zest

Zernike and Spherical harmonic Transforms

Sebastian Sassi

Version 0.1.0

Apr 25, 2025

CONTENTS:

0.1 Getting started

0.1.1 Installation

Before proceeding to installation, it is worth noting that zest mostly does not depend on libraries other than standard lbrary. However, an exception to this is that performing least squares fits of spherical harmonic and Zernike expansions requires linking with LAPACK.

For the installation you need to obtain the source code, e.g., by cloning the git repository. Then, navigate to the source directory

```
git clone https://github.com/sebsassi/zest.git
cd zest
```

If you are familiar with CMake, zest follows a conventional CMake build/install procedure. Even if not, the process is simple: first, create a directory where the library is built, say `build`, and then build the sources in that directory, e.g.,

```
cmake --preset=default
cmake --build build
```

The default configuration here should be adequate. After that you can install the built library from the build directory to our desired location

```
cmake --install build --prefix <install directory>
```

Here `install directory` denotes your preferred installation location.

0.1.2 Basic Usage

To test the installation and take our first steps in using the library, we can create a short program that evaluates the spherical harmonic expansion of a function, rotates it, and prints out the rotated coefficients. Make a file `rotate_sh.cpp` with the following contents

```cpp
#include "zest/sh_glq_transformer.hpp"
#include "zest/rotor.hpp"

#include <cmath>
#include <cstdio>

int main()
{
    auto function = [](double lon, double colat)
    {
        const double x = std::sin(colat)*std::cos(lon);
        return std::exp(-x*x);
    };

    // Evaluate the function on a Gauss-Legendre quadrature grid
    constexpr std::size_t order = 20;
    zest::st::SphereGLQGridPoints points{};
    zest::st::SphereGLQGrid grid
        = points.generate_values(function, order);

    // Transform the grid to obtain its spherical harmonic expansion
    zest::st::GLQTransformerGeo transformer{};
    zest::st::RealSHExpansion expansion
        = transformer.forward_transform(grid, order);
```

(continues on next page)

(continued from previous page)

```
    // Euler angles
    const double alpha = std::numbers::pi/2;
    const double beta = std::numbers::pi/4;
    const double gamma = 0;

    // Rotate the expansion coefficients
    std::array<double, 3> angles = {alpha, beta, gamma};
    zest::WignerdPiHalfCollection wigner(order);
    zest::Rotor rotor{};
    rotor.rotate(expansion, wigner, angles);

    for (std::size_t l = 0; l < expansion.order(); ++l)
    {
        for (std::size_t m = 0; m <= l; ++m)
            std::printf("f[%lu, %lu] = %f", l, m, expansion(l, m));
    }
}
```

Now, to compile the code, we use GCC in this example and link our code with zest

```
g++ -std=c++20 -O3 -mfma -mavx2 -o rotate_sh rotate_sh.cpp -lzest
```

There are few things of note here. First, zest is built on the C++20 standard, and therefore requires a sufficiently modern compiler, which implements the necessary C++20 features. To tell GCC we are using C++20, we give the flag `std=c++20`.

Secondly, the performance of the library is sensitive to compiler optimizations. As a baseline, we use the optimization level `-O3` to enable all architecture-independent optimizations in GCC. On top of that, this example assumes that we are building for an x86 CPU, which supports floating point fused multiply-add operations (`-mfma`) and AVX2 SIMD operations (`-mavx2`). These options form a good performant baseline that should work for all modern x86 CPUs. In general, if you will be running your code on the system you compile it on `-march=native` should be a decent alternative to these options.

0.2 Theoretical background

This section aims to give a brief introduction to the mathematics that underlie this library. This introduction assumes basic mathematical knowledge of linear algebra, integration, and basis functions.

0.2.1 Spherical harmonics

Spherical harmonics, denoted $Y_{lm}(\theta,\varphi)$, are a collection of special functions defined on the sphere. They form a complete orthogonal basis on the sphere S^2, meaning that any square integrable function $f(\theta,\varphi)\in L^2(S^2)$ can be represented as a linear combination of (possibly an infinite number of) spherical harmonics

$$f(\theta,\varphi)=\sum_{l=0}^{\infty}\sum_{|m|\leq l} f_{lm}Y_{lm}(\theta,\varphi),$$

such that

$$\int_{S^2} Y_{lm}(\theta,\varphi)^* Y_{l'm'}(\theta,\varphi)\,d\Omega = N_{lm}\delta_{ll'}\delta_{mm'}.$$

Here δ_{ij} is the Kroenecker delta, and N_{lm} is a normalization constant that depends on the normalization convention of spherical harmonics.

Spherical harmonics come in two forms: complex spherical harmonics, which we denote as Y_l^m with upper index m, and real spherical harmonics which we denote as Y_{lm} with upper index m. This library has been built around real spherical harmonics with currently no support for full complex spherical harmonics. Therefore this introduction is presented in terms of real spherical harmonics, apart from cases where complex spherical harmonics are necessary. The real and complex spherical harmonics are related via a linear transformation

$$Y_{lm} = \begin{cases} \frac{i}{\sqrt{2}}(Y_l^m - (-1)^m Y_l^{-m}) & \text{if} m < 0, \\ Y_l^0 & \text{if} m = 0, \\ \frac{1}{\sqrt{2}}(Y_l^{-m} + (-1)^m Y_l^m) & \text{if} m > 0. \end{cases}$$

Spherical harmonics can be expressed in closed form using the associated Legendre polynomials $P_l^m(x)$. There are multiple possible conventions for writing the spherical harmonic functions depending on two factors: the normalization discussed above, and the presence of the so-called Condon–Shortley phase. For brevity, we do not write down all possible permutations of conventions here. Two conventions of note are the geodesy convention

$$Y_l^m(\theta,\varphi) = \sqrt{(2l+1)\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)e^{im\varphi},$$

and quantum mechanics convention

$$Y_l^m(\theta,\varphi) = (-1)^m \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)e^{im\varphi}.$$

For the geodesy convention, the normalization constant is $N_{lm} = 4\pi$, whereas the quantum mechanics spherical harmonics are unit normalized with $N_{lm} = 1$. The quantum mechanics convention also includes the Condon–Shortley phase factor $(-1)^m$, whereas the geodesy convention doesn't.

This library is convention agnostic to an extent. It supports both Condon–Shortley phase conventions, and allows a choice between unit and 4π normalization, but does not at present support all possible normalization conventions.

For completeness, we also write down the real spherical harmonics in the geodesy convention

$$Y_{lm}(\theta,\varphi) = \begin{cases} \sqrt{2(2l+1)\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)\sin(|m|\varphi) & \text{if} m < 0, \\ \sqrt{2l+1} P_l^m(\cos\theta) & \text{if} m = 0, \\ \sqrt{2(2l+1)\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)\cos(m\varphi) & \text{if} m > 0, \end{cases}$$

It is commonplace to absorb the normalization to the associated Legendre polynomials. Therefore we define

$$\bar{P}_l^m(x) = \sqrt{(2l+1)\frac{(l-m)!}{(l+m)!}} P_l^m(x),$$

for, e.g., the 4π normalization convention.

0.2.2 Spherical harmonic transforms

Spherical harmonic transform here refers to the process of finding the expansion coefficients f_{lm} given a function $f(\theta,\varphi)$. This is in principle straightforward, since it is easy to check that the coefficients can be written as

$$f_{lm} = \frac{1}{N_{lm}} \int_{S^2} f(\theta,\varphi) Y_{lm}(\theta,\varphi)\, d\Omega.$$

If we express the spherical harmonics in terms of the associated Legendre polynomials and trigonometric functions, this can be written as

$$f_{lm} = \frac{1}{N_{lm}} \int_{-1}^{1} \bar{P}_l^m(\cos\theta) \int_0^{2\pi} \begin{Bmatrix} \cos(m\varphi) \\ \sin(|m|\varphi) \end{Bmatrix} f(\theta,\varphi)\, \varphi\, d\cos\theta.$$

The trigonometric functions inside the curly braces denote the two different options. It is worth noting that the inner integral over φ is a Fourier transform.

In practice, given an arbitrary function, the integrals won't generally have a closed form solution, which means that in practice the integrals have to be evaluated numerically. Naively, one could do this by providing a function that evaluates $Y_{lm}(\theta, \varphi)$, and use any numerical integration routine, but this is profoundly inefficient in all aspects.

Exact numerical evaluation of all expansion coefficients is impossible, because that would require evaluating $f(\theta, \varphi)$ at an infinite number of points. To that end, we can seek an approximation given a finite set of points. Here we can rely on so-called numerical quadrature rules. The basic idea is that given a domain X, we can find a collection of points $x_i \in X$, with $i = 1, 2, \dots, N$, and a corresponding set of weights $w_i \in \mathbb{R}$, such that for any polynomial such that we can approximate the integral of a function $f : X \to \mathbb{R}$ with a sum

$$\int_X f(x)\,dx \approx \sum_{i=1}^{N} w_i f(x_i).$$

In particular, it is possible to find a quadrature rule that integrates polynomials up to some order exactly. That is, there exists an integer $M \geq N$ such that for any polynomial $P_M(x)$ of order M, we have an exact equality

$$\int_X P_M(x)\,dx \approx \sum_{i=1}^{N} w_i P_X(x_i).$$

A particular example of such a quadrature rule is Gauss–Legendre quadrature for functions defined on the interval $[-1, 1]$, for which $M = 2N - 1$. The Gauss–Legendre quadrature rule is the basis of the fast spherical harmonic transform.

To return back to the spherical harmonic expansion coefficients, let θ_i, with $i = 0, \dots, L$, be such that $z_i = cos\theta_i \in [-1, 1]$ are the Gauss–Legendre quadrature nodes, and let $\varphi_j = 2\pi j/(2L + 1)$, with $j = 0, \dots, 2L$. We can now write

$$f_{lm} \approx \sum_{i=0} L w_i \bar{P}_l^m(z_i) \sum_{j=0}^{2L} \begin{Bmatrix} \cos(m\varphi_j) \\ \sin(|m|\varphi_j) \end{Bmatrix} f(\theta_i, \varphi_j).$$

Now, if $f_L(\theta, \varphi)$ is a function which can be expressed as a finite linear combination of spherical harmonics such that

$$f_L(\theta, \varphi) = \sum_{l=0} L \sum_{|m|\leq l} f_{lm} Y_{lm}(\theta, \varphi),$$

then the above relation will be exact. Therefore, f_L can be regarded as the best interpolating truncation approximation, up to degree L, for the function f on the grid defined by θ_i and φ_j.

If we consider the number of operations it takes to evaluate all the coefficients up to degree L, we may note that there are $(L + 1)^2$ coefficients, and $(L + 1)(2L + 1)$ grid points. Therefore it appears that it would take $\mathcal{O}(L^4)$ operations to evaluate all coefficients. However, at closer inspection, we may observe that it is possible to first evaluate the intermediate coefficients

$$f_m(\theta_i) = \sum_{j=0}^{2L} \begin{Bmatrix} \cos(m\varphi_j) \\ \sin(|m|\varphi_j) \end{Bmatrix} f(\theta_i, \varphi_j).$$

This is nothing more than a discrete Fourier transform, and the intermediate coefficients can therefore be evaluated in $\mathcal{O}(L^2 \log L)$ operations using a fast Fourier transform. After that, the sums

$$f_{lm} \approx \sum_{i=0}^{L} \bar{P}_l^m(z_i) f_m(\theta_i)$$

can be evaluated in $\mathcal{O}(L^3)$ operations, leaving us with an operation count that only grows as $\mathcal{O}(L^3)$ to evaluate the spherical harmonic transform.

The inverse transform, from the coefficients back to the grid, can be performed using the same set of operations in reverse. That is, we can first compute the intermediate coefficients $f_m(\theta_i)$ by summing f_{lm} over the associated Legendre polynomials, and then perform a fast Fourier transform to get the gridded values $f(\theta_i, \varphi_j)$.

