ZebraDM

Zernike-based Radon transforms

Sebastian Sassi

Version 0.1.0

Apr 29, 2025

CONTENTS:

0.1 Getting started

0.1.1 Installation

Before we can start with the installation of the library, we need to take care of its dependencies. There are two:

- zest (REQUIRED) is a companion library, which provides utilities for performing Zernike and spherical harmonic transforms.
- cubage (OPTIONAL) provides capabilities for multidimensional numerical integration. This library is only used by the numerical integration implementation of the Radon transforms, which are implemented for comparison. If you are not planning on building the included benchmarks, you likely have no need for this.

For all practical purposes, zest is the only dependency you need to care about. Its installation process is straightforward and similar to this library.

After you have installed zest, installation of this library proceeds similarly. To obtain the source, clone the repository

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

If you are familiar with CMake, ZebraDM follows a conventional CMake build/install procedure. Even if not, the process is straightforward. First, build the project

```
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 your 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 isotropic angle integrated transverse and nontransverse Radon transform for a distribution. To do this, crate a file `radon.cpp` with the contents

```
#include "zest/zernike_glq_transformer.hpp"
#include "zebradm/zebra_angle_integrator.hpp"

int main()
{
    auto shm_dist = [](const Vector<double, 3>& v){
        constexpr double disp_sq = 0.4*0.4;
        const double speed_sq = dot(v,v);
        return std::exp(-speed_sq/disp_sq);
    }

    constexpr std::size_t order = 20;
    constexpr double vmax = 1.0;
    zest::zt::ZernikeExpansion dist_expansion
        = zest::zt::ZernikeTransformerOrthoGeo{}.transform(shm_dist, vmax, order);

    std::vector<std::array<double, 3>> vlab = {
        {0.5, 0.5, 0.0}, {0.5, 0.0, 0.5}, {0.0, 0.5, 0.5}
```

(continues on next page)

(continued from previous page)

```
    };

    std::vector<double> vmin = {0.2, 0.3, 0.4};

    std::vector<std::array<double, 2>> out_buffer(vlab.size()*vmin.size());
    zest::MDSpan<std::array<double, 2>, 2> out(
            out_buffer.data(), {vlab.size(), vmin.size()});

    zebra::IsotropicTransverseAngleIntegrator(order)
        .integrate(dist_expansion, vlab, vmin, out);

    for (std::size_t i = 0; i < 0; ++i)
    {
        const double nontransverse = out[i][j][0];
        const double transverse = out[i][j][1];
        for (std::size_t j = 0; j < 0; ++j)
            std::printf("{%f, %f} ", nontransverse, transverse);
        std::printf("\n");
    }
}
```

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

