CurrentStatus

February 8, 2022

Hide the code cells so the notebook remains readable Hidden here is a code snippet that hides all code cells. There are some problems with markdown cells, they seem to disapear after beeing run if one tries to edit them. It was taken from https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ipython-notebook-visualized-with-nbviewer

[1]:
```
%%HTML
<script>
    function luc21893_refresh_cell(cell) {
        if( cell.luc21893 ) return;
        cell.luc21893 = true;
        console.debug('New code cell found...' );

        var div = document.createElement('DIV');
        cell.parentNode.insertBefore( div, cell.nextSibling );
        div.style.textAlign = 'right';
        var a = document.createElement('A');
        div.appendChild(a);
        a.href='#'
        a.luc21893 = cell;
        a.setAttribute( 'onclick', "luc21893_toggle(this); return false;" );

        cell.style.visibility='hidden';
        cell.style.position='absolute';
        a.innerHTML = '[show code]';

    }
    function luc21893_refresh() {
        if( document.querySelector('.code_cell .input') == null ) {
            // it apeears that I am in a exported html
            // hide this code
            var codeCells = document.querySelectorAll('.jp-InputArea')
            codeCells[0].style.visibility = 'hidden';
            codeCells[0].style.position = 'absolute';
            for( var i = 1; i < codeCells.length; i++ ) {
                luc21893_refresh_cell(codeCells[i].parentNode)
            }
            window.onload = luc21893_refresh;
```

```
        }
        else {
            // it apperas that I am in a jupyter editor
            var codeCells = document.querySelectorAll('.code_cell .input')
            for( var i = 0; i < codeCells.length; i++ ) {
                luc21893_refresh_cell(codeCells[i])
            }
            window.setTimeout( luc21893_refresh, 1000 )
        }
    }

    function luc21893_toggle(a) {
        if( a.luc21893.style.visibility=='hidden' ) {
            a.luc21893.style.visibility='visible';
            a.luc21893.style.position='';
            a.innerHTML = '[hide code]';
        }
        else {
            a.luc21893.style.visibility='hidden';
            a.luc21893.style.position='absolute';
            a.innerHTML = '[show code]';
        }
    }

    luc21893_refresh()
</script>
```

<IPython.core.display.HTML object>

[2]:
```
from astropy.io import fits
from pyirf.interpolation import interpolate_energy_dispersion
from pyirf.binning import bin_center
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from mpl_toolkits.axes_grid1 import ImageGrid
from scipy.stats import norm, skewnorm, genpareto
from scipy.spatial import Delaunay
from scipy.interpolate import interp1d, griddata

from glob import glob
import re
import warnings
```

1 Helper Functions

[3]:
```
def find_simplex(grid_points, target_point):
    triangular_grid = Delaunay(grid_points)
    target_simplex = triangular_grid.find_simplex(target_point)
    if target_simplex == -1:
        raise ValueError("The target point lies outside the specified grid.")

    target_simplex_indices = triangular_grid.simplices[target_simplex]

    return target_simplex_indices
```

2 Read the Grid IRF subset

[4]:
```
# Path for the grid IRFs
grid_irfs = glob('Data/IRF_Grid/*')

# Grid Points
grid_points = []
for path in grid_irfs:
    grid_points.append(np.array([int(s) for s in re.findall(r'\d+', path)[1:]]))
grid_points = np.array(grid_points)
grid_points = np.delete(grid_points, 3, axis=0).astype('float')
target = np.array([1/np.cos(31*np.pi/180), 90])

# Use 1/cos
grid_points[:, 0] = 1/np.cos(grid_points[:, 0]*np.pi/180)

# Get the energy dispersions
grid_edisps = []
for path in grid_irfs:
    with fits.open(path) as hdul:
            data = hdul[2].data
    Migration = data['MATRIX'][0]
    grid_edisps.append(Migration)
grid_edisps = np.array(grid_edisps)

# Cache the wanted edisp and delete it from the input templates
target_edisp = grid_edisps[3]

grid_edisps = np.delete(grid_edisps, 3, axis=0)

# Bin edges
with fits.open(grid_irfs[0]) as hdul:
    data = hdul[2].data
    hdr = hdul[2].header
```

```
TrueEnerg = data['ENERG_LO'][0]
TrueEnerg = np.append(TrueEnerg, data['ENERG_HI'][0][-1])

MigraEnerg = data['MIGRA_LO'][0]
MigraEnerg = np.append(MigraEnerg, data['MIGRA_HI'][0][-1])

Theta = data['THETA_LO'][0]
Theta = np.append(Theta, data['THETA_HI'][0][-1])

Migration = data['MATRIX'][0]
```

3 Use the current implementation of the PR

[5]:
```
def lookup(x, len_hist):
    """
    Function to look up the bin-value corresponding to a bin number.

    Parameters
    ----------
    x: numpy.ndarray, shape=(M+L)
    Concatenated array in which the first M values are the M bin-values for a␣
↪histogram consisting of M bins and
    the following L values are L bin-numbers, where the histogram value should␣
↪be looked up.

    len_hist: int
    Number (M) of histogram bins

    Returns
    -------
    looked_up_values: numpy.ndarray, shape=(M)
    Looked up bin-values
    """
    hist = x[:len_hist]
    binnr = x[len_hist:]

    mask = (binnr == 0) | (binnr == len_hist + 1)

    looked_up_values = np.zeros(len(x) - len_hist)
    looked_up_values[~mask] = hist[(binnr[~mask] - 1).astype(int)]

    return looked_up_values
```

[6]:
```
def numerical_quantile(pdf, mids, percentile):
    """
    Approximative quantile computation for histograms
```

```
    Parameters
    ----------
    pdf: numpy.ndarray, shape=(2)
    Input histogram

    mids: numpy.ndarray, shape=(M)
    Bin mids for the input histogram

    percentile: numpy.ndarray, shape=(L)
    Percentages where the quantile is needed

    Returns
    -------
    quantiles: numpy.ndarray, shape=(L)
    Computed quantiles

    """
    if np.sum(pdf) == 0:
        return np.full(len(percentile), np.nan)

    # Compute cdf and interpolate for ppf, Normalize to one first, since the␣
↪sum of a histogram does not
    # need to be one
    cdf = np.cumsum(pdf)
    interp = interp1d(
        cdf / cdf.max(), mids, fill_value=(mids.min(), mids.max()),␣
↪bounds_error=False
    )

    # Lookup ppf value
    return interp(percentile)
```

[7]:
```
def rebin(entries, mids, width):
    """
    Sort L pdf-values into M bins by computing the mean of all entries falling␣
↪into a bin

    Parameters
    ----------
    entries: numpy.ndarray, shape=(2*L)
    Concatenated array in which the first L values are the x-values and
    the following L values pdf(x). This is purely done to make this function␣
↪usable with np.apply_along_axis.

    mids: numpy.ndarray, shape=(M)
    Bin mids for the desired histogram
```

```
    width: float
    Bin width for the desired histogram, all bins are assumed to have the same␣
↪bin-width

    Returns
    -------
    hist: numpy.ndarray, shape=(M)
    Histogram
    """

    # Decompose input
    x = entries[: int(len(entries) / 2)]
    y = entries[int(len(entries) / 2) :]

    # Put into bins, replace nans by 0 that arise in those bins where no␣
↪entries lie (np.mean([]) = np.nan)
    # Catch both warnings that arise when np.mean([]) is called, as this is␣
↪anticipated.

    with warnings.catch_warnings():
        warnings.filterwarnings(action="ignore", message="Mean of empty slice")
        warnings.filterwarnings(
            action="ignore", message="invalid value encountered in␣
↪double_scalars"
        )
        rebinned_histogram = [
            np.mean(y[(x >= low) & (x < up)])
            for low, up in zip(mids - width / 2, mids + width / 2)
        ]

    return np.nan_to_num(rebinned_histogram)
```

