ($R^{(4)}_{mn}$) kinds and their first derivatives with respect to $\xi$ for given order $m$, degree $n$, size parameter $c$, and radial coordinate $\xi$.

This function is an R wrapper for compiled C++ code, which in turn calls the underlying Fortran library (`prolate_swf`) for high-performance numerical computation. All heavy computation is performed in compiled code for speed and accuracy.

## Usage

```
Rmn(m, n, c, xi, kind = 1, precision = "double")
```

## Arguments

| | |
|---|---|
| `m` | Non-negative integer. The order of the spheroidal function ($m \geq 0$). |
| `n` | Non-negative integer. The degree of the spheroidal function ($n \geq m$). |
| `c` | Numeric. The scalar size parameter (also denoted $\gamma$ in some references). |
| `xi` | Numeric. The radial coordinate at which to evaluate the function. Must satisfy $\xi \geq 1$. |
| `kind` | Integer. Specifies which kind of radial function to compute: `1` (first kind), `2` (second kind), `3` (third kind/outgoing), or `4` (fourth kind/incoming). Default is `1`. |
| `precision` | Character. Either `"double"` (default) or `"quad"`. Controls the floating-point precision used in the underlying Fortran computation. |
| | `"double"` Uses standard double-precision (64-bit) arithmetic. Fastest and sufficient for most applications. |
| | `"quad"` Uses quadruple-precision (128-bit) arithmetic for higher numerical accuracy in challenging parameter regimes (e.g., large $m$, $n$, $c$, or near singularities). Computation is significantly slower and requires a package build with native quad precision support. |

## Details

The prolate spheroidal radial functions are solutions to the radial part of the scalar Helmholtz equation in prolate spheroidal coordinates. They satisfy the differential equation:

$$(\xi^2 - 1)\frac{d^2 R_{mn}}{d\xi^2} + 2\xi\frac{dR_{mn}}{d\xi} - \left(\lambda_{mn} - c^2\xi^2 + \frac{m^2}{\xi^2 - 1}\right) R_{mn} = 0$$

where $\lambda_{mn}$ is the separation constant (eigenvalue).

**Function kinds:**

- `kind = 1`: Radial function of the first kind $R^{(1)}_{mn}(c, \xi)$. Regular at the origin; analogous to spherical Bessel functions $j_n$.
- `kind = 2`: Radial function of the second kind $R^{(2)}_{mn}(c, \xi)$. Singular at the focal points ($\xi = 1$); analogous to spherical Neumann functions $y_n$.
- `kind = 3`: Radial function of the third kind (outgoing Hankel-type) $R^{(3)}_{mn}(c, \xi) = R^{(1)}_{mn}(c, \xi) + iR^{(2)}_{mn}(c, \xi)$. Used for outgoing wave solutions.
- `kind = 4`: Radial function of the fourth kind (incoming Hankel-type) $R^{(4)}_{mn}(c, \xi) = R^{(1)}_{mn}(c, \xi) - iR^{(2)}_{mn}(c, \xi)$. Used for incoming wave solutions.

**Domain restrictions:**

- The radial coordinate must satisfy $\xi \geq 1$ (prolate domain).
- The order must satisfy $m \geq 0$.
- The degree must satisfy $n \geq m$.

**Normalization:** The functions use the Morse-Feshbach normalization by default, where the radial functions reduce to spherical Bessel/Neumann functions as $c \to 0$.

**Implementation:** This function is an R wrapper for a compiled C++ interface (`Rmn_cpp`), which itself wraps the Fortran subroutine `profcn` from the `prolate_swf` library developed by Arnie Lee Van Buren and Jeffrey Boisvert. The underlying algorithm uses a combination of forward and backward recursion with the Bouwkamp eigenvalue method for high accuracy across wide parameter ranges. The C++ layer manages memory, precision selection, and data conversion between R and Fortran for robust and efficient computation.

## Value

A list containing:

`value` The function value $R^{(k)}_{mn}(c, \xi)$. Real for `kind = 1` or `2`; complex for `kind = 3` or `4`.

`derivative` The first derivative $\frac{d}{d\xi}R^{(k)}_{mn}(c, \xi)$. Real for `kind = 1` or `2`; complex for `kind = 3` or `4`.

## References

Van Buren, A. L. and Boisvert, J. E. "Prolate Spheroidal Wave Functions." GitHub repository: https://github.com/MathieuandSpheroidalWaveFunctions/prolate_swf

Morse, P. M. and Feshbach, H. (1953). *Methods of Theoretical Physics*. McGraw-Hill, New York. Chapter 21.

Flammer, C. (1957). *Spheroidal Wave Functions*. Stanford University Press.

NIST Digital Library of Mathematical Functions. Chapter 30: Spheroidal Wave Functions. https://dlmf.nist.gov/30

## See Also

Smn for prolate spheroidal angular functions.

## Examples

```
# First kind radial function
Rmn(m = 2, n = 3, c = 1, xi = 1.5, kind = 1)

# Second kind radial function
Rmn(m = 0, n = 2, c = 5, xi = 2.0, kind = 2)

# Third kind (outgoing) radial function
Rmn(m = 1, n = 1, c = 2, xi = 1.2, kind = 3)

# Fourth kind (incoming) radial function
Rmn(m = 1, n = 1, c = 2, xi = 1.2, kind = 4)

# Double precision (default)
Rmn(m = 2, n = 3, c = 1, xi = 1.5)
```

---

# sardine *Sample sardine shape with fully inflated swimbladder*

---

## Description

A pre-generated SBF scatterer containing all information required for target strength modeling. The packaged geometry follows the sardine entry distributed through the NOAA Fisheries KRM reference collection and archived in the echoSMs resources. The object metadata identifies the target as *Sardinops sagax caerulea* following Conti and Demer (2003).

## Usage

```
data(sardine)
```

## Format

A named list with the following components:

**metadata** Relevant and identifying metadata (list).

**model_parameters** Specified model parameters (list).

**model** Model outputs and results (list).

**body** A list with:

- `rpos`: Position matrix (x, yw, zU, zL; m).
- `sound_speed`: Flesh sound speed ($c_{body}$, m/s).
- `density`: Flesh density ($\rho_{body}$, kg/m$^3$).
- `theta`: Orientation relative to transducer ($\theta_{body}$, radians).

**bladder** A list with:

- `rpos`: Position matrix (x, yw, zU, zL; m).
- `sound_speed`: Bladder sound speed ($c_{bladder}$, m/s).
- `density`: Bladder density ($\rho_{bladder}$, kg/m$^3$).
- `theta`: Orientation relative to transducer ($\theta_{bladder}$, radians).

**shape_parameters** A named list with:

- `body`: A list describing the body:
  - `length`: Body length (m).
  - `ncyl`: Number of discrete cylinders.
  - `theta_units`: Units for orientation angle.
  - `length_units`: Units for length.
- `bladder`: A list describing the swimbladder:
  - `length`: Bladder length (m).
  - `ncyl`: Number of discrete cylinders.
  - `theta_units`: Units for orientation angle.
  - `length_units`: Units for length.

## Value

A pre-generated `SBF` scatterer object.

## Source

NOAA Fisheries KRM model reference collection (https://www.fisheries.noaa.gov/data-tools/krm-model) and the archived `echoSMs` resource set (https://github.com/ices-tools-dev/echoSMs).

## References

Conti, S.G., and Demer, D.A. (2003). Wide-bandwidth acoustical characterization of anchovy and sardine from reverberation measurements in an echoic tank. *ICES Journal of Marine Science*, 60, 617-624. doi:10.1016/S10543139(03)000560

## Examples

```
data("sardine", package = "acousticTS")
sardine
```

## SBF *Swimbladdered fish (SBF) object/class.*

### Description

A S4 class that provides slots to contain relevant animal metadata for parameterizing models for swimbladdered fish (SBF) that are partitioned into two sets of discretized cylinders: the body and the swimbladder. Both shapes comprise independent position matrices, material properties, orientations, and other relevant shape-related data and metadata. See Scatterer for a more detailed description on how this S4 object is organized.

### Value

An object of the SBF S4 class.

### See Also

Scatterer

### Examples

```
methods::getClass("SBF")
```

## sbf_generate *Generate a SBF-class object.*

### Description

Generate a SBF-class object.