```
g++ -std=c++20 -O3 -march=native -o radon radon.cpp -lzebradm -lzest
```

There are a 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, apart from linking with this library, don't forget to link with the dependencies. In this case, zest.

Finally, 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 enables architecture specific optimizations with the `-march=native flag`. This is generally advisable if your code will be running on the same machine it is built on. However, the situation is different if you expect to be running the same executable on machines with potentially different architectures. For typical x86, fused multiply-add operations `-mfma` and AVX2 SIMD operations `-mavx2`, should be available on most hardware and are sufficient for near optimal performance.

0.2 Theory – Zernike-based Radon transforms

The motivation and theory of the Zernike-based Radon transforms is described in detail in the article arxiv:2504.19714, which introduces this method. This sections aims to give a brief introduction to the topic and the methods emplyed by this library.

In short, ZebraDM is a library that exists primarily for fast evaluation of integrals of the form

$$\overline{\mathscr{R}}[f](w,\vec{x}_0) = \int_{S^2} S(w,\hat{n}) \int_B f(\vec{x}+\vec{x}_0)\delta(\vec{x}\cdot\hat{n}-w)\, d^3x\, d\Omega,$$

and also

$$\overline{\mathscr{R}^{\perp}}[f](w,\vec{x}_0) = \int_{S^2} S(w,\hat{n}) \int_B (x^2-(\vec{x}\cdot\hat{n})^2) f(\vec{x}+\vec{x}_0)\delta(\vec{x}\cdot\hat{n}-w)\, d^3x\, d\Omega.$$

The inner integral is over the unit ball

$$B = \{\vec{x}\in\mathbb{R}^3 \mid \|\vec{x}\| \le 1\},$$

and the outer integral is over the sphere S^2.

These integrals are, in the context of this library called the *angle-integrated Radon* and *transverse Radon transform*, respectively. This naming is so because the inner integral of the first equation is precisely the conventional three-dimensional Radon transform of the function $f(\vec{x} + \vec{x}_0)$. The parameter $\vec{x}_0$ here is an arbitrary offset applied on the distribution. The functions $f(\vec{x})$ and $S(w, \hat{n})$ may, in general, be defined in coordinate systems that differ by an arbitrary rotation, although this library currently only implements solutions where they differ by a rotation about the z-axis.

As described in more detail in the article, intergals of this form occur in computation of expected dark matter event rates in dark matter direct detection experiments. In this context $f(\vec{x})$ represents the dark matter velocity distribution with $\vec{x}$ effectively corresponding to the dark matter velocity in the laboratory frame, and $\vec{x}_0$ representing the velocity of the laboratory frame relative to the average motion of the dark matter. The unit vector $\hat{n}$ corresponds to the direction of momentum transfer in dark matter scattering, and w corresponds to an energy parameter whose definition depends on the context. The function $S(w, \hat{n})$, in turn, corresponds to a detector response function.

0.2.1 Radon transforms

The Radon transform is a map, which maps a function f defined on $\mathbb{R}^n$ onto the space of $(n-1)$-dimensional hyperplanes on $\mathbb{R}^n$. In 3D, this means mapping the function f onto the space of planes on $\mathbb{R}^3$. The Radon transform in this case is given by the integral formula

$$\mathscr{R}[f](w, \hat{n}) = \int_{\mathbb{R}^3} \delta(\vec{x} \cdot \hat{n} - w) f(\vec{x})\, d^3x,$$

where δ is the Dirac delta-function. The delta-function forces the integration to be over a plane whose normal vector is $\hat{n}$, and whose distance from the origin is w. These two parameters together uniquely define a plane in $\mathbb{R}^3$.

The function f doesn't have to be supported on entirity of $\mathbb{R}^3$. If f is nonzero only on some measurable subset $U \subset \mathbb{R}^3$, the we may define

$$\mathscr{R}[f](w, \hat{n}) = \int_{U} \delta(\vec{x} \cdot \hat{n} - w) f(\vec{x})\, d^3x.$$

This library, in particular, assumes that f is zero outside the unit ball. For practical purposes in terms of computation with finite precision floating point numbers, this is not much of a restriction at all. For example, any square-integrable function on $\mathbb{R}^3$ can be clamped to zero beyond some finite radius R such that its Radon transform will remain the same up to some precision. Then the coordinates can be rescaled such that $R \to 1$. Most numerical problems in $\mathbb{R}^3$ can therefore be restricted to a ball without losses, and in terms of functions supported on a ball, the rescaling lets us assume a unit ball without loss of generality.

Since this library deals with angle-integrated Radon transforms, i.e., interals of the Radon transform over the unit vectors $\hat{n}$, it is useful to think of the planes as the tangent planes of spherical shells of radius w at the points $w\hat{n}$. Then the angle-integrated Radon transform maps f onto these spherical shells parametrized by w. Due to this identification, the parameter w is known as the *shell* parameter, or `shell` for short in the code.

An important property of Radon transforms is that if we use f to define a new offset function $f_{\vec{x}_0}(\vec{x}) = f(\vec{x}+\vec{x}_0)$, then

$$\mathscr{R}[f_{\vec{x}_0}](w, \hat{n}) = \mathscr{R}[f](w + \vec{x}_0 \cdot \hat{n}, \hat{n}).$$

Therefore handling Radon transforms of functions with arbitrary offsets is straightforward to deal with.

It is worth noting that if f is zero outside of the unit ball, then the Radon transform is zero for $w > 1$, because none of the planes tangent to the outer shells intersect the unit ball. If we take into account the arbitrary offset $\vec{x}_0$, then this implies that the Radon transform is zero for $|w + \vec{x}_0 \cdot \hat{n}| > 1$. This means that the angle-integrated Radon transform is nonzero only for the shell parameters $w \leq 1 + x_0$, where x_0 is the length of $\vec{x}_0$.

0.2.2 Zernike expansions

The core idea behind the methods of this library is the expansion of the function $f(\vec{x})$ in a basis of so-called Zernike functions $Z_{nlm}(\vec{x})$, which are orthogonal on the unit ball. Thus

$$f(\vec{x}) = \sum_{nlm} f_{nlm} Z_{nlm}(\vec{x}).$$

Transformations of Zernike functions into the expansion coefficients are implemented in a companion library zest. Zernike functions are described in further detail in the documentation of zest. For purposes of this library, their most important property is that their Radon transform has a closed-form expression

$$\mathscr{R}[Z_{nlm}](w, \hat{n}) = \frac{2\pi}{(n+1)(n+2)}(1-w^2)C_n^{3/2}(w)Y_{lm}(\hat{n}).$$

Here $C_n^{3/2}(w)$ are so-called Gegenbauer polynomials. It is possible to express the term $(1-w^2)C_n^{3/2}(w)$ as a linear combination of two Legendre polynomials. Given this, the Radon transform of the shifted function $f(\vec{x}+\vec{x}_0)$ is

$$\mathscr{R}[f](w, \hat{n}) = 2\pi \sum_{nlm} f'_{nlm} P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n}).$$

where

$$f'_{nlm} = \frac{1}{2n+3} f_{nlm} - \frac{1}{2n-1} f_{n-2,lm},$$

where the second term is neglected for $n = 0, 1$ and $f_{nlm} = 0$ for $n > L$ and $l > n$. This formula for f'_{nlm} essentially defines the Radon transform in the Zernike coefficient space.

0.2.3 Angle integrals

Although the Radon transform formula for the Zernike coefficients is useful by itself and is also implemented in this library, a majority of this library is focused on computing integrals of the Radon transform (potentially multiplied by a response function $S(w, \hat{n})$) over the directions $\hat{n}$. To this end, it is important to notice that all dependence on $\hat{n}$ in the Radon transform is in the basis functions

$$P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n}).$$

A mild complication is the potential presence of the response function $S(w, \hat{n})$. This is dealt with via the observation that we have a collection of functions

$$f'_n(\hat{n}) = \sum_{lm} f'_{nlm} Y_{lm}(\hat{n}).$$

These functions can be multiplied by $S(w, \hat{n})$. However, as mentioned above, $f'_n(\hat{n})$ and $S(w, \hat{n})$ may be defined in coordinate systems differing by a rotation. Therefore, in practice, they first need to be rotated to a matching coordinate system. In any case, defining

$$f_n^S(w, \hat{n}) = f_n^{(R)}(\hat{n}) S^{(R')}(w, \hat{n}),$$

where R and R' denote rotations applied on the functions, we end up back at

$$S(w, \hat{n})\mathscr{R}[f](w, \hat{n}) = 2\pi \sum_{nlm} f^S_{nlm} P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n}).$$

The outcome is therefore that we only ever need to integrate

$$\int P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n})\, d\Omega.$$

This integral simplifies if the integration coordinates can be chosen such that the z-axis is in the direction of $\vec{x}_0$, which requires rotation of the coefficients f^S_{nlm}. With that the problem reduces to the evaluation of

$$A_{nl}(w, x_0) = \int_{-1}^{1} P_n(w + x_0 z) P_l(z)\, dz,$$

such that

$$\overline{\mathscr{R}}[f](w, \vec{x}_0) = 2\pi \sum_{nlm} f_{nlm}^{S;R''} A_{nl}(w, x_0).$$

Here R'' denotes the rotation to the integration coordinates.

Integration of the transverse Radon transform proceeeds much in the same way, except there the additional term $x^2 - (\vec{x} \cdot \hat{n})^2$ needs to be dealt with. The problem can be reduced to integration of multiple conventional Radon transforms by means of some recursion relations of Zernike functions, after which the computation proceeds much in the same way as discussed above.

A notable fact is that in the decomposition of the transverse Radon transform to conventional Radon transforms, one of them happens to be the just $\mathscr{R}[f]$. Therefore, evaluation of the transverse Radon transform of $f(\vec{x})$ always gives the nontransverse Radon transform for free. The library takes advantage of this fact, and so methods that evaluate the angle-integrated transverse Radon transform always return both the nontransverse, and the transverse result.

0.3 Usage – Integrating a Radon transform

This section goes through the core parts of the library and its usage via an example of evaluating the angle-integrated Radon transform of a distribution with a response function.

The angle integration is implemented in four core classes:

- `zdm::zebra::IsotropicAngleIntegrator`
- `zdm::zebra::AnisotropicAngleIntegrator`
- `zdm::zebra::IsotropicTransverseAngleIntegrator`
- `zdm::zebra::AnisotropicTransverseAngleIntegrator`

The classes with `Transverse` compute the angle-integrated transverse Radon transform in addition to the nontransverse case, whereas the others only compute the nontransverse case. The classes marked with `Anisotropic` are used when we have an anisotropic response function, whereas the `Isotropic` classes are for the special case where the response function is istropic (or, alternatively, when we have no response function).

We will mainly consider `zdm::zebra::AnisotropicAngleIntegrator` here. The isotropic integrators are simpler, because they don't need to deal with the presence of a response function, and the transverse integrators in turn essentially only differ by the fact that they return two numbers where the nontransverse integrators return one.

The first order of business is to initialize our integrator

```cpp
#include <zebradm/zebra_angle_integrator.hpp>

int main()
{
    constexpr std::size_t dist_order = 30;
    constexpr std::size_t resp_order = 60;

    // ...

    zdm::zebra::AnisotropicAngleIntegrator integrator(dist_order, resp_order);

    // ...
}
```

All the integrators are located in the header `zebradm/zebra_angle_integrator.hpp`. The parameters `dist_order` and `resp_order` are the orders of the Zernike and spherical harmonic expansions of the distribution and response functions, respectively. One can also default initialize the integrator

```
zdm::zebra::AnisotropicAngleIntegrator integrator{};
```

Initializing with the order parameter preallocates some buffers. However, when the integrator is used, it will also read these paramters off the expansions it is given, and adjust its buffers accordingly, so whether to use default initialization or not is a matter of preference.

In order to use the integrator, we will need to prepare the data needed to compute the Radon transform. First and foremost, we need a Zernike expansion representing our distribution function. Typically, the Zernike expansion is computed from some mathematical expression for the distribution, so we will define one