[8]:
```
def interp_hist_quantile(
    edges, hists, m, m_prime, axis, normalize, quantile_resolution=1e-3
):
    """
    Function that wraps up the quantile PDF interpolation procedure [1] adopted␣
↪for histograms.

    Parameters
    ----------
    edges: numpy.ndarray, shape=(M+1)
    Common array of bin-edges (along the abscissal ("x") axis) for the M bins␣
↪of the input templates

    hists: numpy.ndarray, shape=(2,...,M,...)
```

```
    Array of M bin-heights (along the ordinate ("y") axis) for each of the 2␣
↪input templates.
    The distributions to be interpolated (e.g. MigraEnerg for the IRFs Energy␣
↪Dispersion) is expected to
    be given at the dimension specified by axis.

    m: numpy.ndarray, shape=(2)
    Array of the 2 morphing parameter values corresponding to the 2 input␣
↪templates. The pdf's qunatiles
    are expected to vary linearly between these two reference points.

    m_prime: float
    Value for which the interpolation is performed (target point)

    axis: int
    Axis along which the pdfs used for interpolation are located

    normalize: string or None
    Mode of normalisation to account for the approximative nature of the␣
↪interpolation. "sum" normalizes
    the interpolated histogram to a sum of 1, "weighted_sum" to an integral of␣
↪1. None does not apply any
    normalization.

    quantile_resolution: float
    Interpolated quantile spacing, defaults to 1/1000

    Returns
    -------
    f_new: numpy.ndarray, shape=(...,M,...)
    Interpolated histograms

    References
    ----------
    .. [1] B. E. Hollister and A. T. Pang (2013). Interpolation of Non-Gaussian␣
↪Probability Distributions
           for Ensemble Visualization
           https://engineering.ucsc.edu/sites/default/files/technical-reports/
↪UCSC-SOE-13-13.pdf
    """
    # determine quantiles step
    percentages = np.arange(0, 1 + 0.5 * quantile_resolution,␣
↪quantile_resolution)
    mids = bin_center(edges)
```

```
    quantiles = np.apply_along_axis(numerical_quantile, axis, hists, mids,␣
↪percentages)

    # interpolate quantiles step
    # First: compute alpha from eq. (6), the euclidean norm between the two␣
↪grid points has to be one,
    # so normalize to ||m1-m0|| first
    dist = np.linalg.norm(m[1] - m[0])
    g = m / dist
    p = m_prime / dist
    alpha = np.linalg.norm(p - g[0])

    # Second: Interpolate quantiles as in eq. (10)
    q_bar = (1 - alpha) * quantiles[0] + alpha * quantiles[1]

    # evaluate interpolant PDF values step
    # The original PDF (only given as histogram) has to be re-evaluated at the␣
↪quantiles determined above
    binnr = np.digitize(quantiles, edges)
    helper = np.concatenate((hists, binnr), axis=axis)
    V = np.apply_along_axis(lookup, axis, helper, hists.shape[axis])

    # Compute the interpolated histogram at positions q_bar as in eq. (12), set␣
↪V_bar to
    # zero when both template PDFs are zero
    # V_bar = V[0] * V[1] / ((1 - alpha) * V[1] + alpha * V[0])
    a = V[0] * V[1]
    b = (1 - alpha) * V[1] + alpha * V[0]
    V_bar = np.divide(a, b, out=np.zeros_like(a), where=b != 0)

    # Create temporary axis to imitate the former shape, as one dimension was␣
↪lost through the interpolation and
    # therefore axis might not longer be correct
    q_bar = q_bar[np.newaxis, :]
    V_bar = V_bar[np.newaxis, :]

    # Shift interpolated pdf back into the original histogram bins as V_bar is␣
↪by construction given at
    # positions q_bar.
    width = np.diff(edges)
    helper = np.concatenate((q_bar, V_bar), axis=axis)
    interpolated_histogram = np.apply_along_axis(rebin, axis, helper, mids,␣
↪width)

    # Re-Normalize, as the normalisation is lost due to approximate nature of␣
↪this method
```

```python
    # Set norm to nan for empty histograms to avoid division through 0
    if normalize == "sum":
        norm = np.sum(interpolated_histogram, axis=axis)
        norm = np.repeat(
            np.expand_dims(norm, axis), interpolated_histogram.shape[axis],␣
↪axis=axis
        )
        norm[norm == 0] = np.nan
    elif normalize == "weighted_sum":
        norm = np.sum(interpolated_histogram * width, axis=axis)
        norm = np.repeat(
            np.expand_dims(norm, axis), interpolated_histogram.shape[axis],␣
↪axis=axis
        )
        norm[norm == 0] = np.nan
    elif normalize is None:
        norm = 1

    # Normalize and squeeze the temporary axis from the result
    return np.nan_to_num(interpolated_histogram / norm).squeeze()
```

[9]:
```python
def interpolate_energy_dispersion_original(
    bin_edges, energy_dispersions, grid_points, target_point, axis, normalize
):
    """
    Takes a grid of dispersion matrixes for a bunch of different parameters
    and interpolates it to given value of those parameters

    Parameters
    ----------
    bin_edges: np.ndarray
        bin edges of the energy migrations, have to be equidistant
    energy_dispersions: np.ndarray
        grid of energy migrations
    grid_points: np.ndarray
        array of parameters corresponding to energy_dispersions, of shape␣
↪(n_grid_points, n_interp_dim)
    target_point: np.ndarray
        values of parameters for which the interpolation is performed, of shape␣
↪(n_interp_dim)
    axis: int
        axis along which the energy-migration pdfs used for interpolation are␣
↪located
    normalize: str
        Mode of normalisation to account for the approximative nature of the␣
↪interpolation. "sum" normalizes
```

```
        the interpolated histogram to a sum of 1, "weighted_sum" to an integral␣
↪of 1. None does not apply any
        normalization.

    Returns
    -------
    matrix_interp: np.ndarray
        Interpolated dispersion matrix 3D array with shape (n_energy_bins,␣
↪n_migration_bins, n_fov_offset_bins)

    Raises
    ------
    ValueError if number of grid- and template-points is not matching

    ValueError if target_point is outside grid

    ValueError if grid dimension is > 2
    """
    # Test grid and energy_dispersions for matching dimensions
    if len(grid_points) != energy_dispersions.shape[0]:
        raise ValueError("Number of grid- and template-points not matching.")

    # If grid is nD with n>2: Raise Error.
    if (not np.isscalar(grid_points[0])) and (len(grid_points[0]) > 2):
        raise ValueError("Grid Dimension > 2")

    # If data is 1D: directly interpolate between next neighbors
    if np.isscalar(grid_points[0]):
        # Sort arrays to find the pair of next neighbors to the target_point
        sorting_indizes = np.argsort(grid_points)
        sorted_grid = grid_points[sorting_indizes]
        sorted_template = energy_dispersions[sorting_indizes]
        input_pos = np.digitize(target_point, sorted_grid)

        if (input_pos == len(grid_points)) or (input_pos == -1):
            raise ValueError("The target point lies outside the specified grid.
↪")

        neighbors = np.array([input_pos - 1, input_pos])

        # Get matching pdfs and interpolate
        grid_point_subset = sorted_grid[neighbors]

        template_subset = sorted_template[neighbors]
        return interp_hist_quantile(
            bin_edges, template_subset, grid_point_subset, target_point, axis,␣
↪normalize
```

```
        )

    # Else for 2D: Chain 1D interpolations to arive at the desired point

    # Find simplex (triangle in this 2D case) of grid-points in which the target point lies.
    # Use Delaunay-Transformation to costruct a triangular grid from grid_points.
    triangular_grid = Delaunay(grid_points)
    target_simplex = triangular_grid.find_simplex(target_point)
    if target_simplex == -1:
        raise ValueError("The target point lies outside the specified grid.")

    target_simplex_indices = triangular_grid.simplices[target_simplex]
    target_simplex_vertices = grid_points[target_simplex_indices]

    # Use the segment between the two vertices of the triangle that are closest to the target_point to construct
    # the intermediate point m_tilde. m_tilde will be the point where the line between
    # the most distant vertex and the target point crosses the remaining triangle side. This construction assures
    # minimal distance between m_tilde and the target point and should thus minimize interpolation error.
    distances = np.linalg.norm(target_point - target_simplex_vertices, axis=1)

    sorting_indizes = np.argsort(distances)
    sorted_vertices = target_simplex_vertices[sorting_indizes]
    sorted_indices = target_simplex_indices[sorting_indizes]

    # Construct m_tilde. This needs one to solve a problem of the form Ax = b
    A = np.array(
        [target_point - sorted_vertices[-1], -sorted_vertices[1] + sorted_vertices[0]]
    ).T
    b = sorted_vertices[0] - sorted_vertices[-1]
    m_tilde = sorted_vertices[-1] + np.linalg.solve(A, b)[0] * (
        target_point - sorted_vertices[-1]
    )

    # Interpolate to m_tilde
    template_subset = energy_dispersions[[sorted_indices[0], sorted_indices[1]], :]
    grid_point_subset = grid_points[[sorted_indices[0], sorted_indices[1]], :]
    interpolated_hist_tilde = interp_hist_quantile(
        bin_edges, template_subset, grid_point_subset, m_tilde, axis, normalize
```

```
    )

    # Interpolate to target_point, reshape as axes with length 1 are lost in␣
↪the computation and the shapes would not be matching
    template_subset = np.array(
        [
            interpolated_hist_tilde.reshape(energy_dispersions.shape[1:]),
            energy_dispersions[sorted_indices[-1]],
        ]
    )
    grid_point_subset = np.array([m_tilde, grid_points[sorted_indices[-1]]])

    # Reshape result as axes with length 1 are lost in the computation and the␣
↪shapes would not be matching the input shape
    return interp_hist_quantile(
        bin_edges, template_subset, grid_point_subset, target_point, axis,␣
↪normalize
    ).reshape(energy_dispersions.shape[1:])
```