### Usage

```
sbf_generate(
  x_body = NULL,
  w_body = NULL,
  zU_body = NULL,
  zL_body = NULL,
  x_bladder = NULL,
  w_bladder = NULL,
  zU_bladder = NULL,
  zL_bladder = NULL,
  sound_speed_body = NULL,
  sound_speed_bladder = NULL,
  g_body = NULL,
  h_body = NULL,
```

```
  g_bladder = NULL,
  h_bladder = NULL,
  density_body = NULL,
  density_bladder = NULL,
  theta_body = pi/2,
  theta_bladder = pi/2,
  theta_units = "radians",
  length_units = "m",
  ID = NULL,
  body_shape = NULL,
  bladder_shape = NULL
)
```

## Arguments

| | |
|---|---|
| x_body | Vector containing along-body axis (m). |
| w_body | Vector containing across-body axis (m). |
| zU_body | Vector containing dorsal-body axis (m). |
| zL_body | Vector containing ventral-body axis (m). |
| x_bladder | Vector containing along-bladder axis (m). |
| w_bladder | Vector containing across-bladder axis (m). |
| zU_bladder | Vector containing dorsal-bladder axis (m). |
| zL_bladder | Vector containing ventral-bladder axis (m). |
| sound_speed_body | Flesh sound speed ($c_{body}$, m $s^{-1}$). |
| sound_speed_bladder | Bladder sound speed (c, m $s^-1$). |
| g_body | Body density contrast. |
| h_body | Body sound speed contrast. |
| g_bladder | Bladder density contrast. |
| h_bladder | Bladder sound speed contrast. |
| density_body | Flesh density ($\rho_{body}$, kg $m^3$). |
| density_bladder | Bladder density ($\rho$, kg $m^3$). |
| theta_body | Angle of body relative to wavefront ($\theta_body$, radians). |
| theta_bladder | Angle of bladder relative to wavefront ($\theta_bladder$, radians). |
| theta_units | Compatibility argument. Scatterer constructors now assume radians and ignore non-SI alternatives. |
| length_units | Compatibility argument. Scatterer constructors now assume meters and ignore non-SI alternatives. |
| ID | Optional metadata identifier. |
| body_shape | Optional pre-built Shape for the body; overrides x_body/w_body/zU_body/zL_body if supplied. |
| bladder_shape | Optional pre-built Shape for the bladder; overrides x_bladder/w_bladder/zU_bladder/zL_bladder if supplied. |

## Details

The recommended interface is to supply pre-built Shape objects through body_shape and bladder_shape, then specify material properties through either contrasts (g_*, h_*) or absolute density/sound-speed values. Legacy coordinate-vector inputs are retained for backward compatibility and are converted internally to Shape objects before the scatterer is built. Character-based shape dispatch is deprecated; build the component geometry first with a Shape constructor and then pass the resulting Shape object.

Body/bladder width vectors (w_body, w_bladder) default to zeros when missing. Resulting rpos matrices always carry the row names x_body, w_body, zU_body, zL_body for the body and x_bladder, w_bladder, zU_bladder, zL_bladder for the bladder.

## Value

Generates a SBF-class object.

## See Also

SBF

## Examples

```
# Manual body/bladder coordinates
sbf_generate(
  x_body = seq(0, 0.1, length.out = 5),
  w_body = rep(0, 5),
  zU_body = seq(0.001, 0.002, length.out = 5),
  zL_body = -seq(0.001, 0.002, length.out = 5),
  x_bladder = seq(0.02, 0.09, length.out = 4),
  w_bladder = rep(0, 4),
  zU_bladder = rep(0.0015, 4),
  zL_bladder = rep(-0.0015, 4),
  sound_speed_body = 1500,
  sound_speed_bladder = 340,
  density_body = 1040,
  density_bladder = 1.2
)

# Using pre-built shapes
body_shape <- arbitrary(
  x_body = c(0, 0.1), zU_body = c(0.001, 0.002),
  zL_body = c(-0.001, -0.002)
)
bladder_shape <- arbitrary(
  x_bladder = c(0.02, 0.09), w_bladder = c(0, 0),
  zU_bladder = c(0.0015, 0.0015),
  zL_bladder = c(-0.0015, -0.0015)
)
sbf_generate(
  body_shape = body_shape, bladder_shape = bladder_shape,
  sound_speed_body = 1500, sound_speed_bladder = 340,
  density_body = 1040, density_bladder = 1.2
```

)

---

`Scatterer` *Scatterer-class object for target strength estimation*

---

## Description

The `acousticTS` package uses a variety of defined S4-class objects comprising different types of scatterers, such as fish with gas-filled swimbladders (SBF) and fluid-like crustaceans (FLS).

## Value

An object from the `Scatterer` S4 class hierarchy.

## Slots

`metadata` List containing relevant metadata

`model_parameters` Model parameters necessary for predicting TS

## Data Organization

`metadata`**:** A `list` comprising any identifying information associated with the scatterer. The default metadata entry includes `ID` that uses a default `character` value of `"UID"` (i.e. `ID = "UID"`). This can otherwise be formatted in any manner for book keeping purposes.

`body/bladder`**:** A `list` that includes information relevant to the scatterer's position vector, material properties, tilt/orientation, etc. For some scatterers, this may only include `body`, but other targets may have an additional parameter such as `bladder`. Generally, each entry includes:

- rpos: the relevant position vector ($r_0$) that includes axes such as `x`, `y`, `z`, etc., that depend on the type of scatterer being used.
- `radius`: in some cases this includes the radius measurements for each cylinders depending on the type of scatterer object.
- `theta`: the orientation of the scatterer relative to the transmitting transducer or sound source ($\theta_{animal}$) that can be represented either by degrees or radians, although all functions require radians.
- `g`, `h`: material properties that represent the density and sound speed contrasts (g and h, respectively) relative to the ambient/surrounding fluid. Some targets may instead have standard sound speed ($c_{animal}$, m $s^{-1}$) and density ($\rho_{animal}$, kg $m^3$).

`shape_parameters`**:** A `list` that includes metadata pertaining to the shape of the scatterer and any other features of interest (e.g. gas-filled swimbladder). Generally, each entry includes: overall body length, the number of discrete cylinders that make up the shape (if applicable), and the units related to both $\theta_{animal}$ (e.g. rad, °) and length (e.g. mm, m).

`model_parameters`**:** A `list` that contains relevant model parameterization once an object has been initialized for modeling $\sigma_{bs}$. This is typically broken up into three categories:

- `parameters`: A `list` that includes information such as frequency (Hz), acoustic wavenumber (i.e. k), etc.
- `medium`: A `data.frame` including information such as the material properties of the ambient medium.
- `scatterer`: A `list` containing summarized information used to parameterize certain scattering models.

`model`: A `list` that collects model results from one or more models in the linear domain (i.e. $\sigma_{bs}$).

### Supported Scatterers

- `Elastic-based scatterers` (ELA)
- `Composite multi-component scatterers` (CSC)
- `Backboned fish` (BBF)
- `Calibration spheres` (CAL)
- `Elastic-shelled scatterers` (ESS)
- `Fluid-like scatterers` (FLS)
- `Gas-filled scatterers` (GAS)
- `Swimbladdered fish` (SBF)

### Examples

```
methods::getClass("Scatterer")
```

---

## shear *Calculate the shear modulus (G)*

---

### Description

Calculates the shear modulus (G) from two of the three other elastic moduli: bulk modulus (K), Young's modulus (E), or Poisson's ratio ($\nu$). Assumes 3D material properties.

The relationships used are:

$$G = \frac{3KE}{9K - E}$$

$$G = \frac{3K(1 - 2\nu)}{2(1 + \nu)}$$

$$G = \frac{E}{2(1 + \nu)}$$

### Usage

```
shear(K = NULL, E = NULL, nu = NULL)
```

### Arguments

| | |
|---|---|
| K | Bulk modulus (Pa). |
| E | Young's modulus (Pa). |
| nu | Poisson's ratio (Dimensionless). |

### Value

Shear modulus (G, Pa).