```
#include <cmath>
#include <array>

constexpr std::array<double, 3> var = {0.5, 0.6, 0.7};
auto dist_func = [&](double lon, double colat, double r)
{
    const std::array<double, 3> x = {
        r*std::sin(colat)*std::cos(lon),
        r*std::sin(colat)*std::sin(lon),
        r*std::cos(colat)
    };

    return std::exp(-(x[0]*x[0]/var[0]) + x[1]*x[1]/var[1] + x[2]*x[2]/var[2]);
};
```

This is a C++ lambda function describing an anisotropic Gaussian distribution. The function takes three doubles denoting the three spherical coordinates. The distribution function must have either this signature, or an alternative signature which takes a single `std::array<double, 3>`, denoting the Cartesian three-vector `x`. Defining the distribution function as a lambda, because additional parameters can be taken as captures, as is the case with `dispersion` here.

The business of Zernike and spherical harmonic transforms and expansions is handled by the library zest. We can use zest's `zest::zt::ZernikeTransformer` to accomplish this. As a more general purpose library, zest supports multiple conventions for normalization and the Condon–Shortley phase. In ZebraDM the conventions are chosen to be such that the spherical harmonics are 4π-normalized and defined without the Condon–Shortley phase, and the radial Zernike polynomials are fully normalized. Multiple aliases of the basic types are defined by zest for different combinations of conventions, and so the correct transformer for Zernike expansions compatible with ZebraDM is `zest::zt::ZernikeTransformerNormalGeo`. We can use this to easily get the Zernike expansion of our distribution

```
#include <zest/zernike_glq_transformer.hpp>

constexpr double radius = 2.0;
zest::zt::RealZernikeExpansionNormalGeo zernike_transformer{};
zest::zt::RealZernikeExpansionNormalGeo distribution
    = zernike_transformer{}.transform(dist_func, radius, dist_order);
```

The Zernike functions are defined on the unit ball, but we can obviously scale any ball to a unit ball. The `radius` parameter here does exactly that. It is the radius of the ball on which our function is defined, so that `zest::zt::ZernikeTransformer` can do the scaling for you.

The next problem is to define our response function. For purposes of this demonstration, we use an arbitrary function

```
constexpr std::array<double, 3> a = {0.5, 0.5, 0.5};
auto resp_func = [&](double shell, double lon, double colat)
{
    const std::array<double, 3> dir = {
        std::sin(colat)*std::cos(lon),
```

(continues on next page)

(continued from previous page)

```
        std::sin(colat)*std::sin(lon),
        std::cos(colat)
    };

    return std::exp(-min_speed*(zdm::linalg(dir, a)));
};
```

The argument `shell` here is same as the shell parameter w (see the section on theoretical background), which in dark matter direct detection literature is often denoted $v_{\min}$. In nuclear scattering of dark matter this is the minimum speed needed from dark matter to give the nucleus recoil momentum equal to the momentum transfer.

The angle-integrated Radon transform in this library is defined on a collection of shell parameters. We therefore need to decide upon the collection of shell parameters. As discussed in the theoretical background section, the geometry of the situation means that if our distribution has offset $\vec{x}_0$, then the angle-integrated Radon transform goes to zero for $w > 1 + x_0$. Therefore, to determine an appropriate maximum value for the shell parameter, we will need to determine our offsets. In a real problem the offsets would come from somewhere. For example, in the context of dark matter direct detection they are the velocities of the laboratory relative to the dark matter distribution. For purposes of this example, we will generate a random list of vectors of some length

```
#include <random>
#include <vector>

std::vector<std::array<double, 3>> generate_offsets(std::size_t count, double
 ↪offset_len)
{
    std::mt19937 gen;
    std::uniform_real_distribution rng_dist{0.0, 1.0};

    std::vector<std::array<double, 3>> offsets(count);
    for (std::size_t i = 0; i < count; ++i)
    {
        const double ct = 2.0*rng_dist(gen) - 1.0;
        const double st = std::sqrt((1.0 - ct)*(1.0 + ct));
        const double az = 2.0*std::numbers::pi*rng_dist(gen);
        offsets[i] = {offset_len*st*std::cos(az), offset_len*st*std::sin(az), ct};
    }

    return offsets;
}
```

Alongside this, we can create a similar function that generates a vector of shell parameters

```
std::vector<double> generate_shells(std::size_t count, double offset_len)
{
    const double max_shell = 1.0 + offset_len;

    std::vector<double> shells(count);
    for (std::size_t i = 0; i < count; ++i)
        shells[i] = max_shell*double(i)/double(count - 1);

    return shells;
}
```

Then we can generate the offsets and shells

```
constexpr double offset_len = 0.5;
constexpr double offset_count = 10;
constexpr double shell_count = 50;

std::vector<std::array<double, 3>> offsets
    = generate_offsets(offset_count, offset_len);
std::vector<double> shells = generate_shells(shell_count, offset_len);
```

Now that we actually have the shells, we can compute the spherical harmonic transforms of the shells on the response functions. For this purpose, the header `zebradm/zebra_util.hpp` provides the container `zdm::SHExpansionVector` for storing a collection of spherical harmonic expansions in a single buffer, as well as the class `zdm::zebra::ResponseTransformer` for computing the spherical harmonic expansions.

```
zdm::zebra::ResponseTransformer response_transformer{};
zdm::SHExpansionVector response
    = response_transformer.transform(resp_func, shells, resp_order);
```

At this point we are almost ready to use the integrator. We still need two things, however. First is a vector of rotation angles for each offset, because not only can the distribution be defined in coordinates with an arbitrary offset, but it can also have a rotation relative to the coordinates in which the response is defined.

In principle, the distribution and response functions could be defined in coordinate systems which differ from each other by an arbitrary 3D rotation. However, arbitrary 3D rotations of spherical harmonic expansions are expensive, so the transformer has been limited to doing rotations about the z-axis per offset. With that said, nothing stops you from applying arbitrary global rotations on the expansions of the distribution and response before handing them off to the integrator. You can rotate Zernike and spherical harmonic transforms by arbitrary Euler angles with the `zest::Rotor` class

```
#include <numbers>

#include <zest/rotor.hpp>

zest::WignerdPiHalfCollection wigner(resp_order);
zest::Rotor rotor(resp_order);
constexpr std::array<double, 3> euler_angles = {
    std::numbers::pi/2, std::numbers::pi/3, std::numbers::pi/4
};

for (std::size_t i = 0; i < response.extent(); ++i)
    rotor.rotate(response[i], wigner, euler_angles, zest::RotatioType::coordinate);
 ↪
```

The variable `wigner` holds some constant Wigner D-matrices needed for the rotation. The last argument in turn tells the rotor whether we are rotating the coordinate system (active), or the object (passive). You can read more about this in the zest documentation.

With that said, here we can just create a nice full rotation

```
std::vector<double> generate_rotation_angles(std::size_t offset_count)
{
    std::vector<double> rotation_angles(offset_count);
    for (std::size_t i = 0; i < offset_count; ++i)
        rotation_angles[i]
            = 2.0*std::numbers::pi*double(i)/double(offset_count - 1);
    return rotation_angles;
}
```

and then generate the rotation angles

```
std::vector<double> rotation_angles = generate_rotation_angles(offset_count);
```

Now, the last remaining thing we need is a buffer to put the results in