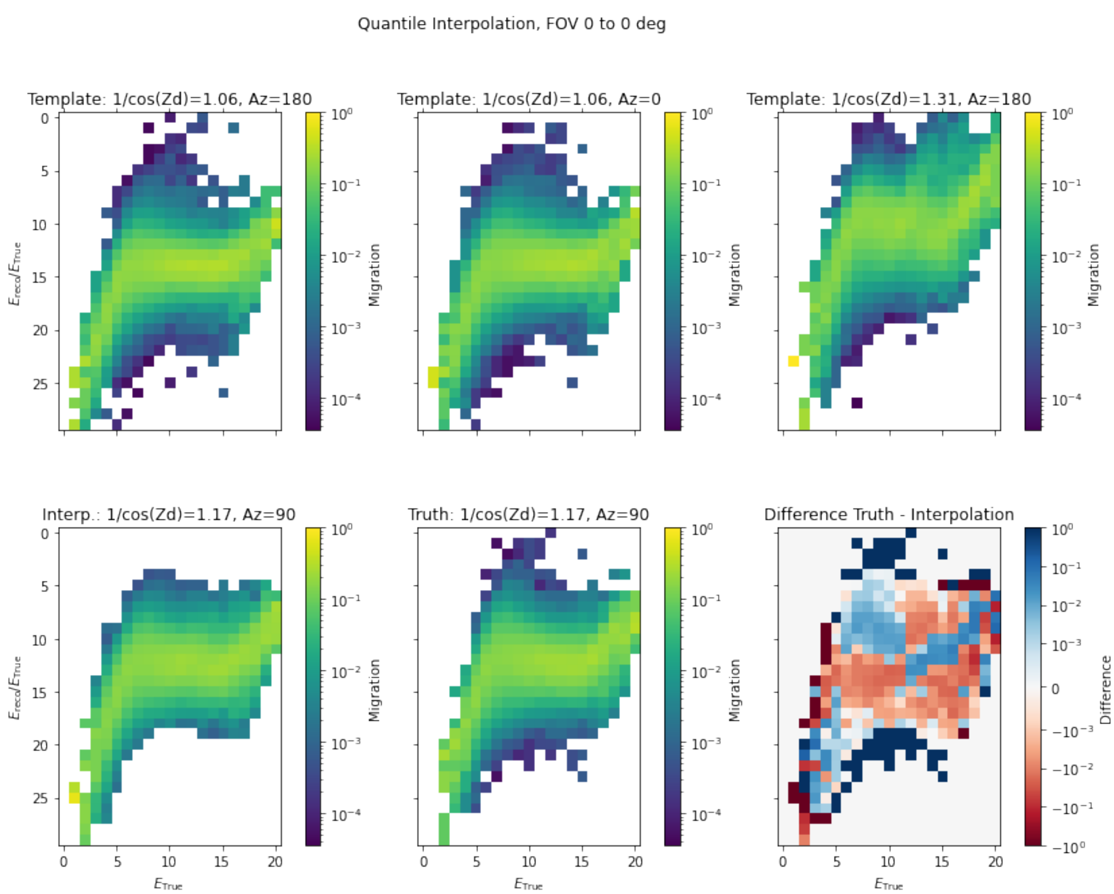
[10]:
```
grid_inter = interpolate_energy_dispersion_original(MigraEnerg,
                                                    grid_edisps,
                                                    grid_points,
                                                    target,
                                                    -2,
                                                    normalize = 'sum')
# Compare truth and interpolation, here by simple distance
dist =  np.nan_to_num((target_edisp.squeeze() - grid_inter.squeeze()))
original_res_sum = np.sum(np.abs(dist))

# Set certain scenerios to 1 (all information missing/non estimated) or -1␣
↪(information estimated that is not there)
#dist[(target_edisp.squeeze() == 0) & (grid_inter.squeeze() == 0)] = 0
dist[(target_edisp.squeeze() == 0) & (grid_inter.squeeze() != 0)] = -1
dist[(target_edisp.squeeze() != 0) & (grid_inter.squeeze() == 0)] = 1

target_matrices = np.array([grid_inter.squeeze(), target_edisp.squeeze(), dist])

original_result = grid_inter.squeeze()
original_dist = dist
```

[11]:
```
simplex_ind = find_simplex(grid_points, target)
matrices = np.array([grid_edisps[simplex_ind]]).squeeze()

#matrices = np.swapaxes(matrices, 0, 1).reshape(-1, *matrices.shape[-2:])
fig = plt.figure(figsize=(30, 10))
```

```
fig.suptitle(f"Quantile Interpolation, FOV {Theta[0]:.0f} to {Theta[1]:.0f}␣
↪deg")

grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,3),
                 axes_pad=1,
                 share_all=False,
                 cbar_location="right",
                 cbar_mode="each",
                 cbar_size="7%",
                 cbar_pad=0.25,
                 )

viridis = matplotlib.cm.get_cmap('viridis', 256)
newcolors = viridis(np.linspace(0, 1, 256))
white = np.array([1, 1, 1, 1])
newcolors[:1, :] = white
newcmp = ListedColormap(newcolors)


print(matrices.shape)
for ax, matrix in zip([grid[0], grid[1], grid[2]], matrices):
    mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
↪max()))
    #mat = ax.matshow(matrix, cmap="viridis", vmin=0, vmax=1)
    ax.cax.colorbar(mat).solids.set_edgecolor("face")
    ax.cax.set_ylabel('Migration')
    ax.cax.toggle_label(True)

for ax, matrix in zip([grid[3], grid[4], grid[5]], target_matrices):
    if ax != grid[5]:
        mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
↪max()))
        ax.cax.colorbar(mat).solids.set_edgecolor("face")
        ax.cax.set_ylabel('Migration')
        ax.cax.toggle_label(True)
    else:
        mat = ax.matshow(matrix, cmap='RdBu', norm=matplotlib.colors.
↪SymLogNorm(linthresh=1e-3))
        ax.cax.colorbar(mat).solids.set_edgecolor("face")
        ax.cax.set_ylabel('Difference')
        ax.cax.toggle_label(True)

grid[0].set_title(fr"Template: 1/cos(Zd)={grid_points[simplex_ind[0]][0]:.2f},␣
↪Az={grid_points[simplex_ind[0]][1]:.0f}")
```

```
grid[1].set_title(fr"Template: 1/cos(Zd)={grid_points[simplex_ind[1]][0]:.2f},␣
↪Az={grid_points[simplex_ind[1]][1]:.0f}")
grid[2].set_title(fr"Template: 1/cos(Zd)={grid_points[simplex_ind[2]][0]:.2f},␣
↪Az={grid_points[simplex_ind[2]][1]:.0f}")

grid[3].set_title(fr"Interp.: 1/cos(Zd)={target[0]:.2f}, Az={target[1]:.0f}")
grid[4].set_title(fr"Truth: 1/cos(Zd)={target[0]:.2f}, Az={target[1]:.0f}")
grid[5].set_title(f"Difference Truth - Interpolation")

grid[-1].xaxis.set_ticks_position('bottom')
grid[-2].xaxis.set_ticks_position('bottom')
grid[-3].xaxis.set_ticks_position('bottom')

grid[-1].set_xlabel(r"$E_{\mathrm{True}}$")
grid[-2].set_xlabel(r"$E_{\mathrm{True}}$")
grid[-3].set_xlabel(r"$E_{\mathrm{True}}$")

grid[0].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")
grid[3].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")
plt.savefig('2D_Interp_Real_Data.png', bbox_inches='tight');
```

```
(3, 30, 21)
```



Red colors: Interpolation > Truth, Blue colors: Interpolation < Truth

Values of 1 indicate bins where all information is missing in the interpolation and the true IRF contains information. Values of -1 indicate bins where information is estimated yet the true IRF does not contain any.