0.2.3 Zernike functions

The (3D) Zernike functions are a collection of functions that form an orthogonal basis on the unit ball B, defined as the points $x \in \mathbb{R}^3$ such that $\|x\| \leq 1$. It is worth noting that, conventionally, "Zernike functions" or "Zernike polynomials", refers to an analogous collection of 2D functions that form a basis on the unit disk. Here we will refer to the 3D functions as simply "Zernike functions".

The Zernike functions can be written as

$$Z_{nlm}^{(\alpha)}(\rho,\theta,\varphi) = R_{nl}^{(\alpha)}(\rho)Y_{lm}(\theta,\varphi),$$

where the radial functions can be defined using Jacobi polyonomials $P_n^{\alpha,\beta}(x)$ as

$$R_{nl}^{(\alpha)}(\rho) = (1-\rho^2)^\alpha \rho^l P_{(n-l)/2}^{(\alpha,l+1/2)}(2\rho^2 - 1).$$

The parameter α defines multiple families of Zernike polynomials. For practical purposes, the family defined by $\alpha = 0$ is the simplest to deal with, and is what is used by this library. For this reason, we will denote the Zernike functions in this family simply by Z_{nlm}.

An important point about Zernike functions is that because the indices $(n-l)/2$ of the Jacobi polynomials must be nonnegative integers, n and l are restricted to having the same parity. That is, if n is even, then l must be even, and if n is odd, then l must be odd.

Since the Zernike functions form an orthogonal basis, any function on the unit ball can be written as

$$f(\rho,\theta,\varphi) = \sum_{\frac{1}{2}(n-l)\in\mathbb{N}} \sum_{|m|\leq l} f_{nlm} Z_{nlm}(\rho,\theta,\varphi),$$

and we have an orthogonality relation

$$\int_B Z_{nlm}(\rho,\theta,\varphi) Z_{n'l'm'}(\rho,\theta,\varphi)\, dV = N_{nlm}\delta_{nn'}\delta_{ll'}\delta_{mm'}.$$

The ambiquity about the phase and normalization of spherical harmonics naturally applies to Zernike functions, but there is an additional ambiquity over the normalization of the radial Zernike functions themselves. Per the definition of $R_{nl}^{(\alpha)}(\rho)$ given above, we have an orthogonality relation

$$\int_0^1 R_{nl}^{(\alpha)}(\rho) R_{n'l}^{(\alpha)}(\rho) \frac{\rho^2\, d\rho}{(1-\rho^2)^\alpha} = N_{nl}^{(\alpha)}\delta_{nn'}.$$

with

$$N_{nl}^{(\alpha)} = \frac{1}{2(n+\alpha+3/2)} \frac{((n-l)/2+1)_\alpha}{((n-l)/2+l+3/2)_\alpha}.$$

The notation $(x)_\alpha$ is the Pochammer symbol, but we don't need to worry about it much, because under $\alpha = 0$ the expression reduces to

$$N_{nl}^{(0)} = \frac{1}{2n+3}.$$

Like in the case of spherical harmonics, this normalization can be absorbed into the definition of $R_{nl}^{(\alpha)}(\rho)$ to get unit-normalized Zernike functions. Both conventions are supported by zest.

0.2.4 Zernike transforms

Just as we have the spherical harmonic transform to obtain the spherical harmonic expansion coefficients of a function defined on the sphere, we have a Zernike transform to obtain the Zernike expansion coefficients of a function on the ball. Similarly to the case of spherical harmonics, it is straightforward to see that the Zernike expansion coefficients of $f(\rho,\theta,\varphi)$ are given by

$$f_{nlm} = \int_B f(\rho,\theta,\varphi) Z_{nlm}(\rho,\theta,\varphi)\rho^2\, d\rho\, d\Omega.$$

The numerical Zernike transform algorithm is effectively the same as the spherical harmonic transform presented earlier with an extra dimension on the grid. That is, in addition to the points θ_i, and φ_j, we can define the points ρ_k, with $k = 0, \ldots, L+1$, such that $\rho_k = (x_k + 1)/2$, where x_k are Gauss–Legendre nodes. Note that the radial direction requires one node more than the θ direction, because the radial integral comes with an extra factor of ρ. We can now write

$$f_{nlm} \approx \sum_{k=0}^{L+1} \sum_{i=0} L w_k w_i \rho_k^2 R_{nl}(\rho_k) \bar{P}_l^m(z_i) \sum_{j=0}^{2L} \begin{Bmatrix} \cos(m\varphi_j) \\ \sin(|m|\varphi_j) \end{Bmatrix} f(\rho_k, \theta_i, \varphi_j).$$

As is the case with the spherical harmonic transform, this transform can be performed stepwise, first computing intermediate coefficients $f_m(\rho_k, \theta_i)$, by doing the innermost sum, then the intermediate coefficients $f_{lm}(\rho_k)$ from the middle sum, and then finally f_{nlm} are obtained by doing the last sum. This means that the entire transformation can be performed with $\mathcal{O}(L^4)$ operations.

0.2.5 Rotations

It is common that we have a function expressed in terms of a spherical harmonic expansion

$$f(\theta, \varphi) = \sum_{l=0}^{\infty} \sum_{|m| \leq l} f_{lm} Y_{lm}(\theta, \varphi).$$

and we express it in terms of coordinates (θ', φ'), which are related to the coordinates (θ, φ) by a rotation R. The challenge is then to find coefficients f'_{lm}, which express f in the rotated coordinate system,

$$f(\theta', \varphi') = \sum_{l=0}^{\infty} \sum_{|m| \leq l} f'_{lm} Y_{lm}(\theta', \varphi').$$

The spherical harmonics in the different coordinate systems are related by a linear transformation,

$$Y_l^m(\theta', \varphi') = \sum_{|m'| \leq l} D_{mm'}^{(l)}(R)^* Y_l^{m'}(\theta, \varphi).$$

Here $D_{mm'}^{(l)}(R)$ are the elements of the Wigner D-matrix. Note that this relation is specifically for the complex spherical harmonics. There is a corresponding matrix for real spherical harmonics.

The above relation can be used to find the coefficients f'^m_l of the complex spheric1 harmonic expansion,

$$f'^m_l = \sum_{|m| \leq l} D_{mm'}^{(l)}(R)^* f_l^m.$$

In principle, for a given rotation defined, e.g., by Euler angles, it is possible to compute the elements of the D-matrix and perform the matrix multiplication to get the rotation. However, an alternative approach used by zest avoids computing the D-matrix. This approach relies on two facts. First, that given Euler angles α, β, and *gamma*, the D-matrix can be expressed as

$$D_{mm'}^{(l)}(\alpha, \beta, \gamma) = e^{-im\alpha} d_{mm'}^{(l)}(\beta) e^{-im'\gamma},$$

where $d_{mm'}^{(l)}(\beta)$ are coefficients of the Wigner (small) d-matrix. The Euler angles here specifically are in the ZYZ convention, where α and γ correspond to rotations about the Z-axis, and β corresponds to a rotation about the Y-axis. The second fact is that a rotation about the Y-axis by angle β can be written as a rotation about the X-axis by 90 degrees, followed by a rotation about the Z-axis by the angle β, followed by another rotation about the X-axis by 90 degrees in the opposite direction to the first one. This is the ZXZXZ method, which has the property that all the rotations by variable angles are about the Z-axis, for which the D-matrix is diagonal. The X-rotations in turn can be expressed in terms of the d-matrix elements $d_{mm'}^{(l)}(\pi/2)$, which need to be computed once, and can be reused for all rotations.

When it comes to Zernike expansions, there is nothing special about the rotations compared to the spherical harmonic case, because the rotation only operates on the angular part. Therefore once we have determined how we apply the rotations to spherical harmonic coefficients, we get the corresponding rotations on Zernike coefficients for free.

0.3 Anatomy of zest

This section of the documentation outlines the core features of zest, and their usage, motivating some of the architectural decisions and giving guidance on best practices. Knowledge of the contents of the section on theoretical background is assumed in this section.

0.3.1 Layouts – complex multidimensional indexing

In dealing with spherical harmonic and Zernike expansions, one runs into nontrivial indexing schemes. Spherical harmonics are indexed by the pair of integers (l, m), for which the condition $|m| \leq l$ applies. With some cutoff $l \leq L$, these index are organized in a triangle in the plane. The Zernike functions, on the other hand, are indexed by the triple (n, l, m), with not only the condition $|m| \leq l \leq n$, but also that $(n - l)/2$ must be an integer, which forces n and l to have the same parity. These indices are organized in a tetrahedron with holes in it where n and l do not satisfy the parity condition.

Mapping these multidimensional index schemes onto a one-dimensional buffer is a challenging endeavor. The simplest solution would be to use a conventional multidimensional array to store the elements corresponding to the indices, but this means that there will be elements in the buffer that are never accessed. For example, the index triple $(1, 2, 3)$ doesn't correspond to any Zernike function, and therefore there is never need to access the corresponding element. For spherical harmonics, half of the elements in the buffer would never be accessed, and for Zernike functions up 83% of the elements would never be accessed. This is both wasteful in terms of memory usage, and bad for cache utilization.

Fortunately, it is relatively straightforward to create more compact schemes. for storing elements. For example, spherical harmonic coefficients can be stored sequentially by mapping the index pair (l, m) to the one-dimensional index $i = l(l + 1) + m$ without any wasted memory. An alternative is to map pairs $\{|m|, -|m|\}$ onto the indices $l(l+1)/2+|m|$. This wastes a small amount of memory because $m = 0$ maps to both elements of the pair, but has some desirable properties for iteration over the buffer.

To deal with differing indexing schemes in a unified and flexible manner, zest uses a system of *layouts*. A 2D layout, for example, is a type with the structure

```
struct SomeLayout
{
    using index_type;
    using size_type;
    using IndexRange;
    using SubLayout; // optional

    static constexpr LayoutTag layout_tag;

    static size_type size(size_type order);
    static size_type idx(index_type l, index_type m);
};
```

The function `size` gives the total number of elements in an index set as determined by the parameter `order`. The exact definition of `order` depends on the layout. For example, for spherical harmonic coefficients cut off at some degree L such that $|m| \leq l \leq L$, `order` is by convention equal to $L + 1$. The case for Zernike functions is analogous. For the layout of a basic 1D array, `order` would just be the size. The convention is such that `order = 0` always corresponds to an empty layout.

The function `idx` gives the index in the one-dimensional buffer corresponding to the pair (l, m), in this case. For example, it could return $l(l + 1) + m$ for a spherical harmonic layout.

The constant `layout_tag` is used by zest to identify what type of index geometry the layout is intended to represent.

The member type `SubLayout` is a layout of one dimension lower to facilitate accessing lower-dimensional slices of the index set. For example, for a layout for spherical harmonic coefficients, it would be a 1D layout for to the row of m values that correspond to a single l.

Finally, the member type `IndexRange` leads to another concept in zest, the concept of *index ranges*. Because of potential nontrivial restrictions on indices—e.g., that n and l must have the same parity when dealing with Zernike functions—iterating through index sets by hand is error prone. Consider, for example, iterating l for Zernike functions

```
for (std::size_t l = n % 2; l < n; l += 2)
    // Do things
```

For someone used to writing C-style for-loops, it is easy to go through the motions and write `l = 0` instead of `l = n % 2`, or `++l` instead of `l += 2`, either of which will lead to hard to diagnose bugs. Furthermore, this way it is difficult to write generic code, which can iterate both spherical harmonic and Zernike indices. Index ranges allow the use of range-based for-loops instead

```
SomeLayout::IndexRange indices;
for (auto l : indices)
    // Do things
```

0.3.2 Containers and views

For handling expansion coefficients and quadrature grids, zest presents a number of containers and views. Containers are objects which own the underlying buffer they refer to, i.e., they are responsible for allocation and deallocation of the buffer. Views are objects which do not own the buffer they refer to; they simply give a *view* to a buffer owned by some other object.