```
#include <zest/md_array.hpp>

zest::MDArray<double, 2> out({offset_count, shell_count});
```

If we were dealing with one of the `Transverse` integrators, then then we would have to use `std::array<double, 2>` as the element type of `out` instead to store the nontransverse–transverse pair.

With this, we finally have everything in place to integrate the angle-integrated Radon transform

```
integrator.integrate(
        distribution, response, offsets, rotation_angles, shells, out);
```

Now, this is almost it. However, there is one point which need to be accounted for. Earlier we set the parameter `radius = 2.0` indicating to the Zernike transformer that our distribution is defined in a ball of radius two. However, the Radon transform is always evaluated on the unit ball. This means that if we defined the unit ball coordinates $\vec{x} = \vec{r}/R$, where R is our radius, then

$$\mathscr{R}[f](w,\hat{n}) = \int \delta(\vec{r}\cdot\hat{n} - w)f(\vec{r})\,d^3r = R^2\int \delta(\vec{x}\cdot\hat{n} - w/R)f(R\vec{x})\,d^3x.$$

That is, in practice, not only do we need to divide our original shell parameters by the radius (which we didn't do here because we just generated the scaled parameters directly), but we also have to multiply our result by the radius squared

```
for (auto& element : out.flatten())
    element *= radius*radius;
```

If we were also evaluating the transverse Radon transform, we would likewise have to multiply it by the fourth power of the radius.

And this is it. We have successfully computed the angle-integrated Radon transform of of our distribution, combined with an anisotropic response function, for a set of shells and offset–angle pairs. In summary, here is the full source code of our program

```
#include <array>
#include <vector>
#include <cmath>
#include <numbers>
#include <random>
#include <cstdio>

#include <zest/zernike_glq_transformer.hpp>
#include <zest/md_array.hpp>
#include <zest/rotor.hpp>

#include <zebradm/zebra_angle_integrator.hpp>
#include <zebradm/linalg.hpp>

std::vector<std::array<double, 3>>
generate_offsets(std::size_t count, double offset_len)
{
    std::mt19937 gen;
    std::uniform_real_distribution rng_dist{0.0, 1.0};

    std::vector<std::array<double, 3>> offsets(count);
```

(continues on next page)

(continued from previous page)

```
    for (std::size_t i = 0; i < count; ++i)
    {
        const double ct = 2.0*rng_dist(gen) - 1.0;
        const double st = std::sqrt((1.0 - ct)*(1.0 + ct));
        const double az = 2.0*std::numbers::pi*rng_dist(gen);
        offsets[i] = {offset_len*st*std::cos(az), offset_len*st*std::sin(az), ct};
    }

    return offsets;
}

std::vector<double> generate_rotation_angles(std::size_t offset_count)
{
    std::vector<double> rotation_angles(offset_count);
    for (std::size_t i = 0; i < offset_count; ++i)
        rotation_angles[i]
            = 2.0*std::numbers::pi*double(i)/double(offset_count - 1);
    return rotation_angles;
}


std::vector<double> generate_shells(std::size_t count, double offset_len)
{
    const double max_shell = 1.0 + offset_len;

    std::vector<double> shells(count);
    for (std::size_t i = 0; i < count; ++i)
        shells[i] = max_shell*double(i)/double(count - 1);

    return shells;
}

int main()
{
    constexpr std::array<double, 3> var = {0.5, 0.6, 0.7};
    auto dist_func = [&](double lon, double colat, double r)
    {
        const std::array<double, 3> x = {
            r*std::sin(colat)*std::cos(lon),
            r*std::sin(colat)*std::sin(lon),
            r*std::cos(colat)
        };

        return std::exp(-(x[0]*x[0]/var[0]) + x[1]*x[1]/var[1] + x[2]*x[2]/var[2]);
↪
    };

    constexpr std::array<double, 3> a = {0.5, 0.5, 0.5};
    auto resp_func = [&](double shell, double lon, double colat)
    {
        const std::array<double, 3> dir = {
            std::sin(colat)*std::cos(lon),
            std::sin(colat)*std::sin(lon),
            std::cos(colat)
        };

```

(continues on next page)

(continued from previous page)

```cpp
        return std::exp(-shell*(zdm::dot(dir, a)));
    };

    constexpr double offset_len = 0.5;
    constexpr std::size_t offset_count = 10;
    constexpr std::size_t shell_count = 50;

    std::vector<std::array<double, 3>> offsets
        = generate_offsets(offset_count, offset_len);
    std::vector<double> rotation_angles = generate_rotation_angles(offset_count);
    std::vector<double> shells = generate_shells(shell_count, offset_len);

    constexpr double radius = 2.0;
    constexpr std::size_t dist_order = 30;
    zest::zt::ZernikeTransformerNormalGeo zernike_transformer{};
    zest::zt::RealZernikeExpansionNormalGeo distribution
        = zernike_transformer.transform(dist_func, radius, dist_order);

    constexpr std::size_t resp_order = 60;
    zdm::zebra::ResponseTransformer response_transformer{};
    zdm::SHExpansionVector response
        = response_transformer.transform(resp_func, shells, resp_order);

    constexpr std::array<double, 3> euler_angles = {
        std::numbers::pi/2, std::numbers::pi/3, std::numbers::pi/4
    };

    zest::WignerdPiHalfCollection wigner(resp_order);
    zest::Rotor rotor(resp_order);
    for (std::size_t i = 0; i < response.extent(); ++i)
        rotor.rotate(response[i], wigner, euler_angles,
↪zest::RotationType::coordinate);

    zdm::zebra::AnisotropicAngleIntegrator integrator(dist_order, resp_order);

    zest::MDArray<double, 2> out({offset_count, shell_count});
    integrator.integrate(
            distribution, response, offsets, rotation_angles, shells, out);

    for (auto& element : out.flatten())
        element *= radius*radius;

    for (std::size_t i = 0; i < out.extent(0); ++i)
    {
        for (std::size_t j = 0; j < out.extent(0); ++j)
            std::printf("%.7e", out(i,j));
        std::printf("\n");
    }
}
```

0.4 Example: DM–nucleon scattering

In the nonrelativistic effective theory framework of nuclear scattering, the direct detection scattering rate for dark matter scattering off nuclear targets can be expressed in the form

$$\frac{d^2R_S}{dEd\Omega} = \frac{1}{64\pi^2}\frac{\rho_0}{m_{\rm DM}^3 m_{\rm N}^2}(F(q^2)\mathscr{R}[f](\hat{q}, v_{\rm min}) + F_\perp(q^2)\mathscr{R}_\perp[f](\hat{q}, v_{\rm min})).$$

Here ρ_0 is the local dark matter density, $m_{\rm DM}$ is the mass of the dark matter particle, $m_{\rm N}$ is the mass of the target nucleus, $\vec{q}$ is the momentum transfer to the nucleus, and

$$v_{\rm min} = \frac{q}{2\mu_{\rm DM,N}},$$

with $\mu_{\rm DM,N}$ is the reduced mass of the DM–nucleus system. See the article arxiv:2504.19714 for a detailed discussion.

The functions $F(q^2)$ and $F_\perp(q^2)$ depend on the effective theory couplings of dark matter to nucleons, and on the nuclear response functions. We do not concern ourselves with their details here, however, but it is worth noting that in the limit of small momentum transfer they can be regarded as polynomials in q^2.

The functions $\mathscr{R}[f]$ and $\mathscr{R}_\perp[f]$ denote the Radon and transverse Radon transforms of the dark matter velocity distribution f in the laboratory frame, as defined in the theory section.

This example demonstrates how this library could be used to calculate the energy-differential event rate

$$\frac{dR}{dE} = \int S(\vec{q})\frac{d^2R_S}{dEd\Omega}\,d\Omega.$$

0.4.1 Disclaimer

There are a number of input quantities whose origins we need to consider for the full calculation. Apart from the parameters discussed above, there is also the laboratory velocity relative to the dark matter distribution, $\vec{v}_{\rm lab}$, which appears in the formula for the lab frame velocity distribution $f(\vec{v} + \vec{v}_{\rm lab})$.

Furthermore, there is a hidden parameter, which is the relative orientation of the coordinate systems in which the distribution $f(\vec{v})$ is defined, and the coordinate system of the response $S(\vec{q})$. Conventionally, the former is in galactic coordinates, while the latter is in lab coordinates, and this needs to be dealt with.

This library is primarily focused on the computationally challenging part of evaluating the Radon transforms in a timely manner, and does not provide general facilities for evaluation of the other input parameters. Therefore, this example is about what form the inputs will need to take so that they can be used with this library. Parts that are outside the scope of this library are assumed as given.

0.4.2 Inputs

We will assume that there exists a number of functions that evaluate the input parameters we need to compute the angle-integrated Radon transforms, and it is up to the user of the lbrary how those functions are implemented in practice. These magic functions are as follows