Pro: Mosty small difference between Truth and Interpolation

Contra: Biased results for low true energies, overestimates tails and underestimates peak for high true energies, Does not really scale beyond 2D hidden variables as a Deleauny Triagulation has to be performed to find the simplex in which the target point lies.

4 Quantile Interpolation approach that scales better to nD

[12]:
```
'''
Basic idea: Hollister and Pang [1] give a quick wrapup of the steps needed to␣
↪interpolate via their method.
We follow these steps losely but use existing implementations for e.g.␣
↪interpolation in nD.
'''
def interpolate_pdf(edges, pdfs, m, mprime, axis):
    pdfs = np.swapaxes(pdfs, axis, -1)
    cdfs = np.cumsum(pdfs, axis=-1)

    cdfs = cdfs/np.expand_dims(np.max(cdfs, axis=-1), axis=-1)

    quantiles = np.linspace(0, 1, 1000)

    bin_mids = bin_center(edges)

    # create ppf values from cdf samples via interpolation of the cdfs,␣
↪determine quantile steps of [1]
    ppfs_resampled = np.apply_along_axis(lambda cdf: interp1d(cdf, bin_mids,␣
↪bounds_error=False,
                                                                             ␣
↪fill_value='extrapolate')(quantiles),
                                         -1, cdfs)

    # nD interpolation of ppf values, interpolate quantiles step of [1]
    ppf_interpolant = griddata(m, ppfs_resampled, mprime)

    # recalculate pdf values, evaluate interpolant PDF values step of [1]
    pdf_interpolant = np.diff(quantiles)/np.diff(ppf_interpolant, axis=-1)
```

```
    # Unconventional solution to make this usable with np.apply_along_axis for␣
↪readability
    xyconcat = np.concatenate((ppf_interpolant[...,:-1] + np.
↪diff(ppf_interpolant),
                               np.nan_to_num(pdf_interpolant)), axis=-1)

    # Interpolate pdf samples and evaluate at bin edges, weight with the␣
↪bin_width to estimate
    # correct bin height via the midpoint rule formulation of the trapezoidal␣
↪rule
    result = np.apply_along_axis(lambda xy: np.diff(edges)*np.
↪nan_to_num(interp1d(xy[:int(len(xy)/2)],
                                                                             ␣
↪ xy[int(len(xy)/2):],
                                                                             ␣
↪ bounds_error=False,
                                                                             ␣
↪ fill_value=(0,0))(bin_mids)),
                                                                        -1,␣
↪xyconcat)

    # Renormalize histogram to a sum of 1
    norm = np.sum(result, axis=-1)
    norm = np.repeat(
        np.expand_dims(norm, -1), result.shape[-1], axis=-1
    )
    norm[norm == 0] = np.nan

    return np.swapaxes(np.nan_to_num(result / norm).squeeze(), axis, -1)
```

[13]:

```
grid_inter = interpolate_pdf(MigraEnerg, grid_edisps, grid_points, target, -2)

dist =  np.nan_to_num((target_edisp.squeeze() - grid_inter.squeeze()))
new_res_sum = np.sum(np.abs(dist))

dist[(target_edisp.squeeze() == 0) & (grid_inter.squeeze() == 0)] = 0
dist[(target_edisp.squeeze() == 0) & (grid_inter.squeeze() != 0)] = -1
dist[(target_edisp.squeeze() != 0) & (grid_inter.squeeze() == 0)] = 1

target_matrices = np.array([grid_inter.squeeze(), target_edisp.squeeze(),␣
↪dist]);

new_result = grid_inter.squeeze()
new_dist = dist
```

```
/tmp/ipykernel_341115/1710923620.py:9: RuntimeWarning: invalid value encountered
in true_divide
```

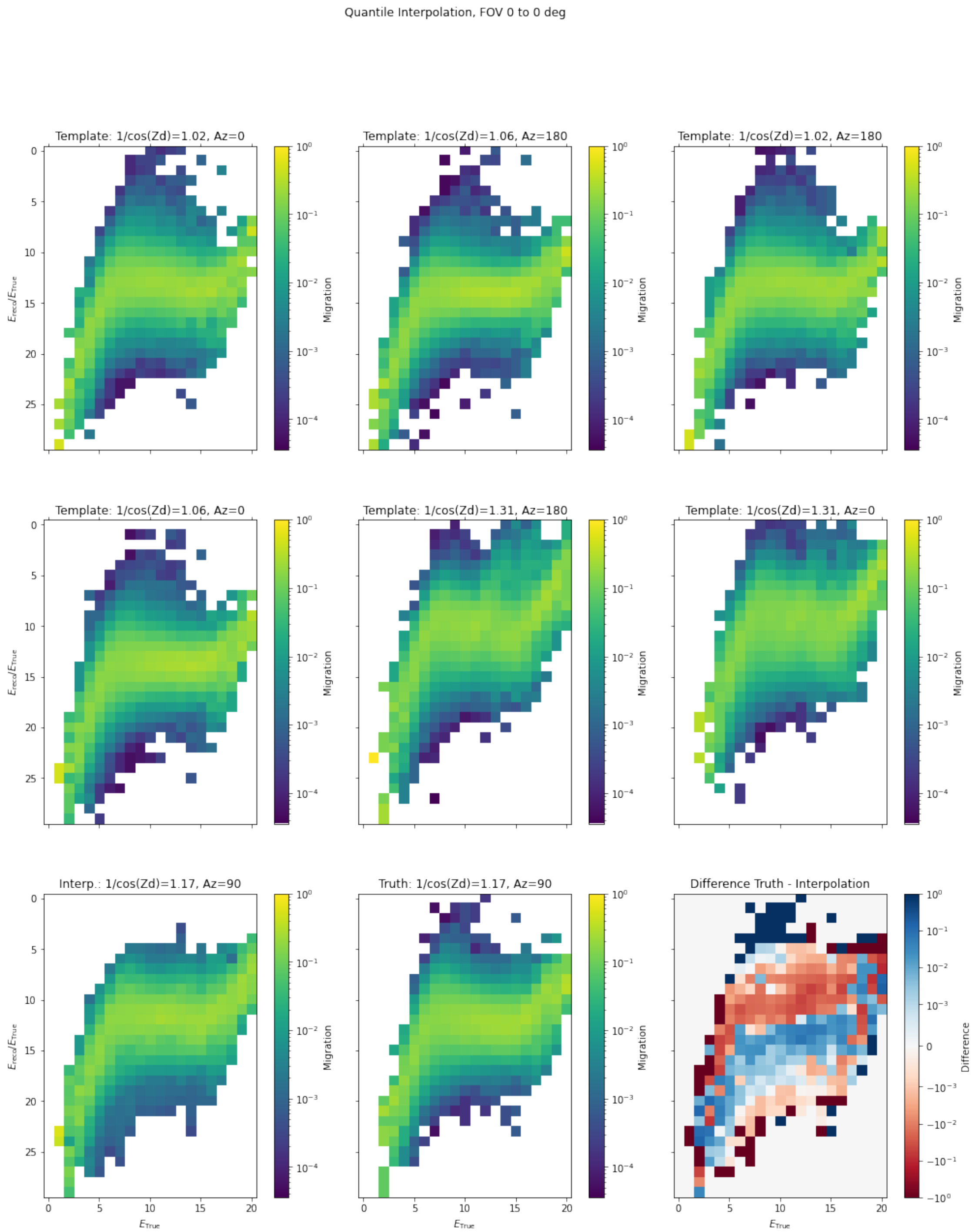
```
  cdfs = cdfs/np.expand_dims(np.max(cdfs, axis=-1), axis=-1)
/home/runedominik/.local/miniconda3/envs/working/lib/python3.9/site-
packages/scipy/interpolate/interpolate.py:623: RuntimeWarning: divide by zero
encountered in true_divide
  slope = (y_hi - y_lo) / (x_hi - x_lo)[:, None]
/home/runedominik/.local/miniconda3/envs/working/lib/python3.9/site-
packages/scipy/interpolate/interpolate.py:626: RuntimeWarning: invalid value
encountered in multiply
  y_new = slope*(x_new - x_lo)[:, None] + y_lo
/tmp/ipykernel_341115/1710923620.py:27: RuntimeWarning: invalid value
encountered in add
  xyconcat = np.concatenate((ppf_interpolant[…,:-1] +
np.diff(ppf_interpolant),
```

