EnvironmentSTUDIOflow

Release x.y.unknown

Tom Clark

Sep 04, 2019

Contents

EnvironmentSTUDIO flow (**es-flow** for short) is a library for analysing and modeling atmospheric and marine boundary layers.

While **es-flow** can be used for any turbulent boundary layer analysis, its main focus is for wind and tidal site characterisation - generating the 'best fit' of analytical models to measured velocity data.

A key strength of **es-flow** is the adem. This extremely robust method allows users to:

- determine detailed turbulence information from instruments like LiDAR
- characterise turbulence and shear beyond tip height of even the biggest offshore wind turbines
- characterise **coherent structure** in turbulence, crucially important for fatigue loading in wind turbines.

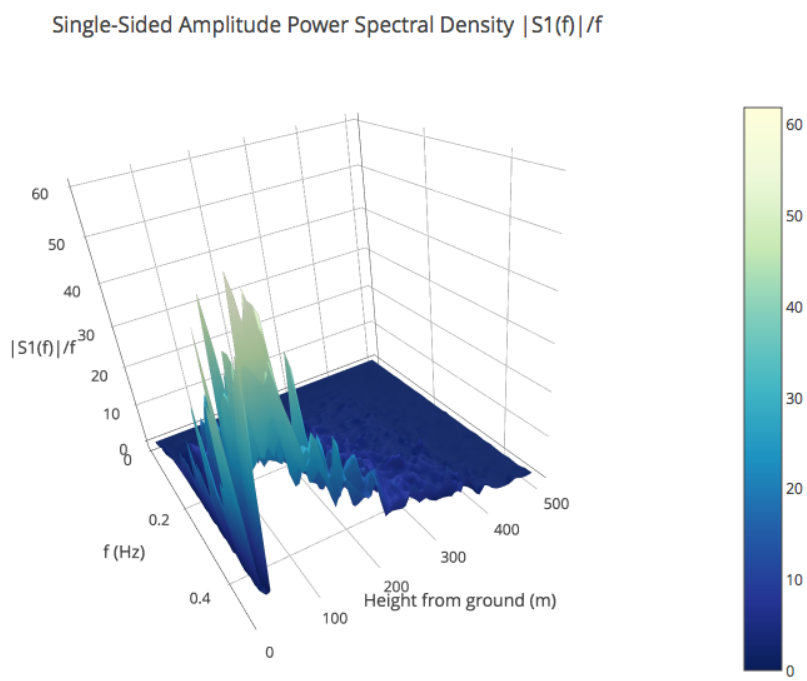


Fig. 1: Atmospheric Boundary Layer PSD determined with **es-flow**.

CHAPTER 1

Aims

The **es-flow** library provides:[1]

1. **Parameterised profiles**
    - Mean Velocity (using power law, logarithmic law, MOST or Lewkowicz relations)
    - Mean Veer (Using Monin-Obukhov approach)
    - Reynolds Stress `u'w'` (using Lewkowicz relations)
    - Reynolds Stress `u'u'`, `u'v'`, `u'w'`, `v'v'`, `v'w'`, `w'w'` (using the adem)
    - Spectra `Sij` (using Kaimal, von Karman)
    - Spectra `Sij` (using adem)
    - Integral turbulent intensity and lengthscale `I`, `l`
2. **Best fit parameter sets**
    - To describe the above profiles analytically (given measured velocity data from an instrument).

In future, generation of artificial flow fields for simulation purposes might be considered. This would overlap with - or replace -

- `.wnd` fields input to BEM or FVM models like Bladed, FAST and TurbineGRID
- Inlet boundary conditions for DES or LES codes

[1] Not all of this functionality is implemented yet!

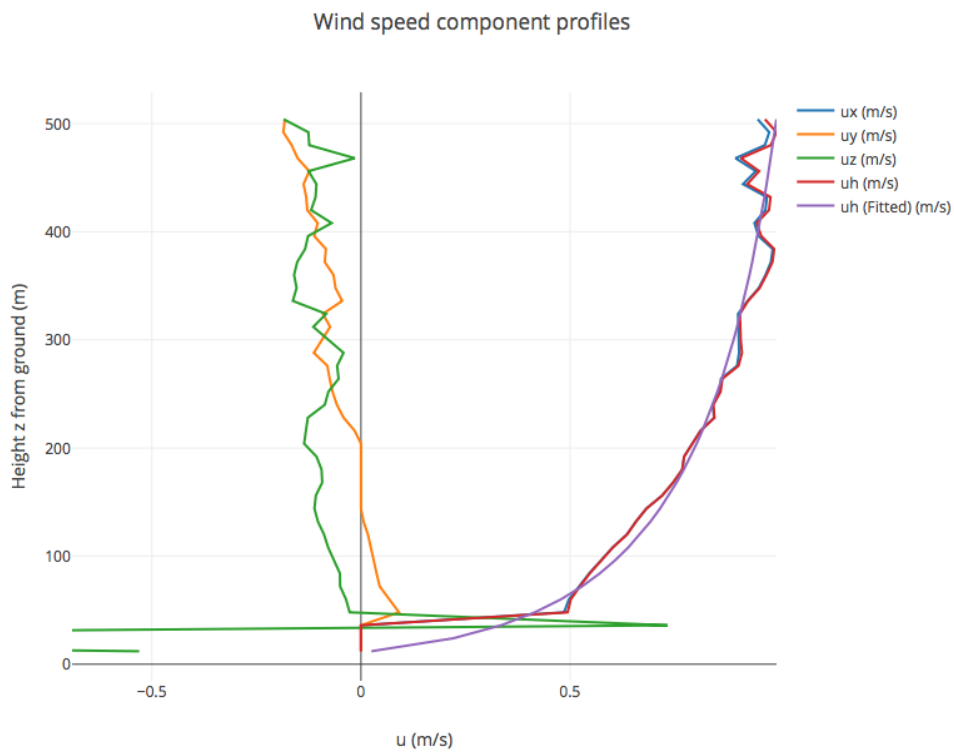


Fig. 1: Atmospheric Boundary Layer velocity profile, fitted with **es-flow**.

CHAPTER 2

Uses

At Octue, **es-flow** is used to:

- Provide a basis for developing and validating new processes, like the adem, for characterising Atmospheric Boundary Layers.
- Process LiDAR and ADCP datasets from raw instrument files.
- Apply windowed analyses to time series data, for flow characterisation.
- Generate load cases for FAST, Bladed and our own TurbineGRID aerodynamic wind turbine analysis tools.

We'd like to hear about your use case. Please get in touch!

We use the GitHub Issue Tracker to manage bug reports and feature requests.

2.1 Analytical Models

2.1.1 Velocity Relations

es-flow uses a range of different velocity relations, so you can choose which is preferable. Fitting different profiles returns a metric of accuracy; but this isn't the be-all and end-all. Always choose which formulation best represents the physics in play!

Power Law

Power law

Logarithmic

Log law

MOST

Most law

Lewkowicz

Lewkowicz 1982, modified by P&M. Include the Reynolds Stress profile

2.1.2 Veer Relations

Monin obukhov

2.1.3 Spectral Relations

Kaimal

Attached-Detached Eddy Method

Describe ADEM here.

2.2 Examples

Here, we look at example use cases for the library, and show how to achieve them in python and C++. Many of these are copied straight from the unit test cases, so you can always check there to see how everything hooks up.

> **Warning:** To plot the figures in the examples, we use (in python) octue's sdk, which includes plotly, or the fledgling C++ plotting library (also by this author) cpplot, either of which you'll need to install yourself - or use your preferred plotting library (e.g. matplotlib).

2.2.1 Obtaining a velocity profile

Given a set of vertical coordinates `z_i` and a set of input parameters for an analytical model, how can we determine the corresponding mean velocity profile?

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

#include "relations/velocity.h"

using namespace es;

int main(const int argc, const char **argv) {

    // Basic boundary layer parameters
    double pi_coles = 0.42;
    double kappa = 0.41;
```

(continues on next page)

(continued from previous page)

```
    double u_inf = 20.0;
    double shear_ratio = 23.6;
    double delta = 1000.0;

    // Get velocity at height z, where z is a simple double scalar
    double z_scalar = 1.0;
    double speed1 = lewkowicz_speed(z_scalar, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta);
    std::cout << "checked scalar double operation (U = " << speed1 << " m/s)" <<␣
↪std::endl;

    // Get velocity at a range of heights z
    double low = 1;
    double high = 100;
    int n_bins = 10;
    VectorXd z = VectorXd::LinSpaced(n_bins, low, high);
    VectorXd speed = lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio, delta);
    std::cout << "checked VectorXd operation: "<< speed.transpose() << std::endl;

    return 0;
}
```

Python

```
import numpy as np
import es

def main():

    // Basic test parameters
    pi_coles = 0.42
    kappa = 0.41
    u_inf = 20.0
    shear_ratio = 23.6
    delta = 1000.0

    // Check that it works for a z value scalar
    z = 10.0
    speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
    print('checked scalar operation (U = ', speed, 'm/s)')

    // Check that it works for a numpy array input (vertically spaced z)
    low = 1
    high = 100
    n_bins = 10
    z = np.linspace(low, high, n_bins)
    speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
    print('checked array operation:', speed)

    return
```

2.2.2 Fitting a velocity profile

Imagine we have a vertical profile of velocity `u(z)`, determined by an instrument such as a vertical LiDAR in VAD mode.

Say there are 30 data points, spaced 10m apart vertically. In reality, each data point is an ensemble of many thousands of doppler velocity readings, from a region of vertical space above and below the `z` coordinate, known as 'bins'.

We wish to determine the parameter set for an analytical profile that best characterises the atmosphere at that point in time. Importantly:

- We may need to fix some of the parameters, to help constrain the solver or because we know them already.
- The better our first guess, the quicker the algorithm will converge, and the more likely it is to find a sensible answer.

Under the hood, `es-flow` uses a Levenberg-Marquadt solution to find the best fit (**'Google's amazing ceres-solver library <>'_** is used for this). The gnarly details are wrapped for ease of use, and you can provide an initial guess and fix certain parameters.

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include "fit.h"
#include "relations/velocity.h"
#include "definitions.h"

using namespace es;

int main(const int argc, const char **argv) {

    // 'True' profile parameters
    double pi_coles = 0.42;
    double kappa = KAPPA_VON_KARMAN;
    double u_inf = 20;
    double shear_ratio = 23.6;
    double delta_c = 1000;

    // Simulate measured data by taking the true profile and adding noise
    Eigen::ArrayXd z = Eigen::ArrayXd::LinSpaced(40, 1, 400);
    Eigen::ArrayXd u_original(40);
    Eigen::ArrayXd u_noisy(40);
    Eigen::ArrayXd u_fitted(40);
    u_original = lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio, delta_c);
    u_measured = u_original + ArrayXd::Random(40) / 4;
    std::cout << z.transpose() << std::endl <<std::endl;
    std::cout << u_measured.transpose() << std::endl <<std::endl;

    // Fit to find the value of parameters.
    // NB you can constrain different parameters, and use values other than the␣
↪default,
    // see the docs for fit_lewkowicz_speed.
    Eigen::Array<double, 5, 1> fitted = fit_lewkowicz_speed(z, u_measured);
    u_fitted = lewkowicz_speed(z, fitted(0), fitted(1), fitted(2), fitted(3),␣
↪fitted(4));

    // Sum of squares error, for exact and fitted. They should be similar, with␣
↪fitted being lower.
```

(continues on next page)

(continued from previous page)

```
    double lsq_error_noisy = pow(u_original-u_measured, 2.0).sum();
    double lsq_error_fitted = pow(u_fitted-u_measured, 2.0).sum();
    std::cout << "Sqd error (correct - noisy): " << lsq_error_noisy << std::endl;
    std::cout << "Sqd error (fitted - noisy) (should be lower): " << lsq_error_fitted
↪<< std::endl;

    return 0;
}
```

Python

```
import numpy as np
import es

def main():

    # WARNING - THIS IS WHAT I WANT, NOT WHAT I HAVE
    # (I'm using this section to sketch a future object oriented python API out!)

    // Simulate measured data by taking a true profile and adding noise
    pi_coles = 0.42
    kappa = 0.41
    delta = 1000.0
    u_inf = 20.0
    shear_ratio = 23.6
    z = np.linspace(1, 400, 40)
    true_speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
    measured_speed = true_speed + 3 * (np.random(40) - 0.5)

    // Make an initial guess and help the solver by constraining boundary layer␣
↪thickness and setting the von karman constant
    initial_guess = np.array([0.5, 0.41, np.amax(measured_speed), 20, 1000)
    fix_params = numpy.array([0, 1, 0, 0, 1], dtype=bool)

    // Run the fitting process
    prof = es.LewkowiczProfile()
    prof.fit(z, measured_speed, initial_guess, fit_params)
    print(prof.params)

    return
```

2.2.3 Smearing and de-smearing

Say we have an instrument like a vertical profiling LiDAR, which takes measurements from a volume, rather than a point. If we capture a profile (e.g. of velocity computed by the VAD technique), each point is the average velocity in a vertical 'bin'. However, if the shear (`du/dz`) changes through that bin, then the average velocity is biased… strictly speaking, the instrument applies a filter that changes the recorded velocity profile.

In order to:

1. compare like-for like when validating against another instrument like a met-mast,
2. correctly determine parameters for an analytic fit to the measurements and
3. post-process measured velocity data to *de-filter* it,

we need to be able to *smear* an analytically generated profile in the same way that measuring it with an instrument would.

Applying smear to a velocity profile

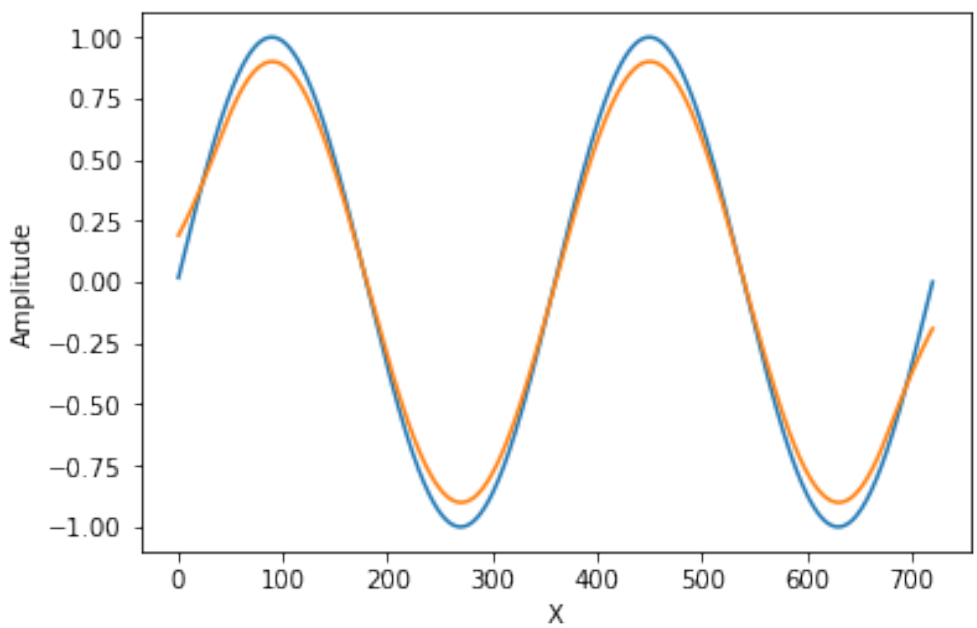
For a general timeseries signal, or an image, etc, this is done by convolution - the following python script (you can copy/paste into a jupyter notebook to see it live) shows the basics, in this case a sin(x) signal being smeared out, as if measured by an instrument whose viewport is 90 degrees wide:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x = np.linspace(1, 720, 720)
sig = np.sin((x*2*3.14159/360))

# The kernel is a 'box filter'. It's like taking an average over a 'bin' of width 90␣
↪degrees on the x axis.
ker = np.ones(90)
ker = ker/sum(ker)

convolved = np.convolve(sig, ker, 'same')

fig, ax = plt.subplots(1, 1)
ax.plot(x,sig)
ax.plot(x,convolved)
ax.set_xlabel('X')
ax.set_ylabel('Amplitude')
```



Do you see the problem at the ends? For our use cases, we can't ignore errors near the endpoints. Sigh.

This is solved for you by `es-flow`, which integrates analytical profiles over a specified vertical range, or 'bin'. At present, we implement a top hat filter, which is pretty representative of the way in which LiDAR and SODAR units bin their measurements. Please feed back if it'd be helpful to have other (or custom) kernel shapes!

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

#include "relations/velocity.h"
#include "utilities/smear.h"

using namespace es;

int main(const int argc, const char **argv) {

    // Basic boundary layer parameters
    double pi_coles = 0.42;
    double kappa = 0.41;
    double u_inf = 20.0;
    double shear_ratio = 23.6;
    double delta = 1000.0;

    // Bin size (m in the z direction)
    double bin_size = 10;

    // Get 'correct' and 'smeared' to compare
    Eigen::ArrayXd z = Eigen::ArrayXd::LinSpaced(40, 1, 400);
    Eigen::ArrayXd speed = lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta);
    Eigen::ArrayXd speed_smeared = lewkowicz_speed_smeared(z, bin_size, pi_coles,␣
↪kappa, u_inf, shear_ratio, delta);

    std::cout << "Actual wind speed: " << speed.transpose() << std::endl;
    std::cout << "Wind speed measured with a binning instrument (es-flow): " << speed_
↪smeared.transpose() << std::endl;

    return 0;
}
```

Python

```
import numpy as np
import es

def main():

    # Basic boundary layer parameters
    pi_coles = 0.42
    kappa = 0.41
    u_inf = 20.0
    shear_ratio = 23.6
    delta = 1000.0

    # Bin size (m in the z direction)
    bin_size = 10

    # Get 'correct' and 'smeared' to compare
    z = np.linspace(1, 400, 40)
    speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
```

(continues on next page)

(continued from previous page)

```
    speed_smeared = es.relations.lewkowicz_speed_smeared(z, pi_coles, kappa, u_inf,␣
↪shear_ratio, delta)

    print('Actual wind speed:', speed)
    print('Wind speed measured with a binning instrument (es-flow):', speed_smeared)

    return
```

De-smearing a measured profile

OK, so lets say we have measured profile data. We know our instrument has smeared the actual profile somewhat, and we want to correct for this.

Argh, but we can't! **De-smearing is equivalent to a deconvolution.** Not only is this not guaranteed to be numerically stable, but the problem isn't well conditioned for the case where the data points are spaced further apart from the bins... We'd need to interpolate the measured signal to a higher spatial resolution, deconvolve then interpolate back down. That's subject to significant error, since noise/spikes/artifacts in some vertical locations pollute their neighbours.

Never fear! Once you're fitted an analytical profile to the noisy data, `es-flow` makes it simple to determine a correction for the measured data.

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

#include "relations/velocity.h"
#include "utilities/smear.h"

using namespace es;

int main(const int argc, const char **argv) {

    /* First, fit an analytic profile to your data... see "Fitting a velocity profile
↪" to get the following:
     * u_original
     * u_noisy
     * fitted_params
     * u_fitted
     */

    // Use the difference between the analytical distribution and its smeared␣
↪equivalent to correct the data
    Eigen::ArrayXd u_fitted_smeared = lewkowicz_speed_smeared(z, bin_size, fitted(0),␣
↪fitted(1), fitted(2), fitted(3), fitted(4));
    Eigen::ArrayXd corrector = u_fitted - u_fitted_smeared;
    Eigen::ArrayXd u_measured_corrected = u_noisy + corrector;

    std::cout << "Measured wind speed: " << u_measured.transpose() << std::endl;
    std::cout << "Postprocessed (de-smeared) measurements: " << u_measured_corrected.
↪transpose() << std::endl;

    return 0;
}
```

Python

```
import numpy as np
import es

def main():

    # First, fit an analytic profile to your data... see "Fitting a velocity profile"␣
↪to get the following:
    # u_original
    # u_noisy
    # fitted_params
    # u_fitted

    # Use the difference between the analytical distribution and its smeared␣
↪equivalent to correct the data
    u_fitted_smeared = lewkowicz_speed_smeared(z, bin_size, fitted[0], fitted[1],␣
↪fitted[2], fitted[3], fitted[4])
    corrector = u_fitted - u_fitted_smeared
    u_measured_corrected = u_noisy + corrector

    print('Measured wind speed:', u_measured);
    print('Postprocessed (de-smeared) measurements:', u_measured_corrected)

    return
```

2.2.4 Getting profile derivatives

OK, so you fitted a profile to experimental data and have the parameter set that best represents the atmosphere at that time. But you're a smart cookie who's calculating other complex relations, and you also need the derivative with respect to height.

Central differencing? No… there's a much more robust and accurate way. We simply use automatic differentiation:

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

#include "relations/velocity.h"

using namespace es;
using namespace Eigen;

int main(const int argc, const char **argv) {

    typedef Eigen::AutoDiffScalar<Eigen::VectorXd> ADScalar;
    ADScalar ads_z;
    ADScalar ads_speed;
    VectorXd dspeed_dz;
    dspeed_dz.setZero(n_bins);

    for (int k = 0; k < n_bins; k++) {
        ads_z.value() = z[k];
        ads_z.derivatives() = Eigen::VectorXd::Unit(1, 0);
        ads_speed = power_law_speed(ads_z, u_ref, z_ref, alpha);
        dspeed_dz[k] = ads_speed.derivatives()[0];
```

(continues on next page)

(continued from previous page)

```
    }

    std::cout << "speed = ["      << speed.transpose()      << "];" << std::endl;
    std::cout << "dspeed_dz = [" << dspeed_dz.transpose() << "];" << std::endl;

    return 0;
}
```

Python

```
# WARNING - I'm afraid no API for this has been created in python yet. TODO!
def main():
    return
```

2.2.5 Obtaining a spectral profile

The Atmospheric Boundary Layer has a turbulent spectrum which varies with height. At each height in a spectral profile, the amplitude of the spectrum varies with wavenumber (frequency). There are also six terms in a turbulent spectrum `S_i_j` (not 9, since the tensor is symmetric, i.e. `S_1_2 = S_2_1`).

> **Warning:** You'd expect a spectral profile to be a tensor with dimension `nz x nf x 6`. But, we don't do that... instead we actually return 6 `nz x nf` arrays.
>
> Yes, we know this is a hassle. Why? Two reasons: - It's actually bloody annoying doing tensors with Eigen, because their tensor library is undocumented and very feature-light in terms of indexing (it's an unsupported extension, which I firmly believe whould be part of core), so C++ users can expect some unnecessarily heavy syntax if you try. What a drag. Let's see what happens in the next few releases.
>
> - Pybind doesn't bind ‘‘Eigen::Tensor‘‘s. Sure, I could fork it and add tensors (this issue, open at the time of writing, gives some hints) but would rather be working on core features here. If a contributor wants to do the work of sorting pybind to work with ‘‘Eigen::Tensor‘‘s then I'll happily update at this end.

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/AutoDiff>

#include "relations/velocity.h"

using namespace es;

int main(const int argc, const char **argv) {

    // Boundary layer parameters (full ADEM set)
    double pi_coles = 0.42;
    double kappa = 0.41;
    double u_inf = 20.0;
    double shear_ratio = 23.6;
    double delta = 1000.0;
    double beta = 0.0;
    double zeta = 0.0;
    Eigen::ArrayXd parameters = Eigen::ArrayXd(7);
```

(continues on next page)

(continued from previous page)

```
    parameters << pi_coles, kappa, u_inf, shear_ratio, delta, beta, zeta;

    // Set z at which you want the spectra
    Eigen::ArrayXd z = Eigen::VectorXd::LinSpaced(40, 1, 400);

    // Set the frequency range in Hz you want
    Eigen::ArrayXd f = Eigen::VectorXd::LinSpaced(100, 0.01, 20);

    // Allocate output arrays to contain the spectral tensor
    Eigen::ArrayXXd spectrum_1_1 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd spectrum_1_2 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd spectrum_1_3 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd spectrum_2_2 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd spectrum_2_3 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd spectrum_3_3 = Eigen::ArrayXXd(z.cols(), f.cols());

    // Calculate the spectra
    adem_spectra(
        z,
        f,
        parameters,
        spectrum_1_1,
        spectrum_1_2,
        spectrum_1_3,
        spectrum_2_2,
        spectrum_2_3,
        spectrum_3_3
        );

    // Printing to the command line sucks. Let's do something beautiful...


    return 0;
}
```

Python

```
import numpy as np
import es

def main():

    // Basic test parameters
    pi_coles = 0.42
    kappa = 0.41
    u_inf = 20.0
    shear_ratio = 23.6
    delta = 1000.0

    // Check that it works for a z value scalar
    z = 10.0
    speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
    print('checked scalar operation (U = ', speed, 'm/s)')

    // Check that it works for a numpy array input (vertically spaced z)
    low = 1
```

(continues on next page)

(continued from previous page)

```
    high = 100
    n_bins = 10
    z = np.linspace(low, high, n_bins)
    speed = es.relations.lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio,␣
↪delta)
    print('checked array operation:', speed)

    return
```

2.2.6 Fitting a turbulent spectrum to LiDAR data

Using the Spectral relations provided by the Attached-Detached Eddy Method relation_adem, we can fit the spectrum generated in the ADEM process to a *partial* spectrum obtainable from LiDAR. The spectrum obtainable from LiDAR is of course band limited, so we fit to the valid part of the LiDAR spectrum.

C++

```
#include <Eigen/Dense>
#include <Eigen/Core>

#include "relations/velocity.h"
#include "relations/spectra.h"

using namespace es;

int main(const int argc, const char **argv) {

    // Create simulated 'measured' dataset (see 'Obtaining a spectral profile', above)
    // whose bandwidth goes up to 3Hz, with a noise up to 20% of magnitude applied to␣
↪the 'true' spectra
    double pi_coles = 1.19;
    double kappa = 0.41;
    double u_inf = 20.0;
    double shear_ratio = 28.1;
    double delta = 1000.0;
    double beta = 1.45;
    double zeta = 2.18;
    Eigen::ArrayXd parameters = Eigen::ArrayXd(7);
    parameters << pi_coles, kappa, u_inf, shear_ratio, delta, beta, zeta;
    Eigen::ArrayXd z = Eigen::VectorXd::LinSpaced(40, 1, 500);
    true_u = lewkowicz_speed(z, pi_coles, kappa, u_inf, shear_ratio, delta_c);
    measured_u = true_u + ArrayXd::Random(40) / 4;
    Eigen::ArrayXd f_measured = Eigen::VectorXd::LinSpaced(100, 0.01, 3);
    Eigen::ArrayXd true_u(40);
    Eigen::ArrayXd measured_u(40);
    Eigen::ArrayXXd true_spectrum_1_1 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd true_spectrum_1_2 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd true_spectrum_1_3 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd true_spectrum_2_2 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd true_spectrum_2_3 = Eigen::ArrayXXd(z.cols(), f.cols());
    Eigen::ArrayXXd true_spectrum_3_3 = Eigen::ArrayXXd(z.cols(), f.cols());
    adem_spectra(
        z,
        f,
        parameters,
```

(continues on next page)

(continued from previous page)

```
        true_spectrum_1_1,
        true_spectrum_1_2,
        true_spectrum_1_3,
        true_spectrum_2_2,
        true_spectrum_2_3,
        true_spectrum_3_3
    );
    Eigen::ArrayXXd measured_spectrum_1_1 = true_spectrum_1_1 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);
    Eigen::ArrayXXd measured_spectrum_1_2 = true_spectrum_1_2 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);
    Eigen::ArrayXXd measured_spectrum_1_3 = true_spectrum_1_3 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);
    Eigen::ArrayXXd measured_spectrum_2_2 = true_spectrum_2_2 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);
    Eigen::ArrayXXd measured_spectrum_2_3 = true_spectrum_2_3 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);
    Eigen::ArrayXXd measured_spectrum_3_3 = true_spectrum_3_3 *␣
↪(Eigen::ArrayXXd::Random(z.cols(), f.cols()) * 0.4 + 1);

    // We can use the velocity to fit most of the parameters in the model, so do that␣
↪first (most robust)
    // Here, we set the default fixed parameters (von karman constant and boundary␣
↪layer thickness).
    Eigen::Array<double, 5, 1> fitted_params_u;
    Eigen::Array<bool, 5, 1> fixed_params_u;
    fitted_params_u << 0.5, KAPPA_VON_KARMAN, measured_u.maxCoeff();, 20, 1000;
    fixed_params_u << false, true, false, false, true;
    fit_lewkowicz_speed(z, measured_u, fixed_params_u, fitted_params_u, true);

    // We use those, plus beta=0, zeta=0 (fully equilibrium boundary layer) as an␣
↪initial guess to the spectrum
    // fit - keeping the ones we've already defined fixed.
    Eigen::Array<double, 7, 1> fitted_params_s;
    Eigen::Array<bool, 7, 1> fixed_params_s;
    fitted_params_s <<
        fitted_params_u(0),
        fitted_params_u(1),
        fitted_params_u(2),
        fitted_params_u(3),
        fitted_params_u(4),
        0.0,
        0.0;
    fixed_params_s << true, true, true, true, true, false, false;
    fit_adem_spectra(
        z,
        measured_spectrum_1_1,
        measured_spectrum_1_2,
        measured_spectrum_1_3,
        measured_spectrum_2_2,
        measured_spectrum_2_3,
        measured_spectrum_3_3,
        fixed_params_u,
        fitted_params_u,
        true
    );
    std::cout << "Fitted spectral parameters:" << std::endl;
```

(continues on next page)

(continued from previous page)

```
    std::cout << fitted_params.transpose() << std::endl;