```
std::vector<double>
time_interval(std::string_view start, std::string_view end, std::size_t count);

std::vecto<std::array<double, 3>>
compute_lab_velocities_equatorial(std::span<double> times);

std::vector<double> compute_earth_rotation_angles(std::span<double> times);
zdm::Matrix<double, 3, 3> rotation_matrix_from_equatorial_to_galactic();
std::array<double, 3> euler_angles_from_lab_to_polar(double lon, double lat);

std::vector<std::array<double, 2>>
```

(continues on next page)

(continued from previous page)

```
compute_eft_responses(std::span<double> momentum_transfers);

zdm::SHExpansionVector
get_detector_response(
    std::span<double> momentum_transfers, std::size_t resp_order);

DistributionParams get_distribution_params();

double velocity_distribution(
    const std::array<double, 3>& velocity, const DistributionParams& params);
```

This is approximately the scope of things we'd need to implement to get to the point where we can evaluate the angle-integrated Radon transform. Some of the the things here have been simplified so that we don't get bogged down in irrelevant details. For example, in reality the function `compute_eft_responses` would also take as parameters the EFT coefficients, parameters of the target nucleus, and so on. But in this example, we do not care so much about the implementations or inputs to these functions, just their outputs.

There is already one important aspect that can be observed here. For performance reasons, the library requires the velocity distribution and response to have coordinate systems whose z-axes are aligned, but which may differ by an angle in the xy-plane. This is no problem, because the Earth's axis of rotation does not change, to a good approximation, on time scales at which direct detection experiments operate. Therefore, aligning the z-axes of the coordinates of the velocity distribution only requires a time-independent rotation on both. The velocity distribution is rotated to the equatorial coordinate system (technically, the geocentric celestial refernce system), and the response is rotated to a horizontal coordinate system at the north pole (which I call "polar" coordinate system here for brevity, and which technically is the international terrestrial reference system). These two coordinate systems then differ by the Earth rotation angle.

The velocity distribution here is assumed to have a parametric formula, but the response is defined a bit more ambiguously, because in reality it would likely be generated from some numerical data, so we just have a function, which gives a collection of spherical harmonic expansions given a collection of momentum transfer magnitudes.

To avoid having to specify a definition of time, its origin and units, I have defined a function, which just gives us a number of times given a start date, end date, and a count

```
std::vector<double> times = time_interval("2000-01-01", "2001-01-02", 24);
```

Doesn't matter what units these are in, because we will just use them to compute things we actually care about

```
std::vector<std::array<double, 3>> v_lab_eq = lab_velocities_equatorial(times);
std::vector<double> era = earth_rotation_angles(times);
```

Then we have the static coordinate transforms

```
zdm::Matrix<double, 3, 3> equ_to_gal = rotation_matrix_from_equatorial_to_
 ↪galactic();

constexpr double lon = 0.5*std::numbers::pi;
constexpr double lat = 0.25*std::numbers::pi;
std::array<double, 3> lab_to_polar = euler_angles_from_lab_to_polar(lon, lat);
```

Apart from a very conveniently located detector site, it's notable that we define one of these rotations in terms of a rotation matrix, and the other in terms of Euler angles. The reason for that is that we are going to apply them under different circumstances.

Next we wish to generate a collection of energies at which the energy differential event rate will be evaluated. However, something we need to take into account is that the energy is bound by the inequality

$$v_{\text{min}} \leq v_{\text{lab}} + v_{\text{esc}},$$

where v_{esc} is the escape velocity. When this inequality does not hold, the event rate is zero, so there is no point computing the event rate outside this range. We could just generate the v_{min} values directly, but that may not be desirable if we want the energies to be equispaced, since v_{min} is not a linear function of energy

$$v_{\mathrm{min}} = \sqrt{\frac{m_{\mathrm{N}}E}{2\mu_{\mathrm{DM,N}}}}.$$

However, this gives a straightforward upperbound for the energy

$$E \leq \frac{2\mu_{\mathrm{DM,N}}}{m_{\mathrm{N}}}(v_{\mathrm{lab}} + v_{\mathrm{esc}})^2.$$

This is straightforward enough to implement here

```
std::vector<double> generate_energies(
    double reduced_mass, double nuclear_mass, double v_lab_max, double v_esc,
    std::size_t count)
{
    const double v_minmax = v_lab_max + v_esc;
    const double emax
        = std::sqrt(2.0*reduced_mass/nuclear_mass)*v_minmax*v_minmax;

    std::vector<double> energies(count);
    for (std::size_t i = 0; i < count; ++i)
        energies[i] = emax*double(i)/double(count - 1);

    return energies;
}

std::vector<double> vmin_from(
    std::span<double> energies, double reduced_mass, double nuclear_mass)
{
    const double prefactor = nuclear_mass/(2.0*reduced_mass);

    std::vector<double> vmin(energies.size());
    for (std::size_t i = 0; i < count; ++i)
        vmin[i] = std::sqrt(prefactor*energies[i]);

    return vmin;
}

std::vector<double> momentum_transfers_from(
    std::span<double> energies, double nuclear mass)
{
    const double prefactor = 2.0*nuclear_mass;

    std::vector<double> momentum_transfers(energies.size());
    for (std::size_t i = 0; i < count; ++i)
        momentum_transfers[i] = std::sqrt(prefactor*energies[i]);

    return momentum_transfers;
}
```

The value of `v_lab_max` we need to calculate from the list of lab velocities we generated above

```
double maximum_lab_velocity(std::span<std::vector<double, 3>> lab_velocities)
{
    double v_lab_sq_max = 0.0;
    for (const auto& v_lab : lab_velocities)
```

(continues on next page)

(continued from previous page)

```
    {
        const double v_sq = zdm::dot(v_lab, v_lab);
        v_lab_sq_max = std::max(v_lab_sq_max, v_sq);
    }

    return std::sqrt(v_lab_sq_max);
}
```

Now we can generate the $v_{\min}$ and momentum transfer values we are after