[14]:
```
matrices1 = np.array([grid_edisps[:3]]).squeeze()
matrices2 = np.array([grid_edisps[3:]]).squeeze()

#matrices = np.swapaxes(matrices, 0, 1).reshape(-1, *matrices.shape[-2:])
fig = plt.figure(figsize=(30, 20))

fig.suptitle(f"Quantile Interpolation, FOV {Theta[0]:.0f} to {Theta[1]:.0f}␣
↪deg")

grid = ImageGrid(fig, 111,
                 nrows_ncols=(3,3),
                 axes_pad=1,
                 share_all=False,
                 cbar_location="right",
                 cbar_mode="each",
                 cbar_size="7%",
                 cbar_pad=0.25,
                 )

viridis = matplotlib.cm.get_cmap('viridis', 256)
newcolors = viridis(np.linspace(0, 1, 256))
white = np.array([1, 1, 1, 1])
newcolors[:1, :] = white
newcmp = ListedColormap(newcolors)


print(matrices.shape)
for ax, matrix in zip([grid[0], grid[1], grid[2]], matrices1):
    mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
↪max()))
    #mat = ax.matshow(matrix, cmap="viridis", vmin=0, vmax=1)
    ax.cax.colorbar(mat).solids.set_edgecolor("face")
```

```
    ax.cax.set_ylabel('Migration')
    ax.cax.toggle_label(True)

for ax, matrix in zip([grid[3], grid[4], grid[5]], matrices2):
    mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
 ↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
 ↪max()))
    #mat = ax.matshow(matrix, cmap="viridis", vmin=0, vmax=1)
    ax.cax.colorbar(mat).solids.set_edgecolor("face")
    ax.cax.set_ylabel('Migration')
    ax.cax.toggle_label(True)

for ax, matrix in zip([grid[6], grid[7], grid[8]], target_matrices):
    if ax != grid[8]:
        mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
 ↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
 ↪max()))
        ax.cax.colorbar(mat).solids.set_edgecolor("face")
        ax.cax.set_ylabel('Migration')
        ax.cax.toggle_label(True)
    else:
        mat = ax.matshow(matrix, cmap='RdBu', norm=matplotlib.colors.
 ↪SymLogNorm(linthresh=1e-3))
        ax.cax.colorbar(mat).solids.set_edgecolor("face")
        ax.cax.set_ylabel('Difference')
        ax.cax.toggle_label(True)

grid[0].set_title(fr"Template: 1/cos(Zd)={grid_points[0][0]:.2f},␣
 ↪Az={grid_points[0][1]:.0f}")
grid[1].set_title(fr"Template: 1/cos(Zd)={grid_points[1][0]:.2f},␣
 ↪Az={grid_points[1][1]:.0f}")
grid[2].set_title(fr"Template: 1/cos(Zd)={grid_points[2][0]:.2f},␣
 ↪Az={grid_points[2][1]:.0f}")
grid[3].set_title(fr"Template: 1/cos(Zd)={grid_points[3][0]:.2f},␣
 ↪Az={grid_points[3][1]:.0f}")
grid[4].set_title(fr"Template: 1/cos(Zd)={grid_points[4][0]:.2f},␣
 ↪Az={grid_points[4][1]:.0f}")
grid[5].set_title(fr"Template: 1/cos(Zd)={grid_points[5][0]:.2f},␣
 ↪Az={grid_points[5][1]:.0f}")

grid[6].set_title(fr"Interp.: 1/cos(Zd)={target[0]:.2f}, Az={target[1]:.0f}")
grid[7].set_title(fr"Truth: 1/cos(Zd)={target[0]:.2f}, Az={target[1]:.0f}")
grid[8].set_title(f"Difference Truth - Interpolation")

grid[-1].xaxis.set_ticks_position('bottom')
grid[-2].xaxis.set_ticks_position('bottom')
```

```
grid[-3].xaxis.set_ticks_position('bottom')

grid[-1].set_xlabel(r"$E_{\mathrm{True}}$")
grid[-2].set_xlabel(r"$E_{\mathrm{True}}$")
grid[-3].set_xlabel(r"$E_{\mathrm{True}}$")

grid[0].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")
grid[3].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")
grid[6].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")

plt.savefig('2D_Interp_Real_Data.png', bbox_inches='tight');
```

```
(3, 30, 21)
```



Pro: Scales easily beyond 2D hidden variables, can pass all templates at once without any need to find a simplex.

Contra: Clear Bias (in this example towards higher reco-bin numbers) over all bins of true energy visible.

5 Comparison of both approaches

Compare the sum of absolute residuals

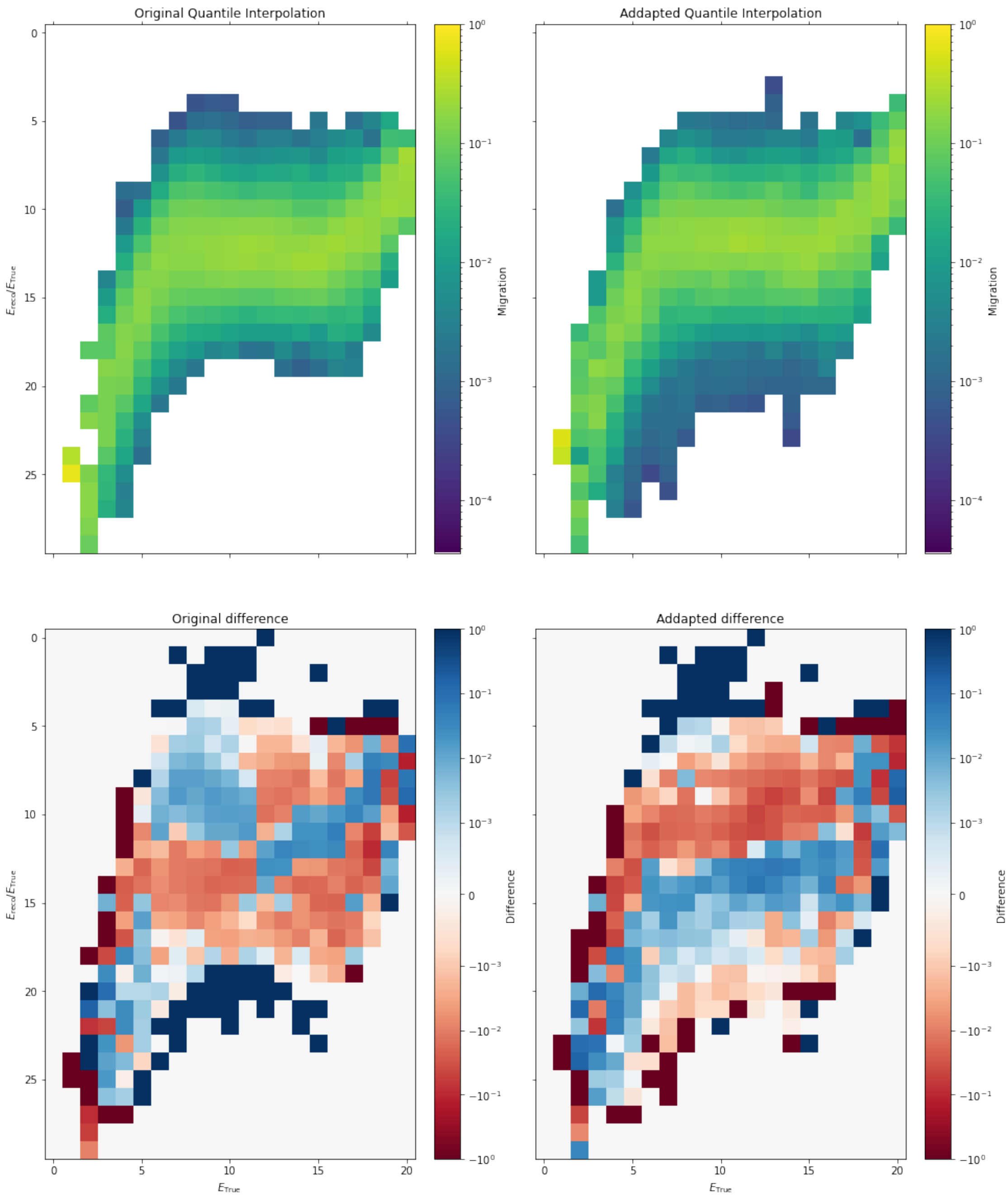
[15]:
```
print("Original Quantile Interpolation Method: ", original_res_sum, "\n"
      "Addapted Quantile Interpolation Method: ", new_res_sum)
```

```
Original Quantile Interpolation Method:  5.480000967160334
Addapted Quantile Interpolation Method:  6.333022842617732
```