}
```

Python

```
# WARNING - I'm afraid no API for this has been created in python yet. TODO!
def main():
    return
```

2.3 File Formats

Add file format details here.

2.4 Installation

2.4.1 Third party library installation

Intel MKL

Attention: If you don't wish to use Intel MKL, or need to build for a non-intel architecture, please contact Octue.

Download the Intel MKL library packages. Click on the icon and follow installation instructions. You'll need the administrator password.

The tools are installed in `/opt/intel/`, the `include` directory is `/opt/intel/include`.

Intel TBB

Attention: If you don't wish to use Intel TBB, or need to build for a non-intel architecture, please contact Octue.

Mac OSX

brew install tbb

Linux

Download the Intel TBB library packages. Click on the icon and follow installation instructions, ensuring that the TBBROOT environment variable is set. You'll need the administrator password.

The tools are installed in `/opt/intel/`, the `include` directory is `/opt/intel/include`.

Windows

Follow the Intel TBB instructions, ensuring that the TBBROOT environment variable is set.

matio

Mac OSX

Whatever you do, don't try to fork and build from source - the autoconf is complex and not suitable for OSX. Luckily there's a brew formula:

Linux

Please contact Octue for Linux installation instructions.

Windows

Please contact Octue for Windows installation instructions.

ceres-solver, eigen and glog

Mac OSX

Google's ceres-solver also depends on glog and eigen, so we get three for the price of one:

Linux

Please contact Octue for Linux installation instructions.

Windows

Please contact Octue for Windows installation instructions.

2.4.2 Third party build requirements

Attention: These dependencies are only required if you're building **es-flow** from source.

cxxopts

Mac OSX

To build **es-flow**, `cxxopts` must be placed alongside **es-flow**. From the **es-flow** root directory:

Linux

Please contact Octue for Linux installation instructions.

Windows

Please contact Octue for Windows installation instructions.

NumericalIntegration

Mac OSX

To build **es-flow**, NumericalIntegration must be placed alongside **es-flow**. From the **es-flow** root directory:

Linux

Please contact Octue for Linux installation instructions.

Windows

Please contact Octue for Windows installation instructions.

2.5 Turbulence

es-flow is aimed at characterising shear, veer and turbulence in Atmospheric and Marine Boundary Layers. The concepts of shear and veer are well known in the wind industry, but their closely relate cousin, turbulence, is more of a mystery.

For a thorough understanding of turbulence in the ABL, a lengthy academic career is required. But, for the less patient among us, a few notes, recommended readings and resources are made available here - specific to the wind and tidal stream industries.

1. *Coherent Structures*: Notes on coherent ctructures - an explanation and exploration of this author's pet subject, with some extremely pretty pictures.
2. *Turbulent Effects (Wind)*: The "Turbulence Selector (Wind)" - a review of the many, varied ways in which turbulence can affect plant and operations in the wind industry.
3. *Turbulent Effects (Tidal)*: The "Turbulence Selector (Tidal)" - the same, but for the tidal stream industry.

2.5.1 Coherent Structures

TODO

2.5.2 Turbulent Effects (Wind)

Turbulence has a wide range of effects on wind plant and operations. To help understand how turbulence impacts the industry, we put together an 'effects selector'.

TODO - refer to the poster.

The Turbulent Effects Selector

The selector links: (D) design considerations, (S) turbulent scales, (E) physical effects and (N) the nature of the effect. Hover your pointer over the chart. Whichever label you hover over is linked to related issues: red items affect the item; green items are affected by it.

2.5.3 Turbulent Effects (Tidal)

Turbulence has a wide range of effects on tidal stream plant and operations. The Turbulence in Marine Environments (TiME) project resulted in guidance for the tidal stream power industry, giving a comprehensive review of:

1. Techniques for measuring turbulence in tidal streams.
2. Characterisation of turbulence (including a description of several of the relations in **es-flow**).
3. Different turbulent effects that impact operations.

The Turbulent Effects Selector

One output of the TiME project was the "turbulent effects selector". The selector links: (D) design considerations, (S) turbulent scales, (E) physical effects and (N) the nature of the effect. Hover your pointer over the chart. Whichever label you hover over is linked to related issues: red items affect the item; green items are affected by it.

2.6 License

Octue offers free academic licensing for **es-flow** as well as a commercial subscription. Please contact tom@octue.com for details.

2.6.1 Third Party Libraries

es-flow includes or is linked against the following third party libraries:

matio

A library for reading and writing MAT files.

ceres-solver

Ceres Solver is licensed under the New BSD license, whose terms are as follows.

In no event shall Google Inc. be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

Eigen

Eigen is licensed under the Mozilla Public License v2.

NumericalIntegration

An extension library for Eigen that allows numerical integration, used under the Mozilla Public License v2.

cumtrapz

cumtrapz.h header file utility for numerical integration is adapted from The Biomechanical Toolkit.

The Biomechanical ToolKit Copyright (c) 2009-2013, Arnaud Barr?Š All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name(s) of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

cumsum

cumsum.h header file utility for numerical summation is adapted from The libigl geometry library. under MPL2.0 Copyright (C) 2013 Alec Jacobson.

cxxopts

An argument parser for C++11 under the MIT license.

Copyright (c) 2014 Jarryd Beck

2.7 Version History

2.7.1 Origins

EnvironmentSTUDIO flow began as an internal tool at Octue, for characterising turbulence at tidal power sites. It was originated in MATLAB (then called the 'Flow Characterisation Suite'), and grew incrementally.

Unfortunately, MATLAB is extremely unwieldy if deployed in production, requiring either (complicated and badly limited) cross compilation, expensive cloud sever licenses or build and deployment with MATLAB's 3Gb+ runtime library (prohibitively big for practical use in cloud services).

As the move was made toward object orientation, and usage increased toward requiring a production version that could be deployed without requiring expensive MATLAB licenses, EnvironmentSTUDIO flow project was started in C++.

Gradually, capability is being ported from MATLAB (the original repo having been subsumed within this project and now being progressively deprecated).

2.7.2 0.1.0

This version bump was funded by AURA via the University of Hull. The objective of the work was to validate capabilities using measured data, and the library was refactored to allow this work to happen in collaboration with Univ. Hull and the Offshore Renewable Energy Catapult.

New Features

1. Scope of library reduced to preclude specific instrument data readers, and focus on environmental characterisation.
2. Library API consolidated.
3. ADEM functionality ported from legacy MATLAB and tested to work with C++.
4. Added plotting routines (using cpplot).
5. Documentation and build systems implemented and settled.
6. Google Test used to create test harness for the majority of the library.
7. Turbulent effects selector included into the documentation for viewing by teh Offshore Renewable Energy Catapult

Backward Incompatible API Changes

1. Entire API altered to reflect change in scope; will be much more stable going forward.

Bug Fixes & Minor Changes

n/a

2.7.3 0.0.1

Initial library release - development version. Highly unstable!

The work to develop this stable release and CI flow undertaken to support research funded by AURA via the University of Hull (see upcoming version 0.1.0).

New Features

1. C++ based *flow* library, with *cmake* build system
2. Large chunks of WIP for 0.1.0 release, included here because the work in progress was used to generate the doc builds.
3. Octue's legacy *es-flow* MATLAB library (to be deprecated)
4. Outline documentation using ReadTheDocs
5. Autodoc build system (with automated build for releases in travis-ci)

Backward Incompatible API Changes

1. n/a (Initial release)

Bug Fixes & Minor Changes

1. n/a (Initial Release)

2.8 Bibliography

2.9 Library API

2.9.1 Class Hierarchy

2.9.2 File Hierarchy

2.9.3 Full API

Namespaces

Namespace cpplot

Namespace Eigen

Namespace es

Contents

Classes

Enums

Functions

- *Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, const Array5b&, const Array5d&, bool)*
- *Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, bool)*
- *Function es::fit_power_law_speed*
- *Function es::get_mean_speed*
- *Function es::get_reynolds_stresses*
- *Function es::get_spectra*
- *Function es::get_t2w*
- *Function es::getVariable*
- *Template Function es::lewkowicz_speed*
- *Template Function es::marusic_jones_speed*
- *Template Function es::most_law_speed*
- *Function es::NaiveBiotSavart*
- *Template Function es::operator<<(::std::ostream&, const Reader<DataType>&)*
- *Function es::operator<<(::std::ostream&, const Bins&)*
- *Template Function es::operator<<(::std::ostream&, const Profile<ProfileType>&)*
- *Function es::operator<<(std::ostream&, AdemData const&)*
- *Template Function es::power_law_speed*
- *Function es::readArray32d*
- *Function es::readArray3d*
- *Function es::readArrayXd*
- *Function es::readArrayXXd*
- *Function es::readDouble*
- *Function es::readString*
- *Function es::readTensor3d*
- *Function es::readVector3d*
- *Function es::readVectorXd*
- *Function es::reynolds_stress_13*
- *Template Function es::veer_lhs*
- *Template Function es::veer_rhs*
- *Function es::writeArray32d*
- *Function es::writeArray3d*
- *Function es::writeArrayXd*
- *Function es::writeArrayXXd*
- *Function es::writeDouble*
- *Function es::writeString*

- *Function es::writeTensor3d*

Namespace std

Namespace utilities

Contents

Classes

- *Class CubicSplineInterpolant*
- *Class LinearInterpolant*

Functions

- *Template Function utilities::array_to_tensor*
- *Function utilities::conv*
- *Function utilities::convolution_matrix*
- *Template Function utilities::cumsum*
- *Template Function utilities::cumulative_integrate*
- *Template Function utilities::deconv*
- *Function utilities::diagonal_loading_deconv*
- *Template Function utilities::fft_next_good_size*
- *Template Function utilities::filter(Eigen::ArrayBase<DerivedOut>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&)*
- *Template Function utilities::filter(std::vector<T>&, const std::vector<T>&, const std::vector<T>&, const std::vector<T>&)*
- *Function utilities::lowpass_fft_deconv*
- *Template Function utilities::matrix_to_tensor*
- *Template Function utilities::tensor_dims*
- *Template Function utilities::tensor_to_array*
- *Template Function utilities::tensor_to_matrix*
- *Template Function utilities::trapz(Eigen::ArrayBase<OtherDerived> const&, const Eigen::ArrayBase<Derived>&, Eigen::Index)*
- *Template Function utilities::trapz(const Eigen::ArrayBase<Derived>&, Eigen::Index)*

- *Template Function utilities::trapz(Eigen::ArrayBase<DerivedOut> const&, const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)*
- *Template Function utilities::trapz(const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)*

Typedefs

- *Typedef utilities::ArrayType*
- *Typedef utilities::MatrixType*

Classes and Structs

Struct InvalidEddyTypeException

- Defined in *File exceptions.h*

Inheritance Relationships

Base Type

- `public exception`

Struct Documentation

`struct InvalidEddyTypeException` : **`public`** exception

Public Functions

`const` char ***`what`**`()` **`const`**

Public Members

std::string **`message`** = "Unknown eddy type. Eddy type string must be one of 'A', 'B1', 'B2', 'B3' or 'B4'."

Struct LewkowiczSpeedResidual

- Defined in *File fit.h*

Struct Documentation

`struct LewkowiczSpeedResidual`

Cost functor for fitting lewkowicz speed profiles.

Implements operator() as required by ceres-solver. Allows some parameters to be fixed.

Internal use only (see `fit_lewkowicz_speed`).

Public Functions

LewkowiczSpeedResidual(double *z*, double *u*, **const** *Array5b* &*fixed_params*, **const** *Array5d* &*initial_params*)

Construct the cost functor.

Parameters, where supplied in input `initial_params` and masked in input `fixed_params` are in this order:

- `pi_coles = params[0]`
- `kappa = params[1]`
- `u_inf = params[2]`
- `shear_ratio = params[3]`
- `delta_c = params[4]`

Parameters

- `z`: Observation data point, vertical coordinate in m
- `u`: Observation data point, horizontal speed in m/s
- `fixed_params`: Eigen::Array<bool, 5, 1>, logical mask, true where the parameter is fixed, false where it is allowed to vary
- `initial_params`: Eigen::Array<double, 5, 1> containing initial parameter values. These are used where fixed parameters are specified

template <typename T>
bool **operator()**(T **const** ***const** **parameters*, T **residual*) **const**

Struct NotImplementedException

- Defined in *File exceptions.h*

Inheritance Relationships

Base Type

- `public exception`

Struct Documentation

struct NotImplementedException : **public** exception

Public Functions

const char ***what**() **const**

Public Members

std::string **message** = "Not yet implemented"

Struct PowerLawSpeedResidual

- Defined in *File fit.h*

Struct Documentation

struct PowerLawSpeedResidual

Cost functor for fitting power law speed profiles.

Templated for automatic differentiation.

Internal use only (see `fit_power_law_speed`).

Public Functions

PowerLawSpeedResidual(double *z*, double *u*, double *z_ref* = 1.0, double *u_ref* = 1.0)

template <typename T**>**
bool **operator()**(**const** T ***const** *alpha*, T ***residual*) **const**

Class AdemData

- Defined in *File adem.h*

Class Documentation

class AdemData

Data container for ADEM input parameters and results.

Upcoming refactor

The *AdemData* class is treated as a struct, whose contents are updated by a number of functions… But, the functions operating on this should be refactored into class methods. That way, appropriate validation of the contents can be undertaken prior to application of each function.

This refactor is captured in issue #34.

Public Functions

void **load**(std::string *file_name*, bool *print_var* = true)

Load data from a *.mat file containing eddy signature data.

Parameters

- `file_name`: File name (including relative or absolute path)
- `print_var`: Boolean, default true. Print variables as they are read in (not advised except for debugging!)

void **save** (std::string *filename*)
Save eddy signature data to a *.mat file.

Parameters

- `filename`: File name (including relative or absolute path)

Public Members

std::vector<std::string> **eddy_types** = {"A", "B1+B2+B3+B4"}
Eddy types used to create the results.

double **beta**

double **delta_c**
Atmospheric boundary layer thickness δ_c [m].

double **kappa**
von Karman constant κ. Typically $\kappa = 0.41$.

double **pi_coles**
Coles wake parameter Π.

double **shear_ratio**
Ratio between free-stream and skin friction velocities $S = U_{inf}/U_\tau$.

double **u_inf**
Free-stream velocity $U_{inf}|_{z=\delta_c}$) [m/s].

double **u_tau**
Skin friction velocity [m/s].

double **zeta**
Scaled streamwise derivative ζ of the Coles wake parameter Π.

Eigen::VectorXd **z**
Vertical coordinates used in the analysis [m].

Eigen::VectorXd **eta**
Nondimensionalised vertical coordinates used in the analysis $\eta = z/\delta_c$.

Eigen::VectorXd **lambda_e**
Parameterised nondimensional vertical coordinates used in the analysis.

Eigen::VectorXd **u_horizontal**
Horizontal mean velocity varying with vertical coordinate [m/s].

Eigen::ArrayXXd **reynolds_stress**
Reynolds stress profiles from all eddy types.

Eigen::ArrayXXd **r13a_analytic**
Reynolds Stress profile R_{13A} determined analytically from the parameter set.

Eigen::ArrayXXd **r13b_analytic**
Reynolds Stress profile R_{13B} determined analytically from the parameter set.

Eigen::ArrayXXd **reynolds_stress_a**
Reynolds Stress profiles, contributions from Type A eddies only.

Eigen::ArrayXXd **reynolds_stress_b**
Reynolds Stress profiles, contributions from Type B eddies only.

Eigen::ArrayXXd **k1z**
Wavenumbers k_1z for which spectra are defined at each vertical coordinate.

Eigen::Tensor<double, 3> **psi**
Turbulent Spectra Ψ (corresponding to wavenumber `k1z`) at each vertical coordinate.

Eigen::Tensor<double, 3> **psi_a**
Turbulent Spectra (corresponding to wavenumber `k1z`) at each vertical coordinate from Type A eddies only.

Eigen::Tensor<double, 3> **psi_b**
Turbulent Spectra (corresponding to wavenumber `k1z`) at each vertical coordinate from Type B eddies only.

Eigen::VectorXd **t2wa**
Negated convolution function $-(T^2)\omega$ encapsulating variation of eddy strength and scale for Type A eddies.

Eigen::VectorXd **t2wb**
Negated convolution function $-(T^2)\omega$ encapsulating variation of eddy strength and scale for Type B eddies.

Eigen::VectorXd **residual_a**
Fit residuals from the deconvolution of `t2wa`

Eigen::VectorXd **residual_b**
Fit residuals from the deconvolution of `t2wb`

Eigen::Index **start_idx**

Eigen::ArrayXXd **ja_fine**
Reynolds Stress profile R_{13A} determined analytically from the parameter set, finely interpolated.

Eigen::ArrayXXd **jb_fine**
Reynolds Stress profile R_{13B} determined analytically from the parameter set, finely interpolated.

Eigen::ArrayXd **r13a_analytic_fine**
Reynolds Stress profile R_{13A} determined analytically from the parameter set, finely interpolated.

Eigen::ArrayXd **r13b_analytic_fine**
Reynolds Stress profile R_{13B} determined analytically from the parameter set, finely interpolated.

Eigen::ArrayXd **minus_t2wa_fine**
Negated convolution function $-(T^2)\omega$ encapsulating variation of eddy strength and scale for Type A eddies, on the grid of lambda_fine.

Eigen::ArrayXd **minus_t2wb_fine**
Negated convolution function $-(T^2)\omega$ encapsulating variation of eddy strength and scale for Type B eddies, on the grid of lambda_fine.

Eigen::ArrayXd **lambda_fine**
Linearly, finely, spaced values of λ at which final.

Eigen::ArrayXd **eta_fine**
Linearly, finely, spaced values of λ at which final.

Class BasicLidar

- Defined in *File data_types.h*

Class Documentation

class BasicLidar

Public Functions

void **read**(mat_t *matfp, bool *print_var* = true)

Public Members

const std::string **type** = "lidar_basic"

VectorXd **t**

VectorXd **z**

Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> **u**

Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> **v**

Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> **w**

Eigen::Vector3d **position**

double **half_angle**

std::string **t**

std::string **z**

std::string **u**

std::string **v**

std::string **w**

std::string **position**

std::string **half_angle**

struct es::*BasicLidar*::**[anonymous] units**

Class Bins

- Defined in *File profile.h*

Class Documentation

class Bins

Public Functions

Bins(std::vector<double> *z*)

Bins(std::vector<double> *z*, std::vector<double> *dx*, std::vector<double> *dy*, std::vector<double> *dz*)

Bins(std::vector<double> *z*, double *half_angle_degrees*)

Bins(std::vector<double> *z*, double *half_angle_degrees*, std::vector<double> *dz*)

~Bins()

Public Members

unsigned long **n_bins**

std::vector<double> **z_ctrs**

std::vector<double> **dx**

std::vector<double> **dy**

std::vector<double> **dz**

Template Class DeficitFunctor

- Defined in *File stress.h*

Class Documentation

template <typename *T_scalar*, typename *T_param*>
class DeficitFunctor

Public Functions

DeficitFunctor(**const** double *kappa*, **const** T_param *pi_coles*, **const** double *shear_ratio*, **const** bool *lewkowicz*, **const** bool *deficit_squared* = false)

T_scalar **operator()**(T_scalar *eta*) **const**

Class EddySignature

- Defined in *File signature.h*

Class Documentation

class EddySignature

Data container for Eddy signature tensors.

Public Functions

void **load**(std::string *file_name*, bool *print_var* = false)

Load data from a *.mat file containing eddy signature data.

TODO overload with load(std::vector<std::string> file_names, bool print_var = false){} to load and average multiple signature files

Parameters

- `file_name`: File name (including relative or absolute path)
- `print_var`: Boolean, default true. Print variables as they are read in (not advised except for debugging!)

void **save**(std::string *file_name*)

Save eddy signature data to a *.mat file.

Parameters

- `filename`: File name (including relative or absolute path)

EddySignature **operator+**(**const** *EddySignature* &*c*) **const**

Define overloaded + (plus) operator for eddy signatures.

Return A new EddySignature() with combined signatures of the two eddy types.

Parameters

- `c`: The *EddySignature* to add.

EddySignature **operator***(double *a*) **const**

Define overloaded * (multiply) operator for eddy signatures.

Return new EddySignature() whose signature (g, j) is element-wise multiplied by input a.

Parameters

- `a`: The number to multiply by

EddySignature **operator/**(double *denom*) **const**

Define overloaded / (divide) operator for eddy signatures.

Return A new EddySignature() whose signature (g, j) is element-wise divided by input denom.

Parameters

- `denom`: A number to divide by

Eigen::ArrayXXd **k1z**(Eigen::ArrayXd &*eta*) **const**

Get k1z wavenumber array (wavenumber space for each vertical coord, e.g. 50 x 801).

Return k1z the wavenumber-length matrix

Parameters

- `eta`: vertical heights at which to get the k1z value, normalised (i.e. z/delta)

Eigen::ArrayXXd **getJ**(**const** Eigen::ArrayXd &*locations*, **const** bool *linear* = false) **const**

Interpolate the signature $J_{i,j}(\lambda)$ to new locations in lambda or eta.

To avoid warping the reconstruction, we need to do the convolution and the deconvolution with both the input and the signature on the same, equally spaced, basis. What if we want to do it on a basis different to the positions where the signature was calculated?

This function allows you to get the signature array, j, at different vertical locations

Return [N x 6] signature array Jij interpolated to input locations

Parameters

- `locations`: [N x 1] The lambda (or optionally eta) values at which to retrieve the signature j
- `linear`: boolean If true, locations are on a linear basis (i.e. they are values of eta). Otherwise (default) they are on a logarithmic basis (i.e. they are values of lambda).

void **`computeSignature`**(**`const`** std::string &*type*, **`const`** int *n_lambda* = 200, **`const`** double *dx* = 0.002)

Calculate eddy intensity functions $J_{i,j}$ and $g_{i,j}$.

Eddy intensity functions $J_{i,j}(\lambda)$ are computed for type `A`, `B1`, `B2`, `B3` or `B4`, which are the eddies described in Ref 1. These functions are the signatures (in terms of turbulent fluctuations) that an individual structure will contribute to Reynolds Stress in a boundary layer. They are used in the convolution integral (36) of Perry and Marusic (1995a) to determine Reynolds Stress profiles.

Eddy spectral functions $g_{i,j}(k1z, \lambda)$ are then computed from J. These functions are the signatures (in terms of turbulent fluctuations) that an individual structure will contribute to turbulent spectra in a boundary layer. They are used in the convolution integral (43) of Perry and Marusic (1995a) to determine Spectral Tensor profiles.

This method updates class properties `j`, `lambda`, `eta`, `g`, `eddy_type`, `domain_spacing`, `domain_extents`.

It works by computing induced velocity [u, v, w] of a single eddy structure on a regular grid [x,y,lambda], normalised to the eddy scale and surrounding the unit eddy, then integrating for J and taking ffts for g.

Perry AE and Marusic I (1995a) A wall-wake model for turbulent boundary layers. Part 1. Extension of the attached eddy hypothesis J Fluid Mech vol 298 pp 361-388

Parameters

- `typestring`: one of 'A', 'B1', 'B2', 'B3', 'B4'
- `n_lambdaint`: number of points logarithmically spaced in the z direction, between the outer part of the domain and a location very close to the wall. Default 200.

void **`applySignature`**(**`const`** std::string &*type*, **`const`** int *n_lambda* = 200)

Apply signature defaults.

Eddy intensity functions $J_{i,j}(\lambda)$ is applied for type `A` or `B` eddies, by digitisation of Figure 20 in Perry and Marusic (1995).

Note that these signatures are incomplete:

1. Only four terms are given (I11, I13, I22, I33, the remaining two are assumed to be zero)
2. Eddy spectral functions $g_{i,j}(k1z, \lambda)$ are not documented in P&M 1995, so can't be reproduced here. Thus, signatures defined by this method cannot be used for computation of spectra (and sets arbitrary `domain_spacing` and `domain_extents` values for the `x` and `y` directions).

This method updates class properties `j`, `lambda`, `eta`, `eddy_type`, `domain_spacing`, `domain_extents`.

See Perry AE and Marusic I (1995) A wall-wake model for turbulent boundary layers. Part 1. Extension of the attached eddy hypothesis J Fluid Mech vol 298 pp 361-388

Parameters

- `typestring`: one of 'A', 'B1', 'B2', 'B3', 'B4'

- `n_lambdaint`: number of points logarithmically spaced in the z direction, between the outer part of the domain and a location very close to the wall. Default 200.

Public Members

std::string **eddy_type**
Eddy type (or types, if ensembled) used to create the results.

Eigen::ArrayXd **lambda**
Mapped vertical coordinates used in the analysis.

Eigen::ArrayXd **eta**
Unmapped (cartesian space) coordinates corresponding to the points in `lambda`.

Eigen::Tensor<double, 3> **g**
g (6 coefficients at each vertical coord and wavenumber, e.g 50 x 801 x 6)

Eigen::Array<double, Eigen::Dynamic, 6> **j**
Eddy intensity functions $Jij(lambda)$, which is [n_lambda x 6] in size, with columns ordered as `[J11 J12 J13 J22 J23 J33]`. Note that the lower diagonal of the 3x3 Reynolds Stress Tensor is not included due to symmetry.

Eigen::Array3d **domain_spacing**
Spacing of the regular grid used to create the eddy intensity signatures [dx, dy, dlambda] (note for the third coordinate, points are linearly spaced in a logarithmic domain, so this is d_lambda, not d_z).

Eigen::Array<double, 3, 2> **domain_extents**
Extents of the regular grid placed over the unit eddy to create the eddy intensity signatures, in [x_min, x_max; y_min, y_max; z_min, z_max] form.

Template Class Profile

- Defined in *File profile.h*

Class Documentation

template <class *ProfileType*>
class Profile

Public Functions

Profile(**const** *Bins* &*bins*)

Profile(**const** *Bins* &*bins*, double *x*, double *y*, double *z*)

~Profile()

const std::vector<ProfileType> &**getValues**() **const**

void **setValues**(**const** std::vector<ProfileType> &*values*)

Public Members

template<>
double **position**[3]

Bins **bins**

Template Class Reader

- Defined in *File readers.h*

Class Documentation

template <class DataType>
class Reader

Public Functions

Reader(**const** std::string *&file*)

~Reader()

std::string **logString**() **const**

DataType ***read**(bool *print* = false)

void **checkFileType**(int *level*)

void **checkTimeseries**(int *level*)

double **getWindowDuration**() **const**

void **setWindowDuration**(double *windowDuration*)

int **getWindowSize**() **const**

void **setWindowSize**(int *windowSize*)

void **readWindow**(**const** int *index*)

Protected Attributes

std::string **file**

mat_t ***matfp**

std::string **file_type** = std::string("none")

int **windowSize**

double **windowDuration**

DataType **data**

Class VelocityProfile

- Defined in *File profile.h*

Inheritance Relationships

Base Type

- `public es::Profile< double >` (*Template Class Profile*)

Class Documentation

`class VelocityProfile` : **`public`** es::*Profile*<double>

Public Functions

`VelocityProfile`()

`virtual ~VelocityProfile`()

Class spectra

- Defined in *File spectra.h*

Class Documentation

`class spectra`

Class CubicSplineInterpolant

- Defined in *File interp.h*

Class Documentation

`class CubicSplineInterpolant`

Public Functions

`CubicSplineInterpolant`(Eigen::VectorXd **`const`** &*x_vec*, Eigen::VectorXd **`const`** &*y_vec*)

Performs cubic interpolation (or maximum degree possible, where inputs are shorter than 4 elements)

Initialise an interpolant as follows:

Eigen::VectorXd xvals(3); Eigen::VectorXd yvals(xvals.rows());

xvals << 0, 15, 30; yvals << 0, 12, 17;

CubicSplineInterpolant s(xvals, yvals);

double **operator()** (double *xi*) **const**

Evaluate interpolant at values xi whose yi values are unknown.

Performs cubic interpolation (or maximum degree possible, where inputs are shorter than 4 elements)

Initialise and evaluate an interpolant:

// ... Initialise interpolant (see constructor) ... *CubicSplineInterpolant* s(xvals, yvals); std::cout << s(12.34) << std::endl;

Return double interpolated value yi corresponding to location xi

Parameters

- `xi`: double (or eigen VectorXd) of target x values for the interpolation

Eigen::VectorXd **operator()** (Eigen::VectorXd **const** &*xi_vec*)

Class LinearInterpolant

- Defined in *File interp.h*

Class Documentation

class LinearInterpolant

Public Functions

LinearInterpolant (**const** Eigen::ArrayXd *x_vec*, **const** Eigen::ArrayXd *y_vec*)

Create a linear interpolant.

Initialise an interpolant as follows:

Eigen::ArrayXd xvals(3); Eigen::ArrayXd yvals(xvals.rows());

xvals << 0, 15, 30; yvals << 0, 12, 17;

LinearInterpolant s(xvals, yvals);

double **operator()** (double *xi*) **const**

Evaluate interpolant at values xi whose yi values are unknown.

Out of range xi values are constrained to the endpoints (i.e. nearest neighbour interpolation)

Performs linear interpolation to evaluate values yi at // ... Initialise interpolant (see constructor) ... *LinearInterpolant* s(xvals, yvals); std::cout << s(12.34) << std::endl;

Return double interpolated value yi corresponding to location xi

Parameters

- `xi`: double (or eigen ArrayXd) of target x values for the interpolation

Eigen::ArrayXd **operator()** (Eigen::ArrayXd **const** &*xi_vec*)

Enums

Enum file_type_check_level

- Defined in *File readers.h*

Enum Documentation

enum `es::`**file_type_check_level**

Values:

NONE = 0

OAS_STANDARD = 1

Enum timeseries_check_level

- Defined in *File readers.h*

Enum Documentation

enum `es::`**timeseries_check_level**

Values:

PRESENT = 0

INCREASING = 1

MONOTONIC = 2

STRICTLY_MONOTONIC = 3

Functions

Function es::adem

- Defined in *File adem.h*

Function Documentation

AdemData `es::`**adem**(**const** double *beta*, **const** double *delta_c*, **const** double *kappa*, **const** double *pi_coles*, **const** double *shear_ratio*, **const** double *u_inf*, **const** double *zeta*, **const** *EddySignature* &*signature_a*, **const** *EddySignature* &*signature_b*, bool *compute_spectra* = true)

Compute full turbulent properties from given Attached-Detached Eddy Model parameters.

Uses the using the Lewkowicz (1982) formulation (Perry and Marusic eq.9) of the Coles wake function to determine u_h(z), spectra and Reynolds Stresses from input parameters.

Parameters

- `beta`: The Clauser parameter, representing acceleration/decelaration of the boundary layer

- `delta_c`: The boundary layer thickness in m
- `kappa`: The von Karman constant, typically 0.41.
- `pi_coles`: Coles wake parameter Pi
- `shear_ratio`: Ratio between free stream and friction velocity S = u_inf/u_tau
- `u_inf`: The free stream speed in m/s
- `zeta`: Represents a scaled streamwise derivative of Pi
- `signature_a`: *EddySignature* class loaded with data for type A eddies
- `signature_b`: *EddySignature* class loaded with data for an ensemble of type B eddies

Template Function es::array_size

- Defined in *File variable_readers.h*

Function Documentation

template <size_t *SIZE*, **class** T>
size_t `es::`**array_size**(T (&*arr*)[SIZE])

Return the size (i.e. number of elements) in an array variable Comes from https://coderwall.com/p/nb9ngq/better-getting-array-size-in-c

Example: int arr[] = {1, 2, 3, 4, 5}; std::cout << array_size(arr) << std::endl;

Function es::checkVariableType

- Defined in *File variable_readers.h*

Function Documentation

void `es::`**checkVariableType**(matvar_t **mat_var*, int *matvar_type*)

Template Function es::coles_wake

- Defined in *File velocity.h*

Function Documentation

template <**typename** T_z, **typename** T_param>
T_z `es::`**coles_wake**(T_z **const** &*eta*, T_param **const** &*pi_coles*, **const** bool *corrected* = true)

Compute corner-corrected coles wake parameter W_c .

There are two alternative functional forms for the wake parameter. Perry and Marusic (1995) used the Lewkowicz (1982) wake function, where the second (1/...) term is artificially applied to ensure that the velocity defect function has zero gradient at z= :

$$W_c[\eta, \Pi] = 2\eta^2 (3 - 2\eta) - \frac{1}{\Pi}\eta^2 (1 - \eta)(1 - 2\eta)$$

Note: Jones, Marusic and Perry 2001 refer to this correction as the 'corner' function and use an alternative correction (recommended by Prof. Coles) to the pure wall flow (instead of the wake) which is not a function of Pi, allowing reversion to the original (Coles 1956) wake parameter W :

$$W[\eta, \Pi] = 2\eta^2 (3 - 2\eta)$$

This function implements the former by default!

This function is templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (the latter via template specialisation) of z values.

Parameters

- `eta`: Nondimensional height values
- `pi_coles`: The coles wake parameter Pi
- `corrected`: Boolean flag, default true. If true, return Lewkowicz (1982) corrected wake function. If false, return the original uncorrected wake parameter.

Template Function es::deficit

- Defined in *File velocity.h*

Function Documentation

template <typename T_eta, **typename** T_param>
T_eta `es::deficit`(T_eta **const** &*eta*, **const** double *kappa*, T_param **const** &*pi_coles*, **const** double *shear_ratio*, **const** bool *lewkowicz* = false)

Get the velocity deficit integrand f used in computation of R_{13}.

TODO move into velocity.h, reuse in lewkowicz_speed and jones_speed functions;. In their original derivation of the ADEM, Perry and Marusic 1995 eqn. 2 use the velocity distribution of Lewkowicz 1982 as a basis to determine the shear stress profile. This led to an integrand for the mean velocity deficit profile:

$$f = \frac{U_1 - \overline{U}}{U_\tau} = -\frac{1}{\kappa}\ln(\eta) + \frac{\Pi}{\kappa}W_c[1, \Pi] - \frac{\Pi}{\kappa}W_c[\eta, \Pi],$$

$$W_c[\eta, \Pi] = 2\eta^2 (3 - 2\eta) - \frac{1}{\Pi}\eta^2 (1 - \eta)(1 - 2\eta)$$

Jones, Marusic and Perry (2001) eqn 1.9 uses an alternative formulation, thereby removing the non-physical dependency of the wake factor on . This leads to the velocity deficit function:

$$f = \frac{U_1 - \overline{U}}{U_\tau} = -\frac{1}{\kappa}\ln(\eta) + \frac{1}{3\kappa}\left(\eta^3 - 1\right) + 2\frac{\Pi}{\kappa}\left(1 - 3\eta^2 + 2\eta^3\right)$$

This function is templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (the latter via template specialisation) of z values.

Return

Parameters

- `eta`: Nondimensional height values
- `kappa`: von Karman constant
- `pi_coles`: The coles wake parameter Pi
- `shear_ratio`: Shear ratio S
- `lewkowicz`: Boolean flag, default false. If true, use Lewkowicz (1982) velocity deficit funciton. If false, use Jones et at (2001).

Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, const Array5b&, const Array5d&, bool)

- Defined in *File fit.h*

Function Documentation

Array5d `es::`**`fit_lewkowicz_speed`**`(`**`const`** Eigen::ArrayXd &*z*, **`const`** Eigen::ArrayXd &*u*, **`const`** *Array5b* &*fixed_params*, **`const`** *Array5d* &*initial_params*, bool *print_report* = true`)`

Determine best fit of Lewkowicz relation to input speed profile.

Robustly fits Lewkowicz speed relation $u = f(z, \Pi, \kappa, U_{inf}, S, \delta_c)$ to input data u, z.

An initial guess is required, and parameters can be fixed to help constrain the fitting process. See also fit_lewkowicz_speed(const Eigen::ArrayXd &z, const Eigen::ArrayXd &u, bool print_report = true) for providing a set of default values and fixed parameters.

Parameters are as follows:

- `pi_coles = params[0]`
- `kappa = params[1]`
- `u_inf = params[2]`
- `shear_ratio = params[3]`
- `delta_c = params[4]`

The fitting process uses a Levenberg-Marquadt solver (provided by `ceres-solver`) with Automatic differentiation (for improved numeric stability and convergence over numerical differentiation approaches), and a Cauchy Loss function of parameter 0.5 (for robustness agains outlying data entries).

Return Array5d containing fitted values `[pi_coles, kappa, u_inf, shear_ratio, delta_c]`

Parameters

- `z`: Vertical locations in m (or normalised, if z_ref = 1.0)
- `u`: Observed speeds at corresponding vertical locations, in m/s
- `fixed_params`: Eigen::Array<bool, 5, 1>, logical mask, true where the parameter is fixed, false where it is allowed to vary
- `initial_params`: Eigen::Array<double, 5, 1> containing initial parameter values. These are used where fixed parameters are specified
- `print_report`: bool, default true. If true, the solver prints iterations summary and full final convergence and solution report (for debugging and validation purposes).

Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, bool)

- Defined in *File fit.h*

Function Documentation

Array5d `es::fit_lewkowicz_speed(const` Eigen::ArrayXd *&z*, `const` Eigen::ArrayXd *&u*, bool *print_report* = true`)`

Determine best fit of Lewkowicz relation to input speed profile.

Robustly fits Lewkowicz speed relation $u = f(z, \Pi, \kappa, U_{inf}, S, \delta_c)$ to input data u, z, using 'sensible' initial guesses (for Atmospheric Boundary Layer in Northern European conditions) and fixed parameters to constrain the fitting process:

```
- von Karman constant \form#3 is fixed according to the value in definitions.h.
- Boundary layer thickness \form#24 is fixed to a default value of ``1000 m``.
```

Return Array5d containing fitted values `[pi_coles, kappa, u_inf, shear_ratio, delta_c]`

Parameters

- `z`: Vertical locations in m (or normalised, if z_ref = 1.0)
- `u`: Observed speeds at corresponding vertical locations, in m/s
- `print_report`: bool, default true. If true, the solver prints iterations summary and full final convergence and solution report (for debugging and validation purposes).

Function es::fit_power_law_speed

- Defined in *File fit.h*

Function Documentation

double `es::fit_power_law_speed`(**const** Eigen::ArrayXd *&z*, **const** Eigen::ArrayXd *&u*, **const** double *z_ref* = 1.0, **const** double *u_ref* = 1.0)

Determine best fit of power law to input speed profile.

Return

Parameters

- `z`: Vertical locations in m (or normalised, if z_ref = 1.0)
- `u`:
- `z_ref`: Optional reference height (default 1.0) by which the input z is normalised
- `u_ref`: Optional reference velocity (default 1.0) by which the input u is normalised

Function es::get_mean_speed

- Defined in *File adem.h*

Function Documentation

void `es::get_mean_speed`(*AdemData &data*)

Get the mean speed profile and update the data structure with it.

Parameters

- `data`:

Function es::get_reynolds_stresses

- Defined in *File adem.h*

Function Documentation

void `es::get_reynolds_stresses`(*AdemData &data*, **const** *EddySignature &signature_a*, **const** *EddySignature &signature_b*)

Get the Reynolds Stress distributions from T2w and J distributions.

The ouptut Reynolds Stress matrix is of size output_dim_size = input_dim_size - kernel_dim_size + 1 (requires: input_dim_size >= kernel_dim_size). Legacy MATLAB equivalent is:

```
[R, RA, RB] = getReynoldsStresses(T2wA, T2wB, JA, JB);
```

Parameters

- `data`:

Function es::get_spectra

- Defined in *File adem.h*

Function Documentation

void `es::get_spectra`(*AdemData* &*data*, **const** *EddySignature* &*signature_a*, **const** *EddySignature* &*signature_b*)

Get the premultiplied power spectra.

Computes tensors Ψ_a (`psi_a`) and Ψ_b (`psi_b`) from the scale functions and the eddy signatures, and adds them to the *AdemData* object.

The output spectral tensors have dimension [nz x nk x 6], where `nz` should agree with the number of elements in the `t2w` scale functions, and `nk` represents the number of wavenumbers for which the spectrum is computed.

Parameters

- `data`: *AdemData* object, which must have properties `t2wa`, 't2wb' and `u_tau` defined already.
- `signature_a`: Signature object for Type A eddies
- `signature_b`: Signature object for Type B eddies

Function es::get_t2w

- Defined in *File adem.h*

Function Documentation

void `es::get_t2w`(*AdemData* &*data*, **const** *EddySignature* &*signature_a*, **const** *EddySignature* &*signature_b*)

Get the T^2w distributions from the eddy signatures by deconvolution.

These distributions are used for calculation of Spectra and Stress terms.

Updates `t2wa`, `t2wb`, `lambda_fine`, `residual_a` and `residual_b` in the input data structure.

Parameters

- `data`:
- `signature_a`:
- `signature_b`:

Function es::getVariable

- Defined in *File variable_readers.h*

Function Documentation

matvar_t *`es::getVariable`(mat_t **matfp*, **const** std::string *var_name*, bool *print_var* = true, int *max_rank* = 2)

Get pointer to a variable and check validity. Optionally print variable and check rank.

Return

Parameters

- `matfp`:
- `var_name`:
- `print_var`:
- `max_rank`: The maximum rank of the variable. Default 2 as everything is an array in MATLAB.

Template Function es::lewkowicz_speed

- Defined in *File velocity.h*

Function Documentation

template <typename T_z, **typename** T_param>
T_z `es::`**lewkowicz_speed**(T_z **const** &*z*, T_param **const** &*pi_coles*, T_param **const** &*kappa*, T_param **const** &*u_inf*, T_param **const** &*shear_ratio*, T_param **const** &*delta_c*)

Compute Lewkowicz (1982) velocity profile.

TODO refactor to base it on velocity deficit, keep code DRY TODO Refactor to find usages as VectorXd and change them to array uses; remove the VectorXd template specialization

Used by Perry and Marusic 1995 (from eqs 2 and 7).

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of z values.

Parameters

- `z`: height in m (or Nondimensional heights (eta = z/delta_c) where delta_c = 1.0)
- `pi_coles`: The coles wake parameter Pi
- `kappa`: von Karman constant
- `u_inf`: Speed of flow at z = delta (m/s)
- `shear_ratio`: Shear / skin friction velocity ratio (shear_ratio = u_inf / u_tau)
- `delta_c`: Boundary layer thickness in m, used to normalise z. Defaults to 1.0.

Template Function es::marusic_jones_speed

- Defined in *File velocity.h*

Function Documentation

template <typename T_z, **typename** T_pi_j>
T_z `es::`**marusic_jones_speed**(T_z **const** &*z*, T_pi_j **const** *pi_j*, **const** double *kappa*, **const** double *z_0*, **const** double *delta*, **const** double *u_inf*, **const** double *u_tau*)

Compute speed profile according to Marusic and Jones' relations.

TODO refactor to base it on velocity deficit, keep code DRY

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of z values.

Speed is computed as:

$$\frac{\overline{U}}{U_\tau} = \frac{1}{\kappa}\ln\left(\frac{z+z_0}{k_s}\right) + Br + \frac{\Pi_j}{\kappa}W_c[\eta,\Pi_j]$$

$$W_c[\eta,\Pi_j] = 2\eta^2\left(3-2\eta\right) - \frac{1}{3\Pi_j}\eta^3$$

$$\eta = \frac{z+z_0}{\delta+z_0}$$

which reduces to the defecit relation:

$$U_D^* = \frac{U_\infty - \overline{U}}{U_\tau} = -\frac{1}{\kappa}\ln\left(\eta\right) + \frac{1}{3\kappa}\left(\eta^3-1\right) + 2\frac{\Pi_j}{\kappa}\left(1-3\eta^2+2\eta^3\right)$$

Parameters

- `z`: Height(s) in m at which you want to get speed.
- `pi_j`: Jones' modification of the Coles wake factor. Double or AutoDiffScalar type accepted.
- `kappa`: von Karman constant
- `z_0`: Roughness length - represents distance of hypothetical smooth wall from actual rough wall z0 = 0.25k_s (m)
- `delta`: Boundary layer thickness (m)
- `u_inf`: Speed of flow at z = delta (m/s)
- `u_tau`: Shear / skin friction velocity (governed by ratio parameter S = u_inf / u_tau)

Template Function es::most_law_speed

- Defined in *File velocity.h*

Function Documentation

template <typename T>
T `es::`**most_law_speed**(T **const** &*z*, **const** double *kappa*, **const** double *d*, **const** double *z0*, **const** double *L*)

Compute speed profile according to the MOST law.

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of z values.

MOST law speed is computed as:

TODO

Parameters

- `z`: Height value(s) at which you want to get speed (m)
- `kappa`: von Karman constant
- `d`: Zero plane offset distance (e.g. for forest canopies) (m)
- `z0`: Roughness length (m)
- `L`: Monin-Obukhov length (m)

Function es::NaiveBiotSavart

- Defined in *File biot_savart.h*

Function Documentation

Matrix3Xd `es::`**`NaiveBiotSavart`**(Matrix3Xd *startNodes*, Matrix3Xd *endNodes*, Matrix3Xd *locations*, VectorXd *gamma*, VectorXd *effective_core_radius_squared*)

O(N^2) 'Naive' application of the biot savart law to determine the action of many vortex lines on many control points.

Outputs:

Parameters

- `startNodes`: [3 x n] Start nodes of vortex lines in x, y, z coords
- `endNodes`: [3 x n] End nodes of vortex lines in x, y, z coords
- `locations`: [3 x p] Locations in x, y, z coords at which induction is required
- `gamma`: [n x 1] Circulation of each vortex line
- `rcEffSqd`: [n x 1] Square of the effective vortex core radius for each vortex line

induction[3 x p] double Induction in u, v, w directions of the input vortex lines at the input control points

Function es::operator<<(std::ostream&, AdemData const&)

- Defined in *File adem.h*

Function Documentation

std::ostream &`es::`**`operator<<`**(std::ostream &*os*, *AdemData* **const** &*data*)

Print information about the *AdemData* attributes to ostream using the << operator.

Return os The same ostream, with the data representation added to the stream

Parameters

- `os`: The ostream to print to
- `data`: The AdemDataclass instance

Template Function es::operator<<(::std::ostream&, const Reader<DataType>&)

- Defined in *File readers.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "es::operator<<" with arguments (::std::ostream&, const Reader<DataType>&) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- std::ostream &es::operator<<(std::ostream&, AdemData const&)
- std::ostream &es::operator<<(std::ostream&, const Bins&)
- template <class DataType>
  std::ostream &es::operator<<(std::ostream&, const Reader<DataType>&)
- template <class ProfileType>
  std::ostream &es::operator<<(std::ostream&, const Profile<ProfileType>&)
```

Function es::operator<<(::std::ostream&, const Bins&)

- Defined in *File profile.cpp*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "es::operator<<" with arguments (::std::ostream&, const Bins&) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- std::ostream &es::operator<<(std::ostream&, AdemData const&)
- std::ostream &es::operator<<(std::ostream&, const Bins&)
- template <class DataType>
  std::ostream &es::operator<<(std::ostream&, const Reader<DataType>&)
- template <class ProfileType>
  std::ostream &es::operator<<(std::ostream&, const Profile<ProfileType>&)
```

Template Function es::operator<<(::std::ostream&, const Profile<ProfileType>&)

- Defined in *File profile.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "es::operator<<" with arguments (::std::ostream&, const Profile<ProfileType>&) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- std::ostream &es::operator<<(std::ostream&, AdemData const&)
- std::ostream &es::operator<<(std::ostream&, const Bins&)
- template <class DataType>
  std::ostream &es::operator<<(std::ostream&, const Reader<DataType>&)
- template <class ProfileType>
  std::ostream &es::operator<<(std::ostream&, const Profile<ProfileType>&)
```

Template Function es::power_law_speed

- Defined in *File velocity.h*

Function Documentation

template <typename T, **typename** Talf>
T `es::`**power_law_speed**(T **const** &*z*, **const** double *u_ref*, **const** double *z_ref*, Talf **const** &*alpha*)

Compute speed profile according to the power law.

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of z values (to obtain shear profile u/dz) and/or alpha values (to obtain variational jacobean for fitting parameter alpha).

Power law speed is computed as:

$$\frac{\overline{U}}{\overline{U}_{ref}} = \left(\frac{z}{z_{ref}}\right)^{\alpha}$$

Parameters

- `z`: Height(s) in m at which you want to get speed.
- `u_ref`: Reference speed in m/s
- `z_ref`: Reference height in m
- `alpha`: Power law exponent. Must be of same type as input z (allows autodifferentiation).

Function es::readArray32d

- Defined in *File variable_readers.h*

Function Documentation

Array32d `es::`**readArray32d**(mat_t **matfp*, **const** std::string *var_name*, bool *print_var*)

Function es::readArray3d

- Defined in *File variable_readers.h*

Function Documentation

Array3d `es::`**readArray3d**(mat_t **matfp*, **const** std::string *var_name*, bool *print_var*)

Function es::readArrayXd

- Defined in *File variable_readers.h*

Function Documentation

ArrayXd `es::readArrayXd`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readArrayXXd

- Defined in *File variable_readers.h*

Function Documentation

ArrayXXd `es::readArrayXXd`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readDouble

- Defined in *File variable_readers.h*

Function Documentation

double `es::readDouble`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readString

- Defined in *File variable_readers.h*

Function Documentation

std::string `es::readString`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readTensor3d

- Defined in *File variable_readers.h*

Function Documentation

Tensor<double, 3> `es::readTensor3d`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readVector3d

- Defined in *File variable_readers.h*

Function Documentation

Vector3d `es::readVector3d`(mat_t *matfp, `const` std::string *var_name*, bool *print_var*)

Function es::readVectorXd

- Defined in *File variable_readers.h*

Function Documentation

VectorXd `es::readVectorXd`(mat_t **matfp*, **const** std::string *var_name*, bool *print_var*)

Function es::reynolds_stress_13

- Defined in *File stress.h*

Function Documentation

void `es::reynolds_stress_13`(Eigen::ArrayXd &*r13_a*, Eigen::ArrayXd &*r13_b*, **const** double *beta*, **const** Eigen::ArrayXd &*eta*, **const** double *kappa*, **const** double *pi_coles*, **const** double *shear_ratio*, **const** double *zeta*)

Compute Horizontal-Vertical Reynolds Stress R13 profile.

Gets Reynolds Stress profiles due to Type A and B eddies.

Can use either the Lewkowicz formulation for velocity deficit (per Perry and Marusic 1995) or

TODO - presently, the integrations here are sensitive to the chosen eta distribution. Refactor according to issue #38

TODO - Determine whether it will be an advantage to template this function so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of eta values.

References

[1] Perry AE and Marusic I (1995) A wall-wake model for turbulent boundary layers. Part 1. Extension of the attached eddy hypothesis J Fluid Mech vol 298 pp 361-388

Future Improvements

[1] Optional different mean profile formulations including account for the free surface

[2] Support directional variation with height; i.e. compatible with mean profile formulations using U(y)

[3] Added formulation for contribution of smaller Type C eddies

[4] For the given wall formulation, can f be calculated more efficiently? See schlichting and gersten p. 593

Parameters

- `beta`: Clauser parameter , representing acceleration/decelaration of the boundary layer
- `eta`: Nondimensional vertical coordinates at which you want to get stress R_{13}. Values must ascend but not necessarily be monotonic.
- `kappa`: von Karman constant.
- `pi_coles`: Coles wake parameter Π
- `shear_ratio`: Ratio between free-stream and skin friction velocities $S = U_{inf}/U_\tau$
- `zeta`: Scaled streamwise derivative ζ of the Coles wake parameter Π

Template Function es::veer_lhs

- Defined in *File veer.h*

Function Documentation

template <typename T>
T `es::veer_lhs`(T **const** *&ui*, **const** double *ui_g*, **const** double *phi*)

Computes the left hand side of the veer relations given u(z) or v(z).

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of u values.

The veer relations are:

$$2|\mathbf{\Omega}|sin(\phi)(\overline{v}_g - \overline{v}) = \frac{\partial}{\partial z}\left(\nu\frac{\partial\overline{u}}{\partial z} - \overline{u'w'}\right)$$
$$2|\mathbf{\Omega}|sin(\phi)(\overline{u}_g - \overline{u}) = \frac{\partial}{\partial z}\left(\nu\frac{\partial\overline{v}}{\partial z} - \overline{v'w'}\right)$$

Parameters

- `ui`: Mean velocity component at a given height (m/s)
- `ui_g`: Mean geostrophic velocity component outside the atmospheric boundary layer (m/s)
- `phi`: Latitude (degrees)

Template Function es::veer_rhs

- Defined in *File veer.h*

Function Documentation

template <typename T>
T `es::veer_rhs`(T *&ui*, T *&uiu3_bar*, **const** double *nu*)

Computes the right hand side of the veer relations given u(z) or v(z).

See `veer_lhs()` for the full relation.

Templated so that it can be called with active scalars (allows use of autodiff), doubles/floats, Eigen::Arrays (directly) or Eigen::VectorXds (via template specialisation) of u values.

Parameters

- `ui`: Mean velocity component at a given height (m/s){
- `uiu3_bar`: Mean cross term of unsteady velocity components $\overline{u_i'u_3'}$ (m^2/s^2)
- `nu`: Kinematic viscosity of the fluid (m^2/s)

Function es::writeArray32d

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeArray32d`(mat_t *mat_fp, **const** std::string &var_name, **const** Eigen::Array<double, 3, 2> &var)

Function es::writeArray3d

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeArray3d`(mat_t *mat_fp, **const** std::string &var_name, **const** Eigen::Array3d &var)

Function es::writeArrayXd

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeArrayXd`(mat_t *mat_fp, **const** std::string &var_name, **const** Eigen::ArrayXd &var)

Function es::writeArrayXXd

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeArrayXXd`(mat_t *mat_fp, **const** std::string &var_name, **const** Eigen::ArrayXXd &var)

Function es::writeDouble

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeDouble`(mat_t *mat_fp, **const** std::string &var_name, **const** double var)

Function es::writeString

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeString`(mat_t *mat_fp, `const` std::string &var_name, std::string &var)

Function es::writeTensor3d

- Defined in *File variable_writers.h*

Function Documentation

void `es::writeTensor3d`(mat_t *mat_fp, `const` std::string &var_name, `const` Tensor<double, 3> &var)

Function main

- Defined in *File main.cpp*

Function Documentation

Warning: doxygenfunction: Cannot find function "main" in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml

Template Function utilities::array_to_tensor

- Defined in *File tensors.h*

Function Documentation

`template <typename` Scalar, typename... *Dims*>
auto `utilities::array_to_tensor`(`const` ArrayType<Scalar> &matrix, Dims... *dims*)

convert Eigen Tensor<> to an Array<>

```
int main () {
    Eigen::Tensor<double,4> my_rank4 (2,2,2,2);
    my_rank4.setRandom();

    Eigen::ArrayXd         myarray =  Tensor_to_Array(my_rank4, 4,4);
    Eigen::Tensor<double,3> my_rank3 =  Array_to_Tensor(mymatrix, 2,2,4);

    std::cout << my_rank3 << std::endl;
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Return

Function utilities::conv

- Defined in *File conv.h*

Function Documentation

Eigen::VectorXd `utilities::conv`(**const** Eigen::VectorXd &*input*, **const** Eigen::VectorXd &*kernel*)

Convolve an input vector with a kernel, returning an output of the same length as the input.

Uses zero-padded fft based output

Parameters

- `out`:
- `input`:
- `kernel`:

Function utilities::convolution_matrix

- Defined in *File conv.h*

Function Documentation

Eigen::MatrixXd `utilities::convolution_matrix`(**const** Eigen::VectorXd &*k*, **const** Eigen::Index *n*)

Determines the convolution matrix C for impulse response (kernel) vector `k`.

convolution_matrix(k, n) * x is equivalent to conv(k, x)

C is a toeplitz matrix where the upper diagonal is entirely zero, the lower diagonal is zero for cases where

TODO sparse alternative. Large signals produce epically big, mostly empty, matrices.

Example:

```
Eigen::VectorXd k(5)
Eigen::VectorXd x(7)
k << 1, 2, 3, 2, 1;
x << 1, 2, 1, 2, 1, 2, 1;
Eigen::MatrixXd c = convolution_matrix(k, 7);
std::cout << "Convolved result c * x: " << c * x << std::endl
```

Parameters

- `k`: Column vector input
- `n`: Desired dimension of the output matrix. i.e. if convolving with vector x of length 8, n should be 8.

- `c`: Convolution matrix

Template Function utilities::cumsum

- Defined in *File cumsum.h*

Function Documentation

```
template <typename DerivedX, typename DerivedY>
EIGEN_STRONG_INLINE void utilities::cumsum(Eigen::PlainObjectBase< DerivedY > & y, const E
```

Template Function utilities::cumulative_integrate

- Defined in *File integration.h*

Function Documentation

```
template <typename T_functor>
Eigen::ArrayXd utilities::cumulative_integrate(const Eigen::ArrayXd &x, const T_functor &functor, const int max_subintervals = 200)
```

Cumulative numerical integration of a functor in 1D, using gauss-Kronrod61 integration.

Operates in the first dimension (colwise). Applied by reference or returns by value.

Return The cumulative integration of f with respect to x, over the domain in input x

Template Parameters

- `T_functor`: Type of the functor class instance supplied to the integrator

Parameters

- `x`: The domain of x over which to do a cumulative integration. Output values of the cumulative integral are given at these x locations.
- `functor`: The integrand functor. This functor must, given a scalar value of x, return a scalar of the same type containing the value of the integrand function at the input point.
- `max_subintervals`: The maximum number of subintervals allowed in the subdivision process of quadrature functions, for each segment in x. This corresponds to the amount of memory allocated for said functions.

Template Function utilities::deconv

- Defined in *File filter.h*

Function Documentation

```
template <typename DerivedIn, typename DerivedOut>
```

void `utilities::deconv`(Eigen::ArrayBase<DerivedOut> &*z*, **const** Eigen::ArrayBase<DerivedIn> &*b*, **const** Eigen::ArrayBase<DerivedIn> &*a*)

One-dimensional deconvolution.

`z = deconv(b, a)` deconvolves vector `a` out of column `b`, returning a column vector

Function utilities::diagonal_loading_deconv

- Defined in *File conv.h*

Function Documentation

Eigen::VectorXd `utilities::diagonal_loading_deconv`(**const** Eigen::VectorXd &*input*, **const** Eigen::VectorXd &*kernel*, **const** double *alpha* = 0.1)

stably solve Fredholm Integral of the first kind by deconvolution with diagonal loading

Useful site: http://eeweb.poly.edu/iselesni/lecture_notes/least_squares/LeastSquares_SPdemos/deconvolution/html/deconv_demo.html

Return deconvolved signal, same length as input signal.

Parameters

- `input`: Input signal
- `kernel`: Kernel (impulse response) to deconvolve out of the input signal
- `alpha`: Stabilisation parameter alpha. Default 0.1.

Template Function utilities::fft_next_good_size

- Defined in *File conv.h*

Function Documentation

template <typename T>
T `utilities::fft_next_good_size`(**const** T *n*)

Find the next good size for an fft, to pad with minimal number of zeros.

Return M Length to pad the signal to, for optimal FFT performance

Parameters

- `N`: Length of a signal

Template Function utilities::filter(Eigen::ArrayBase<DerivedOut>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&)

- Defined in *File filter.h*

Function Documentation

template <typename DerivedOut, **typename** DerivedIn>
void `utilities::filter`(Eigen::ArrayBase<DerivedOut> *&y*, **const** Eigen::ArrayBase<DerivedIn> *&b*, **const** Eigen::ArrayBase<DerivedIn> *&a*, **const** Eigen::ArrayBase<DerivedIn> *&x*)

One-dimensional digital filter.

Filters the input 1d array (or std::vector<>) `x` with the filter described by coefficient arrays `b` and `a`.

See https://en.wikipedia.org/wiki/Digital_filter

If a(0) is not equal to 1, the filter coefficients are normalised by a(0).

a(0)*y(n-1) = b(0)*x(n-1) + b(1)*x(n-2) + . . . + b(nb)*x(n-nb-1) - a(1)*y(n-2) - . . . - a(na)*y(n-na-1)

Template Function utilities::filter(std::vector<T>&, const std::vector<T>&, const std::vector<T>&, const std::vector<T>&)

- Defined in *File filter.h*

Function Documentation

template <typename T>
void `utilities::filter`(std::vector<T> *&y*, **const** std::vector<T> *&b*, **const** std::vector<T> *&a*, **const** std::vector<T> *&x*)

Function utilities::lowpass_fft_deconv

- Defined in *File conv.h*

Function Documentation

Eigen::VectorXd `utilities::lowpass_fft_deconv`(**const** Eigen::VectorXd *&input*, **const** Eigen::VectorXd *&kernel*, **const** std::string *&flag*, double *stab* = 0.01)

stably solve Fredholm Integral of the first kind by fft based deconvolution with a gaussian low pass filter

Return

Parameters

- `input`:
- `kernel`:
- `stab`: Stabilisation parameter as a proportion of the max magnitude of the kernel. Fixes the lowpass total cutoff point. For example, default 0.01 creates a low pass gaussian filter, between 0 and the point where the FFT of the kernel reaches 1% of its max (usually DC) magnitude. Set <= 0 to circumvent low pass filtering.
- `flag`:

Template Function utilities::matrix_to_tensor

- Defined in *File tensors.h*

Function Documentation

template <typename Scalar, typename... *Dims*>
auto `utilities::matrix_to_tensor`(**const** MatrixType<Scalar> &*matrix*, Dims... *dims*)

convert Eigen Matrix<> to a Tensor

```
int main () {
    Eigen::Tensor<double,4> my_rank4 (2,2,2,2);
    my_rank4.setRandom();

    Eigen::MatrixXd         mymatrix =  Tensor_to_Matrix(my_rank4, 4,4);
    Eigen::Tensor<double,3> my_rank3 =  Matrix_to_Tensor(mymatrix, 2,2,4);

    std::cout << my_rank3 << std::endl;

    return 0;
}
```

Return

Template Function utilities::tensor_dims

- Defined in *File tensors.h*

Function Documentation

template <typename T>
std::string `utilities::tensor_dims`(T &*tensor*)

Return a string representation of tensor dimensions.

Return string The output string like "[2 x 3 x 4]" for a rank 3 tensor

Parameters

- `tensor`: An Eigen::Tensor

Template Function utilities::tensor_to_array

- Defined in *File tensors.h*

Function Documentation

template <typename Scalar, int *rank*, **typename** sizeType>
auto `utilities::tensor_to_array`(**const** Eigen::Tensor<Scalar, rank> &*tensor*, **const** sizeType *rows*, **const** sizeType *cols*)

Convert Eigen Tensor<> to an Array<>

```
int main () {
    Eigen::Tensor<double,4> my_rank4 (2,2,2,2);
    my_rank4.setRandom();

    Eigen::ArrayXd         myarray =  Tensor_to_Array(my_rank4, 4,4);
    Eigen::Tensor<double,3> my_rank3 =  Array_to_Tensor(mymatrix, 2,2,4);

    std::cout << my_rank3 << std::endl;

    return 0;
}
```

Return

Template Function utilities::tensor_to_matrix

- Defined in *File tensors.h*

Function Documentation

template <typename Scalar, int *rank*, **typename** sizeType>
auto `utilities::tensor_to_matrix`(**const** Eigen::Tensor<Scalar, rank> &*tensor*, **const** sizeType *rows*, **const** sizeType *cols*)

convert Eigen Tensor to a Matrix

```
int main () {
    Eigen::Tensor<double,4> my_rank4 (2,2,2,2);
    my_rank4.setRandom();

    Eigen::MatrixXd         mymatrix =  Tensor_to_Matrix(my_rank4, 4,4);
    Eigen::Tensor<double,3> my_rank3 =  Matrix_to_Tensor(mymatrix, 2,2,4);

    std::cout << my_rank3 << std::endl;

    return 0;
}
```

Return

Template Function utilities::trapz(Eigen::ArrayBase<OtherDerived> const&, const Eigen::ArrayBase<Derived>&, Eigen::Index)

- Defined in *File trapz.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "utilities::trapz" with arguments (Eigen::ArrayBase<OtherDerived> const&, const Eigen::ArrayBase<Derived>&, Eigen::Index) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- template <typename Derived, typename OtherDerived>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< OtherDerived > const &
↪, const Eigen::ArrayBase< Derived > &, Eigen::Index)
- template <typename Derived>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< Derived > &,␣
↪Eigen::Index)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< DerivedX > &, const␣
↪Eigen::ArrayBase< DerivedY > &)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< DerivedOut > const &,␣
↪const Eigen::ArrayBase< DerivedX > &, const Eigen::ArrayBase< DerivedY > &)
```

Template Function utilities::trapz(const Eigen::ArrayBase<Derived>&, Eigen::Index)

- Defined in *File trapz.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "utilities::trapz" with arguments (const Eigen::ArrayBase<Derived>&, Eigen::Index) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- template <typename Derived, typename OtherDerived>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< OtherDerived > const &
↪, const Eigen::ArrayBase< Derived > &, Eigen::Index)
- template <typename Derived>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< Derived > &,␣
↪Eigen::Index)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< DerivedX > &, const␣
↪Eigen::ArrayBase< DerivedY > &)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< DerivedOut > const &,␣
↪const Eigen::ArrayBase< DerivedX > &, const Eigen::ArrayBase< DerivedY > &)
```

Template Function utilities::trapz(Eigen::ArrayBase<DerivedOut> const&, const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)

- Defined in *File trapz.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "utilities::trapz" with arguments (Eigen::ArrayBase<DerivedOut> const&, const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- template <typename Derived, typename OtherDerived>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< OtherDerived > const &
↪, const Eigen::ArrayBase< Derived > &, Eigen::Index)
- template <typename Derived>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< Derived > &,␣
↪Eigen::Index)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< DerivedX > &, const␣
↪Eigen::ArrayBase< DerivedY > &)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< DerivedOut > const &,␣
↪const Eigen::ArrayBase< DerivedX > &, const Eigen::ArrayBase< DerivedY > &)
```

Template Function utilities::trapz(const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)

- Defined in *File trapz.h*

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "utilities::trapz" with arguments (const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&) in doxygen xml output for project "My Project" from directory: ./doxyoutput/xml. Potential matches:

```
- template <typename Derived, typename OtherDerived>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< OtherDerived > const &
↪, const Eigen::ArrayBase< Derived > &, Eigen::Index)
- template <typename Derived>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< Derived > &,␣
↪Eigen::Index)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> utilities::trapz(const Eigen::ArrayBase< DerivedX > &, const␣
↪Eigen::ArrayBase< DerivedY > &)
- template <typename DerivedX, typename DerivedY, typename DerivedOut>
  EIGEN_STRONG_INLINE void utilities::trapz(Eigen::ArrayBase< DerivedOut > const &,␣
↪const Eigen::ArrayBase< DerivedX > &, const Eigen::ArrayBase< DerivedY > &)
```

Defines

Define acosd

- Defined in *File definitions.h*

Define Documentation

acosd(x)

Define asind

- Defined in *File definitions.h*

Define Documentation

asind(x)

Define cosd

- Defined in *File definitions.h*

Define Documentation

cosd(x)

Define fourpi

- Defined in *File biot_savart.h*

Define Documentation

fourpi

Define invfourpi

- Defined in *File biot_savart.h*

Define Documentation

invfourpi

Define KAPPA_VON_KARMAN

- Defined in *File definitions.h*

Define Documentation

KAPPA_VON_KARMAN

Define OMEGA_WORLD

- Defined in *File definitions.h*

Define Documentation

OMEGA_WORLD

Define pi

- Defined in *File biot_savart.h*

Define Documentation

pi

Define sind

- Defined in *File definitions.h*

Define Documentation

sind(x)

Define tand

- Defined in *File definitions.h*

Define Documentation

tand(x)

Typedefs

Typedef Array32d

- Defined in *File variable_readers.h*

Typedef Documentation

typedef Eigen::Array<double, 3, 2> **Array32d**

Typedef Array32d

- Defined in *File variable_writers.h*

Typedef Documentation

typedef Eigen::Array<double, 3, 2> **Array32d**

Typedef Array33d

- Defined in *File signature.h*

Typedef Documentation

typedef Eigen::Array<double, 3, 3> **Array33d**

Typedef Array53d

- Defined in *File signature.h*

Typedef Documentation

typedef Eigen::Array<double, 5, 3> **Array53d**

Typedef Array5b

- Defined in *File fit.h*

Typedef Documentation

typedef Eigen::Array<bool, 5, 1> **Array5b**

Typedef Array5d

- Defined in *File fit.h*

Typedef Documentation

typedef Eigen::Array<double, 5, 1> **Array5d**

Typedef utilities::ArrayType

- Defined in *File tensors.h*

Typedef Documentation

using utilities::ArrayType = typedef Eigen::Array<T,Eigen::Dynamic, Eigen::Dynamic>

Typedef utilities::MatrixType

- Defined in *File tensors.h*

Typedef Documentation

using utilities::MatrixType = typedef Eigen::Matrix<T,Eigen::Dynamic, Eigen::Dynamic>

Directories

Directory source

Directory path: `source`

Subdirectories

- *Directory adem*
- *Directory io*
- *Directory relations*
- *Directory utilities*

Files

- *File data_types.h*
- *File definitions.h*
- *File exceptions.h*
- *File fit.h*

- *File flow.h*
- *File how_to.h*
- *File main.cpp*
- *File profile.cpp*
- *File profile.h*

Directory adem

Parent directory (`source`)

Directory path: `source/adem`

Files

- *File adem.h*
- *File biot_savart.h*
- *File signature.h*

Directory io

Parent directory (`source`)

Directory path: `source/io`

Files

- *File readers.h*
- *File variable_readers.h*
- *File variable_writers.h*

Directory relations

Parent directory (`source`)

Directory path: `source/relations`

Files

- *File spectra.cpp*
- *File spectra.h*
- *File stress.h*
- *File veer.h*
- *File velocity.h*

Directory utilities

Parent directory (`source`)

Directory path: `source/utilities`

Files

- *File conv.h*
- *File cumsum.h*
- *File cumtrapz.h*
- *File filter.h*
- *File integration.h*
- *File interp.h*
- *File smear.h*
- *File tensors.h*
- *File trapz.h*

Files

File adem.h

Parent directory (`source/adem`)

Contents

Definition (`source/adem/adem.h`)

Program Listing for File adem.h

Return to documentation for file (`source/adem/adem.h`)

```
/*
 * adem.h Implementation of the Attached-Detached Eddy Method
 *
 * Author:               Tom Clark  (thclark @ github)
 *
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef SOURCE_ADEM_H_
#define SOURCE_ADEM_H_

#include <vector>
#include <iostream>
#include <boost/algorithm/string/classification.hpp> // Include boost::for is_any_of
#include <boost/algorithm/string/split.hpp> // Include for boost::split
#include <Eigen/Dense>
#include <Eigen/Core>
#include <stdexcept>
#include <unsupported/Eigen/CXX11/Tensor>
#include <unsupported/Eigen/FFT>
#include <unsupported/Eigen/Splines>

#include "cpplot.h"

#include "adem/signature.h"
#include "profile.h"
#include "io/variable_readers.h"
#include "relations/stress.h"
#include "relations/velocity.h"
#include "utilities/conv.h"
#include "utilities/interp.h"
#include "utilities/filter.h"
#include "utilities/tensors.h"


using namespace utilities;
using namespace cpplot;


namespace es {


class AdemData {
public:

    std::vector<std::string> eddy_types = {"A", "B1+B2+B3+B4"};

    double beta;

    double delta_c;

    double kappa;

    double pi_coles;

    double shear_ratio;

    double u_inf;

    double u_tau;
```

(continues on next page)

(continued from previous page)

```
    double zeta;

    Eigen::VectorXd z;

    Eigen::VectorXd eta;

    Eigen::VectorXd lambda_e;

    Eigen::VectorXd u_horizontal;

    Eigen::ArrayXXd reynolds_stress;

    Eigen::ArrayXXd r13a_analytic;

    Eigen::ArrayXXd r13b_analytic;

    Eigen::ArrayXXd reynolds_stress_a;

    Eigen::ArrayXXd reynolds_stress_b;

    Eigen::ArrayXXd k1z;

    Eigen::Tensor<double, 3> psi;

    Eigen::Tensor<double, 3> psi_a;

    Eigen::Tensor<double, 3> psi_b;

    Eigen::VectorXd t2wa;

    Eigen::VectorXd t2wb;

    Eigen::VectorXd residual_a;

    Eigen::VectorXd residual_b;

    Eigen::Index start_idx;

    // TODO consider whether we can tidy these variables up

    Eigen::ArrayXXd ja_fine;

    Eigen::ArrayXXd jb_fine;

    Eigen::ArrayXd r13a_analytic_fine;

    Eigen::ArrayXd r13b_analytic_fine;

    Eigen::ArrayXd minus_t2wa_fine;

    Eigen::ArrayXd minus_t2wb_fine;

    Eigen::ArrayXd lambda_fine;

    Eigen::ArrayXd eta_fine;

    void load(std::string file_name, bool print_var = true) {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "Reading adem data from file " << file_name << std::endl;

        // Open the MAT file for reading
        mat_t *matfp = Mat_Open(file_name.c_str(), MAT_ACC_RDONLY);
        if (matfp == NULL) {
            std::string msg = "Error reading MAT file: ";
            throw std::invalid_argument(msg + file_name);
        }

        // Use the variable readers to assist
        k1z                 = readArrayXXd(matfp, "k1z", print_var);
        beta                = readDouble(matfp, "beta", print_var);
        delta_c             = readDouble(matfp, "delta_c", print_var);
        kappa               = readDouble(matfp, "kappa", print_var);
        pi_coles            = readDouble(matfp, "pi_coles", print_var);
        shear_ratio         = readDouble(matfp, "shear_ratio", print_var);
        u_inf               = readDouble(matfp, "u_inf", print_var);
        u_tau               = readDouble(matfp, "u_tau", print_var);
        zeta                = readDouble(matfp, "zeta", print_var);
        z                   = readVectorXd(matfp, "z", print_var);
        eta                 = readVectorXd(matfp, "eta", print_var);
        lambda_e            = readVectorXd(matfp, "lambda_e", print_var);
        u_horizontal        = readVectorXd(matfp, "u_horizontal", print_var);
        reynolds_stress     = readArrayXXd(matfp, "reynolds_stress", print_var);
        reynolds_stress_a   = readArrayXXd(matfp, "reynolds_stress_a", print_var);
        reynolds_stress_b   = readArrayXXd(matfp, "reynolds_stress_b", print_var);
        k1z                 = readArrayXXd(matfp, "k1z", print_var);
        psi                 = readTensor3d(matfp, "psi", print_var);
        psi_a               = readTensor3d(matfp, "psi_a", print_var);
        psi_b               = readTensor3d(matfp, "psi_b", print_var);
        t2wa                = readVectorXd(matfp, "t2wa", print_var);
        t2wb                = readVectorXd(matfp, "t2wb", print_var);
        residual_a          = readVectorXd(matfp, "residual_a", print_var);
        residual_b          = readVectorXd(matfp, "residual_b", print_var);

        // Special handling to split the string of types into a vector of strings
        std::string typestr = readString(matfp, "eddy_types", print_var);
        boost::split(eddy_types, typestr, boost::is_any_of(", "), boost::token_
↪compress_on);

        // Close the file
        Mat_Close(matfp);
        std::cout << "Finished reading adem data" << std::endl;
    }

    void save(std::string filename) {
        std::cout << "Writing signature data..." << std::endl;
        throw std::invalid_argument("Error writing mat file - function not implemented
↪");
    }

};


std::ostream &operator<<(std::ostream &os, AdemData const &data) {
    os << "AdemData() with fields:" << std::endl;
    os << "    eddy_types:        ";
```

(continues on next page)

(continued from previous page)

```
    for (std::vector<std::string>::const_iterator i = data.eddy_types.begin(); i !=␣
↪data.eddy_types.end(); ++i) {
        os << *i << ", ";
    }
    os << std::endl;
    os << "    beta:              " << data.beta << std::endl
       << "    delta_c:           " << data.delta_c << std::endl
       << "    kappa:             " << data.kappa << std::endl
       << "    pi_coles:          " << data.pi_coles << std::endl
       << "    shear_ratio:       " << data.shear_ratio << std::endl
       << "    u_inf:             " << data.u_inf << std::endl
       << "    u_tau:             " << data.u_tau << std::endl
       << "    zeta:              " << data.zeta << std::endl
       << "    z:                 [" << data.z.size() << " x 1]" << std::endl
       << "    eta:               [" << data.eta.size() << " x 1]" << std::endl
       << "    lambda_e:          [" << data.lambda_e.size() << " x 1]" << std::endl
       << "    u_horizontal:      [" << data.u_horizontal.size() << " x 1]" <<␣
↪std::endl
       << "    reynolds_stress:   [" << data.reynolds_stress.rows() << " x " << data.
↪reynolds_stress.cols() << "]" << std::endl
       << "    reynolds_stress_a: [" << data.reynolds_stress_a.rows() << " x " <<␣
↪data.reynolds_stress_a.cols() << "]" << std::endl
       << "    reynolds_stress_b: [" << data.reynolds_stress_b.rows() << " x " <<␣
↪data.reynolds_stress_b.cols() << "]" << std::endl
       << "    k1z:               [" << data.k1z.rows() << " x " << data.k1z.cols() <
↪< "]" << std::endl
       << "    psi:               [" << tensor_dims(data.psi) << "]" << std::endl
       << "    psi_a:             [" << tensor_dims(data.psi_a) << "]" << std::endl
       << "    psi_b:             [" << tensor_dims(data.psi_b) << "]" << std::endl
       << "    t2wa:              [" << data.t2wa.size() << " x 1]" << std::endl
       << "    t2wb:              [" << data.t2wb.size() << " x 1]" << std::endl
       << "    residual_a:        [" << data.residual_a.size() << " x 1]" << std::endl
       << "    residual_b:        [" << data.residual_b.size() << " x 1]" <<␣
↪std::endl;
    return os;
}


void get_t2w(AdemData& data, const EddySignature& signature_a, const EddySignature&␣
↪signature_b) {

    /* On the largest eddy scales:
     * lambda_e is the scale of a given eddy, which contributes to the stresses and␣
↪spectra in the boundary layer.
     *
     * As far as the signatures are concerned, lambda is mapped for a domain spanning␣
↪the actual eddy.
     * For a signature, values lambda < 0 are valid; since an eddy exerts an␣
↪influence a short distance above itself
     * (eddies accelerate flow above them, and decelerate flow below).
     *
     * As far as the deconvolution is concerned, lambda is mapped over the whole␣
↪boundary layer.
     * For a boundary layer profile, values < 0 are not meaningful: R_ij and S_ij␣
↪must definitively be 0 outside
     * the boundary layer.
     *
```