### Examples

```
shear(K = 2.5e9, E = 3e9)
```

---

## simulate_ts *Simulate target strength (TS) with flexible parameterization and batching*

---

### Description

Simulate target strength (TS) with flexible parameterization and batching

### Usage

```
simulate_ts(
  object,
  frequency,
  model,
  n_realizations = NULL,
  parameters = list(),
  batch_by = NULL,
  permute = TRUE,
  parallel = TRUE,
  n_cores = .default_simulation_cores(),
  verbose = TRUE
)
```

### Arguments

| | |
|---|---|
| object | Scatterer-class object. |
| frequency | Frequency (Hz). |
| model | Model name. If multiple models are specified, the output will be a list of data frames, one for each model. |

`n_realizations` Optional number of repeated evaluations of each deterministic grid cell (default `1`). It is a pure repeat/redraw multiplier: every deterministic multi-value parameter is always treated as a grid axis, and `n_realizations` controls how many times each resulting cell is evaluated, redrawing any generating functions each time. The total number of realizations is therefore the size of the Cartesian grid multiplied by `n_realizations` (for example, one length and two radii yield two cells; with `n_realizations = 5` that is ten realizations). Use it to draw repeated samples from generating functions (distributions).

`parameters` List containing the values, distributions, or generating functions of parameter values that inform the TS model. Defaults to an empty list, which runs a single realization of the unmodified object. An explicit `frequency` entry supplies the frequency for each simulation row, overriding the shared `frequency` argument.

`batch_by` Optional. Specifies which parameters in `parameters` to batch over. Simulations will be run for all combinations of these parameter values. Default is `NULL`.

`permute` Logical; controls how varied parameters combine. When `TRUE` (default) every deterministic multi-value parameter is crossed into the full Cartesian grid. When `FALSE` the varied parameters are instead paired (zipped) element-wise, so they advance together rather than combinatorially; in that case every varied parameter must supply the same number of values (length-one parameters are held constant).

`parallel` Logical; whether to parallelize the simulations. Default is `TRUE`.

`n_cores` Optional. Number of CPU cores to use for parallelization. Default is the smaller of 2 cores and `parallel::detectCores() - 1`.

`verbose` Logical; whether to print progress and status messages to the console. Default is `TRUE`.

## Details

`simulate_ts()` is a workflow wrapper around repeated `target_strength()` calls. It supports three broad parameter modes inside `parameters`:

- scalars that are recycled across every realization,
- explicit vectors that are either aligned with the full simulation grid or with one or more batched dimensions, and
- generating functions that are re-evaluated for each realization, and
- structured values such as named target-dimension vectors used by `reforge()` (for example `body_target = c(length = 0.03)`).

If `batch_by = "length"` and `parameters[["length"]]` is a vector of candidate values, then simulations are run for each length value, repeated `n_realizations` times. When multiple parameters are supplied through `batch_by`, the function builds the full Cartesian grid of those parameter values and runs the requested number of realizations inside each batch cell.

Batching is automatic and follows one consistent rule: every deterministic multi-value parameter is a grid axis, whether or not `n_realizations` or `batch_by` are supplied. A bare vector therefore always means "sweep these values" rather than "align one value per realization". `batch_by` remains available to document intent, but naming an axis is no longer required for it to be swept.

To vary parameters *together* (paired rather than crossed) set `permute = FALSE`. The varied parameters are then zipped element-wise and must share the same number of values. For example, `parameters = list(theta_body = c(1, 2), density_body = c(1040, 1050))` yields four runs by default but two paired runs - `(1, 1040)` and `(2, 1050)` - under `permute = FALSE`. Alternatively, values that belong to one `reforge()` target can be paired within a single structured parameter, for example `parameters = list(body_target = list(c(length = 0.02, radius = 0.002),c(length = 0.03, radius = 0.003)))`.

Structured batch values should be wrapped in a list so that each candidate is preserved as one unit. For example, use `parameters = list(body_target = list(c(length = 0.02), c(length = 0.03)))` when batching across multiple explicit `reforge()` targets.

Convenience dimension aliases are also supported for compatible `reforge()` methods. For example, `length_body = 0.03` is treated the same as `body_target = c(length = 0.03)` for fluid-like scatterers, while retaining the original `length_body` column in the returned simulation output.

Parameter names are interpreted in the same way they would be if supplied directly to `target_strength()` or to the relevant object constructor / `reforge()` path. This means `simulate_ts()` can be used for:

- orientation perturbations,
- material-property perturbations,
- morphology studies that trigger shape rebuilding or reforging, and
- side-by-side comparisons across one or more model families.

## Value

A named list of data frames, one for each model. Each data frame contains the realized parameter values together with the modeled acoustic output for each simulated run.

## Parallelization

This function uses `pbapply::pblapply()` for parallelized simulation with progress bars. The current implementation uses PSOCK clusters for worker execution across platforms, including Windows, macOS, and Linux. That means worker processes need access to the package namespace and any required exported objects, and it also means startup overhead is more noticeable for very small simulation jobs than for larger batched runs.

Each worker evaluates complete parameter realizations, passing the frequency vector to the requested models without changing its order or duplicates. To distribute individual frequencies across workers, supply them through `parameters = list(frequency = ...)`. Use `permute = FALSE` to pair those frequencies with other varied parameters. Each worker has its own numerical caches and memory allocation. Increase `n_cores` only when sufficient memory is available for those processes.

## Performance Issues

Including too many parameters from `parameters` within `batch_by` may cause significant performance issues or cause `R` to crash. If intensive simulations are required, consider breaking them into more manageable chunks

## Examples

```
shape_obj <- cylinder(
  length_body = 0.05,
  radius_body = 0.003,
  n_segments = 40
)

obj <- fls_generate(
  shape = shape_obj,
  density_body = 1045,
  sound_speed_body = 1520
)

res <- simulate_ts(
  object = obj,
  frequency = seq(38e3, 50e3, by = 6e3),
  model = "dwba",
  n_realizations = 2,
  parameters = list(
    theta_body = function() runif(1, min = 0.5 * pi, max = pi),
    density_body = 1045
  ),
  parallel = FALSE,
  verbose = FALSE
)

head(res)
```

---

# Smn *Prolate Spheroidal Angular Function of the First Kind,* $S\hat{}1_mn(c, \eta)$

---

## Description

Computes the prolate spheroidal angular function of the first kind, $S^{(1)}_{mn}(c, \eta)$, and its first derivative with respect to $\eta$ for given order $m$, degree $n$, size parameter $c$, and angular coordinate $\eta$.

This function is an R wrapper for compiled C++ code, which in turn calls the underlying Fortran library (`prolate_swf`) for high-performance numerical computation. All heavy computation is performed in compiled code for speed and accuracy.

## Usage

```
Smn(m, n, c, eta, normalize = FALSE, precision = "double")
```

## Arguments

| | |
|---|---|
| `m` | Non-negative integer. The order of the spheroidal function ($m \geq 0$). |
| `n` | Non-negative integer. The degree of the spheroidal function ($n \geq m$). |

`c` Numeric. The scalar size parameter.

`eta` Numeric vector. The angular coordinate(s) at which to evaluate the function. Must satisfy $|\eta| \leq 1$.

`normalize` Logical. If `TRUE`, the angular functions are normalized to have unity norm. If `FALSE` (default), the Meixner-Schäfke normalization is used.

`precision` Character. Either `"double"` (default) or `"quad"`. Controls the floating-point precision used in the underlying Fortran computation.

`"double"` Uses standard double-precision (64-bit) arithmetic. Fastest and sufficient for most applications.

`"quad"` Uses quadruple-precision (128-bit) arithmetic for higher numerical accuracy in challenging parameter regimes (e.g., large $m$, $n$, or near singularities). Computation is significantly slower and requires a package build with native quad precision support.

## Details

The prolate spheroidal angular functions are solutions to the angular part of the scalar Helmholtz equation in prolate spheroidal coordinates. They satisfy the differential equation:

$$(1-\eta^2)\frac{d^2S_{mn}}{d\eta^2} - 2\eta\frac{dS_{mn}}{d\eta} + \left(\lambda_{mn} - c^2\eta^2 + \frac{m^2}{1-\eta^2}\right)S_{mn} = 0$$

where $\lambda_{mn}$ is the separation constant (eigenvalue).