```
std::vector<double> energies = generate_energies(
    reduced_mass, nuclear_mass, maximum_lab_velocity(v_lab_eq), v_esc, 50);
std::vector<double> v_min = vmin_from(energies, reduced_mass, nuclear_mass);
std::vector<double> momentum_transfers
    = momentum_transfer_from(energies, nuclear_mass);
```

0.4.3 Distribution and response

With the momentum transfers, we can get the detector response

```
zdm::SHExpansionVector resp = get_detector_response(momentum_transfers);
```

Now, this response is defined in the lab frame, but we want it in the polar frame, so we need to rotate it. For this we can use the class `zest::Rotor`, which enables rotations of spherical harmonic expansions

```
zest::WignerdPiHalfCollection wigner(std::max(resp_order, dist_order));
zest::Rotor rotor(std::max(resp_order, disp_order));
for (std::size_t i = 0; resp.extent(); ++i)
    rotor.rotate(resp[i], wigner, lab_to_polar, zest::RotationType::coordinate);
```

When it comes to the velocity distribution, we need it to have a specific function signature, which only takes as arguments the spherical coordinates of the velocity itself. The easiest way to do this is to wrap it in a lambda

```
const zdm::Matrix<double, 3, 3> rot_equ_to_gal
    = rotation_matrix_from_equatorial_to_galactic();
const DistributionParams params = get_distribution_params();
auto wrapped_distribution = [&](double lat, double colat, double r)
{
    const std::array<double, 3> v_equ
        = zdm::coordinates::spherical_to_cartesian_phys(lat, colat, r);
    const std::array<double, 3> v_gal = zdm::matmul(rot_equ_to_gal, v_equ);
    return velocity_distribution(v_gal, params);
};
```

We can then take the Zernike transform of the wrapped distribution

```
zest::ZernikeTransformerNormalGeo zernike_transformer{};
zdm::ZernikeExpansion dist
    = zernike_transformer.transform(wrapped_distribution, v_esc, dist_order);
```

Giving `v_esc` as the second paramter here essentially tells the transformer that the velocity distribution is zero for velocities greater than the escape velocity, so that it can internally scale the coordinates to the unit sphere.

0.4.4 Angle-integrated Radon transform

We for the most general dark matter event rate, we need both the nontransverse and transverse Radon transforms, so we choose `zdm::zebra::AnisotropicTransverseAngleIntegrator`

```
zdm::zebra::AnisotropicAngleIntegrator integrator(dist_order, resp_order);
```

Before we go and compute the angle-integrated Radon transforms, there is one very important thing to account for. The integral that our `integrator` computes is defined on the unit ball in the velocity space. In other words, it is computed in a system of units where $v_{\text{esc}} = 1$ by definition. Therefore, we need to scale all our units appropriately. In terms of the input, this means dividing both v_{lab} and v_{min} by v_{esc}

```
const double inv_v_esc = 1.0/v_esc;

std::vector<std::array<double, 3>> u_lab_eq = v_lab_eq;
for (auto& element : u_lab_eq)
    element = zdm::mul(inv_v_esc, element);

std::vector<double> u_min = v_min;
for (auto& element : v_min)
    element *= inv_v_esc;
```

Then these are the inputs to the integrator

```
zest::MDArray<std::array<double, 2>, 2> out({v_lab_eq.size(), v_min.size()});
integrator.integrate(dist, resp, u_lab_eq, era, u_min, out);
```

Again, we need to account for the units in which the velocity integral was computed. This means multiplying the nontransverse Radon transform by v_{esc}^2 and the transverse Radon transform by v_{esc}^4

```
const double v_esc_2 = v_esc*v_esc;
const double v_esc_4 = v_esc_2*v_esc_2;
for (auto& element : out.flatten())
{
    element[0] *= v_esc_2;
    element[1] *= v_esc_4;
}
```

0.4.5 Getting the event rates out

After this, the output is in our original velocity units. After this it is just a matter of multiplying by the EFT responses, adding the results together, and multiplying by the common prefactor.

```
std::vector<double> eft_responses = compute_eft_responses(momentum_transfers);
const double prefactor = event_rate_prefactor(dm_density, dm_mass, nuclear_mass);

zdm::MDArray<double, 2> event_rates(out.extents());
for (std::size_t i = 0; out.extent(0); ++i)
{
    for (std::size_t j = 0; out.extent(1); ++j)
        event_rates(i,j) += prefactor*zdm::dot(eft_responses[j], out(i,j));
}
```

The function `event_rate_prefactor` here is a stand in for the prefactor

$$\frac{1}{64\pi^2}\frac{\rho_0}{m_{\text{DM}}^3 m_{\text{N}}^2}.$$

0.5 Library reference

0.5.1 Zernike-based Radon transforms

Types

`class IsotropicAngleIntegrator`

Angle integrated Radon transforms using the Zernike based Radon transform.

Public Functions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               std::span<const std::array<double, 3>> offsets, std::span<const
               double> shells, zest::MDSpan<double, 2> out)
```

Angle integrated Radon transform of a disitribution on an offset unit ball.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

Note

`distribution` and `offsets` are defined in the same coordinates.

Parameters

- **`distribution`** – Zernike expansion of distribution
- **`offsets`** – offsets of the distribution
- **`shells`** – distances of integration planes from the origin
- **`out`** – output values as 2D array of shape `{offsets.size(), shells.size()}`

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               const std::array<double, 3> &offset, std::span<const double> shells,
               std::span<double> out)
```

Angle integrated Radon transform of a disitribution on an offset unit ball.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

Note

`distribution` and `offset` are defined in the same coordinates.

Parameters

- **`distribution`** – Zernike expansion of the distribution
- **`offset`** – offset of the distribution
- **`shells`** – distances of integration planes from the origin
- **`out`** – output values as 1D array of length `shells.size()`

`class AnisotropicAngleIntegrator`

Angle integrated Radon transforms with anisotropic response function using the Zernike based Radon transform.

Public Functions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               SHExpansionVectorSpan<const std::array<double, 2>> response,
               std::span<const std::array<double, 3>> offsets, std::span<const
               double> rotation_angles, std::span<const double> shells,
               zest::MDSpan<double, 2> out, std::size_t trunc_order =
               std::numeric_limits<std::size_t>::max())
```

Angle integrated Radon transform of a disitribution on a offset unit ball, combined with an angle-dependent response.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

The response is an arbitrary function of the unit normal vector and shell radius. It multiplies the Radon transform before the integration over unit normals.

The response can be defined in a coordinate system, which differs from the response coordinate system by an arbitrary rotation about the z-axis. The angles of these rotations are given in the `rotation_angles` parameter. The rotation angles specifically is the counterclockwise angle from the x-axis of the response coordinate system to the x-axis of the distribution coordinate system.

Assuming the Zernike expansion of the distribution has order `L` and the spherical harmonic expansions of the response have order `K`, the algorithm will internally employ expansions of orders up to `K + L` to avoid aliasing when taking products of the distribution and response. However, in practice, the aliasing could be insignificant. Therefore, the parameter `trunc_order` is provided, which can cap the order of the internal expansions.

Note

The offsets and rotation angles come in pairs. Therefore `offsets` and `rotation_angles` must have the same size.

Note

`distribution` and `offsets` are defined in the same coordinates.