(continues on next page)

(continued from previous page)

```
     * Since values of lambda < 0 are theoretically valid for an eddy signature, the␣
↪signatures as produced by
     * EddySignature().computeSignatures() extend out beyond lambda_e (to z/delta = 1.
↪5, in our implementation).
     * Having an eddy whose scale is of the same size as the boundary layer would␣
↪therefore create an influence outside
     * the boundary layer, breaking all assumptions about boundary layers ever.
     *
     * For us to maintain the fundamental boundary layer top condition that U = U_inf␣
↪and u' = 0 for all z/delta >= 1,
     * we assume that eddies can have no influence above their own scale (following␣
↪Perry & Marusic 1995 pp. 371-2).
     *
     * Clipping the signatures like this is the equivalent of setting the eddy␣
↪signatures to zero where lambda < 0.
     *
     */
    data.start_idx = 0;
    for (Eigen::Index i=0; i<signature_a.lambda.rows(); i++) {
        if (signature_a.lambda(i) >= 0) {
            data.start_idx = i;
            break;
        }
    }
    Eigen::Index n_lambda_signature = signature_a.lambda.rows() - data.start_idx;
    Eigen::ArrayXd lambda_signature = signature_a.lambda.bottomRows(n_lambda_
↪signature);

    /* On the smallest eddy scales:
     *
     * So the smallest scale we want to go to is lambda_1, which can be defined by␣
↪the physics of the problem (i.e. the
     * cutoff beyond which viscosity takes over). This scale is defined by the Karman␣
↪number K_tau.
     *  P&M 1995 again give us an approximate relation: lambda_1 ~= 100 nu/u_tau.
     *
     * As far as this function goes, we take lambda_1 as the smallest scale in the␣
↪eddy signature array, which assumes
     * that signatures were computed on an appropriate grid.
     *
     */
    double d_lambda_fine = signature_a.domain_spacing(2);
    double lambda_min = lambda_signature(0);
    double lambda_max = lambda_signature(n_lambda_signature-1);
    Eigen::Index n_lambda_fine = round((lambda_max - lambda_min)/d_lambda_fine);

    Eigen::ArrayXd lambda_fine = Eigen::ArrayXd::LinSpaced(n_lambda_fine, lambda_min,␣
↪lambda_max);

    // Hard limit on lambda max
    double mymax = log(1.0/0.01);
    Eigen::Index up_to = 0;
    for (Eigen::Index i=0; i<lambda_fine.rows(); i++) {
        if (lambda_fine(i) >= mymax) {
            up_to = i;
            break;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    lambda_fine = lambda_fine.topRows(up_to);

    Eigen::ArrayXd eta_fine = lambda_fine.exp().inverse();

    /* On the signatures:
     *
     * To avoid warping the reconstruction, we need to do the convolution and the␣
↪deconvolution with
     * both the input and the signature on the same, linearly spaced, basis.
     *
     * To do this, we interpolate to a fine distribution and store the fine values.
     *
     */
    Eigen::ArrayXXd ja_fine = signature_a.getJ(lambda_fine);
    Eigen::ArrayXXd jb_fine = signature_b.getJ(lambda_fine);
    Eigen::ArrayXd j13a_fine = ja_fine.col(2);
    Eigen::ArrayXd j13b_fine = jb_fine.col(2);

    // Get an ascending version of eta, containing the point eta=0, for use in␣
↪determining analytic R13 distributions
    Eigen::ArrayXd bounded_eta = Eigen::ArrayXd(eta_fine.rows()+2);
    bounded_eta.setZero();
    bounded_eta.middleRows(1, eta_fine.rows()) = eta_fine.reverse();
    bounded_eta.bottomRows(1) = 1.0;

    // Get Reynolds Stresses, trim the zero point, reverse back so the ordering is␣
↪consistent with lambda coordinates
    Eigen::ArrayXd r13a_fine;
    Eigen::ArrayXd r13b_fine;
    reynolds_stress_13(r13a_fine, r13b_fine, data.beta, bounded_eta, data.kappa, data.
↪pi_coles, data.shear_ratio, data.zeta);
    r13a_fine = r13a_fine.middleRows(1, eta_fine.rows());
    r13b_fine = r13b_fine.middleRows(1, eta_fine.rows());
    r13a_fine.reverseInPlace();
    r13b_fine.reverseInPlace();

    // Produce visual check that eta is ascending exponentially
    ScatterPlot pc = ScatterPlot();
    pc.x = Eigen::ArrayXd::LinSpaced(bounded_eta.rows(), 1, bounded_eta.rows());
    pc.y = bounded_eta;
    Layout layc = Layout("Check that eta ascends exponentially");
    layc.xTitle("Row number");
    layc.yTitle("$\\eta$");
    Figure figc = Figure();
    figc.add(pc);
    figc.setLayout(layc);
    figc.write("check_that_eta_ascends_exponentially.json");

    // Produce a visual check that lambda is ascending linearly
    ScatterPlot pc1 = ScatterPlot();
    pc1.x = Eigen::ArrayXd::LinSpaced(lambda_fine.rows(), 1, lambda_fine.rows());
    pc1.y = lambda_fine;
    Layout layc1 = Layout("Check that lambda ascends linearly");
    layc1.xTitle("Row number");
    layc1.yTitle("$\\lambda$");
    Figure figc1 = Figure();
```

(continues on next page)

(continued from previous page)

```
    figc1.add(pc1);
    figc1.setLayout(layc1);
    figc1.write("check_that_lambda_ascends.json");

    // Double check J13 and interpolations
    ScatterPlot pja = ScatterPlot();
    pja.x = lambda_signature;
    pja.y = signature_a.j.bottomRows(n_lambda_signature).col(2);
    pja.name = "Type A (signature)";
    ScatterPlot pjaf = ScatterPlot();
    pjaf.x = lambda_fine;
    pjaf.y = j13a_fine;
    pjaf.name = "Type A (fine)";
    ScatterPlot pjb = ScatterPlot();
    pjb.x = lambda_signature;
    pjb.y = signature_b.j.bottomRows(n_lambda_signature).col(2);
    pjb.name = "Type B (signature)";
    ScatterPlot pjbf = ScatterPlot();
    pjbf.x = lambda_fine;
    pjbf.y = j13b_fine;
    pjbf.name = "Type B (fine)";
    Layout layj = Layout();
    layj.xTitle("$\\lambda$");
    layj.yTitle("$J_{13}$");
    Figure figj = Figure();
    figj.add(pja);
    figj.add(pjaf);
    figj.add(pjb);
    figj.add(pjbf);
    figj.setLayout(layj);
    figj.write("check_j13.json");

    // Deconvolve out the A and B structure contributions to the Reynolds Stresses
    // NOTE: it's actually -1*T^2w in this variable
    Eigen::ArrayXd minus_t2wa_fine;
    Eigen::ArrayXd minus_t2wb_fine;
//    double stability = 0.005;
//    minus_t2wa_fine = utilities::lowpass_fft_deconv(r13a_fine, j13a_fine, "Type_A",␣
↪stability);
//    minus_t2wb_fine = utilities::lowpass_fft_deconv(r13b_fine, j13b_fine, "Type_B",␣
↪stability);

    double alpha = 0.1;
    minus_t2wa_fine = utilities::diagonal_loading_deconv(r13a_fine, j13a_fine, alpha);
    minus_t2wb_fine = utilities::diagonal_loading_deconv(r13b_fine, j13b_fine, alpha);

    // Test the roundtripping convolution without any of the remapping
    Eigen::ArrayXd r13a_fine_recon = conv(minus_t2wa_fine.matrix(), j13a_fine.
↪matrix()).matrix();
    Eigen::ArrayXd r13b_fine_recon = conv(minus_t2wb_fine.matrix(), j13b_fine.
↪matrix()).matrix();

    // Show Reynolds Stresses behaving as part of validation
    ScatterPlot pa = ScatterPlot();
    pa.x = lambda_fine;
    pa.y = -1.0*r13a_fine;
    pa.name = "Type A (input)";
```

(continues on next page)

(continued from previous page)

```
    ScatterPlot pb = ScatterPlot();
    pb.x = lambda_fine;
    pb.y = -1.0*r13b_fine;
    pb.name = "Type B (input)";

    ScatterPlot par = ScatterPlot();
    par.x = lambda_fine;
    par.y = -1.0*r13a_fine_recon;
    par.name = "Type A (reconstructed)";

    ScatterPlot pbr = ScatterPlot();
    pbr.x = lambda_fine;
    pbr.y = -1.0*r13b_fine_recon;
    pbr.name = "Type B (reconstructed)";

    Layout lay = Layout();
    lay.xTitle("$\\lambda$");
    lay.yTitle("$-\\overline{u_1u_3}/U_\\tau^2$");

    Figure fig = Figure();
    fig.add(pa);
    fig.add(pb);
    fig.add(par);
    fig.add(pbr);
    fig.setLayout(lay);
    fig.write("check_r13_fine_reconstruction.json");

    // Create a plot to show the t2w terms
    Figure figt = Figure();
    ScatterPlot pt = ScatterPlot();
    pt.x = lambda_fine;
    pt.y = minus_t2wa_fine;
    pt.name = "$-T_{A}^2(\\lambda - \\lambda_E)\\omega_{A}(\\lambda-\\lambda_E)$";
    figt.add(pt);
    ScatterPlot pt2 = ScatterPlot();
    pt2.x = lambda_fine;
    pt2.y = minus_t2wb_fine;
    pt2.name = "$-T_{B}^2(\\lambda - \\lambda_E)\\omega_{B}(\\lambda-\\lambda_E)$";
    figt.add(pt2);
    Layout layt = Layout();
    layt.xTitle("$\\lambda - \\lambda_E$");
    figt.setLayout(layt);
    figt.write("check_t2w.json");
    std::cout << "Wrote figure check_t2w" << std::endl;


    // Extend by padding out to the same length as lambda_e.
    // The T^2w distributions converge to a constant at high lambda (close to the␣
↪wall) so padding with the last value
    // in the vector is physically valid. This will result in the Reynolds Stresses␣
↪and Spectra, which are obtained by
    // convolution, having the same number of points in the z direction as the axes␣
↪variables (eta, lambda_e, etc)
//    double pad_value_a = minus_t2wa(minus_t2wa.rows()-1);
//    double pad_value_b = minus_t2wb(minus_t2wb.rows()-1);
//
```

(continues on next page)

(continued from previous page)

```
//    auto last_n = lambda_e.rows() - minus_t2wa.rows();
//    minus_t2wa.conservativeResize(lambda_e.rows(),1);
//    minus_t2wb.conservativeResize(lambda_e.rows(),1);
//    minus_t2wa.tail(last_n) = pad_value_a;
//    minus_t2wb.tail(last_n) = pad_value_b;

    // Store in the data object
    data.lambda_fine = lambda_fine;
    data.r13a_analytic_fine = r13a_fine;
    data.r13b_analytic_fine = r13b_fine;
    data.minus_t2wa_fine = minus_t2wa_fine;
    data.minus_t2wb_fine = minus_t2wb_fine;
    data.ja_fine = ja_fine;
    data.jb_fine = jb_fine;
    data.eta_fine = eta_fine;
    data.eta = eta_fine;

}


void get_mean_speed(AdemData& data) {
    data.u_horizontal = lewkowicz_speed(data.eta, data.pi_coles, data.kappa, data.u_
↪inf, data.u_tau);
}


void get_reynolds_stresses(AdemData& data, const EddySignature& signature_a, const␣
↪EddySignature& signature_b){

    // Allocate the output and set zero
    data.reynolds_stress_a = Eigen::ArrayXXd(data.lambda_fine.rows(), 6);
    data.reynolds_stress_b = Eigen::ArrayXXd(data.lambda_fine.rows(), 6);
    data.reynolds_stress_a.setZero();
    data.reynolds_stress_b.setZero();

    // Column-wise convolution of T2w with J
    for (int k = 0; k < 6; k++) {
        data.reynolds_stress_a.col(k) = conv(data.minus_t2wa_fine.matrix(), data.ja_
↪fine.col(k).matrix());
        data.reynolds_stress_b.col(k) = conv(data.minus_t2wb_fine.matrix(), data.jb_
↪fine.col(k).matrix());
    }

    // Trim the zero-padded ends
    data.reynolds_stress = data.reynolds_stress_a + data.reynolds_stress_b;

    // Check that R13 stacks up against the analytical solution
    Figure fig = Figure();
    Layout lay = Layout();
    ScatterPlot paa = ScatterPlot();
    paa.x = data.eta_fine;
    paa.y = -1.0*data.r13a_analytic_fine;
    paa.name = "Type A - analytic";
    ScatterPlot pba = ScatterPlot();
    pba.x = data.eta_fine;
    pba.y = -1.0*data.r13b_analytic_fine;
    pba.name = "Type B - analytic";
```

(continues on next page)

(continued from previous page)

```
    ScatterPlot par = ScatterPlot();
    par.x = data.eta_fine;
    par.y = -1.0*(data.reynolds_stress_a.col(2));
    par.name = "Type A - reconstructed";
    ScatterPlot pbr = ScatterPlot();
    pbr.x = data.eta_fine;
    pbr.y = -1.0*(data.reynolds_stress_b.col(2));
    pbr.name = "Type B - reconstructed";
    fig.add(paa);
    fig.add(pba);
    fig.add(par);
    fig.add(pbr);
    lay.xTitle("$z/\\delta_{c}$");
    lay.yTitle("$-\\overline{u_1u_3}/U_\\tau^2$");
    fig.setLayout(lay);
    fig.write("check_r13_analytic_and_reconstructed.json");
}


void get_spectra(AdemData& data, const EddySignature& signature_a, const␣
↪EddySignature& signature_b){

    // Initialise the output spectrum tensors
    Eigen::array<Eigen::Index, 3> dims = signature_a.g.dimensions();
    Eigen::array<Eigen::Index, 3> psi_dims = {data.t2wa.rows(), dims[1], dims[2]};
    Eigen::Tensor<double, 3> psi_a(psi_dims);
    Eigen::Tensor<double, 3> psi_b(psi_dims);

    // Map the t2w arrays as vectors, for use with the conv function
    Eigen::Map<Eigen::VectorXd> t2wa_vec(data.t2wa.data(), data.t2wa.rows());
    Eigen::Map<Eigen::VectorXd> t2wb_vec(data.t2wb.data(), data.t2wb.rows());

    std::cout << "DIMS " << dims[0] << " " << dims[1] << " " << dims[2] << std::endl;
    std::cout << "PSI_DIMS " << psi_dims[0] << " " << psi_dims[1] << " " << psi_
↪dims[2] << std::endl;

    // For each of the 6 auto / cross spectra terms
    for (Eigen::Index j = 0; j < dims[2]; j++) {

        auto page_offset_sig = j * dims[0] * dims[1];
        auto page_offset_psi = j * psi_dims[0] * psi_dims[1];

        // For each of the wavenumbers
        for (Eigen::Index i = 0; i < dims[1]; i++) {

            auto elements_offset_sig = (page_offset_sig + i * dims[0]);
            auto elements_offset_psi = (page_offset_psi + i * psi_dims[0]);

            Eigen::Map<Eigen::VectorXd> g_a_vec((double *)signature_a.g.data() +␣
↪elements_offset_sig, dims[0]);
            Eigen::Map<Eigen::VectorXd> g_b_vec((double *)signature_b.g.data() +␣
↪elements_offset_sig, dims[0]);

            Eigen::Map<Eigen::VectorXd> psi_a_vec((double *)psi_a.data() + elements_
↪offset_psi, psi_dims[0]);
            Eigen::Map<Eigen::VectorXd> psi_b_vec((double *)psi_b.data() + elements_
↪offset_psi, psi_dims[0]);
```

(continues on next page)

(continued from previous page)

```
            psi_a_vec = conv(t2wa_vec, g_a_vec).reverse();
            psi_b_vec = conv(t2wb_vec, g_b_vec).reverse();

        }
    }

    // Don't store the sum of the spectra - they're memory hungry and we can always␣
↪add the two together

    // Premultiply by the u_tau^2 term (see eq. 43)
    // auto u_tau_sqd = pow(data.u_tau, 2.0);
    // data.psi_a = u_tau_sqd * psi_a;
    // data.psi_b = u_tau_sqd * psi_b;
    data.psi_a = psi_a;
    data.psi_b = psi_b;

}


AdemData adem(const double beta,
              const double delta_c,
              const double kappa,
              const double pi_coles,
              const double shear_ratio,
              const double u_inf,
              const double zeta,
              const EddySignature& signature_a,
              const EddySignature& signature_b,
              bool compute_spectra=true){

    // Data will contain computed outputs and useful small variables from the large␣
↪signature files
    AdemData data = AdemData();

    // Add parameter inputs to data structure
    data.beta = beta;
    data.delta_c = delta_c;
    data.kappa = kappa;
    data.pi_coles = pi_coles;
    data.shear_ratio = shear_ratio;
    data.u_inf = u_inf;
    data.u_tau = data.u_inf / data.shear_ratio;
    data.zeta = zeta;

    // Deconvolve for T^2w and update the data structure with the outputs
    get_t2w(data, signature_a, signature_b);

    // Get vertical points through the boundary layer (for which the convolution␣
↪functions were defined)
    data.z = data.eta.array() * data.delta_c;

    // Add k1z to the data structure
    Eigen::ArrayXd eta = data.eta.array();
    data.k1z = signature_a.k1z(eta);

    // Get the mean speed profile at the same vertical points and update the data␣
↪structure
```

(continues on next page)

(continued from previous page)

```
    get_mean_speed(data);

    // Determine Reynolds Stresses by convolution and add them to the data structure
    get_reynolds_stresses(data, signature_a, signature_b);

    // Determine Spectra by convolution and add them to the data structure
    if (compute_spectra) {
        get_spectra(data, signature_a, signature_b);
    }

    return data;

}

} /* namespace es */

#endif /* SOURCE_ADEM_H_ */
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `adem/signature.h` (*File signature.h*)
- `boost/algorithm/string/classification.hpp`
- `boost/algorithm/string/split.hpp`
- `cpplot.h`
- `io/variable_readers.h` (*File variable_readers.h*)
- `iostream`
- `profile.h` (*File profile.h*)
- `relations/stress.h` (*File stress.h*)
- `relations/velocity.h` (*File velocity.h*)
- `stdexcept`
- `unsupported/Eigen/CXX11/Tensor`
- `unsupported/Eigen/FFT`
- `unsupported/Eigen/Splines`
- `utilities/conv.h` (*File conv.h*)
- `utilities/filter.h` (*File filter.h*)
- `utilities/interp.h` (*File interp.h*)
- `utilities/tensors.h` (*File tensors.h*)
- `vector`

Namespaces

- *Namespace cpplot*
- *Namespace es*

Classes

- *Class AdemData*

Functions

- *Function es::adem*
- *Function es::get_mean_speed*
- *Function es::get_reynolds_stresses*
- *Function es::get_spectra*
- *Function es::get_t2w*
- *Function es::operator<<(std::ostream&, AdemData const&)*

File biot_savart.h

Parent directory (`source/adem`)

Contents

Definition (`source/adem/biot_savart.h`)

Program Listing for File biot_savart.h

Return to documentation for file (`source/adem/biot_savart.h`)

```
/*
 * biot_savart.h A parallelised biot-savart function for use with the Attached-
↪Detached Eddy Method
 *
 * Author:               Tom Clark  (thclark @ github)
```

(continues on next page)

(continued from previous page)

```
 *
 * Copyright (c) 2014-2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_BIOT_SAVART_H
#define ES_FLOW_BIOT_SAVART_H

#include <tbb/tbb.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <tbb/atomic.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <iostream>
#include <Eigen/Dense>


using namespace Eigen;
using namespace std;


#define pi          3.14159265358979323846264;
#define fourpi      12.566370614359217295385 1;
#define invfourpi   0.079577471545948;


namespace es {


// @cond - Don't document these convenience inline functions or the kernel

// Inverse magnitude of a Vector3d element
static inline double InvMagnitude(Vector3d vec){
    return 1 / vec.norm();
}


// Inverse magnitude of a Vector3d element handling /0 errors to return 0 always
static inline double InvMagnitudeZero(Vector3d vec)
{
    double a = vec.norm();
    if (a == 0.) {
        return 0.;
    } else {
        return 1 / a;
    }
}


// Kernel that applies induction from one vortex line to one control point
static inline Vector3d BiotElementalKernel(Vector3d control_locations, Vector3d start_
↪nodes, Vector3d end_nodes, double gamma, double effective_core_radius_squared)
{
```

(continues on next page)

(continued from previous page)

```
    Vector3d l12 = end_nodes - start_nodes;

    Vector3d r1 = control_locations - start_nodes;

    Vector3d r2 = control_locations - end_nodes;

    Vector3d cross1 = l12.cross(r1);

    // Used invmagnitudezero to solve the problem where the cross product is {0,0,0}␣
↪due to point p being on the line defined by l12 (r1 collinear with l12)
    Vector3d induction;
    induction = InvMagnitudeZero(cross1) * cross1;
    double mag_l12 = InvMagnitude(l12);
    double mag_r1 = InvMagnitudeZero(r1);
    double mag_r2 = InvMagnitudeZero(r2);
    double costheta_1 = l12.dot(r1) * (mag_l12 * mag_r1);
    double costheta_2 = l12.dot(r2) * (mag_l12 * mag_r2);
    double h = (cross1.norm()) * (mag_l12);
    double magu = gamma * h * (costheta_1 - costheta_2) * invfourpi;
    magu /= sqrt(pow(effective_core_radius_squared, 2) + pow(h, 4));
    induction = induction * magu;

    return induction;

};


// @endcond


Matrix3Xd NaiveBiotSavart(Matrix3Xd startNodes, Matrix3Xd endNodes, Matrix3Xd␣
↪locations, VectorXd gamma, VectorXd effective_core_radius_squared){

    assert (startNodes.cols() == gamma.size());
    assert (startNodes.cols() == endNodes.cols());
    assert (startNodes.cols() == effective_core_radius_squared.size());
    Index start = 0;
    Index n_control_point = locations.cols();
    Index n_vortex = gamma.size();
    Index n = n_control_point * n_vortex;

    Matrix3Xd induction(3, n_control_point);
    induction.setZero();

    // Parallel version
    tbb::affinity_partitioner partitioner1;
    tbb::affinity_partitioner partitioner2;

    tbb::parallel_for(start, n_control_point, [&](Index control_i){

        Vector3d p = locations.col(control_i);

        Vector3d induction_accumulator;
        induction_accumulator.setZero();
        tbb::spin_mutex induction_mutex;

        tbb::parallel_for(start, n_vortex, [&](Index vortex_i){
```

(continues on next page)

(continued from previous page)

```
            Vector3d A1 = startNodes.col(vortex_i);
            Vector3d B1 = endNodes.col(vortex_i);
            double a = gamma(vortex_i);
            double rc4 = effective_core_radius_squared(vortex_i);
            Vector3d induction_elemental = BiotElementalKernel(p, A1, B1, a, rc4);
            {
                tbb::spin_mutex::scoped_lock lock(induction_mutex);
                induction_accumulator += induction_elemental;
            }
        }, partitioner1);
        induction.block(0, control_i, 3, 1) = induction_accumulator;

    }, partitioner2);

    /* Serial version
    for (long control_i = 0; control_i < n_control_point; control_i++)
    {
        Vector3d p = locations.block(0,control_i,3,1);
        long vortex_i;
        for (vortex_i = 0; vortex_i < n_vortex; vortex_i++) {
            Vector3d A1 = startNodes.block(0, vortex_i, 3, 1);
            Vector3d B1 = endNodes.block(0, vortex_i, 3, 1);
            double a = gamma(vortex_i);
            double rc4 = effective_core_radius_squared(vortex_i);
            induction.block(0, control_i, 3, 1) = induction.block(0, control_i, 3, 1)␣
↪+ BiotElementalKernel(p, A1, B1, a, rc4);
        };

    }
    */

    return induction;
};

};

#endif
```

Includes

- Eigen/Dense
- iostream
- math.h
- stdio.h
- string.h
- tbb/atomic.h
- tbb/blocked_range.h
- tbb/parallel_for.h
- tbb/tbb.h
- time.h

Included By

- *File signature.h*

Namespaces

- *Namespace Eigen*
- *Namespace es*
- *Namespace std*

Functions

- *Function es::NaiveBiotSavart*

Defines

- *Define fourpi*
- *Define invfourpi*
- *Define pi*

File conv.h

Parent directory (`source/utilities`)

Contents

Definition (`source/utilities/conv.h`)

Program Listing for File conv.h

Return to documentation for file (`source/utilities/conv.h`)

```
/*
 * conv.h Matlab-like convolution
 *
 * Author:               Tom Clark  (thclark @ github)
 *
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_CONV_H
#define ES_FLOW_CONV_H

#include <Eigen/Dense>
#include <Eigen/Core>
#include <unsupported/Eigen/FFT>
#include <math.h>

//#include "cpplot.h"
//using namespace cpplot;

namespace utilities {


template<typename T>
T fft_next_good_size(const T n) {
    T result = n;
    if (n <= 2) {
        result = 2;
        return result;
    }
    while (true) {
        T m = result;
        while ((m % 2) == 0) m = m / 2;
        while ((m % 3) == 0) m = m / 3;
        while ((m % 5) == 0) m = m / 5;
        if (m <= 1) {
            return (result);
        }
        result = result + 1;
    }
}


Eigen::VectorXd conv(const Eigen::VectorXd &input, const Eigen::VectorXd &kernel) {

    // TODO template this function signature to also accept arrays

    // Map the input signature data to tensors (shared memory)
    auto input_len = input.rows();
    auto kernel_len = kernel.rows();

    // Compute cumulative length of input and kernel;
    auto N = input_len + kernel_len - 1;
    auto M = fft_next_good_size(N);

    // Create padded FFT inputs
    Eigen::FFT<double> fft;
    Eigen::VectorXd input_padded(M);
    Eigen::VectorXd kernel_padded(M);
    input_padded.setZero();
    kernel_padded.setZero();
    input_padded.topRows(input_len) = input;
```

(continues on next page)

(continued from previous page)

```
    kernel_padded.topRows(kernel_len) = kernel;

    // Take the forward ffts
    Eigen::VectorXcd input_transformed(M);
    Eigen::VectorXcd kernel_transformed(M);
    fft.fwd(input_transformed, input_padded);
    fft.fwd(kernel_transformed, kernel_padded);

    // Convolve by element-wise multiplication
    Eigen::VectorXcd inter = input_transformed.array() * kernel_transformed.array();

    // Inverse FFT
    Eigen::VectorXd out(M);
    out.setZero();
    fft.inv(out, inter);

    // Crop to the size of the input vector
    out = out.topRows(input_len);
    return out;
}


Eigen::MatrixXd convolution_matrix(const Eigen::VectorXd &k, const Eigen::Index n) {

    Eigen::Index mk = k.rows();
    Eigen::VectorXd toeplitz_col = Eigen::VectorXd::Zero(mk + n - 1);
    toeplitz_col.topRows(mk) = k;

    Eigen::Index mt = toeplitz_col.rows();
    Eigen::MatrixXd toeplitz = Eigen::MatrixXd::Zero(mt, n);

    // Fill the lower diagonal of a toeplitz matrix.
    // If it's fat rectangular we don't need to fill the right-most columns.
    Eigen::Index nt_limit = Eigen::Vector2i(n, mt).minCoeff();
    for (Eigen::Index j = 0; j < nt_limit; j++) {
        Eigen::Index extent = Eigen::Vector2i(mt-j, k.rows()).minCoeff();
        toeplitz.block(j,j, extent, 1) = toeplitz_col.topRows(extent);
    }
    return toeplitz;
}


Eigen::VectorXd diagonal_loading_deconv(const Eigen::VectorXd &input, const␣
↪Eigen::VectorXd &kernel, const double alpha=0.1) {
    Eigen::Index n = input.rows();
    Eigen::MatrixXd eye(n, n);
    eye.setIdentity();
    Eigen::MatrixXd c = convolution_matrix(kernel, n).topRows(n);
    Eigen::MatrixXd ct = c.transpose();
    Eigen::BDCSVD<Eigen::MatrixXd> svd(ct * c + alpha * eye, Eigen::ComputeThinU |␣
↪Eigen::ComputeThinV);
    Eigen::VectorXd x = svd.solve(ct * input);
    return x;
}


Eigen::VectorXd lowpass_fft_deconv(const Eigen::VectorXd &input, const␣
↪Eigen::VectorXd &kernel, const std::string &flag, double stab=0.01) {
```

(continues on next page)

(continued from previous page)

```
    // Map the input signature data to tensors (shared memory)
    auto input_len = input.rows();
    auto kernel_len = kernel.rows();

    // Compute cumulative length of input and kernel;
    auto N = input_len + kernel_len - 1;
    auto M = fft_next_good_size(N);

    // Create padded FFT inputs
    Eigen::FFT<double> fft;
    Eigen::VectorXd input_padded(M);
    Eigen::VectorXd kernel_padded(M);
    input_padded.setZero();
    kernel_padded.setZero();
    input_padded.topRows(input_len) = input;
    kernel_padded.topRows(kernel_len) = kernel;

    // Take the forward ffts
    Eigen::VectorXcd input_transformed(M);
    Eigen::VectorXcd kernel_transformed(M);
    fft.fwd(input_transformed, input_padded);
    fft.fwd(kernel_transformed, kernel_padded);

    // Magnitude of the transformed kernel and input
    Eigen::ArrayXd input_mag = pow(input_transformed.array().real(), 2.0) + pow(input_
↪transformed.array().imag(), 2.0);
    input_mag = pow(input_mag, 0.5);
    Eigen::ArrayXd kernel_mag = pow(kernel_transformed.array().real(), 2.0) +␣
↪pow(kernel_transformed.array().imag(), 2.0);
    kernel_mag = pow(kernel_mag, 0.5);

    // Double check plot
//    Figure fig = Figure();
//    ScatterPlot p3 = ScatterPlot();
//    p3.x = Eigen::ArrayXd::LinSpaced(M, 1, M);
//    p3.y = input_mag;
//    p3.name = "input mag";
//    fig.add(p3);
//    ScatterPlot p2 = ScatterPlot();
//    p2.x = Eigen::ArrayXd::LinSpaced(M, 1, M);
//    p2.y = kernel_mag;
//    p2.name = "kernel mag";
//    fig.add(p2);

    // Deconvolve by element-wise division, stabilising divide-by-0 errors based on␣
↪the 1% of magnitude of the kernel
    Eigen::VectorXcd inter(M);
    inter.setZero();

    if (stab > 0) {
        double kernel_cutoff_magnitude = kernel_mag.maxCoeff() * stab;
        Eigen::Index ctr = 0;
        Eigen::Index location = -1;
        for (ctr = 0; ctr < M; ctr++) {
            if (kernel_mag(ctr) > kernel_cutoff_magnitude) {
                inter(ctr) = input_transformed(ctr) / kernel_transformed(ctr);
```

(continues on next page)

(continued from previous page)

```
            } else {
                if (location == -1) {
                    location = ctr;
                };
//                  break;
                inter(ctr) = 0;
            }
        }
        std::cout << "LOCATION " << location << std::endl;

        // The above works on its own, but produces high frequency ringing, because␣
↪of the sharp cutoff low pass filter.
        // The ctr is now set at the point where we want the lowpass filter to fade␣
↪out entirely, so determine a
        // gaussian form between position 0 and here

//        inter = input_transformed.array() / kernel_transformed.array();

        // TODO this can be done in-place to avoid duplicating memory but we want to␣
↪plot for the time being
        Eigen::ArrayXd map_frequency = Eigen::ArrayXd::LinSpaced(location, -5.0, 5);
        Eigen::ArrayXd low_pass(M);
        low_pass.setZero();
        for (auto i = 0; i < map_frequency.rows(); i++) {
            low_pass(i) = 0.5 * (1.0 - std::erf(map_frequency(i)));
        }

        // We're working with a dual-sided FFT so mirror the filter
        low_pass = low_pass + low_pass.reverse();

        // Apply the low pass filter
        inter = inter.array() * low_pass;

        // Debug plot
//        ScatterPlot p5 = ScatterPlot();
//        p5.x = Eigen::ArrayXd::LinSpaced(low_pass.rows(), 1, low_pass.rows());
//        p5.y = low_pass;
//        p5.name = "lowpass magnitude";
//        fig.add(p5);

    } else {
        inter = input_transformed.array() / kernel_transformed.array();
    }

//    for (auto k = 0; k<M; k++) {
//        if ((k > 4) && (k < M-5)) {
//            inter(k) = 0.0;
//        }
//    }

//    std::cout << "Smoothing the inter" <<std::endl;
//    Eigen::ArrayXd inter_sm_imag(inter.size());
//    inter_sm_imag.segment(2, M-5) = (inter.imag().segment(0, M-5) + inter.imag().
↪segment(1, M-5) + inter.imag().segment(2, M-5) + inter.imag().segment(3, M-5)  +␣
↪inter.imag().segment(4, M-5)) / 5.0;
//    inter_sm_imag(1) = (inter.imag()(0) + inter.imag()(1) + inter.imag()(2))/3.0;
//    inter_sm_imag(M-2) = (inter.imag()(M-3) + inter.imag()(M-2) + inter.imag()(M-
↪1))/3.0;
```

(continues on next page)

(continued from previous page)

```
//    inter.imag() = inter_sm_imag;
//    Eigen::ArrayXd inter_sm_real(inter.size());
//    inter_sm_real.segment(2, M-5) = (inter.real().segment(0, M-5) + inter.real().
↪segment(1, M-5) + inter.real().segment(2, M-5) + inter.real().segment(3, M-5)  +␣
↪inter.real().segment(4, M-5)) / 5.0;
//    inter_sm_real(1) = (inter.real()(0) + inter.real()(1) + inter.real()(2))/3.0;
//    inter_sm_real(M-2) = (inter.real()(M-3) + inter.real()(M-2) + inter.real()(M-
↪1))/3.0;
//    inter.real() = inter_sm_real;


//    Eigen::ArrayXd inter_mag = pow(inter.array().real(), 2.0) + pow(inter.array().
↪imag(), 2.0);
//    inter_mag = pow(inter_mag, 0.5);
    Eigen::ArrayXd inter_mag = inter.array().imag();
//    ScatterPlot p4 = ScatterPlot();
//    p4.x = Eigen::ArrayXd::LinSpaced(M, 1, M);
//    p4.y = inter_mag;
//    p4.name = "inter mag";
//    fig.add(p4);
//
//    fig.write("check_that_fft_deconv_behaves_" + flag + ".json");
    // Inverse FFT
    Eigen::VectorXd out(M);
    out.setZero();
    fft.inv(out, inter);

    // Smooth the output
    std::cout << "Smoothing the out" <<std::endl;
    Eigen::ArrayXd inter_sm_imag(out.size());
    Eigen::ArrayXd out_sm(out.size());
    out_sm.segment(2, M-5) = (out.segment(0, M-5) + out.segment(1, M-5) + out.
↪segment(2, M-5) + out.segment(3, M-5)  + out.segment(4, M-5)) / 5.0;
    out_sm(1) = (out(0) + out(1) + out(2))/3.0;
    out_sm(M-2) = (out(M-3) + out(M-2) + out(M-1))/3.0;
    out = out_sm;