The library comes with a number of containers for easy storage and manipulation of different types of data. For storing spherical harmonic and Zernike expansions of real functions, there are the classes `zest::st::RealSHExpansion` and `zest::zt::RealZernikeExpansion` respectively.

The template parameters of these containers primarily control the various normalization conventions. The parameter `ElementType` is the type of elements in the underlying buffer. There are two main choices here: if `ElementType` is a floating point type (e.g., `double`), this implies that the elements are stored sequentially with m going from $-l$ to l. On the other hand, if `ElementType` is an array-like type of length two, e.g., `std::array<double, 2>`, then the elements are stored in pairs $\{|m|, -|m|\}$ with $|m|$ running from zero to l. The latter option is the default and recommended option when dealing with the quadrature-based transforms, but the former is mandatory for fitting an expansion to data.

For these classes, the library provides a number of convenient aliases for various common combinations of normalization and phase conventions. For spherical harmonics these aliases are

- `zest::st::RealSHExpansionAcoustics`
- `zest::st::RealSHExpansionQM`
- `zest::st::RealSHExpansionGeo`

For Zernike functions there are corresponding aliases for the unnormalized radial functions:

- `zest::zt::RealZernikeExpansionAcoustics`
- `zest::zt::RealZernikeExpansionQM`
- `zest::zt::RealZernikeExpansionGeo`

and furthermore for the normalized radial Zernike polynomials:

- `zest::zt::RealZernikeExpansionNormalAcoustics`
- `zest::zt::RealZernikeExpansionNormalQM`
- `zest::zt::RealZernikeExpansionNormalGeo`

For storage of function values on Gauss–Legendre quadrature grids there are the classes `zest::st::SphereGLQGrid` and `zest::zt::BallGLQGrid` for the sphere and ball, respectively. The `ElementType` parameter here is simply a floating point type. The parameter `LayoutType`, in turn, describes how the multidimensional grid is laid out in memory. This is not something a user of the library generally needs to worry about, because the default layout is the layout that should be used for performing the transforms to expansion coefficients.

Mirroring the convention of the C++ standard library, views to buffers in zest are referred with the word "span". Each of the above containers has a corresponding view. Thus we have `zest::st::RealSHSpan` and `zest::zt::RealZernikeSpan` with the corresponding aliases for different normalization/phase conventions, and `zest::st::SphereGLQGridSpan` and `zest::zt::BallGLQGridSpan` for the quadrature grids.

In additon, for completeness it is worth mentioning the `zest::MDSpan`, which is a general multidimensional array view, and is the base of both `zest::st::SphereGLQGridSpan` and `zest::zt::BallGLQGridSpan`. It is a poor man's alternative to C++23's `std::mdspan`, replicating the part of its interface, which is necessary for this library.

Views are very useful, because they allow for more flexible storage of the expansions and grids. For example, zest does not offer a container for storage of multiple spherical harmonic expansions, and that is by design. If one needed to work with multiple spherical harmonic expansions at the same time—a scenario which is very easy to imagine—they might be tempted to use something like `std::vector` to store the expansions. But this involves multiple memory allocations, one for each expansion, and spreads the expansions across memory, which is not cache friendly and could negatively impact performance if the expansions are small.

Instead, what one should do is allocate one buffer of the expansion's underlying type, which stores all the expansions back to back in the same buffer, and then take views into that buffer to access the different expansions. For example

```
using ExpansionSpan = zest::st::RealSHExpansionQM;

constexpr std::size_t num_expansions = 100;
constexpr std::size_t order = 10;
constexpr std::size_t expansion_size = ExpansionSpan::size(order);

std::vector<std::array<double, 2>>
expansion_buffer(num_expansions*expansion_size);

for (std::size_t i = 0; i < num_expansions; ++i)
{
    ExpansionSpan expansion(expansion_buffer.data() + i*expansion_size, order);

    // ...
}
```

As is conventional in C++ libraries prior to C++23's multidimensional subscript operator, multidimensional views and containers can be indexed with the call operator `operator()`

```
constexpr std::size_t order = 3;
zest::st::RealSHExpansion expansion(order);
expansion(0, 0) = {1.0, 0.0};
expansion(1, 0) = {0.5, 0.0};
expansion(1, 1) = {0.5, -0.5};
expansion(2, 0) = {0.25, 0.0};
expansion(2, 1) = {0.25, -0.25};
expansion(2, 2) = {0.25, -0.25};
```

All multidimensional containers and views in this library allow for lower dimensional subviews to be taken, which reproduce corresponding slices of the data. Specifically, the subscript operator `operator[]` provides access to the lower dimensional subview

```
for (auto l : expansion.indices())
{
    auto expansion_l = expansion[l];
    for (auto m : expansion_m.indices())
```

(continues on next page)

(continued from previous page)

```
    {
        expansion_l[m][0] += 0.1;
        expansion_l[m][1] -= 0.1;
    }
}
```

This example also demonstrates the use of the index ranges discussed in the previous subsection. In fact, the above is the preferred way of iterating over an expansion, because it avoids the errors that could be made in writing the constraints for the indices by hand.

0.3.3 Gauss–Legendre quadrature transformers

At the heart of zest are the Gauss–Legendre quadrature grid based transforms of spherical harmonic and Zernike expansions. These transforms are implemented by the classes `zest::st::GLQTransformer` and `zest::zt::GLQTransformer` for spherical harmonic and Zernike transforms respectively. The normalization and phase convention parameters are the same as those to the respective expansion containers discussed above. To that end, both transformer classes have a set of aliases for some commond combinations of normalization and phase conventions. These are

- `zest::st::GLQTransformerAcoustics`
- `zest::st::GLQTransformerQM`
- `zest::st::GLQTransformerGeo`

for the spherical harmonic transformer as well as

- `zest::zt::GLQTransformerAcoustics`
- `zest::zt::GLQTransformerQM`
- `zest::zt::GLQTransformerGeo`
- `zest::zt::GLQTransformerNormalAcoustics`
- `zest::zt::GLQTransformerNormalQM`
- `zest::zt::GLQTransformerNormalGeo`

for the Zernike transformer. The final parameter `GridLayoutType` in turn is the same as for the corresponding grid containers.

It goes without saying that the transformer must have the same values for these template parameters as the expansion and grid. This is one of the ways zest protects consistency of conventions in transformations.

The transformers come with two methods for performing transformations: `forward_transform` and `backward_transform`. The forward transform transforms a grid to an expansion, and the backward transform is the inverse, transforming an expansion to a grid. Both of these methods have two primary overloads, one which takes both the input and output expansion/grid as arguments and modifies the output

```
transformer.forward_transform(grid, expansion);
transformer.backward_transform(expansion, grid);
```

and one which takes the input expansion/grid and returns the output container

```
auto expansion = transformer.forward_transform(grid, order);
auto new_grid = transformer.backward_transform(expansion, order);
```

Here the method takes the additional parameter `order`. In the case of the forward transform, this parameter is the order of the expansion. Note that the grid has its own order parameter, which is the maximum expansion order that can be taken with that grid. Therefore, the order of the output expansion is `min(order, grid.order())`. On the other hand, in the backward transform, the `order` parameter determines the point at which the summation of the expansion is truncated. The order of `new_grid` will again be `min(order, expansion.order())`.

0.3.4 Rotations

For understanding this subsection discussing the implementation of rotations in zest, reading the corresponding subsection in the theoretical background is highly recommended. In summary, zest implements rotations for both spherical harmonic and Zernike expansions using the ZXZXZ algorithm. This algorithm implements a rotation by Euler angles (α, β, γ) as a series of rotations starting with a rotation about the z-axis by γ, followed by a 90 degree rotation about the new x-axis, followed by a rotation about the new z-axis by β, followed by a -90 degree rotation about the new x-axis, finally followed by a rotation about the new z-axis by α; hence ZXZXZ. This has the advantage that the general form of Wigner's D-matrices never needs to be evaluated. The x-axis rotations are expressible in terms of the d-matrix for a 90 degree rotation, and can be precomputed once, On the other hand, the z-rotations are just diagonal matrices of values $e^{im\theta_i}$, where θ_i is one of (α, β, γ).

With this brief review of the essential facts, zest has a single class `zest::Rotor` for performing the rotations, which has the method `rotate` for performing general rotations and `polar_rotate` for the special case of rotations about the z-axis

```
zest::Rotor rotor(order);
zest::WignerdPiHalfCollection wigner_d_pi2(order);

std::array<double, 3> euler_angles
    = {std::numbers::pi/4, std::numbers::pi/4, std::numbers::pi/4};
rotor.polar_rotate(
    expansion, std::numbers::pi/2, zest::RotationType::coordinate);
rotor.rotate(
    expansion, wigner_d_pi2, euler_angles, zest::RotationType::coordinate);
```

All rotations take as their last argument an enum of type `zest::RotationType`, which has two values `zest::RotationType::object` and `zest::RotationType::coordinate`. These express whether the rotation represents a rotation of an object in space (active rotation) or a rotation of the coordinate system (passive rotation). The polar rotation naturally takes as its argument a single angle, whereas the general rotation takes three Euler angles, given as a standard library array with three elements. Finally, the general rotation takes as its second argument an object of type `zest::WignerdPiHalfCollection`. This object contains the values of the d-matrix for a 90 degree angle, i.e., $\pi/2$, up to some specified order.

0.4 Library reference

0.4.1 Spherical harmonic expansions

Concepts

`template<typename T>`
`concept real_sh_expansion`

#include <real_sh_expansion.hpp> Concept describing a conventional spherical harmonic expansion.

`template<typename T>`
`concept row_skipping_real_sh_expansion`

#include <real_sh_expansion.hpp> Concept describing a spherical harmonic expansion where every other row is skipped.

Types

`template<SHNorm sh_norm_param, SHPhase sh_phase_param, typename ElementType = std::array<double, 2>>`
`class RealSHExpansion`

A container for purely real spherical harmonic data.

Template Parameters

- **`sh_norm_param`** – normalization convention of the spherical harmonics

- **sh_phase_param** – phase convention of the spherical harmonics
- **ElementType** – type of elements

Public Functions

`inline size_type order() const noexcept`

Order of the expansion.

`inline std::span<element_type> flatten() noexcept`

Flattened view of the underlying buffer.

`inline std::span<const element_type> flatten() const noexcept`

Flattened view of the underlying buffer.

`inline void resize(size_type order)`

Change the size of the expansion.

Public Static Functions

`static inline constexpr size_type size(size_type order) noexcept`

Number of data elements for size parameter `order`.

Parameters

order – parameter presenting the size of the expansion

`template<typename ElementType, typename LayoutType, SHNorm sh_norm_param, SHPhase sh_phase_param>`
`class SHLMSpan : public zest::TriangleSpan<ElementType, LayoutType>`

A non-owning view for storing 2D data related to spherical harmonics.

Template Parameters

- **ElementType** – type of elements
- **LayoutType** – layout of the elements
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

`template<typename ElementType, typename LayoutType, SHNorm sh_norm_param, SHPhase sh_phase_param>`
`class SHLMVecSpan : public zest::TriangleVecSpan<ElementType, LayoutType>`

A non-owning view for storing 3D data related to spherical harmonics.