[16]:
```
matrices1 = np.array([original_result, new_result]).squeeze()
matrices2 = np.array([original_dist, new_dist]).squeeze()

#matrices = np.swapaxes(matrices, 0, 1).reshape(-1, *matrices.shape[-2:])
fig = plt.figure(figsize=(20, 20))

grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 cbar_location="right",
                 cbar_mode="each",
                 cbar_size="7%",
                 cbar_pad=0.25,
                 )

viridis = matplotlib.cm.get_cmap('viridis', 256)
newcolors = viridis(np.linspace(0, 1, 256))
white = np.array([1, 1, 1, 1])
newcolors[:1, :] = white
newcmp = ListedColormap(newcolors)


print(matrices.shape)
for ax, matrix in zip([grid[0], grid[1]], matrices1):
    mat = ax.matshow(matrix, cmap=newcmp, norm=matplotlib.colors.
 ↪LogNorm(vmin=matrices[(matrices>0)].min(), vmax=matrices[(matrices>0)].
 ↪max()))
    #mat = ax.matshow(matrix, cmap="viridis", vmin=0, vmax=1)
    ax.cax.colorbar(mat).solids.set_edgecolor("face")
    ax.cax.set_ylabel('Migration')
    ax.cax.toggle_label(True)

for ax, matrix in zip([grid[2], grid[3]], matrices2):
    mat = ax.matshow(matrix, cmap='RdBu', norm=matplotlib.colors.
 ↪SymLogNorm(linthresh=1e-3, vmax=1, vmin=-1))
    ax.cax.colorbar(mat).solids.set_edgecolor("face")
    ax.cax.set_ylabel('Difference')
```

```
    ax.cax.toggle_label(True)


grid[0].set_title(fr"Original Quantile Interpolation")
grid[1].set_title(fr"Addapted Quantile Interpolation")
grid[2].set_title(fr"Original difference")
grid[3].set_title(fr"Addapted difference")

grid[-1].xaxis.set_ticks_position('bottom')
grid[-2].xaxis.set_ticks_position('bottom')

grid[-1].set_xlabel(r"$E_{\mathrm{True}}$")
grid[-2].set_xlabel(r"$E_{\mathrm{True}}$")

grid[0].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")
grid[2].set_ylabel(r"$E_{\mathrm{reco}} / E_{\mathrm{True}}$")

plt.savefig('Comp.png', bbox_inches='tight');
```

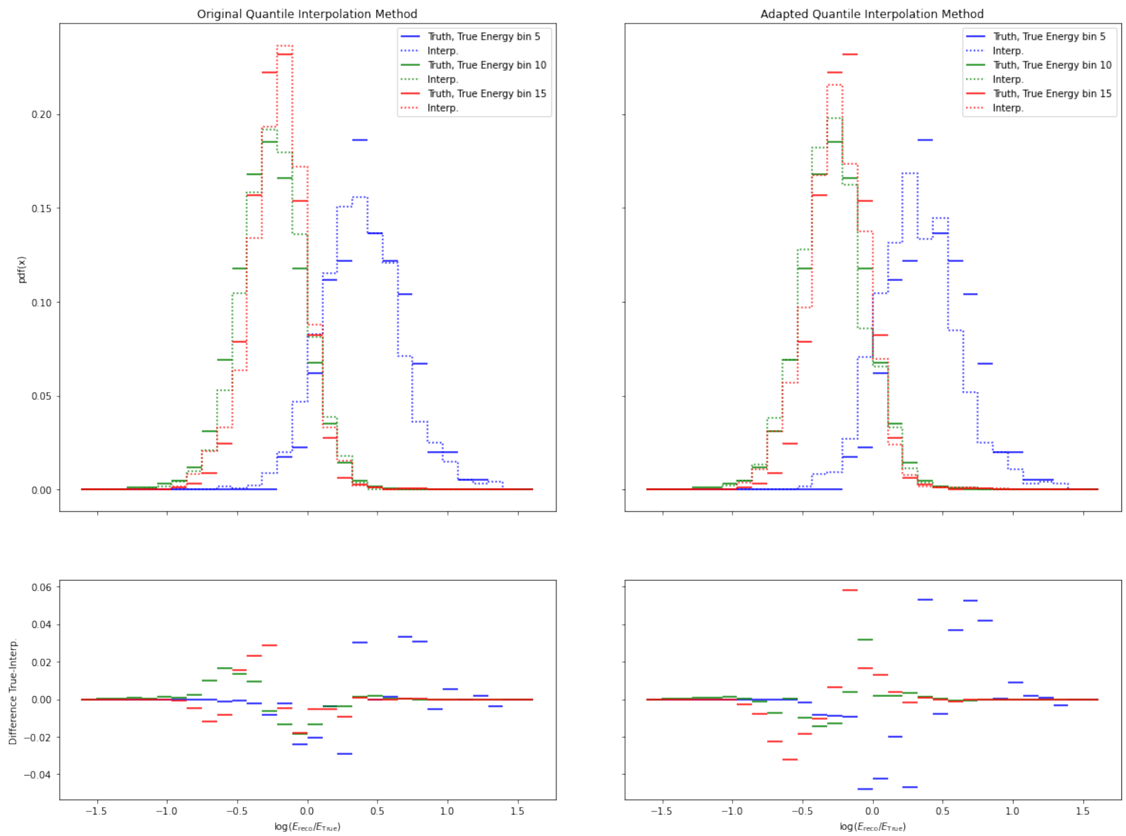
```
(3, 30, 21)
```



[17]:
```
fig = plt.figure(figsize=(20, 15))

grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
```

```
                 )
for i, c in zip([4, 9, 14], ['blue', 'green', 'red']):
    grid[0].hlines(target_edisp.squeeze()[:, i], xmin=np.log(MigraEnerg[:-1]),␣
 ↪xmax=np.log(MigraEnerg[1:]), color=c,
               label=rf"Truth, True Energy bin {i+1}")
    grid[0].step(np.log(MigraEnerg[:-1]), original_result[:, i], color=c,␣
 ↪where='post', linestyle='dotted',
            label=rf"Interp.")

    grid[1].hlines(target_edisp.squeeze()[:, i], xmin=np.log(MigraEnerg[:-1]),␣
 ↪xmax=np.log(MigraEnerg[1:]), color=c,
               label=rf"Truth, True Energy bin {i+1}")
    grid[1].step(np.log(MigraEnerg[:-1]), new_result[:, i], color=c,␣
 ↪where='post', linestyle='dotted',
            label=rf"Interp.")

    grid[2].hlines(target_edisp.squeeze()[:, i]-original_result[:, i],
                   xmin=np.log(MigraEnerg[:-1]), xmax=np.log(MigraEnerg[1:]),␣
 ↪color=c,
               label=rf"Truth-Interp., True Energy bin {i+1}")

    grid[3].hlines(target_edisp.squeeze()[:, i]-new_result[:, i],
                   xmin=np.log(MigraEnerg[:-1]), xmax=np.log(MigraEnerg[1:]),␣
 ↪color=c,
               label=rf"Truth-Interp., True Energy bin {i+1}")
grid[0].legend(loc='best')
grid[1].legend(loc='best')

grid[0].set_title("Original Quantile Interpolation Method")
grid[1].set_title("Adapted Quantile Interpolation Method")

grid[0].set_ylabel('pdf(x)')
grid[2].set_ylabel('Difference True-Interp.')
grid[2].set_xlabel('log($E_{\mathrm{reco}} / E_{\mathrm{True}}$)')
grid[3].set_xlabel('log($E_{\mathrm{reco}} / E_{\mathrm{True}}$)')
```

```
[17]: Text(0.5, 0, 'log($E_{\\mathrm{reco}} / E_{\\mathrm{True}}$)')
```



Results mainly comparable, bias and deviation stronger in new method. Result of trapezoidal rule usage? For a gaussian distribution this trapezoidal integration would overestimate the tails and underestimate the peak.