    // Crop to the size of the input vector
    out = out.topRows(input_len);

    return out;
}

}  /* namespace utilities */



#endif //ES_FLOW_CONV_H
```

Includes

- Eigen/Core
- Eigen/Dense
- math.h

- `unsupported/Eigen/FFT`

Included By

- *File adem.h*
- *File signature.h*

Namespaces

- *Namespace utilities*

Functions

- *Function utilities::conv*
- *Function utilities::convolution_matrix*
- *Function utilities::diagonal_loading_deconv*
- *Template Function utilities::fft_next_good_size*
- *Function utilities::lowpass_fft_deconv*

File cumsum.h

Parent directory (`source/utilities`)

Contents

Definition (`source/utilities/cumsum.h`)

Program Listing for File cumsum.h

Return to documentation for file (`source/utilities/cumsum.h`)

```
/*
 * cumsum.h Cumulative sum of a vector
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
```

(continues on next page)

(continued from previous page)

```
 * ---
 *
 * This file is adapted from part of libigl, a simple c++ geometry processing library.
 *
 * Copyright (C) 2013 Alec Jacobson <alecjacobson@gmail.com>
 *
 * This Source Code Form is subject to the terms of the Mozilla Public License
 * v. 2.0. If a copy of the MPL was not distributed with this file, You can
 * obtain one at http://mozilla.org/MPL/2.0/.
 *
 */

#ifndef ES_FLOW_CUMSUM_H
#define ES_FLOW_CUMSUM_H

#include <Eigen/Core>
#include <Eigen/Dense>


namespace utilities {

/* Computes a cumulative sum down the columns (or across the rows) of X
 *
 * @tparam DerivedX Type of matrix/array X
 * @tparam DerivedY Type of matrix/array Y
 * @param[in] X    m by n Matrix to be cumulatively summed.
 * @param[in] dim  dimension to take cumulative sum (1 goes down columns, 2 across␣
↪rows). Default 1.
 * @param[out] Y   m by n Matrix containing cumulative sum.
 */
template <typename DerivedX, typename DerivedY>
EIGEN_STRONG_INLINE void cumsum(Eigen::PlainObjectBase<DerivedY > & y, const␣
↪Eigen::MatrixBase<DerivedX > & x, const int dim=1) {

    y.resizeLike(x);

    Eigen::Index num_outer = (dim == 1 ? x.cols() : x.rows() );
    Eigen::Index num_inner = (dim == 1 ? x.rows() : x.cols() );

    // This has been optimized so that dim = 1 or 2 is roughly the same cost.
    // (Optimizations assume ColMajor order)
    if (dim == 1) {
        // TODO multithread the outer loop
        for(int o = 0; o<num_outer; o++)
        {
            typename DerivedX::Scalar sum = 0;
            for (int i = 0;i<num_inner;i++) {
                sum += x(i,o);
                y(i,o) = sum;
            }
        }
    } else {
        for (int i = 0; i<num_inner; i++) {
            // TODO multithread the inner loop
            for (int o = 0; o<num_outer; o++) {
                if (i == 0) {
                    y(o,i) = x(o,i);
```

(continues on next page)

(continued from previous page)

```
                } else {
                    y(o,i) = y(o,i-1) + x(o,i);
                }
            }
        }
    }
}

}

#endif //ES_FLOW_CUMSUM_H
```

Includes

- `Eigen/Core`
- `Eigen/Dense`

Namespaces

- *Namespace utilities*

Functions

- *Template Function utilities::cumsum*

File cumtrapz.h

Parent directory (`source/utilities`)

Contents

Definition (`source/utilities/cumtrapz.h`)

Program Listing for File cumtrapz.h

Return to documentation for file (`source/utilities/cumtrapz.h`)

```
/*
 * cumtrapz.h Cumulative Trapezoidal Numerical Integration for Eigen
 *
 * Author:              Tom Clark  (thclark @ github)
 *
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_CUMTRAPZ_H
#define ES_FLOW_CUMTRAPZ_H

#include <Eigen/Dense>


namespace utilities {


template<typename Derived, typename OtherDerived>
EIGEN_STRONG_INLINE void cumsum(Eigen::ArrayBase<OtherDerived> const & out, const␣
↪Eigen::ArrayBase<Derived>& in)
{
    Eigen::ArrayBase<OtherDerived>& out_ = const_cast< Eigen::ArrayBase<OtherDerived>&
↪ >(out);
    out_ = in;
    for (int i = 0 ; i < out_.cols() ; ++i)
    {
        for (int j = 1 ; j < out_.rows() ; ++j)
        {
            out_.coeffRef(j,i) += out_.coeff(j-1,i);
        }
    }
}
// Remove template specialisation from doc (causes duplicate) @cond
template<typename Derived>
EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> cumsum(const Eigen::ArrayBase<Derived>& y)
{
    Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar, Eigen::ArrayBase<Derived>
↪::RowsAtCompileTime, Eigen::ArrayBase<Derived>::ColsAtCompileTime> z;
    cumsum(z,y);
    return z;
}
// @endcond


template<typename Derived, typename OtherDerived>
EIGEN_STRONG_INLINE void cumtrapz(Eigen::ArrayBase<OtherDerived> const & out, const␣
↪Eigen::ArrayBase<Derived>& in)
{
    Eigen::ArrayBase<OtherDerived>& out_ = const_cast< Eigen::ArrayBase<OtherDerived>&
↪ >(out);
    out_.derived().setZero(in.rows(), in.cols());
    cumsum(out_.bottomRows(out_.rows()-1), (in.topRows(in.rows()-1) + in.
↪bottomRows(in.rows()-1)) * 0.5);
}
// Remove template specialisation from doc (causes duplicate) @cond
template<typename Derived>
EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> cumtrapz(const Eigen::ArrayBase<Derived>& y)
```

(continues on next page)

(continued from previous page)

```
{
    Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar, Eigen::ArrayBase<Derived>
↪::RowsAtCompileTime, Eigen::ArrayBase<Derived>::ColsAtCompileTime> z;
    cumtrapz(z,y);
    return z;
}
// @endcond


template<typename DerivedX, typename DerivedY, typename DerivedOut>
EIGEN_STRONG_INLINE void cumtrapz(Eigen::ArrayBase<DerivedOut> const & out, const␣
↪Eigen::ArrayBase<DerivedX>& in_x, const Eigen::ArrayBase<DerivedY>& in_y)
{
    // Input size check
    eigen_assert(in_x.rows() == in_y.rows());
    eigen_assert(in_x.cols() == 1);

    // Get dx for each piece of the integration
    Eigen::Array<typename Eigen::ArrayBase<DerivedX>::Scalar, Eigen::ArrayBase
↪<DerivedX>::RowsAtCompileTime, Eigen::ArrayBase<DerivedX>::ColsAtCompileTime> dx;
    dx = (in_x.bottomRows(in_x.rows()-1) - in_x.topRows(in_x.rows()-1));

    // Get the average heights of the trapezoids
    Eigen::Array<typename Eigen::ArrayBase<DerivedY>::Scalar, Eigen::ArrayBase
↪<DerivedY>::RowsAtCompileTime, Eigen::ArrayBase<DerivedY>::ColsAtCompileTime> inter;
    inter = (in_y.topRows(in_y.rows()-1) + in_y.bottomRows(in_y.rows()-1)) * 0.5;

    // Multiply by trapezoid widths to give areas of the trapezoids.
    // NB Broadcasting with *= only works for arrayX types, not arrayXX types like dx
    //  inter *= dx;
    for (int i = 0 ; i < inter.cols() ; ++i)
    {
        for (int j = 0 ; j < inter.rows() ; ++j)
        {
            inter(j,i) *= dx(j,0);
        }
    }

    // Initialise output
    Eigen::ArrayBase<DerivedOut>& out_ = const_cast< Eigen::ArrayBase<DerivedOut>& >
↪(out);
    out_.derived().setZero(in_y.rows(), in_y.cols());

    // Perform the cumulative sum down the columns
    cumsum(out_.bottomRows(out_.rows()-1), inter);
}
// Remove template specialisation from doc (causes duplicate) @cond
template<typename DerivedX, typename DerivedY, typename DerivedOut>
EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> cumtrapz(const Eigen::ArrayBase<DerivedX>& x, const␣
↪Eigen::ArrayBase<DerivedY>& y)
{
    Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar, Eigen::ArrayBase
↪<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>::ColsAtCompileTime> z;
    cumtrapz(z, x, y);
    return z;
```

(continues on next page)

(continued from previous page)

```
}

}  /* namespace utilities */

#endif //ES_FLOW_CUMTRAPZ_H
```

Includes

- `Eigen/Dense`

File data_types.h

Parent directory (`source`)

Contents

Definition (`source/data_types.h`)

Program Listing for File data_types.h

Return to documentation for file (`source/data_types.h`)

```
/*
 * data_types.h Definitions for different input data types
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_DATA_TYPES_H
#define ES_FLOW_DATA_TYPES_H

#include <string>
#include "matio.h"
#include <eigen3/Eigen/Core>
#include "io/variable_readers.h"


namespace es {
```

(continues on next page)

(continued from previous page)

```
class BasicLidar {
public:
    const std::string type = "lidar_basic";
    VectorXd t;
    VectorXd z;
    Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> u;
    Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> v;
    Eigen::Array<double, Eigen::Dynamic, Eigen::Dynamic> w;
    Eigen::Vector3d position;
    double half_angle;
    struct {
        std::string t;
        std::string z;
        std::string u;
        std::string v;
        std::string w;
        std::string position;
        std::string half_angle;
    }units;

    void read(mat_t *matfp, bool print_var = true) {

        // Straightforward reads
        t = readVectorXd(matfp, "t", print_var);
        z = readVectorXd(matfp, "z", print_var);
        u = readArrayXXd(matfp, "u", print_var);
        v = readArrayXXd(matfp, "v", print_var);
        w = readArrayXXd(matfp, "w", print_var);
        half_angle = readDouble(matfp, "half_angle", print_var);

        // Handle initialisation of position as a two element vector, zero padded␣
↪(elevation = 0) and as a three
        // element vector.
        Eigen::VectorXd pos = readVectorXd(matfp, "position", print_var);
        if (pos.size() == 2) {
            position = Vector3d(pos(0), pos(1), 0.0);
        }else {
            position = Vector3d(pos(0), pos(1), pos(2));
        }

        // TODO read in and tests on units structure

    }

};

} /* namespace es */

#endif // ES_FLOW_DATA_TYPES_H
```

Includes

- eigen3/Eigen/Core

- io/variable_readers.h (*File variable_readers.h*)
- matio.h
- string

Included By

- *File readers.h*

Namespaces

- *Namespace es*

Classes

- *Class BasicLidar*

File definitions.h

Parent directory (source)

Contents

- *Definition (source/definitions.h)*
- *Included By*
- *Defines*

Definition (source/definitions.h)

Program Listing for File definitions.h

Return to documentation for file (source/definitions.h)

```
/*
 * definitions.h Constants, definitions and identities
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_DEFINITIONS_H
#define ES_FLOW_DEFINITIONS_H

// Rotational speed of the world in rad/s
#define OMEGA_WORLD 7.2921159e-05
```

(continues on next page)

(continued from previous page)

```
// von Karman constant
#define KAPPA_VON_KARMAN 0.41

// Degrees based trigonometry
#define sind(x) (sin(fmod((x), 360.0) * M_PI / 180.0))
#define cosd(x) (cos(fmod((x), 360.0) * M_PI / 180.0))
#define asind(x) (asin(x) * 180.0 / M_PI)
#define acosd(x) (acos(x) * 180.0 / M_PI)
#define tand(x) (tan(x * M_PI / 180.0) * M_PI / 180.0)

#endif // ES_FLOW_DEFINITIONS_H
```

Included By

- *File fit.h*
- *File profile.cpp*
- *File stress.h*
- *File veer.h*

Defines

- *Define acosd*
- *Define asind*
- *Define cosd*
- *Define KAPPA_VON_KARMAN*
- *Define OMEGA_WORLD*
- *Define sind*
- *Define tand*

File exceptions.h

Parent directory (`source`)

Contents

Definition (source/exceptions.h)

Program Listing for File exceptions.h

Return to documentation for file (source/exceptions.h)

```
/*
 * exceptions.h Customised exceptions for appropriate and fine grained error handling
 *
 * Author:              Tom Clark  (thclark@github)
 *
 * Copyright (c) 2017-9 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_EXCEPTIONS_H
#define ES_FLOW_EXCEPTIONS_H


#include <exception>


namespace es {


struct NotImplementedException : public std::exception {
    std::string message = "Not yet implemented";
    const char *what() const throw() {
        return message.c_str();
    }
};

struct InvalidEddyTypeException : public std::exception {
    std::string message = "Unknown eddy type. Eddy type string must be one of 'A', 'B1
↪', 'B2', 'B3' or 'B4'.";
    const char *what() const throw() {
        return message.c_str();
    }

};

}

#endif // ES_FLOW_EXCEPTIONS_H
```

Includes

- exception (*File exceptions.h*)

Namespaces

- *Namespace es*

Classes

- *Struct InvalidEddyTypeException*
- *Struct NotImplementedException*

File filter.h

Parent directory (`source/utilities`)

Contents

Definition (`source/utilities/filter.h`)

Program Listing for File filter.h

Return to documentation for file (`source/utilities/filter.h`)

```
/*
 * filter.h One-dimensional polynomial based digital filter for eigen arrayXd
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_FILTER_H
#define ES_FLOW_FILTER_H

#include <vector>
#include <iostream>
#include <Eigen/Dense>
#include <Eigen/Core>
#include <stdexcept>


namespace utilities {


template<typename DerivedOut, typename DerivedIn>
void filter(Eigen::ArrayBase<DerivedOut> &y,
            const Eigen::ArrayBase<DerivedIn> &b,
            const Eigen::ArrayBase<DerivedIn> &a,
```

(continues on next page)

(continued from previous page)

```
            const Eigen::ArrayBase<DerivedIn> &x) {

    // Check for 1D inputs
    eigen_assert((a.cols() == 1) || (a.rows() == 1));
    eigen_assert((b.cols() == 1) || (a.rows() == 1));
    eigen_assert((x.cols() == 1) || (a.rows() == 1));

    // Initialise output
    Eigen::ArrayBase<DerivedOut> &y_ = const_cast< Eigen::ArrayBase<DerivedOut> & >
↪(y);
    y_.derived().setZero(x.rows(), x.cols());

    // Direct Form II transposed standard difference equation
    typename DerivedOut::Scalar tmp;
    Eigen::Index i;
    Eigen::Index j;
    for (i = 0; i < x.size(); i++) {
        tmp = 0.0;
        j = 0;
        y_(i) = 0.0;

        for (j = 0; j < b.size(); j++) {
            if (i - j < 0) continue;
            tmp += b(j) * x(i - j);
        }

        for (j = 1; j < a.size(); j++) {
            if (i - j < 0) continue;
            tmp -= a(j) * y_(i - j);
        }

        tmp /= a(0);
        y_(i) = tmp;
    }
}

template<typename T>
void filter(std::vector<T> &y, const std::vector<T> &b, const std::vector<T> &a,␣
↪const std::vector<T> &x) {

    y.resize(0);
    y.resize(x.size());

    for (int i = 0; i < x.size(); i++) {
        T tmp = 0.0;
        int j = 0;
        y[i] = 0.0;
        for (j = 0; j < b.size(); j++) {
            if (i - j < 0) continue;
            tmp += b[j] * x[i - j];
        }

        for (j = 1; j < a.size(); j++) {
            if (i - j < 0) continue;
            tmp -= a[j] * y[i - j];
        }
```

(continues on next page)

(continued from previous page)

```
        tmp /= a[0];
        y[i] = tmp;
    }
}

template<typename DerivedIn, typename DerivedOut>
void deconv(Eigen::ArrayBase<DerivedOut> &z,
            const Eigen::ArrayBase<DerivedIn> &b,
            const Eigen::ArrayBase<DerivedIn> &a) {

    eigen_assert(a(0) != 0);
    eigen_assert((a.cols() == 1) || (a.rows() == 1));
    eigen_assert((b.cols() == 1) || (b.rows() == 1));

    auto na = a.size();
    auto nb = b.size();

    // Initialise output
    Eigen::ArrayBase<DerivedOut> &z_ = const_cast< Eigen::ArrayBase<DerivedOut> & >
↪(z);

    // Cannot deconvolve a longer signal out of a smaller one
    if (na > nb) {
        std::cout << "Warning! You cannot deconvolve a longer signal (a) out of a␣
↪shorter one (b)!!" << std::endl;
        z_.derived().setZero(1, 1);
        return;
    }
    z_.derived().setZero(nb - na + 1, 1);

    // Deconvolution and polynomial division are the same operations as a digital␣
↪filter's impulse response B(z)/A(z):
    Eigen::Array<typename DerivedOut::Scalar, Eigen::Dynamic, 1> impulse;
    impulse.setZero(nb - na + 1, 1);
    impulse(0) = 1;
    filter(z_, b, a, impulse);
}

}  /* namespace utilities */

#endif //ES_FLOW_FILTER_H
```

Includes

- Eigen/Core
- Eigen/Dense
- iostream
- stdexcept
- vector

Included By

- *File adem.h*
- *File signature.h*

Namespaces

- *Namespace utilities*

Functions

- *Template Function utilities::deconv*
- *Template Function utilities::filter(std::vector<T>&, const std::vector<T>&, const std::vector<T>&, const std::vector<T>&)*
- *Template Function utilities::filter(Eigen::ArrayBase<DerivedOut>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&, const Eigen::ArrayBase<DerivedIn>&)*

File fit.h

Parent directory (`source`)

Contents

Definition (`source/fit.h`)

Program Listing for File fit.h

Return to documentation for file (`source/fit.h`)

```
/*
 * Fit.h Use ceres-solver to fit mean speed and spectra to measured data
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */
```

(continues on next page)

(continued from previous page)

```
#ifndef ES_FLOW_FIT_H
#define ES_FLOW_FIT_H

#include <stdio.h>
#include "ceres/ceres.h"
#include <Eigen/Core>

#include "relations/velocity.h"
#include "definitions.h"

using ceres::AutoDiffCostFunction;
using ceres::DynamicAutoDiffCostFunction;
using ceres::CostFunction;
using ceres::Problem;
using ceres::Solver;
using ceres::Solve;
using ceres::CauchyLoss;


typedef Eigen::Array<bool, 5, 1> Array5b;
typedef Eigen::Array<double, 5, 1> Array5d;


namespace es {


struct PowerLawSpeedResidual {
    PowerLawSpeedResidual(double z, double u, double z_ref=1.0, double u_ref=1.0) : z_
↪(z), u_(u), u_ref_(u_ref), z_ref_(z_ref) {}
    template <typename T> bool operator()(const T* const alpha, T* residual) const {

        residual[0] = u_ - power_law_speed(T(z_), u_ref_, z_ref_, alpha[0]);
        return true;
    }
private:
    // Observations for a sample
    const double u_;
    const double z_;
    const double u_ref_;
    const double z_ref_;
};


double fit_power_law_speed(const Eigen::ArrayXd &z, const Eigen::ArrayXd &u, const␣
↪double z_ref=1.0, const double u_ref=1.0) {

    // Define the variable to solve for with its initial value. It will be mutated in␣
↪place by the solver.
    double alpha = 0.3;

    // TODO Assert that z.size() == u.size()

    // Build the problem
    Problem problem;
    Solver::Summary summary;
    Solver::Options options;
```

(continues on next page)

(continued from previous page)

```
    options.max_num_iterations = 400;
    options.linear_solver_type = ceres::DENSE_QR;
    options.minimizer_progress_to_stdout = true;
//std::cout << options << std::endl;
    // Set up the only cost function (also known as residual), using cauchy loss␣
↪function for
    // robust fitting and auto-differentiation to obtain the derivative (jacobian)
    for (auto i = 0; i < z.size(); i++) {
        CostFunction *cost_function = new AutoDiffCostFunction<PowerLawSpeedResidual,␣
↪1, 1>(
            new PowerLawSpeedResidual(z[i], u[i], z_ref, u_ref));
        problem.AddResidualBlock(cost_function, new CauchyLoss(0.5), &alpha);
    }

    // Run the solver
    Solve(options, &problem, &summary);
    std::cout << summary.BriefReport() << "\n";
    std::cout << "Initial alpha: " << 0.3 << "\n";
    std::cout << "Final   alpha: " << alpha << "\n";

    return alpha;
}


struct LewkowiczSpeedResidual {

    LewkowiczSpeedResidual(double z, double u, const Array5b &fixed_params, const␣
↪Array5d &initial_params) : z_(z), u_(u), fixed_params_(fixed_params), initial_
↪params_(initial_params) { }

    template <typename T> bool operator()(T const* const* parameters, T* residual)␣
↪const {

        // Mask out fixed and variable parameters
        // TODO Once Eigen 3.4.x comes out, reduce this code to use the parameter␣
↪mask as a logical index.
        //  See this issue which will allow such indexing in eigen: http://eigen.
↪tuxfamily.org/bz/show_bug.cgi?id=329#c27
        int param_ctr = 0;
        std::vector<T> params(5);
        for (auto i = 0; i < 5; i++) {
            if (fixed_params_(i)) {
                params[i] = T(initial_params_(i));
            } else {
                params[i] = parameters[param_ctr][0];
                param_ctr += 1;
            }
        }
        T spd = lewkowicz_speed(T(z_), params[0], params[1], params[2], params[3],␣
↪params[4]);
        residual[0] = u_ - spd;
        return true;
    }
private:
    const double u_;
    const double z_;
```

(continues on next page)

(continued from previous page)

```
    Array5b fixed_params_;
    Array5d initial_params_;
};


Array5d fit_lewkowicz_speed(const Eigen::ArrayXd &z, const Eigen::ArrayXd &u, const␣
↪Array5b &fixed_params, const Array5d &initial_params, bool print_report = true) {

    // Initialise a results array, and get a vector of pointers to the 'unfixed'␣
↪parameters (the ones to optimise)
    Array5d final_params(initial_params);
    std::vector<double *> unfixed_param_ptrs;
    for (int p_ctr = 0; p_ctr <5; p_ctr++) {
        if (!fixed_params(p_ctr)) { unfixed_param_ptrs.push_back(final_params.
↪data()+p_ctr); };
    }

    // Build the problem
    Problem problem;
    Solver::Summary summary;
    Solver::Options options;
    options.max_num_iterations = 400;
    options.minimizer_progress_to_stdout = print_report;

    for (auto i = 0; i < z.size(); i++) {

        // Set the stride such that the entire set of derivatives is computed at once␣
↪(since we have maximum 5)
        auto cost_function = new DynamicAutoDiffCostFunction<LewkowiczSpeedResidual,␣
↪5>(
            new LewkowiczSpeedResidual(z[i], u[i], fixed_params, initial_params)
        );

        // Add N parameters, where N is the number of free parameters in the problem
        auto pc = 0;
        for (auto p_ctr = 0; p_ctr < 5; p_ctr++) {
            if (!fixed_params(p_ctr)) {
                cost_function->AddParameterBlock(1);
                pc += 1;
            }
        }
        cost_function->SetNumResiduals(1);
        problem.AddResidualBlock(cost_function, new CauchyLoss(0.5), unfixed_param_
↪ptrs);
    }

    Solve(options, &problem, &summary);

    if (print_report) {
        std::cout << summary.FullReport() << std::endl;
        std::cout << "Initial values: " << initial_params.transpose() << std::endl;
        std::cout << "Final   values: " << final_params.transpose() << std::endl;
    }

    return final_params;
}
```

(continues on next page)

(continued from previous page)

```
Array5d fit_lewkowicz_speed(const Eigen::ArrayXd &z, const Eigen::ArrayXd &u, bool␣
↪print_report = true) {

    // Set default initial parameters
    double pi_coles = 0.5;
    double kappa = KAPPA_VON_KARMAN;
    double shear_ratio = 20;
    double u_inf = u.maxCoeff();
    double delta_c = 1000;
    Array5d initial_params;
    initial_params << pi_coles, kappa, u_inf, shear_ratio, delta_c;

    // Set default fixed parameters (von karman constant and boundary layer thickness)
    Array5b fixed_params;
    fixed_params << false, true, false, false, true;

    return fit_lewkowicz_speed(z, u, fixed_params, initial_params, print_report);

}

} /* namespace es */


#endif // ES_FLOW_FIT_H
```

Includes

- `Eigen/Core`
- `ceres/ceres.h`
- `definitions.h` (*File definitions.h*)
- `relations/velocity.h` (*File velocity.h*)
- `stdio.h`

Namespaces

- *Namespace es*

Classes

- *Struct LewkowiczSpeedResidual*
- *Struct PowerLawSpeedResidual*

Functions

- *Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, const Array5b&, const Array5d&, bool)*

- *Function es::fit_lewkowicz_speed(const Eigen::ArrayXd&, const Eigen::ArrayXd&, bool)*
- *Function es::fit_power_law_speed*

Typedefs

- *Typedef Array5b*
- *Typedef Array5d*

File flow.h

Parent directory (`source`)

Contents

- *Definition* (`source/flow.h`)

Definition (`source/flow.h`)

Program Listing for File flow.h

Return to documentation for file (`source/flow.h`)

```
/*
 * flow.h Sketching out public api for the flow library
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_FLOW_H
#define ES_FLOW_FLOW_H




#endif //ES_FLOW_FLOW_H
```

File how_to.h

Parent directory (`source`)

Contents

- *Definition* (`source/how_to.h`)

Definition (source/how_to.h)

Program Listing for File how_to.h

Return to documentation for file (source/how_to.h)

```
/*
 * how_to.h  Chunks of code that show us how to do things
 *
 *  May not be complete, may not compile. Not included in any build targets. Just␣
↪copy, paste, adapt to our own uses.
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */


// Remove template specialisation from doc (causes duplicate) @cond

/
↪*****************************************************************************************
 * HOW TO DO PIECEWISE CUBIC HERMITE INTERPOLATION
␣
↪*****************************************************************************************
↪


// TODO This is a work in progress. It gives a pchip spline, but not one which is␣
↪shape preserving. Need gradient
//  conditions at the maxima, minima and endpoints.
#include <iostream>
#include <algorithm>
#include <math.h>
#include <Eigen/Core>
#include <Eigen/Dense>
double evaluate(const double xi, const Eigen::VectorXd &x, const␣
↪ceres::CubicInterpolator<ceres::Grid1D<double> > &interpolator){

    const Eigen::Index k = x.rows();
    Eigen::Index n_rows = x.rows()-1;
    Eigen::VectorXd x_percent;
    Eigen::VectorXd diff = x.bottomRows(n_rows) - x.topRows(n_rows);
//    if ((diff <= 0.0).any()) {
//        throw std::invalid_argument("Input values x must be strictly increasing");
//    }
    cumsum(x_percent, diff);
    x_percent = x_percent / x_percent(n_rows-1);
    double x_max = x.maxCoeff();
    double x_min = x.minCoeff();

    // Out of range xi values are constrained to the endpoints
    double bounded_target = std::max(std::min(xi, x_max), x_min);

    // Run a binary search for where the node is in the grid. This makes the␣
↪algorithm O(log(N)) to evaluate one location
```

(continues on next page)

(continued from previous page)

```
// Thanks to: https://stackoverflow.com/questions/6553970/find-the-first-element-in-a-
↪sorted-array-that-is-greater-than-the-target
    Eigen::Index low = 0, high = k; // numElems is the size of the array i.e arr.
↪size()
    while (low != high) {
        Eigen::Index mid = (low + high) / 2; // Or a fancy way to avoid int overflow
        if (x[mid] <= bounded_target) {
            /* This index, and everything below it, must not be the first element
             * greater than what we're looking for because this element is no greater
             * than the element.
             */
            low = mid + 1;
        }
        else {
            /* This element is at least as large as the element, so anything after it␣
↪can't
             * be the first element that's at least as large.
             */
            high = mid;
        }
    }
    low = low - 1;

    /* Now, low and high surround to the element in question. */

    std::cout << low << " " << high << std::endl;

    double node_low = x[low];
    double node_high = x[high];

    std::cout << "node_low " << node_low << std::endl;
    std::cout << "node_high " << node_low << std::endl;
    std::cout << "low " << low << std::endl;
    std::cout << "high " << high << std::endl;


    double proportion = (bounded_target - node_low) / (node_high - node_low);
    double xi_mapped = double(low) + proportion;

    std::cout << "proportion " << proportion << std::endl;
    std::cout << "xi_mapped " << xi_mapped << std::endl;
    double f, dfdx;

    interpolator.Evaluate(xi_mapped, &f, &dfdx);
    std::cout << "evaluated: " << f << std::endl;

    return f;
}

double evaluateDirect(const double xi, const ceres::CubicInterpolator<ceres::Grid1D
↪<double> > &interpolator){
    // This doesn't work because it doesn't map the proportion of the way through␣
↪correctly
    double f, dfdx;

    interpolator.Evaluate(xi, &f, &dfdx);
    std::cout << "evaluated: " << f << std::endl;
```

(continues on next page)

(continued from previous page)

```
    return f;
}

TEST_F(InterpTest, test_pchip_interp_double) {

Eigen::VectorXd x(5);
Eigen::VectorXd y(5);
Eigen::VectorXd xi = Eigen::VectorXd::LinSpaced(100, 0, 5);
Eigen::VectorXd yi(100);
x << 2, 4.5, 8, 9, 10;
y << 6, 1, 10, 12, 19;

const Eigen::Index k = y.rows();
ceres::Grid1D<double> grid(y.data(), 0, k);
// TODO the interpolator, unlike MATLAB's routine, isn't shape preserving, nor does␣
↪it adjust for non-monotonic x.
//  So this works as an interpolant, but is pretty horrid.
ceres::CubicInterpolator<ceres::Grid1D<double> > interpolator(grid);

double x_max = x.maxCoeff();
double x_min = x.minCoeff();
for (Eigen::Index i=0; i<xi.rows(); i++) {
//yi[i] = evaluateDirect(xi[i], interpolator);
//xi[i] = xi[i] * ((x_max - x_min) / k) + x_min;
yi[i] = evaluate(xi[i], x, interpolator);
}


Figure fig = Figure();
ScatterPlot p_xy = ScatterPlot();
p_xy.x = x;
p_xy.y = y;
p_xy.name = "xy";
fig.add(p_xy);
ScatterPlot p_xiyi = ScatterPlot();
p_xiyi.x = xi;
p_xiyi.y = yi;
p_xiyi.name = "xiyi";
fig.add(p_xiyi);

// Add axis labeling
Layout lay = Layout("pchip check");
lay.xTitle("$x$");
lay.yTitle("$y$");
fig.setLayout(lay);

// Write figures
fig.write("check_pchip.json");

}

// TODO Once done, you can wrap this up into an interpolant much like in the interp.h␣
↪file:
```

(continues on next page)

(continued from previous page)

```
#include "ceres/ceres.h"
#include "ceres/cubic_interpolation.h"
#include "glog/logging.h"
#include "utilities/cumsum.h"

ceres::CubicInterpolator<ceres::Grid1D<double> > getInterpolator(Eigen::VectorXd␣
↪const &x_vec, Eigen::VectorXd const &y_vec) {
    const Eigen::Index k = y_vec.rows();
    // TODO file an issue over at ceres-solver. Grid1D is deeply unhelpful. At the␣
↪very least it should accept Eigen::Index instead of int.
    std::cout << "k: " << k << std::endl;
    ceres::Grid1D<double> grid(y_vec.data(), 0, k);
    ceres::CubicInterpolator<ceres::Grid1D<double> > interpolator(grid);
    return interpolator;

}

class PiecewiseCubicHermiteInterpolant {
public:
    // TODO template for arrays
    PiecewiseCubicHermiteInterpolant(Eigen::VectorXd const &x_vec, Eigen::VectorXd␣
↪const &y_vec)
        : k_(y_vec.rows()),
          interpolator_(getInterpolator(x_vec, y_vec)) {
        // TODO assert that x is strictly ascending or strictly descending
        std::cout << "HEREbf" <<std::endl;
        Eigen::Index n_rows = x_vec.rows();
        Eigen::VectorXd x_percent;
        cumsum(x_percent, x_vec.bottomRows(n_rows-1) - x_vec.topRows(n_rows-1));
    }


    double operator()(double xi) const {
        // x values need to be scaled down in extraction as well.
        std::cout << "evaluating" << std::endl;
        //
        double f, dfdx;
        interpolator_.Evaluate(xi, &f, &dfdx);
        std::cout << "evaluated" << std::endl;

        return f;
    }

    Eigen::VectorXd operator()(Eigen::VectorXd const &xi_vec) {
        Eigen::VectorXd yi_vec(xi_vec.rows());
        yi_vec = xi_vec.unaryExpr([this](double xi) { return this->operator()(xi); }).
↪transpose();
        return yi_vec;
    }

private:


//    double scaled_value(double x) const {
//        return (x - x_min) / (x_max - x_min);
//    }
```

(continues on next page)

(continued from previous page)

```
//    Eigen::RowVectorXd scaled_values(Eigen::VectorXd const &x_vec) const {
//        return x_vec.unaryExpr([this](double x) { return scaled_value(x); }).
↪transpose();
//    }

    // Number of samples in the input data
    Eigen::Index k_;

    // Interpolator, initialised with the grid of values
    const ceres::CubicInterpolator<ceres::Grid1D<double> > interpolator_;
};









/
↪*****************************************************************************************
 * HOW TO DIFFERENTIATE USING EIGEN::AUTODIFF
␣
↪*****************************************************************************************
↪


#include <unsupported/Eigen/AutoDiff>
template <typename T>
T myfun2(const double a, T const & b){
    T c = pow(sin(a),2.) + pow(cos(b),2.) + 1.;
    return c;
}

void test_scalar() {
    std::cout << "== test_scalar() ==" << std::endl;
    double a;
    a = 0.3;
    typedef Eigen::AutoDiffScalar<Eigen::VectorXd> AScalar;
    AScalar Ab;
    Ab.value() = 0.5;
    Ab.derivatives() = Eigen::VectorXd::Unit(1,0);
    AScalar Ac = myfun2(a,Ab);
    std::cout << "Result: " << Ac.value() << std::endl;
    std::cout << "Gradient: " << Ac.derivatives().transpose() << std::endl;
}


template <typename T>
T myfun(T const & a, T const & b){
    T c = pow(sin(a),2.) + pow(cos(b),2.) + 1.;
    return c;
}

void test_scalar(){
    std::cout << "== test_scalar() ==" << std::endl;
    // use with normal floats
    double a,b;
```

(continues on next page)

(continued from previous page)

```
    a = 0.3;
    b = 0.5;
    double c = myfun(a,b);
    std::cout << "Result: " << c << std::endl;

    // use with AutoDiffScalar
    typedef Eigen::AutoDiffScalar<Eigen::VectorXd> AScalar;
    AScalar Aa,Ab;
    Aa.value() = 0.3;
    Ab.value() = 0.5;
    Aa.derivatives() = Eigen::VectorXd::Unit(2,0); // This is a unit vector [1, 0]
    Ab.derivatives() = Eigen::VectorXd::Unit(2,1); // This is a unit vector [0, 1]
    AScalar Ac = myfun(Aa,Ab);
    std::cout << "Result: " << Ac.value() << std::endl;
    std::cout << "Gradient: " << Ac.derivatives().transpose() << std::endl;
}


/
↪*****************************************************************************************
 * HOW TO INTEGRATE USING THE NUMERICALINTEGRATION LIBRARY
␣
↪*****************************************************************************************
↪

#include <Eigen/Dense>
#include <Eigen/Core>
#include <iostream>
#include "NumericalIntegration.h"

/* Integrator for the velocity deficit
 *  We consider the example from:
 *
 *       http://www.gnu.org/software/gsl/manual/html_node/Numerical-integration-
↪examples.html
 *
 *       int_0^1 x^{-1/2} log(x) dx = -4
 *
 *  The integrator expects the user to provide a functor as shown below.
 */

template<typename Scalar>
class IntegrandExampleFunctor
{
public:
    IntegrandExampleFunctor(const Scalar alpha):m_alpha(alpha)
    {
        assert(alpha>0);
    }

    Scalar operator()(const Scalar x) const
    {
        assert(x>0);
        return log(m_alpha*x) / sqrt(x);
    }

    void setAlpha(const Scalar alpha)
```

(continues on next page)

(continued from previous page)