Template Parameters

- **ElementType** – type of elements
- **LayoutType** – layout of the elements
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

Type aliases

`using zest::st::RealSHExpansionAcoustics = RealSHExpansion<SHNorm::qm, SHPhase::none>`

Convenient alias for `RealSHExpansion` with orthonormal spherical harmonics and no Condon-Shortley phase.

using zest::st::**RealSHExpansionQM** = RealSHExpansion<SHNorm::qm, SHPhase::cs>

Convenient alias for RealSHExpansion with orthonormal spherical harmonics with Condon-Shortley phase.

using zest::st::**RealSHExpansionGeo** = RealSHExpansion<SHNorm::geo, SHPhase::none>

Convenient alias for RealSHExpansion with 4-pi normal spherical harmonics and no Condon-Shortley phase.

template<sh_packing **PackingType**, SHNorm **sh_norm_param**, SHPhase **sh_phase_param**>

using zest::st::**PackedSHSpan** = SHLMSpan<**typename** PackingType::element_type, **typename** PackingType::Layout, sh_norm_param, sh_phase_param>

A non-owning view of data modeling spherical harmonic data.

Template Parameters

- **PackingType** – type of packing for the elements
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

template<**typename** **ElementType**, SHNorm **sh_norm_param**, SHPhase **sh_phase_param**>

using zest::st::**RealSHSpan** = PackedSHSpan<RealSHPacking<ElementType>, sh_norm_param, sh_phase_param>

A non-owning view of data modeling purely real spherical harmonic data.

Template Parameters

- **ElementType** – type of elements
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

template<**typename** **ElementType**>

using zest::st::**RealSHSpanAcoustics** = RealSHSpan<ElementType, SHNorm::qm, SHPhase::none>

Convenient alias for RealSHSpan with orthonormal spherical harmonics and no Condon-Shortley phase.

template<**typename** **ElementType**>

using zest::st::**RealSHSpanQM** = RealSHSpan<ElementType, SHNorm::qm, SHPhase::cs>

Convenient alias for RealSHSpan with orthonormal spherical harmonics with Condon-Shortley phase.

template<**typename** **ElementType**>

using zest::st::**RealSHSpanGeo** = RealSHSpan<ElementType, SHNorm::geo, SHPhase::none>

Convenient alias for RealSHSpan with 4-pi normal spherical harmonics and no Condon-Shortley phase.

0.4.2 Spherical harmonic transforms

Concepts

template<**typename** **T**>
concept **sphere_glq_grid**

#include <sh_glq_transformer.hpp> Concept enforcing a type to be either SphereGLQGrid or SphereGLQGridSpan.

template<**typename** **AlignmentType** = CacheLineAlignment>

struct LatLonLayout

Longitudinally contiguous layout for storing a Gauss-Legendre quadrature grid.

Template Parameters
AlignmentType – byte alignment of the grid

Public Static Functions

`static inline constexpr std::size_t size(std::size_t order) noexcept`

Number of grid points.

Parameters
order – order of spherical harmonic expansion

`static inline constexpr std::array<std::size_t, 2> shape(std::size_t order) noexcept`

Shape of the grid.

Parameters
order – order of spherical harmonic expansion

`static inline constexpr std::size_t fft_size(std::size_t order) noexcept`

Number of longitudinal Fourier coefficients.

`static inline constexpr std::array<std::size_t, 2> fft_stride(std::size_t order) noexcept`

Stride of the longitudinally Fourier transformed grid.

`static inline constexpr std::size_t lat_size(std::size_t order) noexcept`

Size in latitudinal direction.

`static inline constexpr std::size_t lon_size(std::size_t order) noexcept`

Size in latitudinal direction.

`template<typename AlignmentType = CacheLineAlignment>`

struct LonLatLayout

Latitudinally contiguous layout for storing a Gauss-Legendre quadrature grid.

Template Parameters
AlignmentType – byte alignment of the grid

Public Static Functions

`static inline constexpr std::size_t size(std::size_t order) noexcept`

Number of grid points.

Parameters
order – order of spherical harmonic expansion

`static inline constexpr std::array<std::size_t, 2> shape(std::size_t order) noexcept`

Shape of the grid.

Parameters
order – order of spherical harmonic expansion

`static inline constexpr std::size_t fft_size(std::size_t order) noexcept`

Number of longitudinal Fourier coefficients.

`static inline constexpr std::array<std::size_t, 2> fft_stride(std::size_t order)`

Stride of the longitudinally Fourier transformed grid.

`static inline constexpr std::size_t lat_size(std::size_t order) noexcept`

Size in latitudinal direction.

`static inline constexpr std::size_t lon_size(std::size_t order) noexcept`

Size in longitudinal direction.

`template<typename LayoutType = DefaultLayout>`

`class SphereGLQGridPoints`

Points defining a Gauss-Legendre quadrature grid on the sphere.

Template Parameters

LayoutType – memory layout of the grid

Public Functions

`inline void resize(std::size_t order)`

Change the size of the corresponding grid.

`inline std::array<std::size_t, 2> shape() noexcept`

Shape of the corresponding grid.

`inline std::span<const double> longitudes() const noexcept`

Longitude values of the grid points.

`inline std::span<const double> glq_nodes() const noexcept`

Latitudinal Gauss-Legendre nodes.

`template<sphere_glq_grid GridType, typename FuncType>`
`inline void generate_values(GridType &&grid, FuncType &&f)`

Generate Gauss-Legendre quadrature grid values from a function.

Template Parameters

- **GridType** – type of grid
- **FuncType** – type of function

Parameters

- **grid** – grid to place the values in
- **f** – function to generate values

`template<typename FuncType>`
`inline auto generate_values(FuncType &&f, std::size_t order)`

Generate Gauss-Legendre quadrature grid values from a function.

Template Parameters

FuncType – type of function

Parameters

f – function to generate values

`template<typename ElementType, typename LayoutType = DefaultLayout>`

`class SphereGLQGrid`

Container for Gauss-Legendre quadrature gridded data on the sphere.

Template Parameters

- **ElementType** – type of elements in the grid
- **LayoutType** – grid layout

Public Functions

inline std::size_t **order**() **const noexcept**

Order of spherical harmonic expansion.

inline std::array<std::size_t, 2> **shape**() **const noexcept**

Shape of the grid.

inline std::span<**const** element_type> **flatten**() **const noexcept**

Flattened view of the underlying buffer.

inline std::span<element_type> **flatten**() **noexcept**

Flattened view of the underlying buffer.

inline void **resize**(std::size_t order)

Change the size of the grid.

template<**typename ElementType**, **typename LayoutType** = DefaultLayout>

class SphereGLQGridSpan : **public** zest::MDSpan<ElementType, 2>

A non-owning view on data modeling a Gauss-Legendre quadrature grid on the sphere.

Template Parameters

- **ElementType** – type of elements in the grid
- **LayoutType** – grid layout

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of spherical harmonic expansion.

inline constexpr const std::array<std::size_t, 2> &**shape**() **const noexcept**

Shape of the grid.

inline constexpr std::span<element_type> **flatten**() **const noexcept**

Flattened view of the underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of grid points.

Parameters

order – order of spherical harmonic expansion

static inline constexpr std::array<std::size_t, 2> **shape**(std::size_t order) **noexcept**

Shape of the grid.

Parameters

order – order of spherical harmonic expansion

template<SHNorm **sh_norm_param**, SHPhase **sh_phase_param**, **typename GridLayoutType** = DefaultLayout>

class GLQTransformer

Transformations between a Gauss-Legendre quadrature grid representation and spherical harmonic expansion representation of real data.

Template Parameters

- **sh_norm_param** – normalization convention of spherical harmonics
- **sh_phase_param** – phase convention of spherical harmonics

- **GridLayoutType** – memory layout of the grid

Public Functions

```
inline std::size_t order() const noexcept
```

Order of spherical harmonic expansion.

```
inline void resize(std::size_t order)
```

Resize transformer for specified expansion order.

```
inline void forward_transform(SphereGLQGridSpan<const double, GridLayout> values,
                              RealSHSpan<std::array<double, 2>, sh_norm_param,
                              sh_phase_param> expansion)
```

Forward transform from Gauss-Legendre quadrature grid to spherical harmonic coefficients.

Parameters

- **values** – values on the spherical quadrature grid
- **expansion** – coefficients of the expansion

```
inline void backward_transform(RealSHSpan<const std::array<double, 2>,
                               sh_norm_param, sh_phase_param> expansion,
                               SphereGLQGridSpan<double, GridLayout> values)
```

Backward transform from spherical harmonic expansion to Gauss-Legendre quadrature grid.

Parameters

- **expansion** – coefficients of the expansion
- **values** – values on the spherical quadrature grid

```
template<row_skipping_real_sh_expansion Expansion>
inline void backward_transform(Expansion &&expansion, SphereGLQGridSpan<double,
                               GridLayout> values)
```

Backward transform from spherical harmonic expansion of even or odd parity to Gauss-Legendre quadrature grid.

> **Note**
>
> A spherical harmonic expansion has even/odd parity if the first index of all nonzero coefficients has even/odd parity.

Template Parameters

Expansion – type of expansion

Parameters

- **expansion** – coefficients of the expansion
- **values** – values on the spherical quadrature grid

```
inline RealSHExpansion<sh_norm_param, sh_phase_param> forward_transform(SphereGLQGridSpan<const double, GridLayout> values, std::size_t order)
```

Forward transform from Gauss-Legendre quadrature grid to spherical harmonic coefficients.

Parameters

- **values** – values on the spherical quadrature grid
- **order** – order of expansion

```
inline SphereGLQGrid<double, GridLayout> backward_transform(RealSHSpan<const std::array<double, 2>, sh_norm_param, sh_phase_param> expansion, std::size_t order)
```

Backward transform from spherical harmonic coefficients to Gauss-Legendre quadrature grid.

Parameters

- **values** – values on the spherical quadrature grid
- **expansion** – coefficients of the expansion

```
template<row_skipping_real_sh_expansion Expansion>
inline SphereGLQGrid<double, GridLayout> backward_transform(Expansion &&expansion, std::size_t order)
```

Backward transform from spherical harmonic expansion of even or odd parity to Gauss-Legendre quadrature grid.

> **Note**
>
> A spherical harmonic expansion has even/odd parity if the first index of all nonzero coefficients has even/odd parity.

Template Parameters

Expansion – type of expansion

Parameters

- **expansion** – coefficients of the expansion
- **values** – values on the spherical quadrature grid

```
template<st::SHNorm sh_norm_param, st::SHPhase sh_phase_param, typename GridLayoutType = DefaultLayout>
class SHTransformer
```

High-level interface for taking SH transforms of functions on balls of arbitrary radii.

Template Parameters

- **sh_norm_param** – normalization convention of spherical harmonics
- **sh_phase_param** – phase convention of spherical harmonics
- **GridLayoutType** –

Public Functions

```
inline void resize(std::size_t order)
```

Resize the transformer to work with expansions of different order.

```
template<spherical_function FuncType>
inline void transform(FuncType &&f, RealSHSpan<std::array<double, 2>, sh_norm_param, sh_phase_param> expansion)
```

Get spherical harmonic expansion of a function expressed in spherical coordinates.

Template Parameters
: **FuncType** – type of function

Parameters

- **f** – function to transform
- **expansion** – buffer to store the expansion

```
template<spherical_function FuncType>
inline RealSHExpansion<sh_norm_param, sh_phase_param> transform(FuncType &&f,
                                                                std::size_t
                                                                order)
```

Get spherical harmonic expansion of a function expressed in spherical coordinates.

Template Parameters
: **FuncType** – type of function

Parameters

- **f** – function to transform
- **order** – order of the expansion

Returns
: spherical harmonic expansion

```
template<cartesian_function FuncType>
inline void transform(FuncType &&f, RealSHSpan<std::array<double, 2>,
                      sh_norm_param, sh_phase_param> expansion)
```

Get spherical harmonic expansion of a function expressed in Cartesian coordinates.