Parameters

- **distribution** – Zernike expansion of the distribution
- **response** – spherical harmonic expansions of response on `shells`
- **offsets** – offsets of the distribution
- **rotation_angles** – z-axis rotation angles between distribution and response coordinates
- **shells** – distances of integration planes from the origin
- **out** – output values as 2D array of shape `{offsets.size(), shells.size()}`
- **trunc_order** – maximum truncation order for internal expansions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               SHExpansionVectorSpan<const std::array<double, 2>> response, const
               std::array<double, 3> &offset, double rotation_angle,
               std::span<const double> shells, zest::MDSpan<double, 2> out,
               std::size_t trunc_order =
               std::numeric_limits<std::size_t>::max())
```

Angle integrated Radon transform of a disitribution on a offset unit ball, combined with an angle-dependent response.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

The response is an arbitrary function of the unit normal vector and shell radius. It multiplies the Radon transform before the integration over unit normals.

The response can be defined in a coordinate system, which differs from the response coordinate system by an arbitrary rotation about the z-axis. The angles of these rotations are given in the `rotation_angles` parameter. The rotation angles specifically is the counterclockwise angle from the x-axis of the response coordinate system to the x-axis of the distribution coordinate system.

Assuming the Zernike expansion of the distribution has order `L` and the spherical harmonic expansions of the response have order `K`, the algorithm will internally employ expansions of orders up to `K + L` to avoid aliasing when taking products of the distribution and response. However, in practice, the aliasing could be insignificant. Therefore, the parameter `trunc_order` is provided, which can cap the order of the internal expansions.

Note

`distribution` and `offset` are defined in the same coordinates.

Parameters

- **distribution** – Zernike expansion of the distribution
- **response** – spherical harmonic expansions of response on `shells`
- **offset** – offset of the distribution
- **rotation_angle** – z-axis rotation angle between distribution and response coordinates
- **shells** – distances of integration planes from the origin

- **`out`** – output values as 1D array of size `shells.size()`
- **`trunc_order`** – maximum truncation order for internal expansions

`class IsotropicTransverseAngleIntegrator`

Angle integrated regular and transverse Radon transforms and using the Zernike based Radon transform.

Public Functions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               std::span<const std::array<double, 3>> offsets, std::span<const
               double> shells, zest::MDSpan<std::array<double, 2>, 2> out)
```

Angle integrated transverse and nontransverse Radon transform of a velocity disitribution on an offset unit ball.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane’s nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

Note

`distribution` and `offsets` are defined in the same coordinates.

Parameters

- **`distribution`** – Zernike expansion of the distribution
- **`offsets`** – offsets of the distribution
- **`shells`** – distances of integration planes from the origin
- **`out`** – output values as 2D array of shape `{offsets.size(), shells.size()}`

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               const std::array<double, 3> &offset, std::span<const double> shells,
               std::span<std::array<double, 2>> out)
```

Angle integrated transverse and nontransverse Radon transform of a velocity disitribution on an offset unit ball.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane’s nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

Note

`distribution` and `offset` are defined in the same coordinates.

Parameters

- **`distribution`** – Zernike expansion of the distribution
- **`offset`** – offset of the distribution
- **`shells`** – distances of integration planes from the origin
- **`out`** – output values as 1D array of size `shells.size()`

`class` **`AnisotropicTransverseAngleIntegrator`**

Angle integrated regular and transverse Radon transforms with anisotropic response function using the Zernike based Radon transform.

Public Functions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               SHExpansionVectorSpan<const std::array<double, 2>> response,
               std::span<const std::array<double, 3>> offsets, std::span<const
               double> rotation_angles, std::span<const double> shells,
               zest::MDSpan<std::array<double, 2>, 2> out, std::size_t trunc_order
               = std::numeric_limits<std::size_t>::max())
```

Angle integrated Radon transform of a disitribution on a offset unit ball, combined with an angle-dependent response.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

The response is an arbitrary function of the unit normal vector and shell radius. It multiplies the Radon transform before the integration over unit normals.

The response can be defined in a coordinate system, which differs from the response coordinate system by an arbitrary rotation about the z-axis. The angles of these rotations are given in the `rotation_angles` parameter. The rotation angles specifically is the counterclockwise angle from the x-axis of the response coordinate system to the x-axis of the distribution coordinate system.

Assuming the Zernike expansion of the distribution has order `L` and the spherical harmonic expansions of the response have order `K`, the algorithm will internally employ expansions of orders up to `K + L` to avoid aliasing when taking products of the distribution and response. However, in practice, the aliasing could be insignificant. Therefore, the parameter `trunc_order` is provided, which can cap the order of the internal expansions.

> **Note**
>
> The offsets and rotation angles come in pairs. Therefore `offsets` and `rotation_angles` must have the same size.

> **Note**
>
> `distribution` and `offsets` are defined in the same coordinates.

Parameters

- **distribution** – Zernike expansion of the distribution
- **response** – spherical harmonic expansions of response on `shells`
- **offsets** – offsets of the distribution
- **rotation_angles** – z-axis rotation angles between distribution and response coordinates
- **shells** – distances of integration planes from the origin
- **out** – output values as 2D array of shape `{offsets.size(), shells.size()}`
- **trunc_order** – maximum truncation order for internal expansions

```
void integrate(ZernikeExpansionSpan<const std::array<double, 2>> distribution,
               SHExpansionVectorSpan<const std::array<double, 2>> response, const
               std::array<double, 3> &offset, double rotation_angle,
               std::span<const double> shells, std::span<std::array<double, 2>>
               out, std::size_t trunc_order =
               std::numeric_limits<std::size_t>::max())
```

Angle integrated Radon transform of a disitribution on a offset unit ball, combined with an angle-dependent response.

The 3D Radon transform is defined as an integral over a plane in the 3D space. Any given plane is uniquely determined by its unit normal vector, and its distance to the origin, defined by the plane's nearest point to the origin. The set of unit vectors at a given distance define a spherical shell. Hence the angle-integrated Radon transform is parametrized by the distances, given in the `shells` parameter.

The Radon transform has well-defined transformation properties under affine transforms. In the angle-integrated Radon transform, the only relevant part of the affine transform is the offset. Thus the `offsets` parameter contains an arbitrary collection of offset vectors for the distribution.

The response is an arbitrary function of the unit normal vector and shell radius. It multiplies the Radon transform before the integration over unit normals.

The response can be defined in a coordinate system, which differs from the response coordinate system by an arbitrary rotation about the z-axis. The angles of these rotations are given in the `rotation_angles` parameter. The rotation angles specifically is the counterclockwise angle from the x-axis of the response coordinate system to the x-axis of the distribution coordinate system.

Assuming the Zernike expansion of the distribution has order `L` and the spherical harmonic expansions of the response have order `K`, the algorithm will internally employ expansions of orders up to `K + L` to avoid aliasing when taking products of the distribution and response. However, in practice, the aliasing could be insignificant. Therefore, the parameter `trunc_order` is provided, which can cap the order of the internal expansions.

Note

`distribution` and `offset` are defined in the same coordinates.

Parameters

- **distribution** – Zernike expansion of the distribution
- **response** – spherical harmonic expansions of response on `shells`
- **offset** – offset of the distribution
- **rotation_angle** – z-axis rotation angle between distribution and response coordinates
- **shells** – distances of integration planes from the origin

- **out** – output values as 1D array of size `shells.size()`
- **trunc_order** – maximum truncation order for internal expansions

Functions

`void zdm::zebra::`**`radon_transform`**`(ZernikeExpansionSpan<`**`const`**` std::array<double, 2>> in, ZernikeExpansionSpan<std::array<double, 2>> out)` **`noexcept`**

Apply the Radon transform onto a Zernike expansion.