**Domain restrictions:**

- The angular coordinate must satisfy $|\eta| \leq 1$.
- The order must satisfy $m \geq 0$.
- The degree must satisfy $n \geq m$.

**Normalization:** When `normalize = FALSE` (default), the functions use the Meixner-Schäfke normalization, which matches the normalization of the corresponding associated Legendre functions. When `normalize = TRUE`, the functions are scaled to have unity norm.

**Implementation:** This function is an R wrapper for a compiled C++ interface (`Smn_cpp`), which itself wraps the Fortran subroutine `profcn` from the `prolate_swf` library developed by Arnie Lee Van Buren and Jeffrey Boisvert. The underlying algorithm uses a combination of forward and backward recursion with the Bouwkamp eigenvalue method for high accuracy across wide parameter ranges. The C++ layer manages memory, precision selection, and data conversion between R and Fortran for robust and efficient computation.

## Value

A list containing:

`value` Numeric vector of function values $S_{mn}^{(1)}(c,\eta)$ at each input `eta`.

`derivative` Numeric vector of first derivatives $\frac{d}{d\eta}S_{mn}^{(1)}(c,\eta)$ at each input `eta`.

Equally sized vectors `m` and `n` with more than one entry are evaluated pairwise. For these paired modes, a scalar `eta` gives vectors in `value` and `derivative`. Multiple `eta` values give matrices with one row per mode pair and one column per angular coordinate.

## References

Van Buren, A. L. and Boisvert, J. E. "Prolate Spheroidal Wave Functions." GitHub repository: https://github.com/MathieuandSpheroidalWaveFunctions/prolate_swf

Meixner, J. and Schäfke, F. W. (1954). *Mathieusche Funktionen und Sphäroidfunktionen*. Springer-Verlag, Berlin.

Flammer, C. (1957). *Spheroidal Wave Functions*. Stanford University Press.

NIST Digital Library of Mathematical Functions. Chapter 30: Spheroidal Wave Functions. https://dlmf.nist.gov/30

## See Also

Rmn for prolate spheroidal radial functions.

## Examples

```
# Single evaluation
Smn(m = 2, n = 3, c = 1, eta = 0.5)

# Multiple eta values
Smn(m = 0, n = 2, c = 5, eta = c(-0.5, 0, 0.5))

# With unity normalization
Smn(m = 1, n = 1, c = 2, eta = 0.3, normalize = TRUE)

# Double precision (default)
Smn(m = 2, n = 3, c = 1, eta = 0.5, precision = "double")
```

---

# smooth_shape *Smooth a stored shape or scatterer profile*

---

## Description

Smooth the stored geometry using a centered moving-average filter applied to profile coordinates. This is useful for cleaning digitized outlines before canonicalization or model runs. For canonical Shape objects, the result is returned as an Arbitrary shape because the edited profile is no longer guaranteed to preserve the canonical class geometry.

## Usage

```
smooth_shape(
  object,
  span = 5,
  component = NULL,
  preserve_ends = TRUE,
  containment = c("warn", "error", "ignore")
)
```

### Arguments

| | |
|---|---|
| object | Shape or Scatterer object. |
| span | Centered moving-average span. Even values are rounded up to the next odd integer. |
| component | Optional component name for scatterers. Defaults to the primary geometry ("body" for most scatterers and "shell" for ESS). |
| preserve_ends | Logical; whether to keep the first and last profile points fixed. |
| containment | Containment policy used when a moved swimbladder or backbone is checked against its body: "warn", "error", or "ignore". |

### Value

The modified object. Shape inputs that are smoothed are returned as Arbitrary shapes.

### See Also

inflate_shape(), resample_shape(), brake(), reforge()

### Examples

```
shape_obj <- arbitrary(
  x_body = c(0, 0.01, 0.02, 0.03, 0.04),
  zU_body = c(0, 0.003, 0.006, 0.0035, 0),
  zL_body = c(0, -0.0025, -0.0055, -0.003, 0)
)
smoothed_shape <- smooth_shape(shape_obj, span = 3)
extract(smoothed_shape, c("shape_parameters", "n_segments"))
```

---

## sphere *Creates a sphere.*

---

### Description

Creates a sphere.

### Usage

```
sphere(radius_body, n_segments = 100, diameter_units = "m")
```

### Arguments

| | |
|---|---|
| radius_body | Object radius (m). |
| n_segments | Number of segments to discretize object shape. Defaults to 1e2 segments. |
| diameter_units | Default is "m" for meters |

## Value

Creates position vector for a spherical object of a defined radius.

## See Also

Sphere

## Examples

```
sphere(radius_body = 0.01, n_segments = 50)
```

---

`target_strength` *Wrapper function to model acoustic target strength*

---

## Description

Wrapper function to model acoustic target strength

## Usage

```
target_strength(
  object,
  frequency,
  model,
  verbose = FALSE,
  model_args = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer-class object. |
| `frequency` | Frequency (Hz). |
| `model` | Model name. Available models currently include `"dwba"` (DWBA), `"bbfm"` (BBFM), `"pcdwba"` (PCDWBA), `"sdwba"` (SDWBA), `"fcms"` (FCMS), `"bcms"` (BCMS), `"ecms"` (ECMS), `"hpa"` (HPA), `"krm"` (KRM), `"psms"` (PSMS), `"sphms"` (SPHMS), `"essms"` (ESSMS), `"vesms"` (VESMS), `"trcm"` (TRCM), and `"calibration"` / `"soems"` (SOEMS). |
| `verbose` | Prints current procedural step occurring from model initialization to calculating TS. Defaults to FALSE. |
| `model_args` | Optional named list of per-model argument bundles. Each list name should match one of the requested model names case-insensitively, and each value should be either a named list or a named atomic vector of arguments to apply only to that model. When the same argument is supplied both through `...` and through `model_args[[model_name]]`, the model-specific entry takes precedence. |
| `...` | Additional optional model inputs/parameters. |

## Details

This is the main high-level entry point for running target-strength models in acousticTS. The supplied scatterer object is checked, the requested model or models are initialized, and the resulting outputs are stored back on the same object.

The available model families span exact modal-series solutions and approximation-based solutions. Readers should consult the model-specific help topics for the physical assumptions, valid object types, boundary-condition options, and model-specific arguments used by each implementation:

- DWBA for the distorted-wave Born approximation applied to weakly scattering fluid-like bodies.
- BBFM for a composite flesh-plus-backbone model that combines a DWBA flesh term with an elastic-cylinder backbone term.
- PCDWBA for the phase-compensated distorted-wave Born approximation applied to elongated fluid-like bodies.
- SDWBA for the stochastic distorted-wave Born approximation.
- FCMS for the finite cylinder modal series solution.
- BCMS for the bent-cylinder modal series solution.
- ECMS for the elastic-cylinder modal series solution.
- HPA for the high-pass approximation family.
- KRM for the Kirchhoff-Ray Mode model.
- PSMS for the prolate spheroidal modal series solution.
- SPHMS for the spherical modal series solution.
- ESSMS for the elastic-shelled spherical modal series solution.
- VESMS for the viscous-elastic spherical scattering model applied to gas-filled elastic shells with an external viscous layer.
- TMM for the single-target transition-matrix family.
- TRCM for the two-ray cylindrical model.
- SOEMS for the solid elastic calibration-sphere model, accessed through "calibration" or "soems".

Model-specific inputs are usually passed through .... For example, some models require a boundary argument, HPA uses a method argument, and several models expose additional numerical controls. When several models are requested together, shared arguments may be supplied through ... and per-model overrides may be supplied through model_args. This is useful when different models should share the same seawater properties but only one of them needs an extra stochastic or numerical control:

```
target_strength(
  object,
  frequency,
  model = c("dwba", "sdwba"),
  density_sw = 1026,
  sound_speed_sw = 1478,
  model_args = list(
```

```
    sdwba = list(phase_sd_init = 0.77)
  )
)
```

Model names are normalized internally, so case-insensitive inputs such as "DWBA" and "dwba" resolve to the same family.

## Value

The input scatterer object with requested model parameters, model outputs, and target strength results stored in its model slots.

## See Also

DWBA, BBFM, PCDWBA, SDWBA, FCMS, BCMS, ECMS, HPA, KRM, PSMS, SPHMS, ESSMS, VESMS, TMM, TRCM, SOEMS

## Examples