Template Parameters
: **FuncType** – type of function

Parameters

- **f** – function to transform
- **expansion** – buffer to store the expansion

```
template<cartesian_function FuncType>
inline RealSHExpansion<sh_norm_param, sh_phase_param> transform(FuncType &&f,
                                                                std::size_t
                                                                order)
```

Get spherical harmonic expansion of a function expressed in Cartesian coordinates.

Template Parameters
: **FuncType** – type of function

Parameters
: **f** – function to transform

Returns
: spherical harmonic expansion

Type aliases

```
using zest::st::DefaultLayout = LonLatLayout<>
```

```
template<typename GridLayout = DefaultLayout>
using zest::st::GLQTransformerAcoustics = GLQTransformer<SHNorm::qm, SHPhase::none,
GridLayout>
```

Convenient alias for `GLQTransformer` with orthonormal spherical harmonics and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::st::**GLQTransformerQM** = GLQTransformer<SHNorm::qm, SHPhase::cs, GridLayout>

Convenient alias for GLQTransformer with orthonormal spherical harmonics with Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::st::**GLQTransformerGeo** = GLQTransformer<SHNorm::geo, SHPhase::none, GridLayout>

Convenient alias for GLQTransformer with 4-pi normal spherical harmonics and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::st::**SHTransformerAcoustics** = SHTransformer<SHNorm::qm, SHPhase::none, GridLayout>

Convenient alias for SHTransformer with orthonormal spherical harmonics and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::st::**SHTransformerQM** = SHTransformer<SHNorm::qm, SHPhase::cs, GridLayout>

Convenient alias for SHTransformer with orthonormal spherical harmonics with Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::st::**SHTransformerGeo** = SHTransformer<SHNorm::geo, SHPhase::none, GridLayout>

Convenient alias for SHTransformer with 4-pi normal spherical harmonics and no Condon-Shortley phase.

Template Parameters
GridLayout –

0.4.3 Zernike expansions

Concepts

template<**typename T**>
concept real_zernike_expansion

#include <zernike_expansion.hpp> Concept enforcing a type to be either RealZernikeExpansion or RealZernikeSpan.

Types

template<ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**, **typename ElementType** = std::array<double, 2>>
class RealZernikeExpansion

A container for a Zernike expansion of a real function.

Template Parameters

- **sh_norm_param** – normalization convention of spherical harmonics
- **sh_phase_param** – phase convention of spherical harmonics

Public Functions

inline size_type **order**() **const noexcept**

Order of the expansion.

inline std::span<**const** element_type> **flatten**() **const noexcept**

Flattened view of the underlying buffer.

inline std::span<element_type> **flatten**() **noexcept**

Flattened view of the underlying buffer.

inline void **resize**(size_type order)

Change the size of the expansion.

Public Static Functions

static inline constexpr size_type **size**(size_type order) **noexcept**

Number of data elements for size parameter `order`.

Parameters

order – parameter presenting the size of the expansion

template<**typename ElementType**, **typename LayoutType**, ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**>
class ZernikeSHSpan : **public** zest::TriangleSpan<ElementType, LayoutType>

A non-owning view of the Zernike expansion coefficients of a given radial index value.

Template Parameters

- **ElementType** – type of elements in the view
- **zernike_norm_param** – zernike function normalization convention
- **sh_norm_param** – spherical harmonic normalization convention
- **sh_phase_param** – spherical harmonic phase convention

template<**typename ElementType**, **typename LayoutType**, ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**>
class ZernikeNLMSpan : **public** zest::TetrahedronSpan<ElementType, LayoutType>

A non-owning view for storing 3D data related to Zernike functions.

Template Parameters

- **ElementType** – type of elements
- **LayoutType** – layout of the elements
- **zernike_norm_param** – zernike function normalization convention
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

Type aliases

using zest::zt::**RealZernikeExpansionAcoustics** = RealZernikeExpansion<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::none>

Convenient alias for RealZernikeExpansion with unnormalized Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

using zest::zt::**RealZernikeExpansionNormalAcoustics** = RealZernikeExpansion<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::none>

Convenient alias for RealZernikeExpansion with orthnormal Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

using zest::zt::**RealZernikeExpansionQM** = RealZernikeExpansion<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::cs>

Convenient alias for RealZernikeExpansion with unnormalized Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

using zest::zt::**RealZernikeExpansionNormalQM** = RealZernikeExpansion<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::cs>

Convenient alias for RealZernikeExpansion with orthonormal Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

using zest::zt::**RealZernikeExpansionGeo** = RealZernikeExpansion<ZernikeNorm::unnormed, st::SHNorm::geo, st::SHPhase::none>

Convenient alias for RealZernikeExpansion with unnormalized Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

using zest::zt::**RealZernikeExpansionNormalGeo** = RealZernikeExpansion<ZernikeNorm::normed, st::SHNorm::geo, st::SHPhase::none>

Convenient alias for RealZernikeExpansion with orthonormal Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

template<zernike_packing **PackingType**, ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**>
using zest::zt::**PackedZernikeSpan** = ZernikeNLMSpan<**typename** PackingType::element_type, **typename** PackingType::Layout, zernike_norm_param, sh_norm_param, sh_phase_param>

A non-owning view of data modeling Zernike function data.

Template Parameters

- **PackingType** – type of packing for the elements
- **zernike_norm_param** – zernike function normalization convention
- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

template<**typename ElementType**, ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**>
using zest::zt::**RealZernikeSpan** = PackedZernikeSpan<RealZernikePacking<ElementType>, zernike_norm_param, sh_norm_param, sh_phase_param>

A non-owning view of data modeling purely real Zernike function data.

Template Parameters

- **ElementType** – type of elements
- **zernike_norm_param** – zernike function normalization convention

- **sh_norm_param** – normalization convention of the spherical harmonics
- **sh_phase_param** – phase convention of the spherical harmonics

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanAcoustics = RealZernikeSpan<ElementType, ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::none>`

Convenient alias for `RealZernikeSpan` with unnormalized Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanNormalAcoustics = RealZernikeSpan<ElementType, ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::none>`

Convenient alias for `RealZernikeSpan` with orthonormal Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanQM = RealZernikeSpan<ElementType, ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::cs>`

Convenient alias for `RealZernikeSpan` with unnormalized Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanNormalQM = RealZernikeSpan<ElementType, ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::cs>`

Convenient alias for `RealZernikeSpan` with orthonormal Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanGeo = RealZernikeSpan<ElementType, ZernikeNorm::unnormed, st::SHNorm::geo, st::SHPhase::none>`

Convenient alias for `RealZernikeSpan` with unnormalized Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

`template<typename ElementType>`

`using zest::zt::RealZernikeSpanNormalGeo = RealZernikeSpan<ElementType, ZernikeNorm::normed, st::SHNorm::geo, st::SHPhase::none>`

Convenient alias for `RealZernikeSpan` with orthonormal Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
ElementType – type of elements in the view

0.4.4 Zernike transforms

Types

template<**typename AlignmentType** = CacheLineAlignment>

struct LonLatRadLayout

Layout for storing a Gauss-Legendre quadrature grid.

Template Parameters
AlignmentType – byte alignment of the grid

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of grid points.

Parameters
order – order of Zernike expansion

static inline constexpr std::array<std::size_t, 3> **shape**(std::size_t order) **noexcept**

Shape of the grid.

Parameters
order – order of Zernike expansion

static inline constexpr std::size_t **fft_size**(std::size_t order) **noexcept**

Number of longitudinal Fourier coefficients.

static inline constexpr std::array<std::size_t, 3> **fft_stride**(std::size_t order) **noexcept**

Stride of the longitudinally Fourier transformed grid.

static inline constexpr std::size_t **lat_size**(std::size_t order) **noexcept**

Size in latitudinal direction.

static inline constexpr std::size_t **rad_size**(std::size_t order) **noexcept**

Size in radial direction.

static inline constexpr std::size_t **lon_size**(std::size_t order) **noexcept**

Size in latitudinal direction.

template<**typename LayoutType** = DefaultLayout>

class BallGLQGridPoints

Points defining a grid in spherical coordinates in the unit ball.

Template Parameters
LayoutType – memory layout of the grid

Public Functions

inline void **resize**(std::size_t order)

Change the size of the corresponding grid.

inline std::span<**const** double> **longitudes**() **const noexcept**

Longitude values of the grid points.

inline std::span<**const** double> **rad_glq_nodes**() **const noexcept**

Radial Gauss-Legendre nodes.

`inline std::span<const double> lat_glq_nodes() const noexcept`

Latitudinal Gauss-Legendre nodes.

`template<ball_glq_grid GridType, typename FuncType>`
`inline void generate_values(GridType &&grid, FuncType &&f)`

Generate Gauss-Legendre quadrature grid values from a function.

Template Parameters

- **`GridType`** – type of grid
- **`FuncType`** – type of function

Parameters

- **`grid`** – grid to place the values in
- **`f`** – function to generate values

`template<typename FuncType>`
`inline auto generate_values(FuncType &&f, std::size_t order)`

Generate Gauss-Legendre quadrature grid values from a function.

Template Parameters
`FuncType` – type of function

Parameters
`f` – function to generate values

`template<typename ElementType, typename LayoutType = DefaultLayout>`

`class BallGLQGrid`

Container for gridded data in spherical coordinates in the unit ball.

Template Parameters

- **`ElementType`** – type of elements in the grid
- **`LayoutType`** – grid layout

Public Functions

`inline std::size_t order() const noexcept`

Order of Zernike expansion.

`inline std::array<std::size_t, 3> shape()`

Shape of the grid.

`inline std::span<const element_type> flatten() const noexcept`

Flattened view of the underlying buffer.

`inline std::span<element_type> flatten() noexcept`

Flattened view of the underlying buffer.

`inline void resize(std::size_t order)`

Change the size of the grid.

Public Static Functions

`static inline constexpr std::size_t size(std::size_t order) noexcept`

Number of grid points.

Parameters
`order` – order of Zernike expansion

static inline constexpr std::array<std::size_t, 3> **shape**(std::size_t order) **noexcept**

Shape of the grid.

Parameters
order – order of Zernike expansion

template<**typename ElementType**, **typename LayoutType** = DefaultLayout>

class BallGLQGridSpan : **public** zest::MDSpan<ElementType, 3>

A non-owning view of gridded data in spherical coordinates in the unit ball.

Template Parameters

- **ElementType** – type of elements in the grid
- **LayoutType** – grid layout

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of Zernike expansion.

inline constexpr const std::array<std::size_t, 3> &**shape**() **const noexcept**

Shape of the grid.

inline constexpr std::span<element_type> **flatten**() **const noexcept**

Flattened view of the underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of grid points.

Parameters
order – order of Zernike expansion

static inline constexpr std::array<std::size_t, 3> **shape**(std::size_t order) **noexcept**

Shape of the grid.

Parameters
order – order of Zernike expansion

template<ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**, **typename GridLayoutType** = DefaultLayout>
class GLQTransformer

Class for transforming between a Gauss-Legendre quadrature grid representation and Zernike polynomial expansion representation of data in the unit baal.

Template Parameters

- **zernike_norm_param** – normalization convention of Zernike functions
- **sh_norm_param** – normalization convention of spherical harmonics
- **sh_phase_param** – phase convention of spherical harmonics
- **GridLayoutType** –

Public Functions

inline std::size_t **order**() **const noexcept**

Order of Xernike expansion.

inline void **resize**(std::size_t order)

Resize transformer for specified expansion order.

inline void **forward_transform**(BallGLQGridSpan<**const** double, GridLayout> values, RealZernikeSpan<std::array<double, 2>, zernike_norm_param, sh_norm_param, sh_phase_param> expansion)

Forward transform from Gauss-Legendre quadrature grid to Zernike coefficients.

Parameters

- **values** – values on the ball quadrature grid
- **expansion** – coefficients of the expansion

inline void **backward_transform**(RealZernikeSpan<**const** std::array<double, 2>, zernike_norm_param, sh_norm_param, sh_phase_param> expansion, BallGLQGridSpan<double, GridLayout> values)

Backward transform from Zernike expansion to Gauss-Legendre quadrature grid.