6 Testing on artificial test cases

In the following lines, both methods are tested interpolating (skewed) Gaussians between points $[1, 1]$, $[10, 1]$, $[1, 10]$, $[10, 10]$ to estimate the distribution at $[2, 5]$. The test-cases will vary the dependencies of the gaussian mean and std (as well as a skewed Gaussians skewness parameter) from the hidden variables m_0 and m_1.

The result-plots show mean and std from repeating the test-cases 100 times with new random variates drawn from the testing distributions.

[18]:
```
# Testcases

# Fix sigma and variate mu
def norm_mu_05(m):
    return norm(loc=m[0]**0.5, scale=1)

def norm_mu_1(m):
```

```
    return norm(loc=m[0]**1, scale=1)

def norm_mu_15(m):
    return norm(loc=m[0]**1.5, scale=1)

def norm_mu_2(m):
    return norm(loc=m[0]**2, scale=1)

# Fix mu and variate sigma
def norm_sig_05(m):
    return norm(loc=0, scale=0.25*m[1]**0.5)

def norm_sig_1(m):
    return norm(loc=0, scale=0.25*m[1]**1)

def norm_sig_15(m):
    return norm(loc=0, scale=0.25*m[1]**1.5)

def norm_sig_2(m):
    return norm(loc=0, scale=0.25*m[1]**2)

def norm_sig_1_05(m):
    return norm(loc=5, scale=0.5*m[1]**1)

def norm_sig_1_1(m):
    return norm(loc=5, scale=0.75*m[1]**1)

# Use skewed gaussian, fix mu and sigma, variate skewness parameter
def skewnorm_skew1(m):
    return skewnorm(loc=0, scale=1,
                    a=0.5*m[0]**1)

def skewnorm_skew2(m):
    return skewnorm(loc=0, scale=1,
                    a=0.5*(m[0])**2)

def skewnorm_skewneg1(m):
    return skewnorm(loc=0, scale=1,
                    a=-0.5*m[0]**1)

def skewnorm_skewneg2(m):
    return skewnorm(loc=0, scale=1,
                    a=-0.5*(m[0])**2)

# Variate mu and sigma

def norm_both(m):
```

```
    return norm(loc=m[0], scale=0.25*m[1])

def norm_mu(m):
    return norm(loc=m[0]*m[1], scale=0.25*m[1])

def norm_sig(m):
    return norm(loc=m[0], scale=0.25*m[0]*m[1])

def norm_mu_sig(m):
    return norm(loc=m[0]*m[1], scale=0.25*m[0]*m[1])

def norm_mu_add(m):
    return norm(loc=m[0]+m[1], scale=0.25*m[1])

def norm_mu_sig_add(m):
    return norm(loc=m[0]+m[1], scale=0.25*(m[0]+m[1]))

# Variate mu and sigma but with cos

def norm_both_cos(m):
    return norm(loc=np.cos(m[0]*np.pi/180), scale=np.cos(m[1]*np.pi/180))

def norm_mu_cos(m):
    return norm(loc=np.cos(m[0]*np.pi/180)*np.cos(m[1]*np.pi/180), scale=np.
 ↪cos(m[1]*np.pi/180))

def norm_sig_cos(m):
    return norm(loc=np.cos(m[0]*np.pi/180), scale=np.cos(m[0]*np.pi/180)*np.
 ↪cos(m[1]*np.pi/180))

def norm_mu_sig_cos(m):
    return norm(loc=np.cos(m[0]*np.pi/180)*np.cos(m[1]*np.pi/180), scale=np.
 ↪cos(m[0]*np.pi/180)*np.cos(m[1]*np.pi/180))
```

[19]:
```
def bootstrap_interpolation(func, m, mprime, bins, npdf=500, niter=100):
    orig_interp = np.array([])
    new_interp = np.array([])

    for i in range(niter):
        pdfs = np.array([])
        for row in m:
            pdfs = np.append(pdfs, np.histogram(func(row).rvs(npdf), bins=bins,␣
 ↪density=True)[0])
        pdfs = pdfs.reshape((m.shape[0], len(bins)-1))

        new = interpolate_pdf(bins, pdfs, m, mprime, -1)
        new_interp = np.append(new_interp, new/(np.sum(binwidth*new)))
```

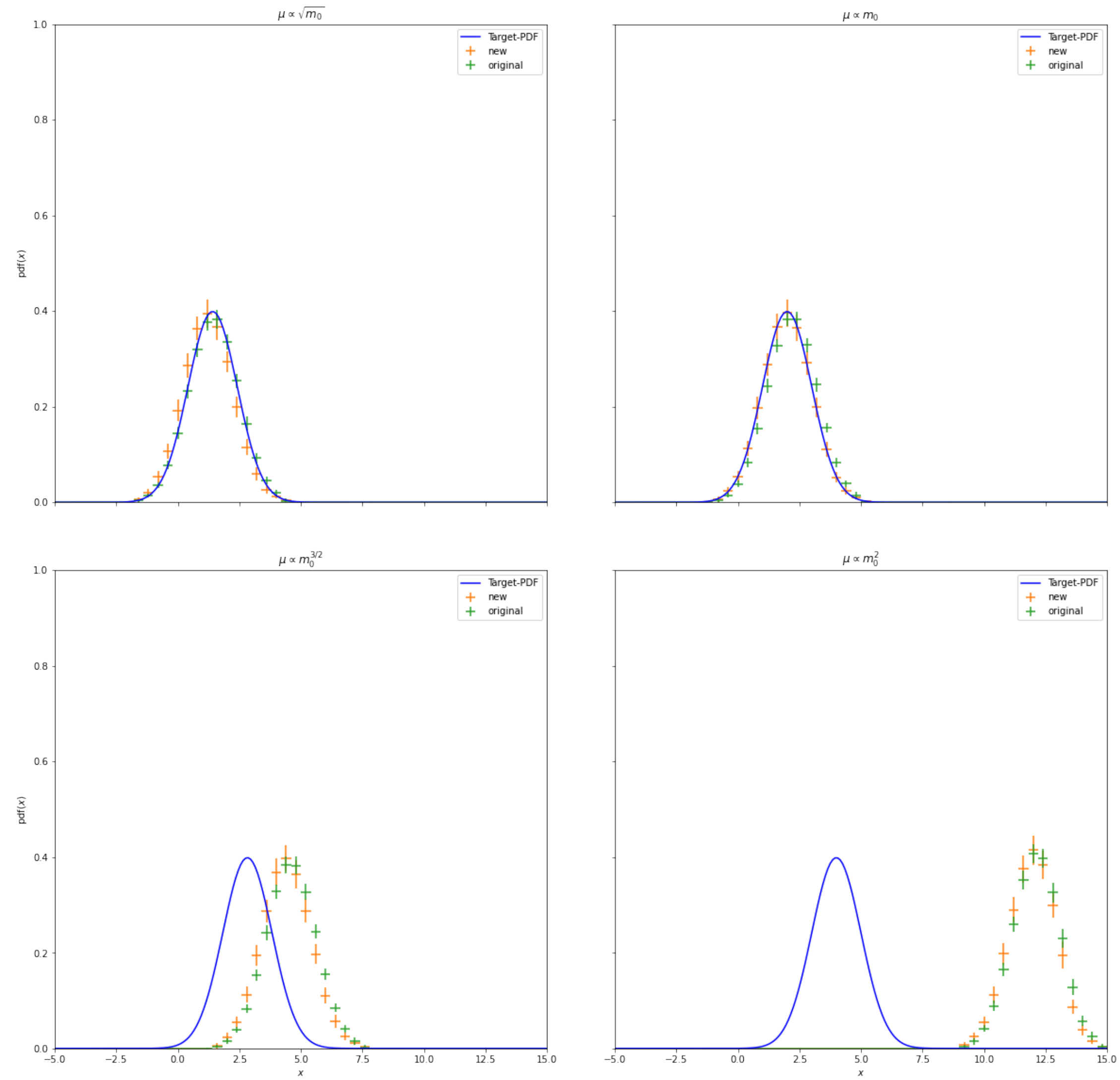
```
        orig_interp = np.append(orig_interp,␣
↪interpolate_energy_dispersion_original(bins, pdfs, m,                                                                       ␣
↪mprime, -1, "weighted_sum"))

    return new_interp.reshape((niter, len(bins)-1)), orig_interp.
↪reshape((niter, len(bins)-1))
```