```
calibration_sphere <- cal_generate(
  material = "WC",
  diameter = 38.1e-3,
  n_segments = 40
)
target_strength(
  calibration_sphere,
  frequency = c(38e3, 120e3),
  model = "calibration"
)
```

---

# tmm_average_orientation

*Orientation-average scattering from a stored TMM object*

---

## Description

Reuses the stored TMM blocks to average the differential scattering cross section over a user-supplied set of incident orientations. By default the receive direction is taken to be the exact monostatic direction for each supplied orientation.

## Usage

```
tmm_average_orientation(
  object,
  theta_body = NULL,
  weights = NULL,
  phi_body = pi,
  theta_scatter = NULL,
```

```
  phi_scatter = NULL,
  distribution = NULL
)
```

## Arguments

| | |
|---|---|
| object | Scatterer-object previously evaluated with target_strength(..., model = "TMM", store_t_matrix = TRUE). |
| theta_body | Numeric vector of incident polar angles (radians). |
| weights | Optional numeric vector of averaging weights. When omitted, equal weights are used. |
| phi_body | Incident azimuth angle(s) (radians). Either scalar or the same length as theta_body. Defaults to pi. |
| theta_scatter | Receive polar angle(s) (radians). Either scalar or the same length as theta_body. Defaults to the monostatic direction. |
| phi_scatter | Receive azimuth angle(s) (radians). Either scalar or the same length as theta_body. Defaults to the monostatic direction. |
| distribution | Optional orientation distribution created by tmm_orientation_distribution. When supplied, it overrides the direct theta_body, weights, and phi_body inputs. |

## Value

A data frame containing the frequency, the orientation-averaged differential backscattering cross section and the corresponding orientation-averaged target strength.

## See Also

tmm_scattering

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
distribution <- tmm_orientation_distribution(n_theta = 5)
tmm_average_orientation(stored, distribution = distribution)
```

`tmm_bistatic_summary` *Summarize bistatic products from a stored TMM object*

## Description

Reuses the stored T-matrix blocks at one stored frequency to compute a higher-level bistatic summary, including forward- and cross-plane slices, peak-scattering direction, backscatter-lobe width, and integrated scattering over coarse angular sectors. In the current package build, this helper is available for the spherical and spheroidal stored branches. Stored cylinders intentionally stop at exact monostatic reuse until a validated retained cylinder angular operator is added.

## Usage

```
tmm_bistatic_summary(
  object,
  frequency = NULL,
  theta_body = NULL,
  phi_body = NULL,
  n_theta = 91,
  n_phi = 181,
  n_psi = 181,
  sectors = NULL,
  drop_dB = 3,
  include_grid = FALSE
)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer-object previously evaluated with `target_strength(..., model = "TMM", store_t_matrix = TRUE)`. |
| `frequency` | Stored frequency (Hz) to summarize. Required when the object contains more than one stored frequency. |
| `theta_body` | Incident polar angle (radians). Defaults to the stored TMM incident angle. |
| `phi_body` | Incident azimuth angle (radians). Defaults to the stored TMM incident angle. |
| `n_theta` | Number of receive polar-angle samples used by the summary grid. |
| `n_phi` | Number of receive azimuth samples used by the summary grid. |
| `n_psi` | Number of forward-centered angular samples used for the local great-circle slices. |
| `sectors` | Optional data frame with columns `sector`, `psi_min`, and `psi_max` (radians). When omitted, three coarse forward/oblique/backward sectors are used. |
| `drop_dB` | Positive dB drop used to define the backscatter-lobe width on the forward-centered slice. |
| `include_grid` | Logical; include the underlying scattering grid in the returned list. |

## Value

A list containing scalar summary metrics, the named slice data frames, sector integrals, and optionally the underlying scattering grid.

## See Also

tmm_scattering_grid, tmm_products

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
tmm_bistatic_summary(
  stored,
  n_theta = 5,
  n_phi = 9,
  n_psi = 9
)
```

---

# tmm_diagnostics *Diagnostics for stored single-target TMM solutions*

---

## Description

Reuses the stored transition-matrix blocks to compute a compact set of numerical and physics-based diagnostics for one or more stored frequencies. The summary combines:

- monostatic reconstruction residuals,
- reciprocity residuals under incident/receive-angle exchange,
- an optical-theorem residual based on forward scattering and the integrated differential cross section,
- block-level conditioning indicators from the stored transition-matrix blocks,
- and, for prolate/oblate targets, an equal-volume sphere-to-spheroid continuation path that checks whether the monostatic response deforms smoothly away from the exact sphere limit.

These checks are meant to help distinguish "the post-processing is self-consistent" from "the retained solve was also numerically comfortable," which is especially helpful for the newer non-spherical TMM branches. For stored cylinders, the current diagnostics are intentionally limited to exact monostatic reconstruction because a validated retained cylinder angular operator is not yet available.

## Usage

```
tmm_diagnostics(
  object,
  frequency = NULL,
  theta_body = NULL,
  phi_body = NULL,
  reciprocity_pairs = NULL,
  n_theta = 61,
  n_phi = 121,
  continuation_steps = 6L
)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer-object previously evaluated with `target_strength(..., model = "TMM", store_t_matrix = TRUE)`. |
| `frequency` | Optional stored frequency or vector of stored frequencies (Hz). Defaults to all stored frequencies. |
| `theta_body` | Incident polar angle (radians) used for the monostatic and optical-theorem checks. Defaults to the stored TMM incident angle. |
| `phi_body` | Incident azimuth angle (radians) used for the monostatic and optical-theorem checks. Defaults to the stored TMM incident angle. |
| `reciprocity_pairs` | Optional data frame giving explicit reciprocity test angles. Must contain `theta_body`, `phi_body`, `theta_scatter`, and `phi_scatter` columns in radians. |
| `n_theta` | Number of polar-angle grid points used by the optical-theorem integration check. |
| `n_phi` | Number of azimuth-angle grid points used by the optical-theorem integration check. |
| `continuation_steps` | Number of equal-volume aspect-ratio steps used for the sphere-to-spheroid continuation check on prolate and oblate targets. Set to `0` or `1` to skip the continuation path. Ignored for non-spheroidal targets. |

## Value

A list with components:

`summary` Per-frequency diagnostic summary.

`block_metrics` Per-frequency block-level conditioning and transpose-residual summaries.

`continuation` Equal-volume sphere-to-spheroid continuation path for spheroidal targets, or `NULL` for other shapes.

## See Also

tmm_scattering, tmm_scattering_grid, tmm_products

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
tmm_diagnostics(stored, n_theta = 5, n_phi = 9)
```

---

# tmm_orientation_distribution

*Build an orientation distribution for stored TMM post-processing*

---

## Description

Creates a validated set of incident angles and normalized weights that can be reused by tmm_average_orientation or tmm_products. The distributions defined here are distributions in theta_body itself rather than isotropic solid-angle distributions.

## Usage

```
tmm_orientation_distribution(
  distribution = c("uniform", "normal", "truncated_normal", "quadrature", "pdf"),
  theta_body = NULL,
  weights = NULL,
  pdf = NULL,
  phi_body = pi,
  mean_theta = pi/2,
  sd_theta = pi/12,
  lower = 0,
  upper = pi,
  n_theta = 91
)
```

### Arguments

| | |
|---|---|
| distribution | Orientation-distribution type. One of "uniform", "normal", "truncated_normal", "quadrature", or "pdf". |
| theta_body | Optional numeric vector of incident polar angles (radians). Required for the "quadrature" and "pdf" pathways. |
| weights | Optional numeric quadrature weights paired with theta_body for distribution = "quadrature". |
| pdf | Optional user-supplied density over theta_body for distribution = "pdf". This can be either a numeric vector the same length as theta_body or a function evaluated at theta_body. |
| phi_body | Incident azimuth angle(s) (radians). Either scalar or the same length as the resolved theta_body grid. |
| mean_theta | Mean angle (radians) for the normal-family distributions. |
| sd_theta | Standard deviation (radians) for the normal-family distributions. |
| lower | Lower bound (radians) for the uniform and truncated-normal distributions. |
| upper | Upper bound (radians) for the uniform and truncated-normal distributions. |
| n_theta | Number of grid points for the analytic distributions. |

### Value

A data frame with normalized orientation weights and class "TMMOrientationDistribution".