Parameters

- **expansion** – coefficients of the expansion
- **values** – values on the ball quadrature grid

inline RealZernikeExpansion<zernike_norm_param, sh_norm_param, sh_phase_param> **forward_transfo**

Forward transform from Gauss-Legendre quadrature grid to Zernike coefficients.

Parameters

- **values** – values on the ball quadrature grid
- **order** – order of expansion

inline BallGLQGrid<double, GridLayout> **backward_transform**(RealZernikeSpan<**const** std::array<double, 2>, zernike_norm_param, sh_norm_param, sh_phase_param> expansion, std::size_t order)

Backward transform from Zernike coefficients to Gauss-Legendre quadrature grid.

Parameters

- **values** – values on the ball quadrature grid

- **expansion** – coefficients of the expansion

template<ZernikeNorm **zernike_norm_param**, st::SHNorm **sh_norm_param**, st::SHPhase **sh_phase_param**, **typename GridLayoutType** = DefaultLayout>
class ZernikeTransformer

High-level interface for taking Zernike transforms of functions on balls of arbitrary radii.

Template Parameters

- **zernike_norm_param** – normalization convention of Zernike functions
- **sh_norm_param** – normalization convention of spherical harmonics
- **sh_phase_param** – phase convention of spherical harmonics
- **GridLayoutType** –

Public Functions

inline void **resize**(std::size_t order)

Resize the transformer to work with expansions of different order.

template<spherical_function **FuncType**>
inline void **transform**(FuncType &&f, double radius, RealZernikeSpan<std::array<double, 2>, zernike_norm_param, sh_norm_param, sh_phase_param> expansion)

Get Zernike expansion of a function expressed in spherical coordinates.

Template Parameters
FuncType – type of function

Parameters

- **f** – function to transform
- **radius** – radius of the ball f is defined on
- **expansion** – buffer to store the expansion

template<spherical_function **FuncType**>
inline RealZernikeExpansion<zernike_norm_param, sh_norm_param, sh_phase_param> **transform**(FuncType &&f, double radius, std::size_t order)

Get Zernike expansion of a function expressed in spherical coordinates.

Template Parameters
FuncType – type of function

Parameters

- **f** – function to transform
- **radius** – radius of the ball f is defined on
- **order** – order of the expansion

Returns
Zernike expansion

template<cartesian_function **FuncType**>

inline void **transform**(FuncType &&f, double radius, RealZernikeSpan<std::array<double, 2>, zernike_norm_param, sh_norm_param, sh_phase_param> expansion)

Get Zernike expansion of a function expressed in Cartesian coordinates.

Template Parameters
FuncType – type of function

Parameters

- **f** – function to transform
- **radius** – radius of the ball f is defined on
- **expansion** – buffer to store the expansion

template<cartesian_function **FuncType**>
inline RealZernikeExpansion<zernike_norm_param, sh_norm_param, sh_phase_param> **transform**(Func-Type &&f, dou-ble ra-dius, std::or-der)

Get spherical harmonic expansion of a function expressed in Cartesian coordinates.

Template Parameters
FuncType – type of function

Parameters

- **f** – function to transform
- **radius** – radius of the ball f is defined on
- **order** – order of the expansion

Returns
Zernike expansion

Type aliases

using zest::zt::**DefaultLayout** = LonLatRadLayout<>

template<**typename GridLayout** = DefaultLayout>
using zest::zt::**GLQTransformerAcoustics** = GLQTransformer<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::none, GridLayout>

Convenient alias for GLQTransformer with unnormalized Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>
using zest::zt::**GLQTransformerNormalAcoustics** = GLQTransformer<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::none, GridLayout>

Convenient alias for GLQTransformer with orthonorml Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**GLQTransformerQM** = GLQTransformer<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::cs, GridLayout>

Convenient alias for GLQTransformer with unnormalized Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**GLQTransformerNormalQM** = GLQTransformer<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::cs, GridLayout>

Convenient alias for GLQTransformer with orthonormal Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**GLQTransformerGeo** = GLQTransformer<ZernikeNorm::unnormed, st::SHNorm::geo, st::SHPhase::none, GridLayout>

Convenient alias for GLQTransformer with unnormalized Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**GLQTransformerNormalGeo** = GLQTransformer<ZernikeNorm::normed, st::SHNorm::geo, st::SHPhase::none, GridLayout>

Convenient alias for GLQTransformer with orthonormal Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerAcoustics** = ZernikeTransformer<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::none, GridLayout>

Convenient alias for ZernikeTransformer with unnormalized Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerNormalAcoustics** = ZernikeTransformer<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::none, GridLayout>

Convenient alias for ZernikeTransformer with orthonormal Zernike functions, orthonormal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerQM** = ZernikeTransformer<ZernikeNorm::unnormed, st::SHNorm::qm, st::SHPhase::cs, GridLayout>

Convenient alias for ZernikeTransformer with unnormalized Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerNormalQM** = ZernikeTransformer<ZernikeNorm::normed, st::SHNorm::qm, st::SHPhase::cs, GridLayout>

Convenient alias for ZernikeTransformer with orthonormal Zernike functions, orthonormal spherical harmonics, and Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerGeo** = ZernikeTransformer<ZernikeNorm::unnormed, st::SHNorm::geo, st::SHPhase::none, GridLayout>

Convenient alias for ZernikeTransformer with unnormalized Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

template<**typename GridLayout** = DefaultLayout>

using zest::zt::**ZernikeTransformerNormalGeo** = ZernikeTransformer<ZernikeNorm::normed, st::SHNorm::geo, st::SHPhase::none, GridLayout>

Convenient alias for ZernikeTransformer with orthonormal Zernike functions, 4-pi normal spherical harmonics, and no Condon-Shortley phase.

Template Parameters
GridLayout –

0.4.5 Spherical harmonic and Zernike conventions

Enums

enum class zest::st::**SHPhase**

Spherical harmonic phase conventions.

Values:

enumerator none

enumerator cs

enum class zest::st::**SHNorm**

Spherical harmonic normalization conventions.

Values:

enumerator geo

geodesy (4 pi) normalization

enumerator qm

quantum mechanics (unit norm) normalization

enum class zest::zt::**ZernikeNorm**

Zernike polynomial normalizations.

Values:

enumerator normed

enumerator unnormed

0.4.6 Rotations

Enums

enum class zest::**RotationType**

Describes whether rotation applies to object or coordinate system.

Values:

enumerator object

object is rotated

enumerator coordinate

coordinate system is rotated

Types

class Rotor

Rotations of spherical harmonic and Zernike coefficients.

Public Functions

```
template<st::real_sh_expansion ExpansionType>
inline void rotate(ExpansionType &&expansion, const zest::WignerdPiHalfCollection
                   &wigner_d_pi2, const std::array<double, 3> &euler_angles,
                   RotationType type)
```

General rotation of a real spherical harmonic expansion via Wigner's D-matrix.

The rotation uses an intrinsic ZYZ convention, where the first Euler angle rotates about the Z-axis, the second Euler angle rotates about the new Y-axis, and the third angle rotates about the new Z-axis again. In summary, the convention is: right-handed, intrinsic, ZYZ.

Template Parameters

ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real spherical harmonic expansion
- **wigner_d_pi2** – Wigner d-matrices at pi/2
- **euler_angles** – Euler angles defining the rotation
- **type** – type of rotation

```
template<st::row_skipping_real_sh_expansion ExpansionType>
```

`inline void rotate(ExpansionType &&expansion, const zest::WignerdPiHalfCollection &wigner_d_pi2, const std::array<double, 3> &euler_angles, RotationType type)`

General rotation of an even/odd real spherical harmonic expansion via Wigner's D-matrix.

The rotation uses an intrinsic ZYZ convention, where the first Euler angle rotates about the Z-axis, the second Euler angle rotates about the new Y-axis, and the third angle rotates about the new Z-axis again. In summary, the convention is: right-handed, intrinsic, ZYZ.

Template Parameters

ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real spherical harmonic expansion
- **wigner_d_pi2** – Wigner d-matrices at pi/2
- **euler_angles** – Euler angles defining the rotation
- **type** – type of rotation

`template<zt::real_zernike_expansion ExpansionType>`
`inline void rotate(ExpansionType &&expansion, const zest::WignerdPiHalfCollection &wigner_d_pi2, const std::array<double, 3> &euler_angles, RotationType type)`

General rotation of a real Zernike expansion via Wigner's D-matrix.

The rotation uses an intrinsic ZYZ convention, where the first Euler angle rotates about the Z-axis, the second Euler angle rotates about the new Y-axis, and the third angle rotates about the new Z-axis again. In summary, the convention is: right-handed, intrinsic, ZYZ.

Template Parameters

ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real Zernike expansion
- **wigner_d_pi2** – Wigner d-matrices at pi/2
- **euler_angles** – Euler angles defining the rotation
- **type** – type of rotation

`template<st::real_sh_expansion ExpansionType>`
`inline void rotate(ExpansionType &&expansion, const zest::WignerdPiHalfCollection &wigner_d_pi2, const std::array<std::array<double, 3>, 3> &matrix, RotationType type)`

General rotation of a real spherical harmonic expansion via Wigner's D-matrix.

The rotation uses an intrinsic ZYZ convention, where the first Euler angle rotates about the Z-axis, the second Euler angle rotates about the new Y-axis, and the third angle rotates about the new Z-axis again. In summary, the convention is: right-handed, intrinsic, ZYZ.

Template Parameters

ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real spherical harmonic expansion
- **wigner_d_pi2** – Wigner d-matrices at pi/2
- **euler_angles** – Euler angles defining the rotation
- **type** – type of rotation

`template<zt::real_zernike_expansion ExpansionType>`

inline void **rotate**(ExpansionType &&expansion, **const** zest::WignerdPiHalfCollection &wigner_d_pi2, **const** std::array<std::array<double, 3>, 3> &matrix, RotationType type)

General rotation of a real Zernike expansion via Wigner's D-matrix.

The rotation uses an intrinsic ZYZ convention, where the first Euler angle rotates about the Z-axis, the second Euler angle rotates about the new Y-axis, and the third angle rotates about the new Z-axis again. In summary, the convention is: right-handed, intrinsic, ZYZ.

Template Parameters
ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real Zernike expansion
- **wigner_d_pi2** – Wigner d-matrices at pi/2
- **euler_angles** – Euler angles defining the rotation
- **type** – type of rotation

template<st::real_sh_expansion **ExpansionType**>
inline void **polar_rotate**(ExpansionType &&expansion, double angle, RotationType type)

Rotation about the Z-axis of a real spherical harmonic expansion.

Template Parameters
ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real spherical harmonic expansion
- **angle** – polar rotation angle
- **type** – type of rotation

template<zt::real_zernike_expansion **ExpansionType**>
inline void **polar_rotate**(ExpansionType &&expansion, double angle, RotationType type)

Rotation about the Z-axis of a real Zernike expansion.

Template Parameters
ExpansionType – type of expansion to rotate

Parameters

- **expansion** – real spherical harmonic expansion
- **angle** – polar rotation angle
- **type** – type of rotation

class WignerdPiHalfCollection

Collection of Wigner (small) d-matrices at pi/2.

template<**typename ElementType**>

class WignerdSpan

Non-owning view of a Wigner (small) d-matrix at pi/2.

Template Parameters
ElementType – type of elements in the view

Functions

`constexpr std::array<double, 3> zest::euler_angles_from_rotation_matrix(const std::array<std::array<doub 3>, 3> &rot) noexcept`

Translate rotation matrix into corresponding Euler angles.

Parameters
rot – rotation matrix

Returns
Euler angles in order alpha, beta, gamma

0.4.7 Uniform grids

Types

`class GridEvaluator`

Class for evaluating spherical harmonic expansions on arbitrary grids.