```
    {
        m_alpha = alpha;
    }
private:
    Scalar m_alpha;
};

double do_something(const double arg)
{
    // Define the scalar
    typedef double Scalar;

    // Define the functor
    Scalar alpha=1.;
    IntegrandExampleFunctor<Scalar> inFctr(alpha);

    //define the integrator
    Eigen::Integrator<Scalar> eigIntgtor(200);

    //define a quadrature rule
    Eigen::Integrator<Scalar>::QuadratureRule quadratureRule = Eigen::Integrator
↪<Scalar>::GaussKronrod61;

    //define the desired absolute and relative errors
    Scalar desAbsErr = Scalar(0.);
    Scalar desRelErr = Eigen::NumTraits<Scalar>::epsilon() * 50.;

    //integrate
    Scalar result = eigIntgtor.quadratureAdaptive(inFctr, Scalar(0.),Scalar(1.),␣
↪desAbsErr, desRelErr, quadratureRule);

    //expected result
    Scalar expected = Scalar(-4.);

    //print output
    size_t outputPrecision  = 18;
    std::cout<<std::fixed;
    std::cout<<"result          = "<<std::setprecision(outputPrecision)<<result<
↪<std::endl;
    std::cout<<"exact result    = "<<std::setprecision(outputPrecision)<<expected<
↪<std::endl;
    std::cout<<"actual error    = "<<std::setprecision(outputPrecision)<<(expected-
↪result)<<std::endl;

    return 0.0;
}

TEST_F(MyTest, test_integrator_example) {

// Test a basic integrator out
double arg = 0.0;
double res = do_something(arg);

}
```

(continues on next page)

(continued from previous page)

```
/
↪*****************************************************************************************
 * HOW TO FIT USING CERES-SOLVER
␣
↪*****************************************************************************************
↪


//#include <stdlib.h>
//#include <stdio.h>
//#include <unistd.h>
//#include <stdbool.h>
//#include "matio.h"
#include "ceres/ceres.h"

#include <Eigen/Core>


using ceres::AutoDiffCostFunction;
using ceres::CostFunction;
using ceres::Problem;
using ceres::Solver;
using ceres::Solve;

// A templated cost functor that implements the residual r = 10 -
// x. The method operator() is templated so that we can then use an
// automatic differentiation wrapper around it to generate its
// derivatives.
struct CostFunctor {
    template <typename T> bool operator()(const T* const x, T* residual) const {
        residual[0] = T(10.0) - x[0];
        return true;
    }
};

void my_fitting_func() {

    // Run the ceres solver

    // Define the variable to solve for with its initial value. It will be mutated in␣
↪place by the solver.
    double x = 0.5;
    const double initial_x = x;

    // Build the problem.
    Problem problem;

    // Set up the only cost function (also known as residual). This uses
    // auto-differentiation to obtain the derivative (jacobian).
    CostFunction *cost_function = new AutoDiffCostFunction<CostFunctor, 1, 1>(new␣
↪CostFunctor);
    problem.AddResidualBlock(cost_function, NULL, &x);

    // Run the solver!
    Solver::Options ceroptions;
    ceroptions.minimizer_progress_to_stdout = true;
    Solver::Summary summary;
```

(continues on next page)

(continued from previous page)

```
    Solve(ceroptions, &problem, &summary);
    std::cout << summary.BriefReport() << "\n";
    std::cout << "x : " << initial_x << " -> " << x << "\n";

}


/
↪*****************************************************************************************
 * HOW TO DO AN FFT USING EIGEN
␣
↪*****************************************************************************************
↪


#include <Eigen/Core>
#include <unsupported/Eigen/FFT>


void my_fft_func() {
    size_t dim_x = 28, dim_y = 126;
    Eigen::FFT<float> fft;
    Eigen::MatrixXf in = Eigen::MatrixXf::Random(dim_x, dim_y);
    Eigen::MatrixXcf out;
    out.setZero(dim_x, dim_y);

    for (int k = 0; k < in.rows(); k++) {
        Eigen::VectorXcf tmpOut(dim_x);
        fft.fwd(tmpOut, in.row(k));
        out.row(k) = tmpOut;
    }

    for (int k = 0; k < in.cols(); k++) {
        Eigen::VectorXcf tmpOut(dim_y);
        fft.fwd(tmpOut, out.col(k));
        out.col(k) = tmpOut;
    }
}


/
↪*****************************************************************************************
 * HOW TO DO AN FFT USING MKL DIRECTLY
␣
↪*****************************************************************************************
↪

#include "mkl_dfti.h"

void my_mkl_fft() {
    // Run an example FFT

    // Make meaningless data and a size vector
    float xf[200][100];
    MKL_LONG len[2] = {200, 100};

    // Create a decriptor, which is a pattern for what operation an FFT will undertake
```

(continues on next page)

(continued from previous page)

```
    DFTI_DESCRIPTOR_HANDLE fft;
    DftiCreateDescriptor (&fft, DFTI_SINGLE, DFTI_REAL, 2, len);
    DftiCommitDescriptor(fft);

    // Compute a forward transform, in-place on the data
    DftiComputeForward(fft, xf);

    // Free the descriptor
    DftiFreeDescriptor(&fft);

    std::cout << "FFT COMPLETE\n";
}


/
↪*****************************************************************************************
 * HOW TO PLOT REYNOLDS STRESS PROFILES, EDDY SIGNATURES AND STRENGTH / SCALE␣
↪DISTRIBUTIONS WITH CPPLOT
␣
↪*****************************************************************************************
↪

#include "cpplot.h"

void my_rs_plotting_func() {

    // Plot the Reynolds Stress profiles (comes from get_spectra)
    cpplot::Figure figa = cpplot::Figure();
    for (auto i = 0; i < 6; i++) {
        cpplot::ScatterPlot p = cpplot::ScatterPlot();
        p.x = Eigen::VectorXd::LinSpaced(data.reynolds_stress_a.rows(), 1, data.
↪reynolds_stress_a.rows());
        p.y = data.reynolds_stress_a.col(i).matrix();
        figa.add(p);
    }
    figa.write("test_t2w_rij_a.json");
    //    legend({'R13A'; 'R13B'})
    //    xlabel('\lambda_E')

    cpplot::Figure figb = cpplot::Figure();
    for (auto i = 0; i < 6; i++) {
        cpplot::ScatterPlot p = cpplot::ScatterPlot();
        p.x = Eigen::VectorXd::LinSpaced(data.reynolds_stress_b.rows(), 1, data.
↪reynolds_stress_b.rows());
        p.y = data.reynolds_stress_b.col(i).matrix();
        figb.add(p);
    }
    figb.write("test_t2w_rij_b.json");
    //    legend({'R13A'; 'R13B'})
    //    xlabel('\lambda_E')


    // Plot the eddy signatures
    cpplot::Figure fig2 = cpplot::Figure();
    cpplot::ScatterPlot ja = cpplot::ScatterPlot();
    ja.x = Eigen::VectorXd::LinSpaced(j13a.rows(), 1, j13a.rows());
    ja.y = j13a.matrix();
```

(continues on next page)

(continued from previous page)

```
    fig2.add(ja);
    cpplot::ScatterPlot jb = cpplot::ScatterPlot();
    jb.x = ja.x;
    jb.y = j13b.matrix();
    fig2.add(jb);
    fig2.write("test_t2w_j13ab.json");
    //    legend({'J13A'; 'J13B'})

    // Plot strength and scale distributions
    cpplot::Figure fig3 = cpplot::Figure();
    cpplot::ScatterPlot twa = cpplot::ScatterPlot();
    twa.x = Eigen::VectorXd::LinSpaced(minus_t2wa.rows(), 1, minus_t2wa.rows());
    twa.y = minus_t2wa.matrix();
    fig3.add(twa);
    cpplot::ScatterPlot twb = cpplot::ScatterPlot();
    twb.x = twa.x;
    twb.y = minus_t2wb.matrix();
    fig3.add(twb);
    fig3.write("test_t2w_t2wab.json");
    //    legend({'T^2\omegaA'; 'T^2\omegaB'})

}

/
↪*****************************************************************************************
 * HOW TO DO MAT FILE READ WRITE
␣
↪*****************************************************************************************
↪

#include "matio.h"

// Create the output file
mat_t *matfp;
matfp = Mat_CreateVer(options["o"].as<std::string>().c_str(), NULL, MAT_FT_MAT73);
if ( NULL == matfp ) {
std::string msg = "Error creating MAT file: ";
throw  msg + options["o"].as<std::string>();
}

// We haven't written any data - just close the file
Mat_Close(matfp);

std::cout << "MATIO TEST COMPLETE" << std::endl;




// @endcond
```

File integration.h

Parent directory (source/utilities)

Contents

Definition (source/utilities/integration.h)

Program Listing for File integration.h

Return to documentation for file (source/utilities/integration.h)

```
/*
 * integration.h Convenient wrappers around tbs1980's Numerical Integration module␣
↪for Eigen
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_INTEGRATION_H
#define ES_FLOW_INTEGRATION_H

#include <Eigen/Dense>
#include <Eigen/Core>
#include "NumericalIntegration.h"


namespace utilities {


template<typename T_functor>
Eigen::ArrayXd cumulative_integrate(const Eigen::ArrayXd &x, const T_functor &functor,
↪ const int max_subintervals=200)
{
    typedef double ScalarType;
    Eigen::ArrayXd integral(x.rows());
    integral.setZero();

    // Define the integrator and quadrature rule
    Eigen::Integrator<ScalarType> eigIntgtor(max_subintervals);
    Eigen::Integrator<ScalarType>::QuadratureRule quadratureRule = Eigen::Integrator
↪<ScalarType>::GaussKronrod61;

    // Define the desired absolute and relative errors
    auto desAbsErr = ScalarType(0.0);
    ScalarType desRelErr = Eigen::NumTraits<ScalarType>::epsilon() * 50.;
```

(continues on next page)

(continued from previous page)

```
    // Integrate to each value in eta
    for (Eigen::Index i = 0; i < x.rows()-1; i++) {
        integral[i+1] = eigIntgtor.quadratureAdaptive(functor, ScalarType(x[i]),␣
↪ScalarType(x[i+1]), desAbsErr, desRelErr, quadratureRule) + integral[i];
    }
    return integral;
}


}  /* namespace utilities */

#endif //ES_FLOW_INTEGRATION_H
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `NumericalIntegration.h`

Included By

- *File stress.h*

Namespaces

- *Namespace utilities*

Functions

- *Template Function utilities::cumulative_integrate*

File interp.h

Parent directory (`source/utilities`)

Contents

Definition (source/utilities/interp.h)

Program Listing for File interp.h

Return to documentation for file (source/utilities/interp.h)

```
/*
 * interp.h Interpolation tools
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_INTERP_H
#define ES_FLOW_INTERP_H

#include <iostream>
#include <math.h>
#include <Eigen/Core>
#include <Eigen/Dense>
#include <unsupported/Eigen/Splines>


namespace utilities {


class LinearInterpolant {
public:
    LinearInterpolant(const Eigen::ArrayXd x_vec, const Eigen::ArrayXd y_vec)
      : x_min(x_vec.minCoeff()), x_max(x_vec.maxCoeff()), x(x_vec), y(y_vec) {}

    double operator()(double xi) const {
        double bounded_xi = std::max(std::min(xi, x_max), x_min);
        Eigen::Index node = findNode(bounded_xi);
        if (node == x.size()-1) {
            return y[node];
        }
        double proportion = (bounded_xi - x[node]) / (x[node + 1] - x[node]);
        return proportion * (y[node + 1] - y[node]) + y[node];
    }

    Eigen::ArrayXd operator()(Eigen::ArrayXd const &xi_vec) {
        Eigen::ArrayXd yi_vec(xi_vec.rows());
        yi_vec = xi_vec.unaryExpr([this](double xi) { return this->operator()(xi); }).
↪transpose();
        return yi_vec;
    }

private:

    // Find the index of the data value in x_vec immediately before the required xi␣
↪value
    Eigen::Index findNode(const double bounded_xi) const {

        // Run a binary search to find the adjacent node in the grid. This is␣
↪O(log(N)) to evaluate one location.
```

(continues on next page)

(continued from previous page)

```
        // https://stackoverflow.com/questions/6553970/find-the-first-element-in-a-
↪sorted-array-that-is-greater-than-the-target
        Eigen::Index low = 0;
        Eigen::Index high = x.size();
        while (low != high) {
            Eigen::Index mid = (low + high) / 2; // Or a fancy way to avoid int␣
↪overflow
            if (x[mid] <= bounded_xi) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        return low - 1;
    }

    // Boundaries of the vector
    double x_min;
    double x_max;

    // Hold a copy of the data in case it changes outside during the life of the␣
↪interpolant
    Eigen::ArrayXd x;
    Eigen::ArrayXd y;
};


class CubicSplineInterpolant {
public:
    CubicSplineInterpolant(Eigen::VectorXd const &x_vec, Eigen::VectorXd const &y_vec)
        : x_min(x_vec.minCoeff()),
          x_max(x_vec.maxCoeff()),
          spline_(Eigen::SplineFitting<Eigen::Spline<double, 1>>::Interpolate(
              y_vec.transpose(),
              // No more than cubic spline, but accept short vectors.
              std::min<long>(x_vec.rows() - 1, 3),
              scaled_values(x_vec))) {}

    double operator()(double xi) const {
        // x values need to be scaled down in extraction as well.
        return spline_(scaled_value(xi))(0);
    }

    Eigen::VectorXd operator()(Eigen::VectorXd const &xi_vec) {
        Eigen::VectorXd yi_vec(xi_vec.rows());
        yi_vec = xi_vec.unaryExpr([this](double xi) { return spline_(scaled_
↪value(xi))(0); }).transpose();
        return yi_vec;
    }

private:
    double scaled_value(double x) const {
        return (x - x_min) / (x_max - x_min);
    }

    Eigen::RowVectorXd scaled_values(Eigen::VectorXd const &x_vec) const {
        return x_vec.unaryExpr([this](double x) { return scaled_value(x); }).
↪transpose();
```

(continues on next page)

(continued from previous page)

```
    }

    double x_min;
    double x_max;

    // Spline of one-dimensional "points."
    Eigen::Spline<double, 1> spline_;
};


}  /* namespace utilities */


#endif //ES_FLOW_INTERP_H
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `iostream`
- `math.h`
- `unsupported/Eigen/Splines`

Included By

- *File adem.h*
- *File signature.h*

Namespaces

- *Namespace utilities*

Classes

- *Class CubicSplineInterpolant*
- *Class LinearInterpolant*

File main.cpp

Parent directory (`source`)

Contents

- *Definition* (`source/main.cpp`)

- *Includes*
- *Functions*

Definition (`source/main.cpp`)

Program Listing for File main.cpp

Return to documentation for file (`source/main.cpp`)

```
/*
 * main.cpp Command line executable for es-flow
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2016-9 Octue Ltd. All Rights Reserved.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
#include "glog/logging.h"
#include "cxxopts.hpp"


int main(int argc, char* argv[]) {

    try {

        // Handle the input parsing and create the program help page

        // Set the program name (for --help option display) and default option␣
↪behaviour
        cxxopts::Options options("es-flow", "EnvironmentSTUDIO flow library wrapper");
        bool logging = false;

        // Define the command line options, arranged visually (in the --help output)␣
↪in two groups:
        options.add_options()
                ("l,log-file",
                 "Switch on logging, optionally specify the directory to save␣
↪logfiles.",
                 cxxopts::value<std::string>()->implicit_value("logs"), "FILE")
                ("h,help",
                 "Display program help.",
                 cxxopts::value<bool>(), "BOOL");

        options.add_options("Input / output file")
                ("i,input-file", "Name of the input file (read only).", cxxopts::value
↪<std::string>(), "FILE")
                ("o,output-file",
                 "Name of the output results file. Warning - this file will be␣
↪overwritten if it already exists.",
```

(continues on next page)

(continued from previous page)

```
                cxxopts::value<std::string>(), "FILE");

        // Parse the input options
        options.parse(argc, argv);

        if (options.count("help")) {
            std::cout << options.help({"", "Input / output file"}) << std::endl;
            exit(0);
        }

        if (options.count("l")) {
            logging = true;
        }

        if (logging) {
            FLAGS_logtostderr = false;
            FLAGS_minloglevel = 0;
            FLAGS_log_dir = options["l"].as<std::string>();
            std::cout << "Logging to: " << FLAGS_log_dir << std::endl;
            google::InitGoogleLogging(argv[0]);
        }


    } catch (const cxxopts::OptionException &e) {
        std::cout << "Error parsing options: " << e.what() << std::endl;
        exit(1);

    } catch (...) {
        // Handle logging of general exceptions
        auto eptr = std::current_exception();
        try {
            // This deliberately rethrows the caught exception in the current scope,␣
↪so we can log it
            if (eptr) {
                std::rethrow_exception(eptr);
            }
        } catch(const std::exception& e) {
            std::cout << "Caught exception: " << e.what() << std::endl;
        }
        exit(1);

    }

    exit(0);

}
```

Includes

- cxxopts.hpp
- glog/logging.h
- stdbool.h
- stdio.h

- stdlib.h
- unistd.h

Functions

- *Function main*

File profile.cpp

Parent directory (source)

Contents

Definition (source/profile.cpp)

Program Listing for File profile.cpp

Return to documentation for file (source/profile.cpp)

```
/*
 * profile.cpp Profile handling for (e.g.) Velocity, Reynolds Stress and Spectral␣
↪Tensor Profile management
 *
 * Function definitions are included in the .h file, because of the templating␣
↪constraint. See http://stackoverflow.com/questions/495021/why-can-templates-only-be-
↪implemented-in-the-header-file
 *
 * Author:                    Tom Clark (thclark @ github)
 *
 * Copyright (c) 2016-9 Octue Ltd. All Rights Reserved.
 *
 */
#include "profile.h"
#include "math.h"
#include "definitions.h"


namespace es {

Bins::Bins(std::vector<double> z): z_ctrs(z) {
    // Construct from a spacing with default bin size from central difference of the␣
↪spacing
    int i;
    n_bins = z_ctrs.size();
```

(continues on next page)

(continued from previous page)

```
    dx = std::vector<double>(n_bins);
    dy = std::vector<double>(n_bins);
    dz = std::vector<double>(n_bins);
    dz[0] = fabs(z[1]-z[0]);
    for (i = 1; i < n_bins-1; i++) {
        dz[i] = 0.5*fabs(z[i+1]-z[i-1]);
    }
    dz[n_bins-1] = fabs(z[n_bins-1]-z[n_bins-2]);
    for (i = 0; i < n_bins-1; i++) {
        dx[i] = dz[i];
        dy[i] = dz[i];
    }
}

Bins::Bins(std::vector<double> z, std::vector<double> d_x, std::vector<double> d_y,␣
↪std::vector<double> d_z): Bins(z) {
    // Construct from a spacing and bin size in x, y, z
    dx = d_x;
    dy = d_y;
    dz = d_z;
    if ((dx.size() != n_bins) ||  (dy.size() != n_bins) || (dz.size() != n_bins)){
        throw std::range_error("Length of dx, dy or dz does not match the number of␣
↪bins");
    }
}

Bins::Bins(std::vector<double> z, double half_angle_degrees): Bins(z) {
    // Construct from spacing, with bin height from central differencing of the␣
↪spacing and bin widths using a half angle
    int i;
    double tha;
    tha = tand(half_angle_degrees);
    dx = std::vector<double>(z_ctrs.size());
    dy = std::vector<double>(z_ctrs.size());
    for (i = 0; i < n_bins - 1; i++) {
        dx[i] = 2.0 * z[i] * tha;
        dy[i] = dx[i];
    }
}

Bins::Bins(std::vector<double> z, double half_angle_degrees, std::vector<double> d_
↪z): Bins(z, half_angle_degrees) {
    // Construct from spacing, with bin height given and bin widths using a half angle
    int i;
    dz = d_z;
    if (dz.size() != n_bins){
        throw std::range_error("Length of dx, dy or dz does not match the number of␣
↪bins");
    }
}

Bins::~Bins() {
    //Destructor
}

::std::ostream& operator<<(::std::ostream& os, const Bins& bins) {
    // Represent in logs or ostream
```

(continues on next page)

(continued from previous page)

```
    return os << "debug statement for bins class";
}


/* Use profile base class constructor and destructor
VelocityProfile::VelocityProfile() {
// TODO Auto-generated constructor stub
}

VelocityProfile::~VelocityProfile() {
// TODO Auto-generated destructor stub
}

VectorXd VelocityProfile::AutoDiff() {
// Differentiate the velocity profile wrt z

    typedef Eigen::AutoDiffScalar<Eigen::VectorXd> AScalar;
    // AScalar stores a scalar and a derivative vector.

    // Instantiate an AutoDiffScalar variable with a normal Scalar
    double s = 0.3;
    AScalar As(s);

    // Get the value from the Instance
    std::cout << "value: " << As.value() << std::endl;

    // The derivative vector is As.derivatives();

    // Resize the derivative vector to the number of dependent variables
    As.derivatives().resize(2);

    // Set the initial derivative vector
    As.derivatives() = Eigen::VectorXd::Unit(2,0);
    std::cout << "Derivative vector : " << As.derivatives().transpose() << std::endl;

    // Instantiate another AScalar
    AScalar Ab(4);
    Ab.derivatives() = Eigen::VectorXd::Unit(2,1);

    // Do the most simple calculation
    AScalar Ac = As * Ab;

    std::cout << "Result/Ac.value()" << Ac.value() << std::endl;
    std::cout << "Gradient: " << Ac.derivatives().transpose() << std::endl;

    Eigen::VectorXd a(1);
    return a;
}
*/

} /* namespace es */
```

Includes

- `definitions.h` (*File definitions.h*)

- `math.h`
- `profile.h` (*File profile.h*)

Namespaces

- *Namespace es*

Functions

- *Function es::operator<<(::std::ostream&, const Bins&)*

File profile.h

Parent directory (`source`)

Contents

Definition (`source/profile.h`)

Program Listing for File profile.h

Return to documentation for file (`source/profile.h`)

```
/*
 * profile.h Profile handling for (e.g.) Velocity, Reynolds Stress and Spectral␣
↪Tensor Profile management
 *
 * Function definitions are included here, rather than the cpp file, because of the␣
↪templating constraint. See http://stackoverflow.com/questions/495021/why-can-
↪templates-only-be-implemented-in-the-header-file
 *
 * Author:                    Tom Clark (thclark @ github)
 *
 * Copyright (c) 2016-9 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_PROFILE_H
#define ES_FLOW_PROFILE_H
```

(continues on next page)

(continued from previous page)

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>


namespace es {


class Bins {
public:

    // Construct from a spacing with bin sizes set by central differencing of spacing
    Bins(std::vector<double> z);

    // Construct from spacing with bin sizes specified
    Bins(std::vector<double> z, std::vector<double> dx, std::vector<double> dy,␣
↪std::vector<double> dz);

    // Construct from spacing and a half angle in degrees, dz set by central␣
↪differencing of spacing
    Bins(std::vector<double> z, double half_angle_degrees);

    // Construct from spacing and a half angle in degrees, with dz specified
    Bins(std::vector<double> z, double half_angle_degrees, std::vector<double> dz);

    // Destroy
    ~Bins();

    // Vectors of z locations and bin dimensions
    unsigned long n_bins;
    std::vector<double> z_ctrs;
    std::vector<double> dx;
    std::vector<double> dy;
    std::vector<double> dz;

};

// Represent Bins class in logs or ostream
::std::ostream& operator<<(::std::ostream& os, const Bins& bins);

template <class ProfileType>
class Profile {
private:
    std::vector<ProfileType> values;

public:

    // Construct using bins only (zero values and position)
    Profile(const Bins &bins);

    // Construct using bins and a global origin location
    Profile(const Bins &bins, double x, double y, double z);

    // Destroy
```

(continues on next page)

(continued from previous page)

```
    virtual ~Profile();

    // Get the values, whatever type they might be
    const std::vector<ProfileType> &getValues() const;

    void setValues(const std::vector<ProfileType> &values);

    // Properties
    double position[3];
    Bins bins;

};

// Represent Profiles class in logs or ostream
template <class ProfileType>
::std::ostream& operator<<(::std::ostream& os, const Profile<ProfileType>& profile);

template <class ProfileType>
Profile<ProfileType>::Profile(const Bins &bins) : bins(bins) {
    // Construct from just bins, with default position {0.0, 0.0, 0.0}
    position[0] = 0.0;
    position[1] = 0.0;
    position[2] = 0.0;
}

template <class ProfileType>
Profile<ProfileType>::~Profile() {
    // Destructor
}

template <class ProfileType>
Profile<ProfileType>::Profile(const Bins &bins, double x, double y, double z):␣
↪bins(bins) {
    // Construct from bins and a position
    position[0] = x;
    position[1] = y;
    position[2] = z;
}

template <class ProfileType>
void Profile<ProfileType>::setValues(const std::vector<ProfileType> &values) {
    if (values.size() != bins.n_bins) {
        throw std::out_of_range("size of vector 'values' does not equal the number of␣
↪bins for this profile");
    }
    Profile::values = values;
}

template <class ProfileType>
const std::vector<ProfileType> &Profile<ProfileType>::getValues() const {
    return values;
}

//template <class ProfileType>
//const std::vector<ProfileType> &Profile<ProfileType>::getZDerivative() const {
//    // Get the first derivative with respect to Z location using Eigen's Autodiff
//    // Some examples can be found in eigen/unsupported/doc/examples/AutoDiff.cpp
```

(continues on next page)

(continued from previous page)

```
//
//
//    return derivatives;
//}

template <class ProfileType>
::std::ostream& operator<<(::std::ostream& os, const Profile<ProfileType>& profile) {
    // Represent in logs or ostream
    return os << "debug statement for profile class";
}


class VelocityProfile: public Profile<double> {
public:
    VelocityProfile();
    virtual ~VelocityProfile();

//    VectorXd AutoDiff();
};


} /* namespace es */


#endif // ES_FLOW_PROFILE_H
```

Includes

- cstdlib
- iostream
- stdexcept
- string
- vector

Included By

- *File adem.h*
- *File signature.h*
- *File velocity.h*
- *File profile.cpp*
- *File stress.h*
- *File veer.h*

Namespaces

- *Namespace es*

Classes

- *Class Bins*
- *Template Class Profile*
- *Class VelocityProfile*

Functions

- *Template Function es::operator<<(::std::ostream&, const Profile<ProfileType>&)*

File readers.h

Parent directory (`source/io`)

Contents

Definition (`source/io/readers.h`)

Program Listing for File readers.h

Return to documentation for file (`source/io/readers.h`)

```
/*
 * readers.h File readers for timeseries instrument data
 *
 * Author:                    Tom Clark (thclark @ github)
 *
 * Copyright (c) 2016-9 Octue Ltd. All Rights Reserved.
 *
 */
#ifndef ES_FLOW_READERS_H
#define ES_FLOW_READERS_H

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
#include <iostream>
#include <vector>
```

(continues on next page)

(continued from previous page)

```
#include <cstdlib>
#include <string>
#include <stdexcept>
#include "matio.h"
#include "glog/logging.h"
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
#include <eigen3/Eigen/Core>
#include "data_types.h"


namespace es {


// Level of data validation applied to timeseries. Exception is thrown if checks fail
enum timeseries_check_level {
    PRESENT             = 0,    // Checks that data is present and of correct type.␣
↪Extracts start and end timestamps.
    INCREASING          = 1,    // As PRESENT + checks that the timestamp always␣
↪increases.
    MONOTONIC           = 2,    // As INCREASING + checks that timestamp uniformly␣
↪increases, allows data skip (e.g. instrument turned off then back on later).␣
↪Extracts sampling frequency.
    STRICTLY_MONOTONIC  = 3,    // AS INCREASING + checks that timestamp uniformly␣
↪increases, data skipping causes errors. Extracts sampling frequency.
};

// Level of data validation applied to file type. Exception is thrown if checks fail
enum file_type_check_level {
    NONE                = 0,    // No checks
    OAS_STANDARD        = 1,    // Checks that the file contains the 'type' string␣
↪variable
};

template <class DataType>
class Reader {
public:

    Reader(const std::string &file);

    ~Reader();

    std::string logString() const;

    DataType *read(bool print=false);

    void checkFileType(int level);

    void checkTimeseries(int level);

    double getWindowDuration() const;

    void setWindowDuration(double windowDuration);
```

(continues on next page)

(continued from previous page)

```
    int getWindowSize() const;

    void setWindowSize(int windowSize);

    void readWindow(const int index);

protected:

    std::string file;
    mat_t *matfp;
    std::string file_type = std::string("none");
    int windowSize;
    double windowDuration;
    DataType data;

};

template <class DataType>
Reader<DataType>::Reader(const std::string &file) : file(file) {

    // Open the MAT file for reading and save the pointer
    matfp = Mat_Open(file.c_str(), MAT_ACC_RDONLY);
    if (matfp == NULL) {
        std::string msg = "Error reading MAT file: ";
        throw std::invalid_argument(msg + file);
    }

}

template <class DataType>
Reader<DataType>::~Reader() {

    // Close the file on destruction of the Reader
    Mat_Close(matfp);

}

template <class DataType>
void Reader<DataType>::checkFileType(int level){

    // Get the file type from the reserved 'type' variable in the mat file
    matvar_t *type_var = Mat_VarRead(matfp, "type");
    if (type_var == NULL) {
        if (level == OAS_STANDARD) {
            throw std::invalid_argument("Error reading mat file (most likely not an␣
↪OAS standard file format - 'type' variable is missing)");
        }
    } else {
        if (type_var->class_type != MAT_C_CHAR) {
            throw std::invalid_argument("Error reading mat file ('type' variable must␣
↪be a character array)");
        }
        file_type = std::string((const char*)type_var->data, type_var->dims[1]);
    }

}
```

(continues on next page)

(continued from previous page)

```
template <class DataType>
void Reader<DataType>::checkTimeseries(int level) {

    // Check the integrity of the timeseries, automatically determine dt and␣
↪frequency where possible

}

template <class DataType>
double Reader<DataType>::getWindowDuration() const {
    return windowDuration;
}

template <class DataType>
void Reader<DataType>::setWindowDuration(double windowDuration) {
    Reader<DataType>::windowDuration = windowDuration;
}

template <class DataType>
int Reader<DataType>::getWindowSize() const {
    return windowSize;
}

template <class DataType>
void Reader<DataType>::setWindowSize(int windowSize) {
    Reader<DataType>::windowSize = windowSize;
}

template <class DataType>
std::string Reader<DataType>::logString() const {
    return std::string("Object Reader for type ") + file_type + std::string(",␣
↪attached to file ") + file;
}

template <class DataType>
DataType *Reader<DataType>::read(bool print) {
    // Simply invoke the read method of the DataType class.
    if (std::strcmp(data.type.c_str(), file_type.c_str())) {
        std::string msg = "Mat file type '" + file_type + "' incompatible with the␣
↪specified data type '" + data.type + "'";
        throw std::invalid_argument(msg);
    }
    data.read(matfp, print);
    return &data;
}

template <class DataType>
void Reader<DataType>::readWindow(const int index) {
    //<DataType>.read(file_type, matfp);
}

// Represent Reader classes in logs or ostream
template <class DataType>
::std::ostream& operator<<(::std::ostream& os, const Reader<DataType>& reader) {
    // Represent in logs or ostream
    return os << reader.logString();
}
```

(continues on next page)

(continued from previous page)

```
} /* namespace es */


#endif // ES_FLOW_READERS_H
```

Includes

- cstdlib
- data_types.h (*File data_types.h*)
- eigen3/Eigen/Core
- glog/logging.h
- iostream
- matio.h
- stdbool.h
- stdexcept
- stdio.h
- stdlib.h
- string
- unistd.h
- vector

Namespaces

- *Namespace es*

Classes

- *Template Class Reader*

Enums

- *Enum file_type_check_level*
- *Enum timeseries_check_level*

Functions

- *Template Function es::operator<<(::std::ostream&, const Reader<DataType>&)*

File signature.h

Parent directory (`source/adem`)

Contents

Definition (`source/adem/signature.h`)

Program Listing for File signature.h

Return to documentation for file (`source/adem/signature.h`)

```
/*
 * signature.h Eddy Signatures for use with the Attached-Detached Eddy Method
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef SOURCE_ADEM_SIGNATURE_H_
#define SOURCE_ADEM_SIGNATURE_H_

#include <boost/algorithm/string/classification.hpp> // Include boost::for is_any_of
#include <boost/algorithm/string/split.hpp> // Include for boost::split
#include <Eigen/Dense>
#include <Eigen/Core>
#include <math.h>
#include <stdexcept>
#include <unsupported/Eigen/CXX11/Tensor>
#include <unsupported/Eigen/FFT>

#include "adem/biot_savart.h"
#include "profile.h"
#include "relations/stress.h"
#include "relations/velocity.h"
#include "utilities/filter.h"
#include "utilities/conv.h"
#include "utilities/interp.h"
#include "utilities/tensors.h"
#include "utilities/trapz.h"
#include "io/variable_readers.h"
```

(continues on next page)

(continued from previous page)

```
#include "io/variable_writers.h"

#include "cpplot.h"


typedef Eigen::Array<double, 5, 3> Array53d;
typedef Eigen::Array<double, 3, 3> Array33d;


using namespace utilities;
namespace es {


class EddySignature {
public:

    std::string eddy_type;

    Eigen::ArrayXd lambda;

    Eigen::ArrayXd eta;

    Eigen::Tensor<double, 3> g;

    Eigen::Array<double, Eigen::Dynamic, 6> j;

    Eigen::Array3d domain_spacing;

    Eigen::Array<double, 3, 2> domain_extents;

    void load(std::string file_name, bool print_var = false) {
        std::cout << "Reading eddy signature data from file " << file_name <<␣
↪std::endl;

        // Open the MAT file for reading
        mat_t *matfp = Mat_Open(file_name.c_str(), MAT_ACC_RDONLY);
        if (matfp == NULL) {
            std::string msg = "Error reading MAT file: ";
            throw std::invalid_argument(msg + file_name);
        }

        // Use the variable readers to assist
        eddy_type = readString(matfp, "eddy_type", print_var);
        lambda = readArrayXd(matfp, "lambda", print_var);
        eta = readArrayXd(matfp, "eta", print_var);
        domain_spacing = readArray3d(matfp, "domain_spacing", print_var);
        domain_extents = readArray32d(matfp, "domain_extents", print_var);
        g = readTensor3d(matfp, "g", print_var);
        j = readArrayXXd(matfp, "j", print_var);

        // Close the file
        Mat_Close(matfp);
        std::cout << "Finished reading eddy signature (Type " + eddy_type + ")" <<␣
↪std::endl;
    }

    void save(std::string file_name) {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "Writing signature data to file " << file_name << std::endl;

        mat_t *matfp = Mat_CreateVer(file_name.c_str(), NULL, MAT_FT_MAT73);
        if ( NULL == matfp ) {
            throw std::runtime_error("Unable to create/overwrite MAT file '" + file_
↪name + "'");
        }

        // Use the variable writers to assist
        writeString(matfp, "eddy_type", eddy_type);
        writeArrayXd(matfp, "lambda", lambda);
        writeArrayXd(matfp, "eta", eta);
        writeArray3d(matfp, "domain_spacing", domain_spacing);
        writeArray32d(matfp, "domain_extents", domain_extents);
        writeTensor3d(matfp, "g", g);
        writeArrayXXd(matfp, "j", j);

        // Close the file
        Mat_Close(matfp);
        std::cout << "Finished writing eddy signature (Type " + eddy_type + ")" <<␣
↪std::endl;

    }

    EddySignature operator+(const EddySignature& c) const
    {
        EddySignature result;
        result.eddy_type = this->eddy_type + "+" + c.eddy_type;
        result.eta = this->eta;
        result.lambda = this->lambda;
        result.domain_spacing = this->domain_spacing;
        result.domain_extents = this->domain_extents;
        result.g = (this->g + c.g);
        result.j = (this->j + c.j);
        return result;
    }

    EddySignature operator*(double a) const
    {
        EddySignature result;
        result.eddy_type = "(" + this->eddy_type + ")*" + std::to_string(a);
        result.eta = this->eta;
        result.lambda = this->lambda;
        result.domain_spacing = this->domain_spacing;
        result.domain_extents = this->domain_extents;
        result.g = this->g;
        result.j = this->j;
        result.g = result.g * a;
        result.j = result.j * a;
        return result;
    }

    EddySignature operator/(double denom) const
    {
        EddySignature result;
        result.eddy_type = "(" + this->eddy_type + ")/" + std::to_string(denom);
        result.eta = this->eta;
```