### See Also

tmm_average_orientation, tmm_products

### Examples

```
tmm_orientation_distribution(
  distribution = "uniform",
  lower = pi / 3,
  upper = 2 * pi / 3,
  n_theta = 7
)
```

---

## tmm_products *Collect multiple post-processed products from one stored TMM solve*

---

### Description

Provides a higher-level convenience interface for the stored-block TMM workflow. One call can return the monostatic scattering spectrum, an orientation average, and a higher-level bistatic summary without rebuilding the underlying T-matrix solve. For stored cylinders, the currently supported products are the exact monostatic scattering spectrum and the corresponding orientation-averaged monostatic outputs; cylinder bistatic summaries remain unavailable until a validated retained cylinder angular operator is added.

## Usage

```
tmm_products(
  object,
  frequency = NULL,
  theta_body = NULL,
  phi_body = NULL,
  orientation = NULL,
  bistatic_summary = FALSE,
  include_grid = FALSE,
  n_theta = 91,
  n_phi = 181,
  n_psi = 181,
  sectors = NULL,
  drop_dB = 3
)
```

## Arguments

| | |
|---|---|
| object | Scatterer-object previously evaluated with `target_strength(..., model = "TMM", store_t_matrix = TRUE)`. |
| frequency | Stored frequency (Hz) used when `bistatic_summary = TRUE`. |
| theta_body | Incident polar angle (radians) for the monostatic and bistatic products. Defaults to the stored TMM incident angle. |
| phi_body | Incident azimuth angle (radians) for the monostatic and bistatic products. Defaults to the stored TMM incident angle. |
| orientation | Optional orientation distribution created by `tmm_orientation_distribution`. |
| bistatic_summary | Logical; include the output of `tmm_bistatic_summary`. |
| include_grid | Logical; keep the 2D scattering grid inside the bistatic summary output. |
| n_theta | Number of receive polar-angle samples for the bistatic summary. |
| n_phi | Number of receive azimuth samples for the bistatic summary. |
| n_psi | Number of forward-centered angular samples used by the local summary slices. |
| sectors | Optional angular-sector table passed to `tmm_bistatic_summary`. |
| drop_dB | Positive dB drop used to define the backscatter-lobe width. |

## Value

A named list containing the requested post-processed TMM products.

## See Also

`tmm_scattering`, `tmm_average_orientation`, `tmm_bistatic_summary`

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
tmm_products(stored)
```

---

# tmm_scattering *Evaluate scattering from a stored TMM object*

---

## Description

Evaluates the far-field scattering amplitude from a previously computed `TMM` object using the stored transition-matrix blocks. This allows the same retained modal operator to be reused for arbitrary single-target incident and receive-angle combinations without rebuilding the boundary solve.

## Usage

```
tmm_scattering(
  object,
  theta_body = NULL,
  phi_body = NULL,
  theta_scatter = NULL,
  phi_scatter = NULL
)
```

## Arguments

| | |
|---|---|
| `object` | Scatterer-object previously evaluated with `target_strength(..., model = "TMM", store_t_matrix = TRUE)`. |
| `theta_body` | Incident polar angle (radians). Defaults to the stored TMM incident angle. |
| `phi_body` | Incident azimuth angle (radians). Defaults to the stored TMM incident angle. |
| `theta_scatter` | Receive polar angle (radians). Defaults to the exact monostatic direction, `pi - theta_body`. |
| `phi_scatter` | Receive azimuth angle (radians). Defaults to the exact monostatic direction, `phi_body + pi`. |

## Value

A data frame with the frequency, complex scattering amplitude, the corresponding differential cross section, and its level in dB.

## See Also

target_strength(), tmm_average_orientation

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
tmm_scattering(stored)
```

---

# tmm_scattering_grid *Evaluate a 2D scattering grid from a stored TMM object*

---

## Description

Reuses the stored T-matrix blocks to evaluate the far-field scattering response over a two-dimensional receive-angle grid at one stored frequency. This is useful for bistatic scattering maps, heatmaps, and polar-style visualizations without rebuilding the retained modal solve. In the current package build, this helper is available for the spherical and spheroidal stored branches. Stored cylinders intentionally stop at exact monostatic reuse until a validated retained cylinder angular operator is added.

## Usage

```
tmm_scattering_grid(
  object,
  frequency = NULL,
  theta_body = NULL,
  phi_body = NULL,
  theta_scatter = NULL,
  phi_scatter = NULL,
  n_theta = 91,
  n_phi = 181
)
```

## Arguments

| | |
|---|---|
| object | Scatterer-object previously evaluated with `target_strength(..., model = "TMM", store_t_matrix = TRUE)`. |
| frequency | Stored frequency (Hz) to evaluate. Required when the object contains more than one stored frequency. |
| theta_body | Incident polar angle (radians). Defaults to the stored TMM incident angle. |
| phi_body | Incident azimuth angle (radians). Defaults to the stored TMM incident angle. |
| theta_scatter | Optional vector of receive polar angles (radians). Defaults to an evenly spaced grid on `[0, pi]`. |
| phi_scatter | Optional vector of receive azimuth angles (radians). Defaults to an evenly spaced grid on `[0, 2*pi]`. |
| n_theta | Number of default polar-angle grid points when `theta_scatter` is not supplied. |
| n_phi | Number of default azimuth grid points when `phi_scatter` is not supplied. |

## Value

A list containing the stored frequency, the incident angles used to build the grid, the receive-angle vectors, and matrices for the complex scattering amplitude, differential scattering cross section, and its level in dB.

## See Also

tmm_scattering, tmm_average_orientation

## Examples

```
target <- fls_generate(
  shape = sphere(radius_body = 0.005, n_segments = 20),
  g_body = 1,
  h_body = 1
)
stored <- target_strength(
  target,
  frequency = 12e3,
  model = "tmm",
  boundary = "pressure_release",
  store_t_matrix = TRUE
)
tmm_scattering_grid(stored, n_theta = 5, n_phi = 9)
```

translate_shape *Translate a stored shape or scatterer component*

## Description

Shift a Shape object or one geometry-bearing component of a Scatterer without rebuilding it from scratch. This is most useful for re-centering profiles, aligning a stored body to a preferred axial origin, or nudging an internal component before model comparisons.

## Usage

```
translate_shape(
  object,
  x_offset = 0,
  y_offset = 0,
  z_offset = 0,
  component = NULL,
  containment = c("warn", "error", "ignore")
)
```

## Arguments

| | |
|---|---|
| object | Shape or Scatterer object. |
| x_offset | Along-axis translation (m). |
| y_offset | Lateral translation (m). This is ignored for profile-style geometries that do not store an explicit lateral centerline. |
| z_offset | Vertical translation (m). |
| component | Optional component name for scatterers. Defaults to the primary geometry ("body" for most scatterers and "shell" for ESS). |
| containment | Containment policy used when a moved swimbladder or backbone is checked against its body: "warn", "error", or "ignore". |

## Value

The modified object, returned as the same broad object type.

## See Also

reanchor_shape(), offset_component(), brake(), reforge(), extract()

## Examples

```
shape_obj <- cylinder(
  length_body = 0.05, radius_body = 0.003,
  n_segments = 40
)
```

```
moved_shape <- translate_shape(shape_obj, x_offset = 0.01)
range(extract(moved_shape, c("position_matrix", "x")))
```

---

`transmission_coefficient` *Plane wave/plane interface transmission coefficient*

---

## Description

Plane wave/plane interface transmission coefficient

## Usage

```
transmission_coefficient(interface1, interface2, mode = "DWBA")
```

## Arguments

| | |
|---|---|
| `interface1` | Dataframe object containing density (kg/m^3) and sound speed (m/s) values for a boundary/interface (1) |
| `interface2` | Dataframe object containing density (kg/m^3) and sound speed (m/s) values for a boundary/interface (2) |
| `mode` | Two options: coefficient calculation for "DWBA" and "KRM" |

## Value

Pressure-amplitude transmission coefficient at normal incidence.