Public Functions

`explicit GridEvaluator(std::size_t max_order)`

Reserves memory for an expansion of given order.

Parameters
max_order – maximum order of spherical harmonic expansion.

`GridEvaluator(std::size_t max_order, std::size_t lon_size, std::size_t lat_size)`

Reserves memory for a combination of expansion and grid size.

Parameters

- **max_order** – maximum order of spherical harmonic expansion.
- **lon_size** – size of grid in the longitudinal direction.
- **lat_size** – size of grid in the latittudinal direction.

`void resize(std::size_t max_order, std::size_t lon_size, std::size_t lat_size)`

Resize for a combination of expansion and grid size.

Parameters

- **max_order** – maximum order of spherical harmonic expansion.
- **lon_size** – size of grid in the longitudinal direction.
- **lat_size** – size of grid in the latittudinal direction.

`template<real_sh_expansion ExpansionType>`
`inline std::vector<double> evaluate(ExpansionType &&expansion, std::span<const double> longitudes, std::span<const double> colatitudes)`

Evaluate spherical harmonic expansion on a grid.

Parameters

- **expansion** – spherical harmonics expansion.
- **longitudes** – longitude values defining the grid points.
- **colatitudes** – colatitude values defining the grid points.

Returns

`std::vector` containing values of the expansion on the grid. The values are ordered as a 2D array with shape `{longitudes.size(), colatitudes.size()}` in row-major order.

class GridEvaluator

Public Functions

explicit GridEvaluator(std::size_t max_order)

Reserves memory for an expansion of given order.

Parameters

max_order – maximum order of spherical harmonic expansion.

GridEvaluator(std::size_t max_order, std::size_t lon_size, std::size_t lat_size, std::size_t rad_size)

Reserves memory for a combination of expansion and grid size.

Parameters

- **lon_size** – size of grid in the longitudinal direction.
- **max_order** – maximum order of spherical harmonic expansion.
- **lat_size** – size of grid in the latittudinal direction.
- **rad_size** – size of grid in the radial direction.

void **resize**(std::size_t max_order, std::size_t lon_size, std::size_t lat_size, std::size_t rad_size)

Resize for a combination of expansion and grid size.

Parameters

- **max_order** – maximum order of spherical harmonic expansion.
- **lon_size** – size of grid in the longitudinal direction.
- **lat_size** – size of grid in the latittudinal direction.
- **rad_size** – size of grid in the radial direction.

template<real_zernike_expansion **ExpansionType**>
inline std::vector<double> **evaluate**(ExpansionType &&expansion, std::span<**const** double> longitudes, std::span<**const** double> colatitudes, std::span<**const** double> radii)

Evaluate spherical harmonic expansion on a grid.

Parameters

- **expansion** – spherical harmonics expansion.
- **longitudes** – longitude values defining the grid points.
- **colatitudes** – colatitude values defining the grid points.
- **radii** – radius values defining the grid points.

Returns

`std::vector` containing values of the expansion on the grid. The values are ordered as a 3D array with shape `{longitudes.size(), colatitudes.size(), radii.size()}` in row-major order.

0.4.8 Power spectra

Functions

namespace zest

namespace st

Functions

```
template<st::real_sh_expansion ExpansionType>
void cross_power_spectrum(ExpansionType &&a, ExpansionType &&b,
                          std::span<double> out) noexcept
```

Compute cross power spectrum of two spherical harmonic expansions.

Parameters

- **a** – spherical harmonic expansion
- **b** – spherical harmonic expansion
- **out** – output buffer for the cross power spectrum

```
template<st::real_sh_expansion ExpansionType>
std::vector<double> cross_power_spectrum(ExpansionType &&a, ExpansionType
                                         &&b)
```

Compute cross power spectrum of two spherical harmonic expansions.

Parameters

- **a** – spherical harmonic expansions
- **b** – spherical harmonic expansions

Returns

`std::vector` storing the the cross power spectrum

```
template<st::real_sh_expansion ExpansionType>
void power_spectrum(ExpansionType &&expansion, std::span<double> out)
                    noexcept
```

Compute power spectrum of a spherical harmonic expansions.

Parameters

- **expansion** – spherical harmonic expansion
- **out** – output buffer for the power spectrum

```
template<st::real_sh_expansion ExpansionType>
std::vector<double> power_spectrum(ExpansionType &&expansion)
```

Compute power spectrum of a spherical harmonic expansions.

Parameters

expansion – spherical harmonic expansion

Returns

`std::vector` storing the power spectrum

namespace zt

Functions

```
template<zt::real_zernike_expansion ExpansionType>
void power_spectrum(ExpansionType &&expansion, RadialZernikeSpan<double,
                    ExpansionType::zernike_norm> out) noexcept
```

Compute power spectrum of a Zernike expansion.

Parameters

- **expansion** – Zernike expansion.
- **out** – place to store the power spectrum.

```
template<zt::real_zernike_expansion ExpansionType>
```

`std::vector<double> power_spectrum(ExpansionType &&expansion)`

Compute power spectrum of a Zernike expansions.

Parameters

expansion – Zernike expansion

Returns

`std::vector` storing the power spectrum.

0.4.9 Layouts

enum class zest::IndexingMode

Enum for tagging the `m` indexing style for spherical harmonics related things.

Values:

enumerator negative

enumerator nonnegative

enum class zest::Parity

Values:

enumerator even

enumerator odd

enum class zest::LayoutTag

Tag for easily differentiating various layouts.

Values:

enumerator linear

enumerator triangular

enumerator tetrahedral

Concepts

template<typename T>
concept one_dimensional_span

template<typename T>
concept two_dimensional_span

template<typename T>
concept two_dimensional_subspannable

template<typename T>
concept layout_2d

template<typename T>
concept triangular_layout

Types

template<IndexingMode **indexing_mode_param**>

struct StandardLinearLayout

Contiguous 1d layout, which is indexed exactly as you think it is.

```
0 1 2 3 4 5...
```

Template Parameters

indexing_mode_param – determines whether indexing may be negative

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of elements in layout for size parameter `order`.

Parameters

order – parameter presenting the size of the layout

static inline constexpr std::size_t **idx**(index_type l) **noexcept**

Linear index of an element in layout.

struct ParityLinearLayout

Contiguous 1d layout, with indexing according to certain parity.

```
0 2 4 6 8...
```

or

```
1 3 5 7 9...
```

> **Warning**
>
> This indexing implies that adjacent even and odd indices map to the same memory slot. Indexing data with this layout mixing even and odd indices is an error.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of elements in layout for size parameter `order`.

Parameters

order – parameter presenting the size of the layout

static inline constexpr std::size_t **idx**(index_type l) **noexcept**

Linear index of an element in layout.

template<IndexingMode **indexing_mode_param**>

struct TriangleLayout

Contiguous 2D layout with indexing.

```
(0,0)
(1,0) (1,1)
(2,0) (2,1) (2,2)
(3,0) (3,1) (3,2) (3,3)
...
```

or

```
                    (0,0)
             (1,-1) (1,0) (1,1)
      (2,-2) (2,-1) (2,0) (2,1) (2,2)
(3,-3) (3,-2) (3,-1) (3,0) (3,1) (3,2) (3,3)
...
```

Template Parameters
`indexing_mode_param` – determines whether indexing may be negative

Public Static Functions

`static inline constexpr std::size_t size(std::size_t order) noexcept`

Number of elements in layout for size parameter `order`.

Parameters
`order` – parameter presenting the size of the layout

`static inline constexpr std::size_t idx(index_type l, index_type m) noexcept`

Linear index of an element in layout.

`struct OddDiagonalSkippingTriangleLayout`

Contiguous 2D layout with indexing.

```
(0,0)
      (1,1)
(2,0)       (2,2)
      (3,1)       (3,3)
(4,0)       (4,2)       (4,4)
...
```

> **Warning**
>
> This indexing implies that some index combinations are simply not valid. It is erroneous to access data using this layout with indices whose sum is an odd number.

Public Static Functions

`static inline constexpr std::size_t size(std::size_t order) noexcept`

Number of elements in layout for size parameter `order`.

Parameters
`order` – parameter presenting the size of the layout

`static inline constexpr std::size_t idx(std::size_t n, std::size_t l) noexcept`

Linear index of an element in layout.

`template<IndexingMode indexing_mode_param>`

`struct RowSkippingTriangleLayout`

Contiguous 2D layout with indexing.

```
(0,0)

(2,0) (2,1) (2,2)

```

(continues on next page)

(continued from previous page)

```
(4,0) (4,1) (4,2) (4,3) (4,4)
...
```

or

```
(1,0) (1,1)

(3,0) (3,1) (3,2) (3,3)

(5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
...
```

or alternatively

```
                            (0,0)

              (2,-2) (2,-1) (2,0) (2,1) (2,2)

(4,-4) (4,-3) (4,-2) (4,-1) (4,0) (4,1) (4,2) (4,3) (4,4)
...
```

or

```
                            (1,-1) (1,0) (1,1)

              (3,-3) (3,-2) (3,-1) (3,0) (3,1) (3,2) (3,3)

(5,-5) (5,-4) (5,-3) (5,-2) (5,-1) (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
...
```

Note

In this layout the index obtained from a pair `(l,m)` is unique only for `l` of the same parity. Otherwise the index is not unique, e.g., `(0,0)` and `(1,0)` fall on the same index.

Template Parameters

`indexing_mode_param` – determines whether indexing may be negative

Public Static Functions

`static inline constexpr` `std::size_t` **`size`**`(std::size_t order)` **`noexcept`**

Number of elements in layout for size parameter `order`.

Parameters

`order` – parameter presenting the size of the layout

`static inline constexpr` `std::size_t` **`idx`**`(index_type l, index_type m)` **`noexcept`**

Linear index of an element in layout.

`template``<IndexingMode` **`indexing_mode_param`**`>`

`struct ZernikeTetrahedralLayout`

Contiguous 3D layout with indexing.

```
(0,0,0)

```

(continues on next page)

(continued from previous page)

```
(1,1,0) (1,1,1)

(2,0,0)
(2,2,0) (2,2,1) (2,2,2)
...
```

Template Parameters
indexing_mode_param – determines whether indexing may be negative

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of elements in layout for size parameter `order`.

Parameters
order – parameter presenting the size of the layout

static inline constexpr std::size_t **idx**(index_type n, index_type l, index_type m) **noexcept**

Linear index of an element in layout.

template<**typename** **ElementType**, **typename** **LayoutType**>

class LinearSpan

A non-owning one-dimensional view of data elements.

Template Parameters

- **ElementType** – type of data elements
- **LayoutType** – type identifying the data layout

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of data elements for size parameter `order`.

Parameters
order – parameter presenting the size of the span

template<**typename** **ElementType**, **typename** **LayoutType**>

class LinearVecSpan

A non-owning one-dimensional view of one-dimensional segments of data.

Template Parameters

- **ElementType** – type of data elements
- **LayoutType** – type identifying the data layout

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **vec_size**() **const noexcept**

Size of a data segment.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order, std::size_t vec_size) **noexcept**

Number of data elements for size parameter `order`.

Parameters

- **order** – parameter presenting the size of the span
- **vec_size** – number of elements in a single data segment

template<**typename ElementType**>

class ParitySpan

A non-owning view where adjacent even and odd indices refer to the same value. Given index `i`, the corresponding offset in the underlying buffer is given by `i/2`.

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

template<**typename ElementType**, **typename LayoutType**>

class TriangleSpan

A non-owning two-dimensional view of data elements with triangular layout.

Template Parameters

- **ElementType** – type of data elements
- **LayoutType** – type identifying the data layout

Subclassed by *zest::st::SHLMSpan< ElementType, LayoutType, sh_norm_param, sh_phase_param >*, *zest::zt::ZernikeSHSpan< ElementType, LayoutType, zernike_norm_param, sh_norm_param, sh_phase_param >*

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of data elements for size parameter `order`.