Given the Zernike expansion coefficients f_{nlm} of a function $f(\vec{x})$, this computes the coefficients in the corresponding expansion of the offset Radon transform

$$\mathscr{R}[f](w,\hat{n}) = \sum_{nlm} g_{gnlm} P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n}).$$

These coefficients are given by the formula

$$g_{nlm} = \frac{f_{nlm}}{2n+3} - \frac{f_{n-2,lm}}{2n-1}.$$

Parameters

- **in** – input Zernike expansion coefficients
- **out** – output Radon expansion coefficients

`void zdm::zebra::`**`radon_transform_inplace`**`(ZernikeExpansionSpan<std::array<double, 2>> exp)` **`noexcept`**

Apply the Radon transform onto a Zernike expansion.

Given the Zernike expansion coefficients f_{nlm} of a function $f(\vec{x})$, this computes the coefficients in the corresponding expansion of the offset Radon transform

$$\mathscr{R}[f](w,\hat{n}) = \sum_{nlm} g_{gnlm} P_n(w + \vec{x}_0 \cdot \hat{n}) Y_{lm}(\hat{n}).$$

These coefficients are given by the formula

$$g_{nlm} = \frac{f_{nlm}}{2n+3} - \frac{f_{n-2,lm}}{2n-1}.$$

Parameters

exp – Zernike expansion coefficients

0.5.2 Containers and views

Types

`class SHExpansionVector`

Container that packs multiple spherical harmonic expansions of the same order into one buffer.

`template<typename SubspanType>`

`class SuperSpan`

View over contiguous data with arbitrary layout in the last dimensions.

Given `SubspanType` describing contiguous data with arbitrary layout, SuperSpan describes a multidimensional array view over such data. For example, if `SubspanType` describes triangular data.

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

then SuperSpan<SubSpanType> can be used to describe a 2D array of such data

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

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

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

Template Parameters
SubspanType – view describing multidimensional data

template<**typename SubspanType**, std::size_t **rank**>

class MultiSuperSpan

Multidimensional view over contiguous data with arbitrary layout in the last dimensions.

Given SubspanType describing contiguous data with arbitrary layout, MultiSuperSpan describes a multidimensional array view over such data. For example, if SubspanType describes triangular data.

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

then MultiSuperSpan<SubSpanType, 2> can be used to describe a 2D array of such data

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

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

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

Template Parameters

- **SubspanType** – view describing multidimensional data
- **rank** – number of dimensions in the outer view

Type aliases

using zdm::**SHExpansion** = zest::st::RealSHExpansionGeo

Alias for zest::st::RealSHExpansionGeo

using zdm::**ZernikeExpansion** = zest::zt::RealZernikeExpansionNormalGeo

Alias for zest::st::RealSHZernikeExpansionGeo

template<**typename ElementType**>

using zdm::**SHExpansionSpan** = zest::st::RealSHSpanGeo<ElementType>

Alias for zest::st::RealSHSpanGeo<ElementType>

Template Parameters
ElementType –

template<**typename ElementType**>

using `zdm::`**`SHExpansionVectorSpan`** = `SuperSpan<SHExpansionSpan<ElementType>>`

Alias for `zest::st::SuperSpan<SHExpansionSpan<ElementType>>`

Template Parameters
`ElementType` –

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

using `zdm::`**`ZernikeExpansionSpan`** = `zest::zt::RealZernikeSpanNormalGeo<ElementType>`

Alias for `zest::st::RealZernikeSpanNormalGeo<ElementType>`

Template Parameters
`ElementType` –

0.5.3 Coordinate transforms

Functions

`constexpr` `std::array<double, 3> zdm::coordinates::`**`cartesian_to_spherical_geo`**`(`**`const`** `std::array<double, 3> &cartesian)` **`noexcept`**

Convert Cartesian to spherical coordinates in geography convention.

Parameters
`cartesian` – vector of Cartesian coordinates

Returns
spherical harmonic coordinates in order [longitude, latitude, length]

`constexpr` `std::array<double, 3> zdm::coordinates::`**`spherical_to_cartesian_geo`**`(double longitude, double latitude)` **`noexcept`**

Convert spherical angles to Cartesian coordinates in physics convention.

Parameters

- **`longitude`** – longitude angle in radians $[0, 2\pi]$
- **`latitude`** – latitude in radians $[-\pi/2, \pi/2]$

Returns
vector of Cartesian coordinates

`constexpr` `std::array<double, 3> zdm::coordinates::`**`spherical_to_cartesian_geo`**`(double longitude, double latitude, double length)` **`noexcept`**

Convert spherical to Cartesian coordinates in geography convention.

Parameters

- **`longitude`** – longitude angle in radians $[0, 2\pi]$
- **`latitude`** – latitude in radians $[-\pi/2, \pi/2]$
- **`length`** – length of vector

Returns
vector of Cartesian coordinates

`constexpr` `std::array<double, 3> zdm::coordinates::`**`cartesian_to_spherical_phys`**`(`**`const`** `std::array<double, 3> &cartesian)` **`noexcept`**

Convert Cartesian to spherical coordinates in physics convention.

Parameters
`cartesian` – vector of Cartesian coordinates

Returns
spherical harmonic coordinates in order [azimuth, colatitude, length]

`constexpr` `std::array<double, 3> zdm::coordinates::`**`spherical_to_cartesian_phys`**`(double azimuth, double colatitude)` **`noexcept`**

Convert spherical angles to Cartesian coordinates in physics convention.

Parameters

- **`azimuth`** – azimuthal angle in radians $[0, 2\pi]$
- **`colatitude`** – colatitude in radians [0, *pi*]

Returns
vector of Cartesian coordinates

`constexpr` `std::array<double, 3> zdm::coordinates::`**`spherical_to_cartesian_phys`**`(double azimuth, double colatitude, double length)` **`noexcept`**

Convert spherical to Cartesian coordinates in physics convention.

Parameters

- **`longitude`** – longitude angle in radians $[0, 2\pi]$
- **`colatitude`** – colatitude in radians $[0, \pi]$
- **`length`** – length of vector

Returns
vector of Cartesian coordinates

0.5.4 Miscellaneous

Types

`class ResponseTransformer`

Class for spherical harmonic expansion of response functions on multiple shells.

Public Functions

```
template<typename RespType>
inline void transform(RespType &&resp, std::span<const double> shells,
                      SHExpansionVectorSpan<std::array<double, 2>> out)
```

Take spherical harmonic transform of a response function.

Template Parameters

RespType – type of response function

Parameters

- **resp** – response function
- **shells** – shells the spherical harmonic transforms are evaluated on
- **out** – spherical harmonic expansions of the response

```
template<typename RespType>
inline SHExpansionVector transform(RespType &&resp, std::span<const double> shells,
                                   std::size_t order)
```

Take spherical harmonic transform of a response function.

Template Parameters

RespType – type of response function

Parameters

- **resp** – response function
- **shells** – shells the spherical harmonic transforms are evaluated on
- **order** – order of spherical harmonic expansions

Returns

spherical harmonic expansions of the response

INDEX