## Examples

```
seawater <- data.frame(density = 1026, sound_speed = 1480)
animal <- data.frame(density = 1050, sound_speed = 1530)
transmission_coefficient(seawater, animal)
```

---

`unregister_model` *Remove a user-defined target-strength model registration*

---

## Description

Remove a user-defined target-strength model registration

## Usage

```
unregister_model(name, remove_persisted = TRUE)
```

### Arguments

| | |
|---|---|
| name | Canonical model name or alias. |
| remove_persisted | Logical scalar. When TRUE, any persisted registry entry is also removed from the user's config file. |

### Value

Invisibly returns the removed canonical model name.

### Examples

```
if (interactive()) {
  initialize_demo <- function(object, frequency, ...) object
  solve_demo <- function(object) object
  register_model(
    "demo",
    initialize = initialize_demo,
    solver = solve_demo
  )
  unregister_model("demo")
}
```

---

## vecnorm *Calculates the Euclidean norm across each row of a given matrix.*

---

### Description

Calculates the Euclidean norm across each row of a given matrix.

### Usage

```
vecnorm( x )
```

### Arguments

| | |
|---|---|
| x | A matrix with numeric, real values. |

### Value

Calculates the Euclidean norm of a vector.

### Examples

```
values <- matrix(c(1, 2, 3), ncol = 3)
vecnorm(values) # should yield 3.741657
```

wavenumber *Calculate the acoustic wavenumber ($k$) for a given frequency and sound speed.*

## Description

Calculates the acoustic wavenumber ($k$) for a given frequency and sound speed in water. The wavenumber is defined as:

$$k = \frac{2\pi f}{c}$$

where $f$ is the frequency (Hz) and $c$ is the sound speed ($ms^{-1}$). The wavenumber describes the spatial frequency of a sound wave and is fundamental in acoustic calculations.

## Usage

```
wavenumber(frequency, sound_speed)
```

## Arguments

| | |
|---|---|
| frequency | Frequency (f, Hz) |
| sound_speed | Sound speed (c, $m\ s^{-1}$) |

## Value

Calculates the acoustic wavenumber (k) based on the sound speed of water.

## Examples

```
wavenumber(frequency = 38e3, sound_speed = 1500)
```

yc *Cylindrical Bessel function of the second kind, $Y_\nu(x)$, and its respective derivatives*

## Description

Computes the cylindrical Bessel function of the second kind ($Y_\nu(z)$), also known as the Neumann function or Weber function, and its derivatives through `ycdk()`.

## Usage

```
yc(l, n)

ycdk(l, n, k)
```

## Arguments

| | |
|---|---|
| l | Numeric. The order ($\nu$) of the Bessel function. Must be purely real; complex orders are not supported. |
| n | Numeric or complex. The argument ($z$) at which to evaluate the function. Supports purely real or purely imaginary values. General complex arguments ($x+iy$ with $x \neq 0$ and $y \neq 0$) are not supported. |
| k | Non-negative integer. The order of the derivative for ycdk. |

## Details

The cylindrical Bessel function of the second kind satisfies the same differential equation as $J_\nu(z)$:

$$z^2 \frac{d^2 Y_\nu}{dz^2} + z \frac{dY_\nu}{dz} + (z^2 - \nu^2) Y_\nu = 0$$

but represents the linearly independent second solution.

**Supported argument types:**

- **Purely real arguments** ($z = x$, where $x \in \mathbb{R}$): Fully supported for both positive and negative values.
- **Purely imaginary arguments** ($z = iy$, where $y \in \mathbb{R}$): Computed using the identity $Y_\nu(iy) = ie^{i\pi\nu/2} I_\nu(y) - \frac{2}{\pi} e^{-i\pi\nu/2} K_\nu(y)$ for $y > 0$, where $I_\nu$ and $K_\nu$ are modified Bessel functions. Negative imaginary arguments use complex conjugation for real order.
- **General complex arguments** ($z = x + iy$, where $x \neq 0$ and $y \neq 0$): **Not supported**.

**Special cases:**

- $Y_\nu(0) = -\infty$ (singularity at the origin).
- Negative real arguments use the upper lip of the principal branch cut. For $x > 0$, $Y_\nu(-x) = e^{-i\pi\nu} Y_\nu(x) + 2i \cos(\pi\nu) J_\nu(x)$.
- For integer order $n$: $Y_n(-x) = (-1)^n [Y_n(x) + 2i J_n(x)]$.
- k-th derivative (DLMF 10.6.1):

$$\frac{d^k}{dz^k} Y_\nu(z) = \frac{1}{2^k} \sum_{j=0}^{k} (-1)^j \binom{k}{j} Y_{\nu-k+2j}(z)$$

**Derivative:**

$$Y'_\nu(z) = Y_{\nu-1}(z) - \frac{\nu}{z} Y_\nu(z)$$

## Value

A complex vector containing:

- yc: $Y_\nu(z)$
- ycdk(..., k = 1): $Y'_\nu(z)$ (first derivative)
- ycdk(..., k = 2): $Y''_\nu(z)$ (second derivative)
- ycdk: $Y_\nu^{(k)}(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 9.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Bessel's equation: https://dlmf.nist.gov/10.2
- Negative argument identity ($Y_\nu(-z)$): Eq. 10.4.1 at https://dlmf.nist.gov/10.4
- Imaginary argument identity ($Y_\nu(iz)$): Eq. 10.27.8 at https://dlmf.nist.gov/10.27

## See Also

jc for Bessel functions of the first kind, hc for Hankel functions (third kind), ys for spherical Bessel functions of the second kind.

## Examples

```
# Real argument, integer order
yc(0, 1)
yc(1, 2.5)

# Real argument, fractional order
yc(0.5, 3)

# Negative real argument
yc(2, -1.5)

# Purely imaginary argument
yc(1, 1i)
yc(2, 3i)

# Singularity at origin
yc(0, 0) # Returns -Inf

# First derivative
ycdk(1, 2, 1)
```

---

# young *Calculate Young's modulus (E).*

---

## Description

Calculates Young's modulus (E) from two of the three other elastic moduli: bulk modulus (K), shear modulus (G), or Poisson's ratio ($\nu$). Assumes 3D material properties.

The relationships used are:

$$E = \frac{9KG}{3K + G}$$

$$E = 3K(1 - 2\nu)$$

$$E = 2G(1 + \nu)$$

## Usage

```
young(K = NULL, G = NULL, nu = NULL)
```

## Arguments

| | |
|---|---|
| K | Bulk modulus (Pa). |
| G | Shear modulus (Pa). |
| nu | Poisson's ratio (Dimensionless). |

## Value

Young's modulus (E, Pa).

## Examples

```
young(K = 2.5e9, G = 1.1e9)
```

---

# ys *Spherical Bessel function of the second kind, $y_\nu(z)$, and its respective derivatives*

---

## Description

Computes the spherical Bessel function of the second kind ($y_\nu(z)$), also known as the spherical Neumann function, and its k-th derivatives.

## Usage

```
ys(l, n)

ysdk(l, n, k)
```

## Arguments

| | |
|---|---|
| l | Numeric. The order of the spherical Bessel function. Can be integer or fractional. |
| n | Numeric or matrix. The argument ($z$) at which to evaluate the function. If a matrix is provided, the function is applied column-wise. |
| k | Non-negative integer. The order of the derivative for ysdk. |

## Details

The spherical Bessel function of the second kind is related to the cylindrical Bessel function by:

$$y_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+1/2}(z)$$

where $Y_\nu(z)$ is the cylindrical Bessel function of the second kind.

The spherical Bessel functions satisfy the same differential equation as $j_\nu(z)$:

$$z^2 \frac{d^2 y_\nu}{dz^2} + 2z \frac{dy_\nu}{dz} + [z^2 - \nu(\nu+1)] y_\nu = 0$$

**Special cases:**

- $y_\nu(0) = -\infty$ (singularity at the origin).
- $y_0(z) = -\frac{\cos(z)}{z}$
- $y_1(z) = -\frac{\cos(z)}{z^2} - \frac{\sin(z)}{z}$

**Derivatives:**

- First derivative: $y'_\nu(z) = \frac{\nu}{z} y_\nu(z) - y_{\nu+1}(z)$
- Second derivative: $y''_\nu(z) = -\frac{\nu}{z^2} y_\nu(z) + \frac{\nu}{z} y'_\nu(z) - y'_{\nu+1}(z)$

## Value

A numeric vector or matrix (matching the input structure) containing:

- `ys`: $y_\nu(z)$
- `ysd`: $y_l^{(k)'}(z)$ (k-th derivative)

## References

Abramowitz, M. and Stegun, I.A. (Eds.). (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards, Applied Mathematics Series 55. Chapter 10.