Parameters

order – parameter presenting the size of the span

template<**typename ElementType**, **typename LayoutType**>

class TriangleVecSpan

A non-owning two-dimensional view of one-dimensional segments of data with triangular layout.

Template Parameters

- **ElementType** – type of data elements
- **LayoutType** – type identifying the data layout

Subclassed by *zest::st::SHLMVecSpan< ElementType, LayoutType, sh_norm_param, sh_phase_param >*

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **vec_size**() **const noexcept**

Size of a data segment.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order, std::size_t vec_size) **noexcept**

Number of data elements for size parameter `order`.

Parameters

- **order** – parameter presenting the size of the span
- **vec_size** – number of elements in a single data segment

template<**typename ElementType**, **typename LayoutType**>

class TetrahedronSpan

A non-owning three-dimensional view of data elements with triangular layout.

Template Parameters

- **ElementType** – type of elements in the view
- **LayoutType** – layout of the elements

Subclassed by *zest::zt::ZernikeNLMSpan< ElementType, LayoutType, zernike_norm_param, sh_norm_param, sh_phase_param >*

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order) **noexcept**

Number of data elements for size parameter `order`.

Parameters

order – parameter presenting the size of the span

template<**typename ElementType**, **typename LayoutType**>

class TetrahedronVecSpan

A non-owning three-dimensional view of data elements with triangular layout.

Template Parameters

- **ElementType** – type of elements in the view
- **LayoutType** – layout of the elements

Public Functions

inline constexpr std::size_t **order**() **const noexcept**

Order of data layout.

inline constexpr std::size_t **size**() **const noexcept**

Size of the underlying buffer.

inline constexpr element_type ***data**() **const noexcept**

Pointer to underlying buffer.

Public Static Functions

static inline constexpr std::size_t **size**(std::size_t order, std::size_t vec_size) **noexcept**

Number of data elements for size parameter `order`.

Parameters

- **order** – parameter presenting the size of the span
- **vec_size** – number of elements in a single data segment

0.4.10 Indexing

Types

template<std::integral **IndexType**, IndexType **stride_param**>

class IndexIterator

Iterator presenting an infinite arithmetic sequence of integer indices with arbitrary stride.

Template Parameters

- **IndexType** – type of the index
- **stride_param** – stride of the index

Public Functions

inline constexpr IndexIterator &**operator++**() **noexcept**

Increment index by stride.

inline constexpr IndexIterator &**operator--**() **noexcept**

Decrement index by stride.

inline constexpr IndexIterator **operator++**(int) **noexcept**

Increment index by stride.

inline constexpr IndexIterator **operator--**(int) **noexcept**

Decrement index by stride.

inline constexpr IndexIterator &**operator+=**(index_type n) **noexcept**

Increment index by multiple strides.

inline constexpr IndexIterator &**operator-=**(index_type n) **noexcept**

Decrement index by multiple strides.

inline constexpr IndexIterator **operator+**(difference_type n) **const noexcept**

Add n strides to index.

inline constexpr IndexIterator **operator-**(difference_type n) **const noexcept**

Subtract n strides from index.

inline constexpr index_type **operator***() **noexcept**

Get value of index.

inline constexpr index_type **operator[]**(index_type n) **noexcept**

Get value of index n strides forward from current index.

inline constexpr index_type **index**() **const noexcept**

Get value of index.

template<std::integral **IndexType**>

class StandardIndexRange

Range of integer indices.

Template Parameters

IndexType – type of the index

Public Functions

inline explicit constexpr StandardIndexRange(index_type end)

Constructs a range of indices `[0, end)`.

Parameters

end – end of index range

inline constexpr StandardIndexRange(index_type begin, index_type end)

Constructs a range of indices `[begin, end)`.

Parameters

- **begin** – start of index range
- **end** – end of index range

inline constexpr iterator **begin**() **const noexcept**

Iterator to the beginning of the range.

inline constexpr iterator **end**() **const noexcept**

Iterator to the end of the range.

template<std::integral **IndexType**>

class ParityIndexRange

Range of even or odd integer indices.

Template Parameters
IndexType – type of the index

Public Functions

inline explicit constexpr ParityIndexRange(index_type end)

Constructs a range of indices `[end % 2, end)`.

Parameters
end – end of index range

inline constexpr ParityIndexRange(index_type begin, index_type end)

Constructs a range of indices `[2*floor(begin/2) + end % 2, end)`.

Parameters

- **begin** – start of index range
- **end** – end of index range

inline constexpr iterator **begin**() **const noexcept**

Iterator to the beginning of the range.

inline constexpr iterator **end**() **const noexcept**

Iterator to the end of the range.

template<std::signed_integral **IndexType**>

class SymmetricIndexRange

Range of integer indices symmetric about zero.

Template Parameters
IndexType – type of the index

Public Functions

inline explicit constexpr SymmetricIndexRange(index_type end)

Constructs a range of indices `(-end, end)`.

Parameters
end – end of index range

inline constexpr SymmetricIndexRange(index_type begin, index_type end)

Constructs a range of indices `[begin, end)`.

Parameters

- **begin** – start of index range
- **end** – end of index range

inline constexpr iterator **begin**() **const noexcept**

Iterator to the beginning of the range.

inline constexpr iterator **end**() **const noexcept**

Iterator to the end of the range.

0.4.11 Multidimensional arrays

Types

template<**typename ElementType**, std::size_t **rank_param**>

class MDArray

Multidimensional array container.

Template Parameters

- **ElementType** – type of array elements
- **rank_param** – number of array dimensions

template<**typename ElementType**, std::size_t **rank_param**>

class MDSpan

Poor man's mdspan for a non-owning multidimensional array view.

Template Parameters

- **ElementType** – type of array elements
- **rank_param** – number of array dimensions

0.4.12 Gauss–Legendre quadrature

Enums

enum class zest::gl::**GLNodeStyle**

Style of Gauss-Legendre nodes.

Values:

enumerator angle

nodes as angles in the interval [0,pi]

enumerator cos

nodes as consines of the angles in the interval [-1,1]

Concepts

template<**typename T**>
concept gl_layout

#include <gauss_legendre.hpp> Concept for restricting layout of Gauss-Legendre nodes.

Types

struct PackedLayout

Packed layout of Gauss-Legendre nodes.

note Gauss-Legendre nodes on the interval [-1,1] are distributed symmetrically about 0, such that for any node `x` the point `-x` is also a node with the same weight. Therefore the nodes and weights only need to be produced for nonnegative `x`. For the negative portion of the interval the nodes are `-x`, and the weights are given by the corresponding weights.

`struct UnpackedLayout`

Unpacked layout of Gauss-Legendre nodes.

note Gauss-Legendre nodes on the interval [-1,1] are distributed symmetrically about 0, such that for any node `x` the point `-x` is also a node with the same weight. Therefore the nodes and weights only need to be produced for nonnegative `x`. For the negative portion of the interval the nodes are `-x`, and the weights are given by the corresponding weights.

Functions

```
template<gl_layout Layout, GLNodeStyle node_style_param,
std::ranges::random_access_range R>
constexpr void zest::gl::gl_nodes(R &&nodes, std::size_t parity) noexcept
```

Obtain Gauss-Legendre nodes for a given number of nodes.

For `num_nodes < 70` the nodes are read from a precomputed table. For greater numbers of nodes Bogaert's iteration-free method is used: I. Bogaert, Iteration-free computation of Gauss-Legendre quadrature nodes and weights, SIAM J. Sci. Comput., 36 (2014), pp. C1008-C1026).

The nodes returned are accurate to double macine epsilon.

Template Parameters

- **`Layout`** – layout of `nodes`
- **`R`** – type of the range for storing the nodes

Parameters

- **`nodes`** – range for storing the nodes
- **`parity`** – parity of the total number of nodes

```
template<gl_layout Layout, std::ranges::random_access_range R>
constexpr void zest::gl::gl_weights(R &&weights, std::size_t parity) noexcept
```

Obtain Gauss-Legendre weights for a given number of nodes.

For `num_nodes < 70` the nodes are read from a precomputed table. For greater numbers of nodes Bogaert's iteration-free method is used: I. Bogaert, Iteration-free computation of Gauss-Legendre quadrature nodes and weights, SIAM J. Sci. Comput., 36 (2014), pp. C1008-C1026).

The weights returned are accurate to double macine epsilon.

Template Parameters

- **`Layout`** – layout of `weights`
- **`R`** – type of the range for storing the weights

Parameters

- **`weights`** – range for storing the weights
- **`parity`** – parity of the total number of nodes

```
template<gl_layout Layout, GLNodeStyle node_style_param,
std::ranges::random_access_range R>
constexpr void zest::gl::gl_nodes_and_weights(R &&nodes, R &&weights, std::size_t
                                              parity) noexcept
```

Obtain Gauss-Legendre nodes and weights for a given number of nodes.

For `num_nodes < 70` the nodes are read from a precomputed table. For greater numbers of nodes Bogaert's iteration-free method is used: I. Bogaert, Iteration-free computation of Gauss-Legendre quadrature nodes and weights, SIAM J. Sci. Comput., 36 (2014), pp. C1008-C1026).

The nodes and weights returned are accurate to double macine epsilon.

Template Parameters

- **`Layout`** – layout of `nodes` and `weights`
- **`R`** – type of the range for storing the nodes and weights

Parameters

- **`nodes`** – range for storing the nodes
- **`weights`** – range for storing the weights
- **`parity`** – parity of the total number of nodes

0.4.13 Memory

Concepts

`template<typename T>`
`concept valid_simd_alignment`

Types

`template<typename T, valid_simd_alignment Alignment>`

`struct AlignedAllocator`

Aligned memory allocator class.

Template Parameters

- **`T`** – type of allocated object
- **`BYTE_ALIGNMENT`** – number of bytes to align to

Public Functions

`inline T *allocate(std::size_t n)`

Allocate an aligned block of memory that fits n values of `value_type`.

`inline void deallocate(T *p, std::size_t n) noexcept`

Free allocated memory.

`template<typename U>`

`struct rebind`

`template<std::size_t byte_alignment>`

`struct VectorAlignment`

Alignmnet descriptor for SIMD vector alignment.

> **Note**
>
> `byte_alignment` must be a power of two.

Template Parameters

`byte_alignment` – number of bytes to align to

Public Static Functions

template<**typename T**>
static inline constexpr std::size_t **vector_size**() **noexcept**

Number of elements that fit in a SIMD vector of given type.

Template Parameters
T – type of elements

struct NoAlignment

Alignmnet descriptor which doesn't specify any alignment.

Public Static Functions

template<**typename T**>
static inline constexpr std::size_t **vector_size**() **noexcept**

Number of elements that fit in a SIMD vector of given type.

Template Parameters
T – type of elements

Type aliases

using zest::**SSEAlignment** = VectorAlignment<16>

Alias for SSE (16 byte) alignment descriptor.

using zest::**AVXAlignment** = VectorAlignment<32>

Alias for AVX (32 byte) alignment descriptor.

using zest::**AVX512Alignment** = VectorAlignment<64>

Alias for AVX512 (64 byte) alignment descriptor.

using zest::**CacheLineAlignment** = VectorAlignment<64>

Alias for cache line (64 byte) alignment descriptor.

Functions

template<**typename T**, valid_simd_alignment **Alignment**>
constexpr std::size_t zest::**aligned_size**(std::size_t n) **noexcept**

Figure out the number of bytes needed to store a number of elements with given byte alignment.

Template Parameters

- **T** – type of allocated object
- **BYTE_ALIGNMENT** – number of bytes to align to

Parameters
n – number of elements

Returns
number of bytes

INDEX

Z