(continues on next page)

(continued from previous page)

```
        result.lambda = this->lambda;
        result.domain_spacing = this->domain_spacing;
        result.domain_extents = this->domain_extents;
        result.g = this->g;
        result.j = this->j;
        result.g = result.g / denom;
        result.j = result.j / denom;
        return result;
    }

    Eigen::ArrayXXd k1z(Eigen::ArrayXd &eta) const {

        double dx = domain_spacing[0];
        auto nx = Eigen::Index((domain_extents(0,1) - domain_extents(0,0)) / dx) + 1;
        Eigen::ArrayXd k1_delta = Eigen::ArrayXd::LinSpaced(nx, 0, nx-1) * 2.0 * M_PI␣
↪/ dx;
        Eigen::ArrayXXd k1z = k1_delta.replicate(1, eta.rows()) * eta.transpose().
↪replicate(k1_delta.rows(), 1);

        std::cout << "k1_delta_start " << k1_delta(0) << std::endl;
        std::cout << "k1_delta_end " << k1_delta(nx-1) << std::endl;
        std::cout << "eta_start " << eta(0) << std::endl;
        std::cout << "eta_end " << eta(eta.rows()-1) << std::endl;
        std::cout << "k1z_n_rows " << k1z.rows() << std::endl;
        std::cout << "k1z_n_cols " << k1z.cols() << std::endl;

        return k1z;
    }

    Eigen::ArrayXXd getJ(const Eigen::ArrayXd & locations, const bool linear=false)␣
↪const {
        Eigen::ArrayXXd j_fine(locations.rows(), 6);
        if (linear) {
            // The new locations are values of eta

            // Assert locations are in the valid range...
            if ((locations < 0.0).any()) {
                throw std::invalid_argument("Input locations (eta values) must be␣
↪defined for eta >= 0.0");
            }

            // Interpolate on an eta basis
            for (int k = 0; k < 6; k++) {
                utilities::CubicSplineInterpolant s(this->eta.matrix(), this->j.
↪col(k).matrix());
                j_fine.col(k) = s(locations);
            }

        } else {
            // The new locations are values of lambda

            // Interpolate on a lambda basis
            for (int k = 0; k < 6; k++) {
                utilities::CubicSplineInterpolant s(this->lambda.matrix(), this->j.
↪col(k).matrix());
                j_fine.col(k) = s(locations);
            }
```

(continues on next page)

(continued from previous page)

```
        }
        return j_fine;
    };

    void computeSignature(const std::string &type, const int n_lambda=200, const
↪double dx=0.002) {

        // Set the type string
        eddy_type = type;

        // Express eddy structure as line vortices.  See Figure 13 Perry and Marusic
↪1995.
        Array33d eddy;
        if (type == "A") {
            eddy << 0, -0.8, 0,
                    1,  0,   1,
                    0,  0.8, 0;

        } else if (type == "B1") {
            eddy << 0.2, -0.15, 1,
                    0,    0,    0.5,
                    0.09, 0.15, 1;

        } else if (type == "B2") {
            eddy << 0,     0.15, 0.5,
                    0.2,   0,    1,
                    0.11, -0.15, 0.5;

        } else if (type == "B3") {
            eddy << 0.09, -0.15, 1,
                    0,     0,    0.5,
                    0.2,   0.15, 1;

        } else if (type == "B4") {
            eddy << 0.11, 0.15, 0.5,
                    0.20, 0,    1,
                    0,   -0.15, 0.5;
        } else {
            // TODO sort out cpplot import ordering so that I can import exceptions
↪from our own damn library!
//            es::InvalidEddyTypeException e;
//            throw(e);
        }

        // Determine start and end nodes of the eddy...
        Eigen::Array3Xd startNodes(3,4);
        Eigen::Array3Xd endNodes(3,4);
        startNodes.leftCols(2) = eddy.topRows(2).transpose();
        endNodes.leftCols(2) = eddy.bottomRows(2).transpose();

        // ... and its reflection in the wall
        eddy.col(2) *= -1;
        startNodes.rightCols(2) = eddy.bottomRows(2).transpose();
        endNodes.rightCols(2) = eddy.topRows(2).transpose();

        //    % PLOT EDDY STRUCTURE SHAPE
        //    raiseFigure(['Type ' type ' Eddy Structure'])
```

(continues on next page)

(continued from previous page)

```
        //    clf
        //    subplot(1,3,1)
        //    plot3([startNodes(:,1)'; endNodes(:,1)'],[startNodes(:,2)'; endNodes(:,
↪2)'],[startNodes(:,3)'; endNodes(:,3)'],'k-')
        //    axis equal
        //    xlabel('x/\delta')
        //    ylabel('y/\delta')
        //    zlabel('z/\delta')

        // Set up influence domain

        /* Let's just think a second about frequency. For an ABL, the boundary layer␣
↪is (say) delta = 1km thick,
         * and the wind speed is (say) 20m/s, and we want to resolve spectra to a␣
↪frequency of (say) 10Hz,
         * then what eddy scale do we need to go down to?
         *      20/10 = 2 m ...
         * nondimensionalising, that's an eddy scale z/delta of 0.002. So our largest␣
↪value of lambda_e (the eddy scale,
         * smaller as lambda_e increases) should correspond to this z/delta (or even␣
↪finer scale).
         *
         * We thus choose z/delta = 0.002 as our finest eddy scale - that's why input␣
↪param dx is set at 0.002.
         *
         */
        //double lambda_max = log(1/dx);
        std::cout << "UNBODGE THIS" << std::endl;
        double lambda_max = log(500);
        double lambda_min = log(1/1.5);

        // The domain extents are set accordingly
        domain_extents = Eigen::Array<double, 3, 2>::Zero();
        domain_extents << -4, 4,
                          -2, 2,
                           (1.0 / exp(lambda_max)), (1.0 / exp(lambda_min));

        // Logarithmically space the lambda coordinates
        lambda = Eigen::ArrayXd::LinSpaced(n_lambda, lambda_min, lambda_max);
        double d_lambda = lambda(1)-lambda(0);

        // Store real space vertical coordinates too
        eta = lambda.exp().inverse();

        /* So now we need to set the spacing in the physical domain
         * Wavenumber k1 2*pi/L = 2*pi*f/U
         *
         * So for the above estimated parameters of f = 10Hz, U = 20m/s, L = 2.␣
↪Normalised to x/delta, this is 0.002.
         * So we need the grid spacing to be of a similar order in order to capture␣
↪the appropriate wavenumbers...
         */
        domain_spacing << dx, dx, d_lambda;

        // A constant expression used for scaling the Reynolds Stress contributions
        constexpr double u0_sqd = 1.0 / (4.0 * M_PI * M_PI);
```

(continues on next page)

(continued from previous page)

```
        // Streamwise and transverse grids (eta is z)
        Eigen::Index nx = ceil((domain_extents(0,1) - domain_extents(0,0)) / domain_
↪spacing(0));
        Eigen::Index ny = ceil((domain_extents(1,1) - domain_extents(1,0)) / domain_
↪spacing(1));
        Eigen::ArrayXXd x_vec = Eigen::ArrayXd::LinSpaced(nx, domain_extents(0,0),␣
↪domain_extents(0,1));
        Eigen::ArrayXXd y_vec = Eigen::ArrayXd::LinSpaced(ny, domain_extents(1,0),␣
↪domain_extents(1,1));

        // Produce x, y matrices, with y being the quickest changing (down columns)
        // Eigen syntax to replicate and interleave like this is v. awkward, so we␣
↪have to write a loop. Sigh.
        Eigen::Index n_locations = x_vec.rows() * y_vec.rows();
        Eigen::Array3Xd locations(3, n_locations);
        Eigen::Index ctr = 0;
        for (auto i=0; i<x_vec.rows(); i++) {
            for (auto j=0; j<y_vec.rows(); j++) {
                locations(0, ctr) = x_vec(i);
                locations(1, ctr) = y_vec(j);
                ctr++;
            };
        };

        // Set unit circulation and the core radius of 0.05 as recommended by Perry␣
↪and Marusic.
        Eigen::VectorXd gamma = Eigen::VectorXd::Ones(4);
        Eigen::VectorXd effective_core_radius_squared = Eigen::VectorXd::Ones(4) *␣
↪pow(0.05, 2.0);

        j = Eigen::ArrayXXd(n_lambda,6);
        j.setZero();

        // For each vertical coordinate, determine the contribution to the eddy␣
↪signature and spectra
        for (Eigen::Index lam_ctr=0; lam_ctr<n_lambda; lam_ctr++) {
            locations.row(2) = Eigen::ArrayXXd::Constant(1, n_locations, eta(lam_
↪ctr));
            std::cout << "Computing signature at eta=" << eta(lam_ctr) << " (" << lam_
↪ctr+1 << " of " << n_lambda << ")" << std::endl;
            Eigen::Array3Xd induction = NaiveBiotSavart(
                startNodes.matrix(),
                endNodes.matrix(),
                locations.matrix(),
                gamma,
                effective_core_radius_squared
            );

            // We don't want to copy or reallocate the memory... map into the␣
↪induction
            Eigen::Map<Eigen::ArrayXXd, 0, Eigen::InnerStride<3>> u (induction.data(),
↪ y_vec.rows(), x_vec.rows());
            Eigen::Map<Eigen::ArrayXXd, 0, Eigen::InnerStride<3>> v (induction.
↪data()+1, y_vec.rows(), x_vec.rows());
            Eigen::Map<Eigen::ArrayXXd, 0, Eigen::InnerStride<3>> w (induction.
↪data()+2, y_vec.rows(), x_vec.rows());
```

(continues on next page)

(continued from previous page)

```
            // Compute and plot Reynolds Stresses produced by a single eddy
            Eigen::ArrayXXd uu = u * u / u0_sqd;
            Eigen::ArrayXXd uv = u * v / u0_sqd;
            Eigen::ArrayXXd uw = u * w / u0_sqd;
            Eigen::ArrayXXd vv = v * v / u0_sqd;
            Eigen::ArrayXXd vw = v * w / u0_sqd;
            Eigen::ArrayXXd ww = w * w / u0_sqd;

            // We have constant spacing, so use quick version of trapz to integrate␣
↪first across X, then down Y
            j.block<1,1>(lam_ctr, 0) = trapz(trapz(uu,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
            j.block<1,1>(lam_ctr, 1) = trapz(trapz(uv,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
            j.block<1,1>(lam_ctr, 2) = trapz(trapz(uw,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
            j.block<1,1>(lam_ctr, 3) = trapz(trapz(vv,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
            j.block<1,1>(lam_ctr, 4) = trapz(trapz(vw,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
            j.block<1,1>(lam_ctr, 5) = trapz(trapz(ww,2) * domain_spacing(0),1) *␣
↪domain_spacing(1);
        };


    }

    void applySignature(const std::string &type, const int n_lambda=200) {

        // Set the type string
        eddy_type = type;

        // Set arbitrary domain extents
        double lambda_max = log(500);
        double lambda_min = log(1/1.5);
        domain_extents = Eigen::Array<double, 3, 2>::Zero();
        domain_extents << -4, 4,
            -2, 2,
            (1.0 / exp(lambda_max)), (1.0 / exp(lambda_min));

        // Logarithmically space lambda coordinates
        lambda = Eigen::ArrayXd::LinSpaced(n_lambda, lambda_min, lambda_max);
        domain_spacing << 1.0, 1.0, lambda(1)-lambda(0);

        // Store real space vertical coordinates too
        eta = lambda.exp().inverse();

        // Interpolate the signatures that are given.
        j = Eigen::ArrayXXd(n_lambda,6);
        j.setZero();
        Eigen::ArrayXXd i11;
        Eigen::ArrayXXd i13;
        Eigen::ArrayXXd i22;
        Eigen::ArrayXXd i33;
        if (type == "A") {
            i11 = Eigen::ArrayXXd(2, 26);
            i11 << 0, 0.012710542, 0.030316638, 0.05182384, 0.08898147, 0.1300581, 0.
↪18483716, 0.2357049, 0.29636374, 0.35898846, 0.41573855, 0.4783607, 0.5385956, 0.
↪6368718, 0.7073141, 0.76600707, 0.82468724, 0.87943816, 0.91658044, 0.9419913, 0.
↪97133654, 1.0007021, 1.0516082, 1.1260028, 1.2239062, 1.50,
```

(continues on next page)

(continued from previous page)

```
                1.6593906, 1.6418779, 1.611259, 1.5544281, 1.4713508, 1.3926289, 1.
↪3051336, 1.2263831, 1.1476042, 1.08192, 1.0162529, 0.94620186, 0.8586324, 0.7667018,
↪ 0.6747941, 0.5829205, 0.469213, 0.33368298, 0.22440498, 0.1457287, 0.09760853, 0.
↪084422655, 0.07117407, 0.040389933, 0.027004533, 0.0;

            i13 = Eigen::ArrayXXd(2, 21);
            i13 << 0, 0.023503713, 0.05093406, 0.08618198, 0.12728672, 0.17620902, 0.
↪22121488, 0.28186864, 0.3620689, 0.45790172, 0.55568004, 0.64367235, 0.7238267, 0.
↪80006206, 0.86259496, 0.90362066, 0.93487054, 0.9700394, 0.9993566, 1.0423712, 1.50,
                0, -0.061066702, -0.14832275, -0.22682244, -0.2878379, -0.340097, -0.
↪38363394, -0.43149212, -0.4618262, -0.4702808, -0.46126255, -0.43917242, -0.
↪39090434, -0.32954726, -0.24639612, -0.17204118, -0.102081485, -0.045210768, -0.
↪014557855, -0.0030345472, -0.004372481;

            i22 = Eigen::ArrayXXd(2, 23);
            i22 << 0, 0.018947192, 0.03648182, 0.054011352, 0.0754471, 0.10665094, 0.
↪17303112, 0.21991083, 0.28439146, 0.34693173, 0.41142765, 0.48179564, 0.57952243, 0.
↪65573704, 0.7202125, 0.788579, 0.8452064, 0.9115866, 0.9701798, 1.0483195, 1.
↪1304111, 1.2340457, 1.50,
                1.176464, 1.150218, 1.0934712, 1.0323671, 0.966883, 0.8926276, 0.
↪79638124, 0.74817854, 0.6998736, 0.6646518, 0.6294187, 0.5985088, 0.5500107, 0.
↪5016375, 0.4489753, 0.37886125, 0.3044581, 0.20821178, 0.14251183, 0.067983694, 0.
↪028291034, 0.014617034, 0.0043686354;

            i33 = Eigen::ArrayXXd(2, 32);
            i33 << 0.0, 0.023483196, 0.041119818, 0.06269325, 0.09603633, 0.12936921,␣
↪0.17444114, 0.21754721, 0.25868744, 0.30175778, 0.35461667, 0.4133425, 0.46618098,␣
↪0.5346596, 0.608995, 0.6696255, 0.73806334, 0.7908406, 0.83576465, 0.8884807, 0.
↪92553115, 0.95087314, 0.9898843, 1.0288852, 1.0737991, 1.1304673, 1.1793332, 1.
↪2321156, 1.2849082, 1.3552966, 1.4374342, 1.50,
                0.0, 0.012935479, 0.043334138, 0.095496446, 0.1780915, 0.251972, 0.
↪3301416, 0.39960802, 0.46037126, 0.49933675, 0.54696, 0.5945491, 0.6247433, 0.
↪6504893, 0.6674866, 0.6714917, 0.66237944, 0.6402863, 0.59209496, 0.51771456, 0.
↪42599595, 0.35613185, 0.26875913, 0.17267188, 0.115766004, 0.07622104, 0.05415063,␣
↪0.036414765, 0.027393447, 0.013912599, 0.013435401, 0.0043572737;

        } else if (type == "B") {
            i11 = Eigen::ArrayXXd(2, 23);
            i11 << 0, 0.07068844, 0.1666695, 0.27835685, 0.36659724, 0.4334307, 0.
↪48672056, 0.5342177, 0.5837344, 0.62532634, 0.67665803, 0.74737716, 0.8137505, 0.
↪8545449, 0.8894136, 0.93391985, 0.9706649, 1.00945, 1.0521617, 1.1184429, 1.2122818,
↪ 1.3199301, 1.50,
                0.016322415, 0.016322415, 0.018703382, 0.024518738, 0.03472676, 0.
↪056264196, 0.091715895, 0.13412246, 0.18173045, 0.22154763, 0.25700384, 0.27592924,␣
↪0.25842202, 0.23056176, 0.19837682, 0.15315433, 0.11402357, 0.08182956, 0.050494146,
↪ 0.025177978, 0.011945607, 0.0073581133, 0.0017353186;

            i13 = Eigen::ArrayXXd(2, 14);
            i13 << 0, 0.2741542, 0.34076267, 0.42109883, 0.47993597, 0.56440324, 0.
↪6233631, 0.7057494, 0.7918987, 0.8779049, 0.94425774, 1.0145372, 1.1240618, 1.50,
                0, -2.3333919E-4, -0.002682268, -0.0068347994, -0.014507807, -0.
↪036872122, -0.054957043, -0.06691398, -0.06584696, -0.05263271, -0.033390157, -0.
↪015006201, -0.0034729026, -8.676593E-4;

            i22 = Eigen::ArrayXXd(2, 23);
            i22 << 0, 0.14884579, 0.29180932, 0.4054613, 0.46828458, 0.515565, 0.
↪5411826, 0.5746176, 0.6335705, 0.6963938, 0.74149364, 0.7689007, 0.8080032, 0.
↪8490102, 0.89000964, 0.93290585, 0.9660264, 0.99715817, 1.0224689, 1.0459764, 1.
↪0810461, 1.1632366, 1.50,
```

(continues on next page)

(continued from previous page)

```
                0, 0.0023498295, 0.0038403252, 0.012338308, 0.030489888, 0.06258152,␣
↪0.08079781, 0.09726137, 0.12063707, 0.13878864, 0.14566672, 0.14474949, 0.13772498,␣
↪0.12461021, 0.110625885, 0.08968175, 0.07049375, 0.047830947, 0.031265218, 0.
↪019913385, 0.012032943, 0.005803057, 0.0026086513;

            i33 = Eigen::ArrayXXd(2, 20);
            i33 << 0, 0.21936293, 0.31337926, 0.39961416, 0.44083738, 0.48011523, 0.
↪5254081, 0.56480104, 0.64147526, 0.64147526, 0.70819265, 0.7767404, 0.8334498, 0.
↪8939988, 0.9446548, 0.9952725, 1.0459669, 1.0967507, 1.1632637, 1.5000255,
                0, 0.002830162, 0.004290453, 0.009236455, 0.016043989, 0.023722952, 0.
↪0409085, 0.05637945, 0.07693768, 0.07693768, 0.08626908, 0.08693706, 0.081578515, 0.
↪07101401, 0.053551547, 0.033491757, 0.018626625, 0.009821928, 0.0053009023, 0.
↪0017315528;

        } else {
            throw std::invalid_argument("You can only apply signatures for type 'A'␣
↪or type 'B' eddies");
        }

        utilities::LinearInterpolant s11(i11.row(0).transpose(), i11.row(1).
↪transpose());
        utilities::LinearInterpolant s13(i13.row(0).transpose(), i13.row(1).
↪transpose());
        utilities::LinearInterpolant s22(i22.row(0).transpose(), i22.row(1).
↪transpose());
        utilities::LinearInterpolant s33(i33.row(0).transpose(), i33.row(1).
↪transpose());

        j.col(0) = s11(eta);
        j.col(2) = s13(eta);
        j.col(3) = s22(eta);
        j.col(5) = s33(eta);
    }
};

} /* namespace es */

#endif /* SOURCE_ADEM_SIGNATURE_H_ */
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `adem/biot_savart.h` (*File biot_savart.h*)
- `boost/algorithm/string/classification.hpp`
- `boost/algorithm/string/split.hpp`
- `cpplot.h`
- `io/variable_readers.h` (*File variable_readers.h*)
- `io/variable_writers.h` (*File variable_writers.h*)
- `math.h`

- `profile.h` (*File profile.h*)
- `relations/stress.h` (*File stress.h*)
- `relations/velocity.h` (*File velocity.h*)
- `stdexcept`
- `unsupported/Eigen/CXX11/Tensor`
- `unsupported/Eigen/FFT`
- `utilities/conv.h` (*File conv.h*)
- `utilities/filter.h` (*File filter.h*)
- `utilities/interp.h` (*File interp.h*)
- `utilities/tensors.h` (*File tensors.h*)
- `utilities/trapz.h` (*File trapz.h*)

Included By

- *File adem.h*

Namespaces

- *Namespace es*

Classes

- *Class EddySignature*

Typedefs

- *Typedef Array33d*
- *Typedef Array53d*

File smear.h

Parent directory (`source/utilities`)

Contents

- *Definition* (`source/utilities/smear.h`)

Definition (`source/utilities/smear.h`)

Program Listing for File smear.h

Return to documentation for file (`source/utilities/smear.h`)

```
/*
 * smear.h Brief overview sentence
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_SMEAR_H
#define ES_FLOW_SMEAR_H

#endif //ES_FLOW_SMEAR_H
```

File spectra.cpp

Parent directory (`source/relations`)

Contents

Definition (`source/relations/spectra.cpp`)

Program Listing for File spectra.cpp

Return to documentation for file (`source/relations/spectra.cpp`)

```
/*
 * spectra.cpp Brief overview sentence
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#include "spectra.h"
```

Includes

- `spectra.h` (*File spectra.h*)

File spectra.h

Parent directory (`source/relations`)

Contents

Definition (`source/relations/spectra.h`)

Program Listing for File spectra.h

Return to documentation for file (`source/relations/spectra.h`)

```
/*
 * spectra.h Brief overview sentence
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_SPECTRA_H
#define ES_FLOW_SPECTRA_H

class spectra {

};

#endif //ES_FLOW_SPECTRA_H
```

Included By

- *File spectra.cpp*

Classes

- *Class spectra*

File stress.h

Parent directory (`source/relations`)

Contents

Definition (`source/relations/stress.h`)

Program Listing for File stress.h

Return to documentation for file (`source/relations/stress.h`)

```
/*
 * stress.h Atmospheric Boundary Layer Reynolds Stress relations
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_STRESS_H_
#define ES_FLOW_STRESS_H_

#include <Eigen/Dense>
#include <Eigen/Core>
#include <iostream>
#include <iomanip>
#include <math.h>
#include "NumericalIntegration.h"
#include "profile.h"
#include "relations/velocity.h"
#include "utilities/integration.h"

#include "definitions.h"
#include "cpplot.h"

using namespace utilities;
using namespace cpplot;


namespace es {


template<typename T_scalar, typename T_param>
class DeficitFunctor {
private:
    double m_kappa, m_shear_ratio;
    T_param m_pi_coles;
```

(continues on next page)

(continued from previous page)

```
    bool m_lewkowicz, m_deficit_squared;

public:
    DeficitFunctor(const double kappa, const T_param pi_coles, const double shear_
↪ratio, const bool lewkowicz, const bool deficit_squared=false) : m_kappa(kappa), m_
↪pi_coles(pi_coles), m_shear_ratio(shear_ratio), m_lewkowicz(lewkowicz), m_deficit_
↪squared(deficit_squared) {};
    T_scalar operator() (T_scalar eta) const{
        if (m_deficit_squared){
            return pow(deficit(eta, m_kappa, m_pi_coles, m_shear_ratio, m_lewkowicz),␣
↪2.0);

        }
        return deficit(eta, m_kappa, m_pi_coles, m_shear_ratio, m_lewkowicz);
    };
};


void reynolds_stress_13(Eigen::ArrayXd &r13_a, Eigen::ArrayXd &r13_b, const double␣
↪beta, const Eigen::ArrayXd &eta, const double kappa, const double pi_coles, const␣
↪double shear_ratio, const double zeta){

    // Check the input has a valid range; the integration will be messed up otherwise.
    if ((eta(eta.rows()-1) != 1.0) || (eta(0) != 0.0)) {
        throw std::invalid_argument("Input eta must be defined in the range 0, 1␣
↪exactly");
    }

    // TODO Improved control of models - see #61.
    bool lewkowicz = true;

    // Get f between eta = 0 and eta = 1
    Eigen::ArrayXd f = deficit(eta, kappa, pi_coles, shear_ratio, lewkowicz);
    const double d_pi = 0.001 * pi_coles;

    // Do the integrations for ei, with central differencing for numerical␣
↪differentiation of e coefficients 5-7
    DeficitFunctor<double, double> deficit_functor(KAPPA_VON_KARMAN, pi_coles, shear_
↪ratio, true, false);
    DeficitFunctor<double, double> deficit_functor_plus(KAPPA_VON_KARMAN, pi_coles+d_
↪pi, shear_ratio, true, false);
    DeficitFunctor<double, double> deficit_functor_minus(KAPPA_VON_KARMAN, pi_coles-d_
↪pi, shear_ratio, true, false);
    DeficitFunctor<double, double> deficit_squared_functor(KAPPA_VON_KARMAN, pi_coles,
↪ shear_ratio, true, true);
    DeficitFunctor<double, double> deficit_squared_functor_plus(KAPPA_VON_KARMAN, pi_
↪coles+d_pi, shear_ratio, true, true);
    DeficitFunctor<double, double> deficit_squared_functor_minus(KAPPA_VON_KARMAN, pi_
↪coles-d_pi, shear_ratio, true, true);
    Eigen::ArrayXd e1 = utilities::cumulative_integrate(eta, deficit_functor);
    Eigen::ArrayXd e1_plus = utilities::cumulative_integrate(eta, deficit_functor_
↪plus);
    Eigen::ArrayXd e1_minus = utilities::cumulative_integrate(eta, deficit_functor_
↪minus);
    Eigen::ArrayXd e2 = utilities::cumulative_integrate(eta, deficit_squared_functor);
    Eigen::ArrayXd e2_plus = utilities::cumulative_integrate(eta, deficit_squared_
↪functor_plus);
```

(continues on next page)

(continued from previous page)

```
    Eigen::ArrayXd e2_minus = utilities::cumulative_integrate(eta, deficit_squared_
↪functor_minus);
    Eigen::ArrayXd e3 = f * e1;
    Eigen::ArrayXd e4 = eta * f;
    Eigen::ArrayXd e5 = (e1_plus - e1_minus) / (2.0 * d_pi);
    Eigen::ArrayXd e6 = (e2_plus - e2_minus) / (2.0 * d_pi);
    Eigen::ArrayXd e7 = f * e5;

    // Get C1 value (eqn 14) which is the end value of the e1 distribution
    double c1 = e1[e1.rows()-1];

    // Get A coefficients from equations A2 a-d
    Eigen::ArrayXd a1 = e2 - e3 + shear_ratio * (e4 - e1);
    Eigen::ArrayXd a2 = (-2.0 * e2) + e3 + (shear_ratio * e1);
    Eigen::ArrayXd a3 = e6 - e7 - (shear_ratio * e5);
    Eigen::ArrayXd a4 = (2.0 * e2) - e3 + (shear_ratio * e4) - (shear_ratio * 3.0 *␣
↪e1);

    // B coefficients are simply evaluated at eta = 1 (eqn A6)
    double b1 = a1(eta.rows()-1);
    double b2 = a2(eta.rows()-1);
    double b3 = a3(eta.rows()-1);
    double b4 = a4(eta.rows()-1);

    // E1 from (eqn A4). Can't call it E1 due to name conflict with above.
    double e1_coeff = 1.0 / (kappa * shear_ratio + 1.0);

    // TODO Resolve issue #59 here.
    double n = coles_wake(1.0, pi_coles);

    // Compile f_i terms
    Eigen::ArrayXd f1;
    Eigen::ArrayXd f2;
    Eigen::ArrayXd f3;
    f1 = 1.0
        - a1 / (b1 + e1_coeff * b2)
        - e1_coeff * a2 / (b1 + e1_coeff * b2);

    f2 = (e1_coeff * n * a2 * b1
        + a3 * b1
        - e1_coeff * n * a1 * b2
        + e1_coeff * a3 * b2
        - a1 * b3
        - e1_coeff * a2 * b3) / (b1 + e1_coeff * b2);

    f3 = (e1_coeff * a2 * b1
        + a4 * b1
        - e1_coeff * a1 * b2
        + e1_coeff * a4 * b2
        - a1 * b4
        - e1_coeff * a2 * b4) / (b1 + e1_coeff * b2);

    // Convert f2 and f3 into g1 and g2, apply eq. 15
    Eigen::ArrayXd g1 = f2 / shear_ratio;
    Eigen::ArrayXd g2 = -f3 / (c1 * shear_ratio);
    Eigen::ArrayXd minus_r13 = f1 + g1 * zeta + g2 * beta;
```

(continues on next page)

(continued from previous page)

```
    // As a validation check, reproduce figure
    Figure fig = Figure();
    ScatterPlot p1 = ScatterPlot();
    p1.x = eta;
    p1.y = f1;
    p1.name = "f1";
    ScatterPlot p2 = ScatterPlot();
    p2.x = eta;
    p2.y = g1*zeta;
    p2.name = "g1*zeta";
    ScatterPlot p3 = ScatterPlot();
    p3.x = eta;
    p3.y = g2*beta;
    p3.name = "g2*beta";
    ScatterPlot p4 = ScatterPlot();
    p4.x = eta;
    p4.y = minus_r13;
    p4.name = "tau / tau_0 (= -r13)";
    fig.add(p1);
    fig.add(p2);
    fig.add(p3);
    fig.add(p4);
    Layout lay = Layout("Check reproduction of Perry & Marusic 1995 Figure 1");
    lay.xTitle("$\\eta$");
    fig.setLayout(lay);
    fig.write("check_perry_marusic_fig_1");

    Eigen::ArrayXd minus_r13_a(eta.rows());
    if (lewkowicz) {
        // Lewkowicz 1982 shear stress for equilibrium sink flow, Perry and Marusic␣
↪eqn. 51
        minus_r13_a = 1.0
            - (60.0 / 59.0) * eta
            - (20.0 / 59.0) * eta.pow(3.0)
            + (45.0 / 59.0) * eta.pow(4.0)
            - (24.0 / 59.0) * eta.pow(5.0)
            + (60.0 / 59.0) * eta * eta.log();
    } else {
        // Shear stress for `pure` equilibrium sink flow with no correction, Perry␣
↪and Marusic eqn. 53
        minus_r13_a = 1.0 - eta + eta * eta.log();
    }

    // Handle the log(0) singularity
    for (int i = 0; i < minus_r13_a.size(); i++) {
        if (std::isinf(minus_r13_a(i)) || std::isnan(minus_r13_a(i))) {
            minus_r13_a(i) = 1.0;
        }
    }

    // Output correctly signed reynolds stress components
    r13_a = -1.0 * minus_r13_a;
    r13_b = -1.0 * (minus_r13 - minus_r13_a);

}

} /* namespace es */
```

(continues on next page)

(continued from previous page)

```
#endif /* ES_FLOW_STRESS_H_ */
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `NumericalIntegration.h`
- `cpplot.h`
- `definitions.h` (*File definitions.h*)
- `iomanip`
- `iostream`
- `math.h`
- `profile.h` (*File profile.h*)
- `relations/velocity.h` (*File velocity.h*)
- `utilities/integration.h` (*File integration.h*)

Included By

- *File adem.h*
- *File signature.h*

Namespaces

- *Namespace es*

Classes

- *Template Class DeficitFunctor*

Functions

- *Function es::reynolds_stress_13*

File tensors.h

Parent directory (`source/utilities`)

Contents

Definition (source/utilities/tensors.h)

Program Listing for File tensors.h

Return to documentation for file (source/utilities/tensors.h)

```
/*
 * tensors.h Utilities to help with Eigen::Tensors
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_TENSORS_H
#define ES_FLOW_TENSORS_H

#include <unsupported/Eigen/CXX11/Tensor>
#include <iostream>
#include <string>


namespace utilities {


template<typename T>
using  MatrixType = Eigen::Matrix<T,Eigen::Dynamic, Eigen::Dynamic>;


template<typename T>
using  ArrayType = Eigen::Array<T,Eigen::Dynamic, Eigen::Dynamic>;


template<typename Scalar,int rank, typename sizeType>
auto tensor_to_matrix(const Eigen::Tensor<Scalar,rank> &tensor,const sizeType rows,
↪const sizeType cols)
{
    return Eigen::Map<const MatrixType<Scalar>> (tensor.data(), rows,cols);
}
```

(continues on next page)

(continued from previous page)

```
template<typename Scalar, typename... Dims>
auto matrix_to_tensor(const MatrixType<Scalar> &matrix, Dims... dims)
{
    constexpr int rank = sizeof... (Dims);
    return Eigen::TensorMap<Eigen::Tensor<const Scalar, rank>>(matrix.data(), {dims...
↪});
}


template<typename Scalar,int rank, typename sizeType>
auto tensor_to_array(const Eigen::Tensor<Scalar,rank> &tensor,const sizeType rows,
↪const sizeType cols)
{
    return Eigen::Map<const ArrayType<Scalar>> (tensor.data(), rows,cols);
}


template<typename Scalar, typename... Dims>
auto array_to_tensor(const ArrayType<Scalar> &matrix, Dims... dims)
{
    constexpr int rank = sizeof... (Dims);
    return Eigen::TensorMap<Eigen::Tensor<const Scalar, rank>>(matrix.data(), {dims...
↪});
}


template<typename T>
std::string tensor_dims(T &tensor) {
    std::stringstream dims;
    for (auto i = tensor.dimensions().begin(); i != tensor.dimensions().end(); ++i) {
        dims << *i << " x ";
    }
    std::string dim_str = dims.str();
    dim_str.pop_back();
    dim_str.pop_back();
    dim_str.pop_back();
    return dim_str;
}

} /* namespace utilities */

#endif //ES_FLOW_TENSORS_H
```

Includes

- `iostream`
- `string`
- `unsupported/Eigen/CXX11/Tensor`

Included By

- *File adem.h*

- *File signature.h*

Namespaces

- *Namespace utilities*

Functions

- *Template Function utilities::array_to_tensor*
- *Template Function utilities::matrix_to_tensor*
- *Template Function utilities::tensor_dims*
- *Template Function utilities::tensor_to_array*
- *Template Function utilities::tensor_to_matrix*

Typedefs

- *Typedef utilities::ArrayType*
- *Typedef utilities::MatrixType*

File trapz.h

Parent directory (`source/utilities`)

Contents

Definition (`source/utilities/trapz.h`)

Program Listing for File trapz.h

Return to documentation for file (`source/utilities/trapz.h`)

```
/*
 * trapz.h Trapezoidal Numerical Integration for Eigen
 *
 * Author:              Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
```

(continues on next page)