NIST Digital Library of Mathematical Functions. https://dlmf.nist.gov/

- Spherical Bessel functions: https://dlmf.nist.gov/10.47
- Relation to cylindrical Bessel functions: Eq. 10.47.4 at https://dlmf.nist.gov/10.47

## See Also

`yc` for cylindrical Bessel functions of the second kind, `js` for spherical Bessel functions of the first kind, `hs` for spherical Hankel functions.

## Examples

```
# Spherical Bessel function of the second kind
ys(0, 1)
ys(1, 2.5)

# Fractional order
ys(0.5, 3)

# Vector input
ys(0, c(1, 2, 3))

# Singularity at origin
ys(0, 0) # Returns -Inf

# First derivative
ysdk(1, 2, 1)

# Second derivative
ysdk(1, 2, 2)

# 3rd derivative
ysdk(1, 1, 3)

# 4th derivative
ysdk(1, 1, 4)
```

# Index


∗ **bessel**
  hc, 34
  hs, 36
  jc, 39
  js, 41
  yc, 98
  ys, 101
∗ **datasets**
  benchmark_ts, 9
  cod, 15
  krill, 43
  sardine, 68
∗ **derivative**
  Pndk, 52
  Qndk, 59
∗ **elastic**
  bulk, 11
  lame, 44
  pois, 54
  shear, 74
  young, 100
∗ **integration**
  gauss_legendre, 33
∗ **legendre**
  Pn, 51
  Pndk, 52
  Qn, 57
  Qndk, 59
∗ **manipulator**
  brake, 10
  flip_shape, 27
  inflate_shape, 37
  offset_component, 48
  reanchor_shape, 61
  resample_shape, 64
  smooth_shape, 80
  translate_shape, 95
∗ **plotting**
  plot.Scatterer, 49
∗ **quadrature**
  gauss_legendre, 33
∗ **scatterer_type_generation**
  bbf_generate, 7
  cal_generate, 12
  ela_generate, 22
  ess_generate, 24
  fls_generate, 28
  gas_generate, 31
  sbf_generate, 70
∗ **scatterer_type**
  BBF, 6
  CAL, 12
  CSC, 19
  ELA, 21
  ESS, 23
  FLS, 28
  GAS, 31
  SBF, 70
  Scatterer, 73
∗ **shape_generation**
  arbitrary, 5
  canonicalize_shape, 14
  create_shape, 18
  cylinder, 20
  oblate_spheroid, 47
  polynomial_cylinder, 55
  prolate_spheroid, 56
  sphere, 81
∗ **shape**
  brake, 10
  flip_shape, 27
  inflate_shape, 37
  offset_component, 48
  reanchor_shape, 61
  resample_shape, 64
  smooth_shape, 80
  translate_shape, 95
∗ **utility**

extract, 26
plot.Scatterer, 49

along_sum, 4
Arbitrary, *5*
arbitrary, 5
arbitrary(), *15*
available_models, 6

BBF, 6, *8*, *19*, *74*
BBF-class *(*BBF*)*, 6
bbf_generate, 7
BBFM, *82–84*
BCMS, *82–84*
benchmark_ts, 9
brake, 10
brake(), *26*, *38*, *61*, *62*, *81*, *95*
bulk, 11

CAL, 12, *13*, *21*, *23*, *74*
CAL-class *(*CAL*)*, 12
cal_generate, 12
canonicalize_shape, 14
cod, 15
compressibility, 17
create_shape, 18
create_shape(), *15*
CSC, *6*, 19, *74*
CSC-class *(*CSC*)*, 19
Cylinder, *20*
cylinder, *8*, 20
cylinder(), *15*
cylinder(...), *18*

db *(*linear*)*, 45
degrees, 21
DWBA, *82–84*

ECMS, *82–84*
ELA, *12*, 21, *23*, *74*
ELA-class *(*ELA*)*, 21
ela_generate, 22
ESS, *12*, *21*, *23*, 23, *25*, *74*
ESS-class *(*ESS*)*, 23
ess_generate, 24
ESSMS, *82–84*
extract, 26
extract(), *10*, *50*, *61*, *62*, *64*, *95*

FCMS, *82–84*
flip_shape, 27
flip_shape(), *10*, *26*, *38*
FLS, *8*, 28, *30*, *73*, *74*
FLS-class *(*FLS*)*, 28
fls_generate, 28

GAS, 31, *32*, *74*
GAS-class *(*GAS*)*, 31
gas_generate, 31
gauss_legendre, 33

hc, 34, *37*, *40*, *100*
hcdk *(*hc*)*, 34
HPA, *82–84*
hs, *35*, 36, *42*, *102*
hsdk *(*hs*)*, 36

inflate_shape, 37
inflate_shape(), *10*, *26*, *28*, *64*, *81*

jc, *35*, 39, *42*, *100*
jcdk *(*jc*)*, 39
js, *37*, *40*, 41, *102*
jsdk *(*js*)*, 41

krill, 43
KRM, *82–84*

lame, 44
linear, 45

neumann, 46

oblate_spheroid, 47
oblate_spheroid(), *15*
oblate_spheroid(...), *18*
OblateSpheroid, *47*
offset_component, 48
offset_component(), *10*, *26*, *95*

PCDWBA, *82–84*
plot.Scatterer, 49
plot.Scatterer(), *62*
Pn, 51, *53*, *58*
Pndk, *52*, 52
pois, 54
polynomial_cylinder, 55
polynomial_cylinder(...), *18*
PolynomialCylinder, *55*
prolate_spheroid, 56

prolate_spheroid(), *15*
prolate_spheroid(...), *18*
ProlateSpheroid, *57*
PSMS, *82–84*

Qn, *52*, 57, *59*
Qndk, *58*, 59

radians, 60
reanchor_shape, 61
reanchor_shape(), *10*, *26*, *28*, *48*, *95*
reforge, 62
reforge(), *10*, *26*, *38*, *48*, *61*, *64*, *81*, *95*
register_model, 63
resample_shape, 64
resample_shape(), *10*, *26*, *38*, *81*
reset_model_registry, 65
rho, 65
Rmn, 66, *80*

sardine, 68
SBF, *19*, 70, *72–74*
SBF-class *(*SBF*)*, 70
sbf_generate, 70
Scatterer, *6*, *12*, *19*, *21*, *23*, *28*, *31*, *70*, 73
Scatterer-class *(*Scatterer*)*, 73
SDWBA, *82–84*
Shape, *19*
shear, 74
simulate_ts, 75
Smn, *68*, 78
smooth_shape, 80
smooth_shape(), *10*, *26*, *28*, *38*, *64*
SOEMS, *82–84*
Sphere, *82*
sphere, 81
sphere(), *15*
sphere(...), *18*
SPHMS, *82–84*

target_strength, 82
target_strength(), *50*, *63*, *93*
TMM, *83, 84*
tmm_average_orientation, 84, *89–91*, *93, 94*
tmm_bistatic_summary, 86, *91*
tmm_diagnostics, 87
tmm_orientation_distribution, *85*, 89, *91*
tmm_products, *87*, *89, 90*, 90
tmm_scattering, *85*, *89*, *91*, 92, *94*
tmm_scattering(), *50*
tmm_scattering_grid, *87*, *89*, 93
tmm_scattering_grid(), *50*
translate_shape, 95
translate_shape(), *10*, *26*, *28*, *48*, *61*
transmission_coefficient, 96
TRCM, *82–84*

unregister_model, 96

vecnorm, 97
VESMS, *82–84*

wavenumber, 98

yc, *35*, *40*, 98, *102*
ycdk *(*yc*)*, 98
young, 100
ys, *37*, *42*, *100*, 101
ysdk *(*ys*)*, 101