(continued from previous page)

```
 *
 */

#ifndef ES_FLOW_TRAPZ_H
#define ES_FLOW_TRAPZ_H

#include <Eigen/Dense>
#include <stdexcept>


namespace utilities {


template<typename Derived, typename OtherDerived>
EIGEN_STRONG_INLINE void trapz(Eigen::ArrayBase<OtherDerived> const & out, const␣
↪Eigen::ArrayBase<Derived>& in, Eigen::Index direction=1) {
    Eigen::ArrayBase<OtherDerived> &out_ = const_cast< Eigen::ArrayBase<OtherDerived>␣
↪& >(out);
    Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,
                 Eigen::ArrayBase<Derived>::RowsAtCompileTime,
                 Eigen::ArrayBase<Derived>::ColsAtCompileTime> inter;

    if (direction == 1) {
        if (in.rows() == 1) {
            out_.derived().setZero(1, in.cols());
        } else {
            inter = (in.topRows(in.rows() - 1) + in.bottomRows(in.rows() - 1)) * 0.5;
            out_.derived() = inter.colwise().sum();
        }
    } else if (direction == 2) {
        if (in.cols() == 1) {
            out_.derived().setZero(in.rows(), 1);
        } else {
            inter = (in.leftCols(in.cols() - 1) + in.rightCols(in.cols() - 1)) * 0.5;
            out_.derived() = inter.rowwise().sum();
        }
    } else {
        assert(((direction == 1) || (direction == 2)));
    }

}

/*
 * Overload method to return result by value.
 */
template<typename Derived>
EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar,␣
↪Eigen::ArrayBase<Derived>::RowsAtCompileTime, Eigen::ArrayBase<Derived>
↪::ColsAtCompileTime> trapz(const Eigen::ArrayBase<Derived>& y, Eigen::Index␣
↪dimension=1)
{
    Eigen::Array<typename Eigen::ArrayBase<Derived>::Scalar, Eigen::ArrayBase<Derived>
↪::RowsAtCompileTime, Eigen::ArrayBase<Derived>::ColsAtCompileTime> z;
    trapz(z,y, dimension);
    return z;
}
```

(continues on next page)

(continued from previous page)

```
template<typename DerivedX, typename DerivedY, typename DerivedOut>
EIGEN_STRONG_INLINE void trapz(Eigen::ArrayBase<DerivedOut> const & out, const␣
↪Eigen::ArrayBase<DerivedX>& in_x, const Eigen::ArrayBase<DerivedY>& in_y)
{
    // Input size check
    eigen_assert(in_x.rows() == in_y.rows());
    eigen_assert(in_x.cols() == 1);

    // Get dx for each piece of the integration
    Eigen::Array<typename Eigen::ArrayBase<DerivedX>::Scalar, Eigen::ArrayBase
↪<DerivedX>::RowsAtCompileTime, Eigen::ArrayBase<DerivedX>::ColsAtCompileTime> dx;
    dx = (in_x.bottomRows(in_x.rows()-1) - in_x.topRows(in_x.rows()-1));


    // Get the average heights of the trapezoids
    Eigen::Array<typename Eigen::ArrayBase<DerivedY>::Scalar, Eigen::ArrayBase
↪<DerivedY>::RowsAtCompileTime, Eigen::ArrayBase<DerivedY>::ColsAtCompileTime> inter;
    inter = (in_y.topRows(in_y.rows()-1) + in_y.bottomRows(in_y.rows()-1)) * 0.5;

    // Multiply by trapezoid widths. NB Broadcasting with *= only works for arrayX␣
↪types, not arrayXX types like dx
    //  inter *= dx;
    for (int i = 0 ; i < inter.cols() ; ++i)
    {
        for (int j = 0 ; j < inter.rows() ; ++j)
        {
            inter(j,i) *= dx(j,0);
        }
    }

    // Initialise output
    Eigen::ArrayBase<DerivedOut>& out_ = const_cast< Eigen::ArrayBase<DerivedOut>& >
↪(out);
    out_.derived().setZero(1, in_y.cols());

    // Output the column-wise sum
    out_.derived() = inter.colwise().sum();
}

/* @brief Trapezoidal numerical integration with non-unit spacing
 * Overload method to return result by value.
 * TODO add dimension argument, for optional integration along the second dimension.
 */
template<typename DerivedX, typename DerivedY, typename DerivedOut>
EIGEN_STRONG_INLINE Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar,␣
↪Eigen::ArrayBase<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>
↪::ColsAtCompileTime> trapz(const Eigen::ArrayBase<DerivedX>& x, const␣
↪Eigen::ArrayBase<DerivedY>& y)
{
    Eigen::Array<typename Eigen::ArrayBase<DerivedOut>::Scalar, Eigen::ArrayBase
↪<DerivedOut>::RowsAtCompileTime, Eigen::ArrayBase<DerivedOut>::ColsAtCompileTime> z;
    trapz(z, x, y);
    return z;
}

}  /* namespace utilities */
```

(continues on next page)

(continued from previous page)

```
#endif //ES_FLOW_TRAPZ_H
```

Includes

- `Eigen/Dense`
- `stdexcept`

Included By

- *File signature.h*

Namespaces

- *Namespace utilities*

Functions

- *Template Function utilities::trapz(const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)*
- *Template Function utilities::trapz(Eigen::ArrayBase<DerivedOut> const&, const Eigen::ArrayBase<DerivedX>&, const Eigen::ArrayBase<DerivedY>&)*
- *Template Function utilities::trapz(const Eigen::ArrayBase<Derived>&, Eigen::Index)*
- *Template Function utilities::trapz(Eigen::ArrayBase<OtherDerived> const&, const Eigen::ArrayBase<Derived>&, Eigen::Index)*

File variable_readers.h

Parent directory (`source/io`)

Contents

Definition (source/io/variable_readers.h)

Program Listing for File variable_readers.h

Return to documentation for file (source/io/variable_readers.h)

```
/*
 * variable_readers.h Helper functions for reading data from mat files to Eigen␣
↪arrays and vectors.
 *
 * References:
 *
 *   [1] Eigen: http://eigen.tuxfamily.org/dox/index.html
 *
 *   [2] matio: https://sourceforge.net/projects/matio/
 *
 *   [3] eigen-matio: https://github.com/tesch1/eigen-matio
 *
 * Future Improvements:
 *
 *   [1] Extension to structs and a range of different types
 *
 *   [2] More elegant implementation based on type, or possibly use of eigen-matio␣
↪(ref [3])
 *
 * Author:                    Tom Clark (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_VARIABLE_READERS_H
#define ES_FLOW_VARIABLE_READERS_H

#include <iostream>
#include <string>
#include "matio.h"
#include <eigen3/Eigen/Core>
#include <unsupported/Eigen/CXX11/Tensor>

using Eigen::Array;
using Eigen::ArrayXd;
using Eigen::Array3d;
using Eigen::ArrayXXd;
using Eigen::Vector3d;
using Eigen::VectorXd;
using Eigen::Tensor;
using Eigen::Dynamic;
typedef Eigen::Array<double, 3, 2> Array32d;


// TODO the Eigen variable readers are getting very unwieldy. Template them!

namespace es {


template<size_t SIZE, class T> inline size_t array_size(T (&arr)[SIZE]) {
```

(continues on next page)

(continued from previous page)

```
    return SIZE;
}

void checkVariableType(matvar_t *mat_var, int matvar_type) {

    // Throw error if the requested data type does not match the saved data type
    if (mat_var->class_type != matvar_type) {
        std::string msg = "Error reading mat file (" + std::string(mat_var->name) + "␣
↪incorrect variable type)";
        throw std::invalid_argument(msg);
    }

}

matvar_t * getVariable(mat_t *matfp, const std::string var_name, bool print_var =␣
↪true, int max_rank = 2) {

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = Mat_VarRead(matfp, var_name.c_str());
    if (mat_var == NULL) {
        std::string msg = "Error reading mat file (most likely incorrect file type: "␣
↪+ var_name + " variable is missing)";
        throw std::invalid_argument(msg);
    }

    // Optionally print the variable information to terminal
    if (print_var) {
        std::cout << "Reading variable: " << std::endl;
        Mat_VarPrint(mat_var, true);
    }

    // Check the rank
    if (mat_var->rank > max_rank) {
        std::string msg = "Rank of variable " + var_name + " exceeds required rank of
↪" + std::to_string(max_rank) ;
        throw std::invalid_argument(msg);
    }

    return mat_var;
}

std::string readString(mat_t *matfp, const std::string var_name, bool print_var) {

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Read the const char from the file, instantiate as std::string
    std::string var = std::string((const char *) mat_var->data, mat_var->dims[1]);

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;
}

Vector3d readVector3d(mat_t *matfp, const std::string var_name, bool print_var) {

    // Get the variable's structure pointer and check validity
```

(continues on next page)

(continued from previous page)

```
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Check for three elements always
    if (mat_var->dims[1]*mat_var->dims[0] != 3) {
        std::string msg = "Number of elements in variable '" + var_name + "' not␣
↪equal to 3";
        throw std::invalid_argument(msg);
    }

    // Read a vector, with automatic transpose to column vector always.
    Vector3d var;
    if (mat_var->dims[0] == 1) {
        if (print_var) std::cout << "Converting row vector '" << mat_var->name << "'␣
↪to column vector" << std::endl;
    }
    var = Vector3d();
    double *var_d = (double *) mat_var->data;
    long int i = 0;
    for (i = 0; i < 3; i++) {
        var[i] = var_d[i];
    }

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;

}


Array3d readArray3d(mat_t *matfp, const std::string var_name, bool print_var) {
    return readVector3d(matfp, var_name, print_var).array();
}


VectorXd readVectorXd(mat_t *matfp, const std::string var_name, bool print_var) {

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Read a vector, with automatic transpose to column vector always.
    VectorXd var;
    if (mat_var->dims[0] == 1) {
        if (print_var) std::cout << "Converting row vector '" << mat_var->name << "'␣
↪to column vector" << std::endl;
        var = VectorXd(mat_var->dims[1], mat_var->dims[0]);
    } else {
        var = VectorXd(mat_var->dims[0], mat_var->dims[1]);
    }

    // Copy the data into the native Eigen types. However, we can also map to data␣
↪already in
    // memory which avoids a copy. Read about mapping here:
    // http://eigen.tuxfamily.org/dox/group__TutorialMapClass.html
    // TODO consider mapping to reduce peak memory overhead
    double *var_d = (double *) mat_var->data;
    long int i = 0;
    for (i = 0; i < var.rows()*var.cols(); i++) {
```

(continues on next page)

(continued from previous page)

```
        var[i] = var_d[i];
    }

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;

}


ArrayXd readArrayXd(mat_t *matfp, const std::string var_name, bool print_var) {
    return readVectorXd(matfp, var_name, print_var).array();
}


Array32d readArray32d(mat_t *matfp, const std::string var_name, bool print_var) {

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Check for three elements always
    if ((mat_var->dims[0] != 3) || (mat_var->dims[1] != 2)) {
        std::string msg = "Variable '" + var_name + "' must be of size 3 x 2";
        throw std::invalid_argument(msg);
    }

    // Declare and size the array
    Eigen::Array<double, 3, 2> var;
    var = Eigen::Array<double, 3, 2>();

    // Copy the data into the native Eigen types. However, we can also map to data␣
↪already in
    // memory which avoids a copy. Read about mapping here:
    // http://eigen.tuxfamily.org/dox/group__TutorialMapClass.html
    // TODO consider mapping to reduce peak memory overhead
    double *var_d = (double *) mat_var->data;
    long int i = 0;
    for (i = 0; i < var.rows()*var.cols(); i++) {
        var(i) = var_d[i];
    }

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;

}


ArrayXXd readArrayXXd(mat_t *matfp, const std::string var_name, bool print_var) {

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Read an array. We always read as eigen arrays, not matrices, as these are far␣
↪more flexible.
    // Can easily cast to matrix type later if linear algebra functionality is needed.
```

(continues on next page)

(continued from previous page)

```
    ArrayXXd var;
    var = ArrayXXd(mat_var->dims[0],  mat_var->dims[1]);

    // Copy the data into the native Eigen types. However, we can also map to data␣
↪already in
    // memory which avoids a copy. Read about mapping here:
    // http://eigen.tuxfamily.org/dox/group__TutorialMapClass.html
    // TODO consider mapping to reduce peak memory overhead
    double *var_d = (double *) mat_var->data;
    long int i = 0;
    long int j = 0;
    long int ind = 0;
    for (j = 0; j < var.cols(); j++) {
        for (i = 0; i < var.rows(); i++) {
            var(i, j) = var_d[ind];
            ind = ind+1;
        }
    }

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;

}

Tensor<double, 3> readTensor3d(mat_t *matfp, const std::string var_name, bool print_
↪var) {

    // Get the variable's structure pointer and check validity for rank 3
    matvar_t *mat_var = getVariable(matfp, var_name, print_var, 3);

    // Read the tensor
    // TODO this code typecasts from size_t to long int, meaning possible data loss␣
↪for very large arrays. Add a check.
    Eigen::Index dim0 = mat_var->dims[0];
    Eigen::Index dim1 = mat_var->dims[1];
    Eigen::Index dim2 = mat_var->dims[2];
    // std::cout  << "Initialising Tensor of size: " << dim0 << ", " << dim1 << ", "
↪<< dim2 << std::endl;
    Eigen::Tensor<double, 3> var = Eigen::Tensor<double, 3>(dim0, dim1, dim2);
    var.setZero();

    // Copy the data into the native Eigen types. However, we can also map to data␣
↪already in
    // memory which avoids a copy. Read about mapping here:
    // http://eigen.tuxfamily.org/dox/group__TutorialMapClass.html
    // TODO consider mapping to reduce peak memory overhead
    double *var_d = (double *) mat_var->data;
    long int i = 0;
    long int j = 0;
    long int k = 0;
    long int ind = 0;
    for (k = 0; k < var.dimensions()[2]; k++) {
        for (j = 0; j < var.dimensions()[1]; j++) {
            for (i = 0; i < var.dimensions()[0]; i++) {
                var(i, j, k) = var_d[ind];
                ind = ind + 1;
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;

}

double readDouble(mat_t *matfp, const std::string var_name, bool print_var){

    // Get the variable's structure pointer and check validity
    matvar_t *mat_var = getVariable(matfp, var_name, print_var);

    // Read a single element double
    double *var_d = (double *) mat_var->data;
    double var = var_d[0];

    // Free the data pointer and return the new variable
    Mat_VarFree(mat_var);
    return var;
}


} /* end namespace */


#endif // ES_FLOW_VARIABLE_READERS_H
```

Includes

- eigen3/Eigen/Core
- iostream
- matio.h
- string
- unsupported/Eigen/CXX11/Tensor

Included By

- *File adem.h*
- *File signature.h*
- *File data_types.h*

Namespaces

- *Namespace es*

Functions

- *Template Function es::array_size*
- *Function es::checkVariableType*
- *Function es::getVariable*
- *Function es::readArray32d*
- *Function es::readArray3d*
- *Function es::readArrayXd*
- *Function es::readArrayXXd*
- *Function es::readDouble*
- *Function es::readString*
- *Function es::readTensor3d*
- *Function es::readVector3d*
- *Function es::readVectorXd*

Typedefs

- *Typedef Array32d*

File variable_writers.h

Parent directory (`source/io`)

Contents

Definition (`source/io/variable_writers.h`)

Program Listing for File variable_writers.h

Return to documentation for file (`source/io/variable_writers.h`)

```
/*
 * variable_writers.h Helper functions for writing data to mat files from Eigen␣
↪arrays and vectors.
 *
 * References:
 *
 *   [1] Eigen: http://eigen.tuxfamily.org/dox/index.html
 *
 *   [2] matio: https://sourceforge.net/projects/matio/
 *
 *   [3] eigen-matio: https://github.com/tesch1/eigen-matio
 *
 * Future Improvements:
 *
 *   [1] Extension to structs and a range of different types
 *
 *   [2] More elegant implementation based on type, or possibly use of eigen-matio␣
↪(ref [3])
 *
 * Author:                    Tom Clark (thclark @ github)
 *
 * Copyright (c) 2019 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_VARIABLE_WRITERS_H
#define ES_FLOW_VARIABLE_WRITERS_H

#include <iostream>
#include <string>
#include <Eigen/Dense>
#include <Eigen/Core>
#include <Eigen/Dense>
#include <unsupported/Eigen/CXX11/Tensor>

#include "matio.h"

using Eigen::Array;
using Eigen::Array3d;
using Eigen::ArrayXXd;
using Eigen::Vector3d;
using Eigen::VectorXd;
using Eigen::Tensor;
using Eigen::Dynamic;


typedef Eigen::Array<double, 3, 2> Array32d;


// TODO the Eigen variable writers are getting very unwieldy. Template them!


namespace es {


void writeString(mat_t *mat_fp, const std::string &var_name, std::string &var) {
```

(continues on next page)

(continued from previous page)

```
    // Create the variable
    size_t dims[2] = {1, var.length()};
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_CHAR,
        MAT_T_UTF8,
        2,
        dims,
        static_cast<void*>(const_cast<char*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeArray3d(mat_t *mat_fp, const std::string &var_name, const Eigen::Array3d &
↪var) {

    // Create the variable
    size_t dims[2] = {3, 1};
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
        2,
        dims,
        static_cast<void*>(const_cast<double*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeArrayXd(mat_t *mat_fp, const std::string &var_name, const Eigen::ArrayXd &
↪var) {

    // Create the variable
    size_t dims[2];
    dims[0] = var.rows();
    dims[1] = 1;
    matvar_t *mat_var = Mat_VarCreate(
```

(continues on next page)

(continued from previous page)

```
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
        2,
        dims,
        static_cast<void*>(const_cast<double*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeArrayXXd(mat_t *mat_fp, const std::string &var_name, const Eigen::ArrayXXd &
↪var) {

    // Create the variable
    size_t dims[2];
    dims[0] = var.rows();
    dims[1] = var.cols();
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
        2,
        dims,
        static_cast<void*>(const_cast<double*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeArray32d(mat_t *mat_fp, const std::string &var_name, const Eigen::Array
↪<double, 3, 2> &var) {

    // Create the variable
    size_t dims[2] = {3, 2};
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
```

(continues on next page)

(continued from previous page)

```
        2,
        dims,
        static_cast<void*>(const_cast<double*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeTensor3d(mat_t *mat_fp, const std::string &var_name, const Tensor<double, 3>
↪ &var) {

    // Create the variable
    size_t dims[3];
    dims[0] = var.dimensions()[0];
    dims[1] = var.dimensions()[1];
    dims[2] = var.dimensions()[2];
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
        3,
        dims,
        static_cast<void*>(const_cast<double*>(var.data())),
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


void writeDouble(mat_t *mat_fp, const std::string &var_name, const double var){

    // Create the variable
    size_t dims[2] = {1, 1};
    matvar_t *mat_var = Mat_VarCreate(
        var_name.c_str(),
        MAT_C_DOUBLE,
        MAT_T_DOUBLE,
        3,
        dims,
        static_cast<void*>(const_cast<double*>(&var)),
```

(continues on next page)

(continued from previous page)

```
        0
    );
    if (nullptr == mat_var ) {
        throw std::runtime_error("Unable to write variable '" + var_name + "' to file.
↪");
    }

    // Write the data and free the pointer
    Mat_VarWrite(mat_fp, mat_var, MAT_COMPRESSION_NONE);
    Mat_VarFree(mat_var);

}


} /* end namespace */


#endif // ES_FLOW_VARIABLE_WRITERS_H
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `iostream`
- `matio.h`
- `string`
- `unsupported/Eigen/CXX11/Tensor`

Included By

- *File signature.h*

Namespaces

- *Namespace es*

Functions

- *Function es::writeArray32d*
- *Function es::writeArray3d*
- *Function es::writeArrayXd*
- *Function es::writeArrayXXd*
- *Function es::writeDouble*
- *Function es::writeString*

- *Function es::writeTensor3d*

Typedefs

- *Typedef Array32d*

File veer.h

Parent directory (`source/relations`)

Contents

Definition (`source/relations/veer.h`)

Program Listing for File veer.h

Return to documentation for file (`source/relations/veer.h`)

```
/*
 * veer.h Atmospheric Boundary Layer Veer relations
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2015-9 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_VEER_H_
#define ES_FLOW_VEER_H_

#include <Eigen/Dense>
#include <Eigen/Core>

#include "definitions.h"
#include "profile.h"


namespace es {


template <typename T>
T veer_lhs(T const & ui, const double ui_g, const double phi){
    T lhs = 2.0*OMEGA_WORLD*sind(phi)*(ui_g - ui);
    return lhs;
```

(continues on next page)

(continued from previous page)

```
}


template <typename T>
T veer_rhs(T & ui, T & uiu3_bar, const double nu){
    T rhs, mixing_term;
    mixing_term = nu*ui.getZDerivative() + uiu3_bar;
    rhs = mixing_term.getZDerivative();
    return rhs;
}

} /* namespace es */

#endif /* ES_FLOW_VEER_H_ */
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `definitions.h` (*File definitions.h*)
- `profile.h` (*File profile.h*)

Namespaces

- *Namespace es*

Functions

- *Template Function es::veer_lhs*
- *Template Function es::veer_rhs*

File velocity.h

Parent directory (`source/relations`)

Contents

Definition (source/relations/velocity.h)

Program Listing for File velocity.h

Return to documentation for file (source/relations/velocity.h)

```
/*
 * velocity.h Atmospheric Boundary Layer velocity profile relations
 *
 * Author:               Tom Clark  (thclark @ github)
 *
 * Copyright (c) 2013-9 Octue Ltd. All Rights Reserved.
 *
 */

#ifndef ES_FLOW_VELOCITY_H_
#define ES_FLOW_VELOCITY_H_

#include <Eigen/Dense>
#include <Eigen/Core>
#include "profile.h"
#include <iostream>


namespace es {

template <typename T, typename Talf>
T power_law_speed(T const & z, const double u_ref, const double z_ref, Talf const &␣
↪alpha){
    T z_norm = z / z_ref;
    T speed = pow(z_norm, alpha) * u_ref;
    return speed;
};

// Remove template specialisations from doc (causes duplicate) @cond
Eigen::VectorXd power_law_speed(Eigen::VectorXd const & z, const double u_ref, const␣
↪double z_ref, const double alpha) {
    Eigen::VectorXd z_norm = z.array() / z_ref;
    Eigen::VectorXd speed = pow(z_norm.array(), alpha) * u_ref;
    return speed;
};
// @endcond


template <typename T>
T most_law_speed(T const & z, const double kappa, const double d, const double z0,␣
↪const double L){
    std::cout << "MOST Law not implemented yet" << std::endl;
    T speed;
    return speed;
}

// Remove template specialisation from doc (causes duplicate) @cond
Eigen::VectorXd most_law_speed(Eigen::VectorXd const & z, const double kappa, const␣
↪double d, const double z0, const double L){
    std::cout << "MOST Law not implemented yet" << std::endl;
    Eigen::VectorXd speed;
```

(continues on next page)

(continued from previous page)

```
    return speed;
};
// @endcond


template <typename T_z, typename T_pi_j>
T_z marusic_jones_speed(T_z const & z, T_pi_j const pi_j, const double kappa, const␣
↪double z_0,
                        const double delta, const double u_inf, const double u_tau){
    T_z eta = (z + z_0) / (delta + z_0);
    T_z eta_cubed = pow(eta, 3.0);
    T_z term1 = log(eta) / kappa;
    T_z term2 = (eta_cubed - 1.0) / (3.0 * kappa);
    T_z term3 = 2.0 * pi_j * (1.0 - pow(eta, 2.0) * 3.0 + eta_cubed * 2.0) / kappa;
    T_z u_deficit = term2 - term1 + term3;
    T_z speed = u_inf - u_deficit * u_tau;
    return speed;
};

// Remove template specialisation from doc (causes duplicate) @cond
template <typename T_pi_j>
Eigen::VectorXd marusic_jones_speed(Eigen::VectorXd const & z, T_pi_j const pi_j,␣
↪const double kappa, const double z_0,
                                    const double delta, const double u_inf, const double u_
↪tau){
    Eigen::VectorXd eta = (z.array() + z_0) / (delta + z_0);
    Eigen::VectorXd eta_cubed = eta.array().cube();
    Eigen::VectorXd term1 = eta.array().log() / kappa;
    Eigen::VectorXd term2 = (eta_cubed.array() - 1.0) / (3.0 * kappa);
    Eigen::VectorXd term3 = 2.0 * pi_j * (1.0 - eta.array().square() * 3.0 + eta_
↪cubed.array() * 2.0) / kappa;
    Eigen::VectorXd u_deficit = term2 - term1 + term3;
    Eigen::VectorXd speed = u_inf - u_deficit.array() * u_tau;
    return speed;
};
//@endcond


template <typename T_z, typename T_param>
T_z coles_wake(T_z const &eta, T_param const &pi_coles, const bool corrected=true){
    T_z wake_param, eta_sqd;
    eta_sqd = pow(eta, 2.0);
    if (corrected) {
        wake_param = 2.0 * eta_sqd * (3.0 - 2.0 * eta)
            - eta_sqd * (1.0 - eta) * (1.0 - 2.0 * eta) / pi_coles;
    } else {
        wake_param = 2.0 * eta_sqd * (3.0 - 2.0 * eta);
    }
    return wake_param;
};

// Remove template specialisation from doc (causes duplicate) @cond
template <typename T_param>
Eigen::VectorXd coles_wake(Eigen::VectorXd const &eta, T_param const &pi_coles, const␣
↪bool corrected=true){
    Eigen::VectorXd wake_param, eta_sqd;
    eta_sqd = eta.array().pow(2.0);
```

(continues on next page)

(continued from previous page)

```
    if (corrected) {
        wake_param = 2.0 * eta_sqd.array() * (3.0 - 2.0 * eta.array())
            - eta_sqd.array() * (1.0 - eta.array()) * (1.0 - 2.0*eta.array()) / pi_
↪coles;
    } else {
        wake_param = 2.0 * eta_sqd.array() * (3.0 - 2.0 * eta.array());
    }
    return wake_param;
};
//@endcond


// Do not document @cond
/* Template wrapper for std::isinf to kill off problems where ceres::Jet is used␣
↪(autodifferentiation) instead of
 * a double. Call it is_dbl_inf rather than isinf to avoid accidental use.
 */
bool is_double_inf(double in) {
    return std::isinf(in);
};

template <typename T>
bool is_double_inf(T in) {
    return false;
};
// @endcond


template <typename T_eta, typename T_param>
T_eta deficit(T_eta const &eta, const double kappa, T_param const & pi_coles, const␣
↪double shear_ratio, const bool lewkowicz=false) {
    T_eta f, ones;
    ones = 1.0;
    f = -1.0 * log(eta)/ kappa;
    if (lewkowicz) {
        f = f + (pi_coles/kappa) * coles_wake(ones, pi_coles)
              - (pi_coles/kappa) * coles_wake(eta, pi_coles);
    } else {
        f = f + (pow(eta, 3.0) - 1.0)/(3.0*kappa)
              + 2.0*pi_coles*(1.0 - 3.0*pow(eta, 2.0) + 2.0*pow(eta, 3.0))/kappa;
    }
    if (is_double_inf(f)) {
        f = shear_ratio;
    };
    return f;
}
// Remove template specialisation from doc (causes duplicate) @cond
template <typename T_param>
Eigen::ArrayXd deficit(const Eigen::ArrayXd &eta, const double kappa, T_param const &␣
↪pi_coles, const double shear_ratio, const bool lewkowicz=false) {
    Eigen::ArrayXd f;
    Eigen::ArrayXd ones;
    ones.setOnes(eta.size());
    f = -1.0 * log(eta)/ kappa;
    if (lewkowicz) {
        f = f + (pi_coles/kappa) * coles_wake(ones, pi_coles)
            - (pi_coles/kappa) * coles_wake(eta, pi_coles);
```

(continues on next page)

(continued from previous page)

```
    } else {
        f = f + (eta.pow(3.0) - 1.0)/(3.0*kappa)
            + 2.0*pi_coles*(1.0 - 3.0*eta.pow(2.) + 2.0*eta.pow(3.0))/kappa;
    }
    for (int k = 0; k < f.size(); k++) {
        if (std::isinf(f[k])) {
            f(k) = shear_ratio;
        }
    }
    return f;
}
// @endcond


template <typename T_z, typename T_param>
T_z lewkowicz_speed(T_z const & z, T_param const & pi_coles, T_param const & kappa, T_
↪param const & u_inf, T_param const & shear_ratio, T_param const &delta_c) {
    T_z f, speed, eta;
    eta = z / delta_c;
    T_param u_tau = u_inf / shear_ratio;
    T_z term1 = log(eta) / (-1.0*kappa);
    T_z term2 = pi_coles * coles_wake(T_z(1.0), pi_coles, true) / kappa;
    T_z term3 = pi_coles * coles_wake(eta, pi_coles, true) / kappa;
    f = term1 + term2 - term3;
    // TODO why isn't this set directly to shear_ratio?
    if (is_double_inf(f)) {
        f = u_inf / u_tau;
    };
    speed = u_inf - f * u_tau;
    return speed;
};

// Remove template specialisation from doc (causes duplicate) @cond
template <typename T_param>
Eigen::VectorXd lewkowicz_speed(Eigen::VectorXd const & z, T_param const &pi_coles, T_
↪param const &kappa, T_param const &u_inf, T_param const &shear_ratio, T_param const␣
↪&delta_c=1.0){
    Eigen::VectorXd f, speed, eta;
    eta = z.array() / delta_c;
    T_param u_tau = u_inf/shear_ratio;
    Eigen::VectorXd term1 = eta.array().log() / (-1.0*kappa);
    double term2 = pi_coles * coles_wake(1.0, pi_coles) / kappa;
    Eigen::VectorXd term3 = pi_coles * coles_wake(eta, pi_coles).array() / kappa;
    f = term1.array() + term2 - term3.array();
    for (int k = 0; k < f.size(); k++) {
        if (std::isinf(f[k])) {
            f(k) = u_inf / u_tau;
        }
    }
    speed = u_inf - f.array() * u_tau;
    return speed;
};

template <typename T_param>
Eigen::ArrayXd lewkowicz_speed(Eigen::ArrayXd const & z, T_param const &pi_coles, T_
↪param const &kappa, T_param const &u_inf, T_param const &shear_ratio, T_param const␣
↪&delta_c=1.0){
```

(continues on next page)

(continued from previous page)

```
    Eigen::ArrayXd f, speed, eta;
    eta = z / delta_c;
    T_param u_tau = u_inf/shear_ratio;
    Eigen::ArrayXd term1 = eta.log() / (-1.0*kappa);
    double term2 = pi_coles * coles_wake(1.0, pi_coles) / kappa;
    Eigen::ArrayXd term3 = pi_coles * coles_wake(eta, pi_coles) / kappa;
    f = term1 + term2 - term3;
    for (int k = 0; k < f.size(); k++) {
        if (std::isinf(f[k])) {
            f(k) = u_inf / u_tau;
        }
    }
    speed = u_inf - f * u_tau;
    return speed;
};
// @endcond


} /* namespace es */


#endif /* ES_FLOW_VELOCITY_H_ */
```

Includes

- `Eigen/Core`
- `Eigen/Dense`
- `iostream`
- `profile.h` (*File profile.h*)

Included By

- *File adem.h*
- *File signature.h*
- *File fit.h*
- *File stress.h*

Namespaces

- *Namespace es*

Functions

- *Template Function es::coles_wake*
- *Template Function es::deficit*
- *Template Function es::lewkowicz_speed*

- *Template Function es::marusic_jones_speed*
- *Template Function es::most_law_speed*
- *Template Function es::power_law_speed*

Bibliography

[Agarwal] S. Agarwal, N. Snavely, S. M. Seitz and R. Szeliski, **Bundle Adjustment in the Large**, *Proceedings of the European Conference on Computer Vision*, pp. 29–42, 2010.

[Bjorck] A. Bjorck, **Numerical Methods for Least Squares Problems**, SIAM, 1996

[Brown] D. C. Brown, **A solution to the general problem of multiple station analytical stereo triangulation**, Technical Report 43, Patrick Airforce Base, Florida, 1958.

[ByrdNocedal] R. H. Byrd, J. Nocedal, R. B. Schanbel, **Representations of Quasi-Newton Matrices and their use in Limited Memory Methods**, *Mathematical Programming* 63(4):129–-156, 1994.

[ByrdSchnabel] R.H. Byrd, R.B. Schnabel, and G.A. Shultz, **Approximate solution of the trust region problem by minimization over two dimensional subspaces**, *Mathematical programming*, 40(1):247–263, 1988.

[Chen] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, **Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate**, *TOMS*, 35(3), 2008.

[Conn] A.R. Conn, N.I.M. Gould, and P.L. Toint, **Trust region methods**, *Society for Industrial Mathematics*, 2000.

[GolubPereyra] G.H. Golub and V. Pereyra, **The differentiation of pseudo-inverses and nonlinear least squares problems whose variables separate**, *SIAM Journal on numerical analysis*, 10(2):413–432, 1973.

[HartleyZisserman] R.I. Hartley & A. Zisserman, **Multiview Geometry in Computer Vision**, Cambridge University Press, 2004.

[KanataniMorris] K. Kanatani and D. D. Morris, **Gauges and gauge transformations for uncertainty description of geometric structure with indeterminacy**, *IEEE Transactions on Information Theory* 47(5):2017-2028, 2001.

[Keys] R. G. Keys, **Cubic convolution interpolation for digital image processing**, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 29(6), 1981.

[KushalAgarwal] A. Kushal and S. Agarwal, **Visibility based preconditioning for bundle adjustment**, *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[Kanzow] C. Kanzow, N. Yamashita and M. Fukushima, **Levenberg–Marquardt methods with strong local convergence properties for solving nonlinear equations with convex constraints**, *Journal of Computational and Applied Mathematics*, 177(2):375–397, 2005.

[Levenberg] K. Levenberg, **A method for the solution of certain nonlinear problems in least squares**, *Quart. Appl. Math*, 2(2):164–168, 1944.

[LiSaad] Na Li and Y. Saad, **MIQR: A multilevel incomplete qr preconditioner for large sparse least squares problems**, *SIAM Journal on Matrix Analysis and Applications*, 28(2):524–550, 2007.

[Madsen] K. Madsen, H.B. Nielsen, and O. Tingleff, **Methods for nonlinear least squares problems**, 2004.

[Mandel] J. Mandel, **On block diagonal and Schur complement preconditioning**, *Numer. Math.*, 58(1):79–93, 1990.

[Marquardt] D.W. Marquardt, **An algorithm for least squares estimation of nonlinear parameters**, *J. SIAM*, 11(2):431–441, 1963.

[Mathew] T.P.A. Mathew, **Domain decomposition methods for the numerical solution of partial differential equations**, Springer Verlag, 2008.

[NashSofer] S.G. Nash and A. Sofer, **Assessing a search direction within a truncated newton method**, *Operations Research Letters*, 9(4):219–221, 1990.

[Nocedal] J. Nocedal, **Updating Quasi-Newton Matrices with Limited Storage**, *Mathematics of Computation*, 35(151): 773–782, 1980.

[NocedalWright] J. Nocedal & S. Wright, **Numerical Optimization**, Springer, 2004.

[Oren] S. S. Oren, **Self-scaling Variable Metric (SSVM) Algorithms Part II: Implementation and Experiments**, Management Science, 20(5), 863-874, 1974.

[Ridders] C. J. F. Ridders, **Accurate computation of F'(x) and F'(x) F''(x)**, Advances in Engineering Software 4(2), 75-76, 1978.

[RuheWedin] A. Ruhe and P.A. Wedin, **Algorithms for separable nonlinear least squares problems**, Siam Review, 22(3):318–337, 1980.

[Saad] Y. Saad, **Iterative methods for sparse linear systems**, SIAM, 2003.

[Stigler] S. M. Stigler, **Gauss and the invention of least squares**, *The Annals of Statistics*, 9(3):465-474, 1981.

[TenenbaumDirector] J. Tenenbaum & B. Director, **How Gauss Determined the Orbit of Ceres**.

[TrefethenBau] L.N. Trefethen and D. Bau, **Numerical Linear Algebra**, SIAM, 1997.

[Triggs] B. Triggs, P. F. Mclauchlan, R. I. Hartley & A. W. Fitzgibbon, **Bundle Adjustment: A Modern Synthesis**, Proceedings of the International Workshop on Vision Algorithms: Theory and Practice, pp. 298-372, 1999.

[Wiberg] T. Wiberg, **Computation of principal components when data are missing**, In Proc. *Second Symp. Computational Statistics*, pages 229–236, 1976.

[WrightHolt] S. J. Wright and J. N. Holt, **An Inexact Levenberg Marquardt Method for Large Sparse Nonlinear Least Squares**, *Journal of the Australian Mathematical Society Series B*, 26(4):387–403, 1985.

Index

A

C

E

F

I

K

O

P

S

T

U