[20]:
```
m = np.array([[ 1,  1],
              [10,  1],
              [ 1, 10],
              [10, 10]])
bins = np.linspace(-100, 100, 501)
binmids = bins[:-1] + np.diff(bins)
binwidth = np.diff(bins)
mprime = np.array([2, 5])
xrange=np.linspace(-100, 100, 2000)
```

6.0.1 Varying mean, fixed std

[21]:
```
import warnings
warnings.filterwarnings("ignore")

fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
                 )

titles = [r'$\mu \propto \sqrt{m_0}$', r'$\mu \propto m_0$',
          r'$\mu \propto m_0^{3/2}$', r'$\mu \propto m_0^2$']

for ax, func, title in zip(grid, [norm_mu_05, norm_mu_1, norm_mu_15,␣
↪norm_mu_2], titles):
    new_interp , orig_interp = bootstrap_interpolation(func, m, mprime, bins)

    new_mean = np.mean(new_interp, axis=0)
    new_std = np.std(new_interp, ddof=1, axis=0)

    orig_mean = np.mean(orig_interp, axis=0)
    orig_std = np.std(orig_interp, ddof=1, axis=0)

    ax.errorbar(binmids, new_mean, xerr=binwidth/2, yerr=new_std, fmt='none',␣
↪color='C1', label="new")
```

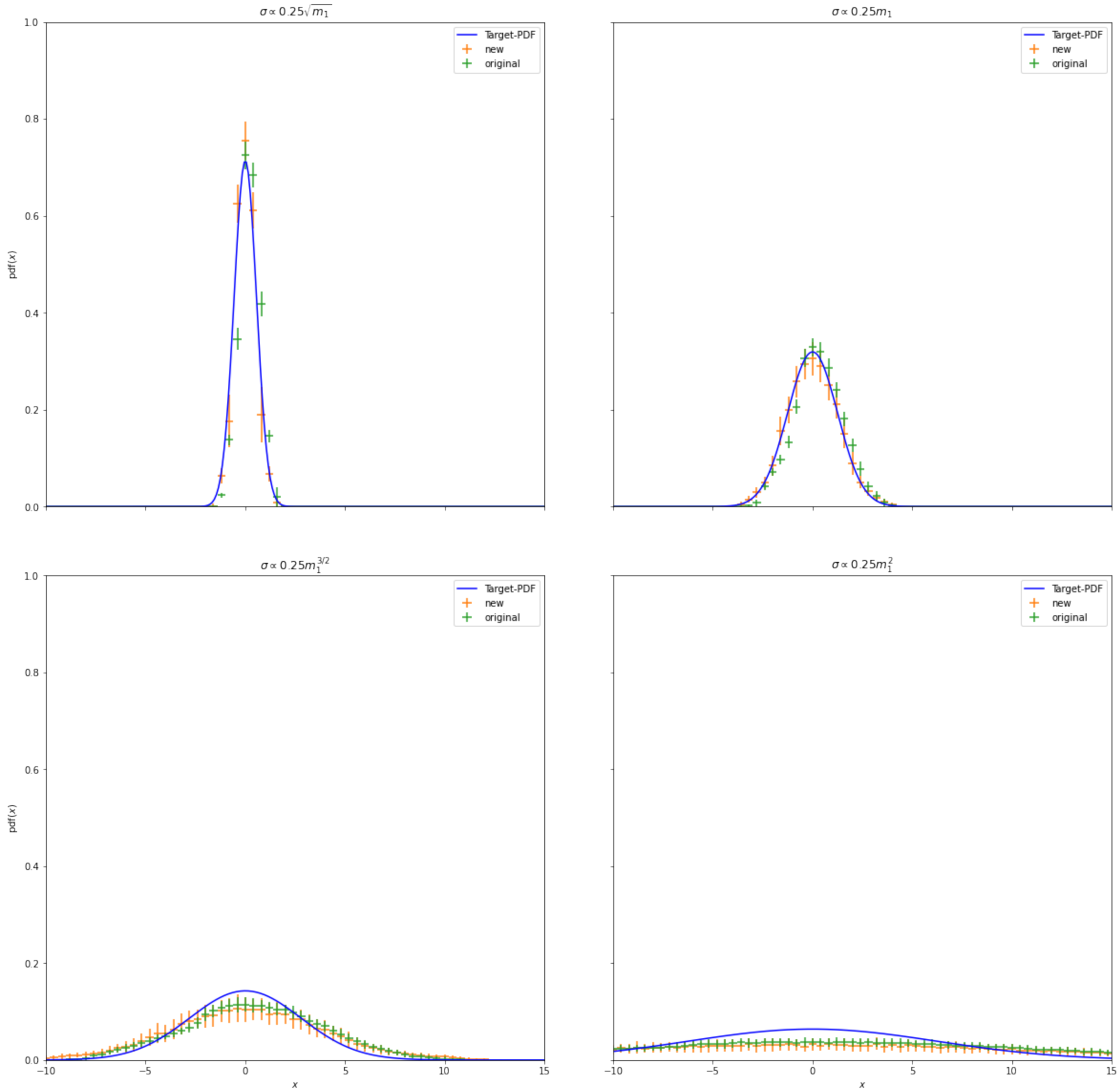
```
    ax.errorbar(binmids, orig_mean, xerr=binwidth/2, yerr=orig_std, fmt='none', color='C2', label="original")
    ax.plot(xrange, func(mprime).pdf(xrange), 'b-', label='Target-PDF')
    ax.legend(loc='best')
    ax.set_ylim(0,1)
    ax.set_xlim(-5, 15)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'pdf($x$)')
    ax.set_title(title)
```



Both methods work fine for linear changes in the distribution mean. New method slightly better.

6.0.2 Fixed mean, varying std

[22]:
```
fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
                 )

titles = [r'$\sigma \propto 0.25 \sqrt{m_1}$', r'$\sigma \propto 0.25 m_1$',
          r'$\sigma \propto 0.25 m_1^{3/2}$', r'$\sigma \propto 0.25 m_1^2$']

for ax, func, title in zip(grid, [norm_sig_05, norm_sig_1, norm_sig_15,␣
↪norm_sig_2], titles):
    new_interp , orig_interp = bootstrap_interpolation(func, m, mprime, bins)

    new_mean = np.mean(new_interp, axis=0)
    new_std = np.std(new_interp, ddof=1, axis=0)

    orig_mean = np.mean(orig_interp, axis=0)
    orig_std = np.std(orig_interp, ddof=1, axis=0)

    ax.errorbar(binmids, new_mean, xerr=binwidth/2, yerr=new_std, fmt='none',␣
↪color='C1', label="new")
    ax.errorbar(binmids, orig_mean, xerr=binwidth/2, yerr=orig_std, fmt='none',␣
↪color='C2', label="original")
    ax.plot(xrange, func(mprime).pdf(xrange), 'b-', label='Target-PDF')
    ax.legend(loc='best')
    ax.set_ylim(0,1)
    ax.set_xlim(-10, 15)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'pdf($x$)')
    ax.set_title(title)
```

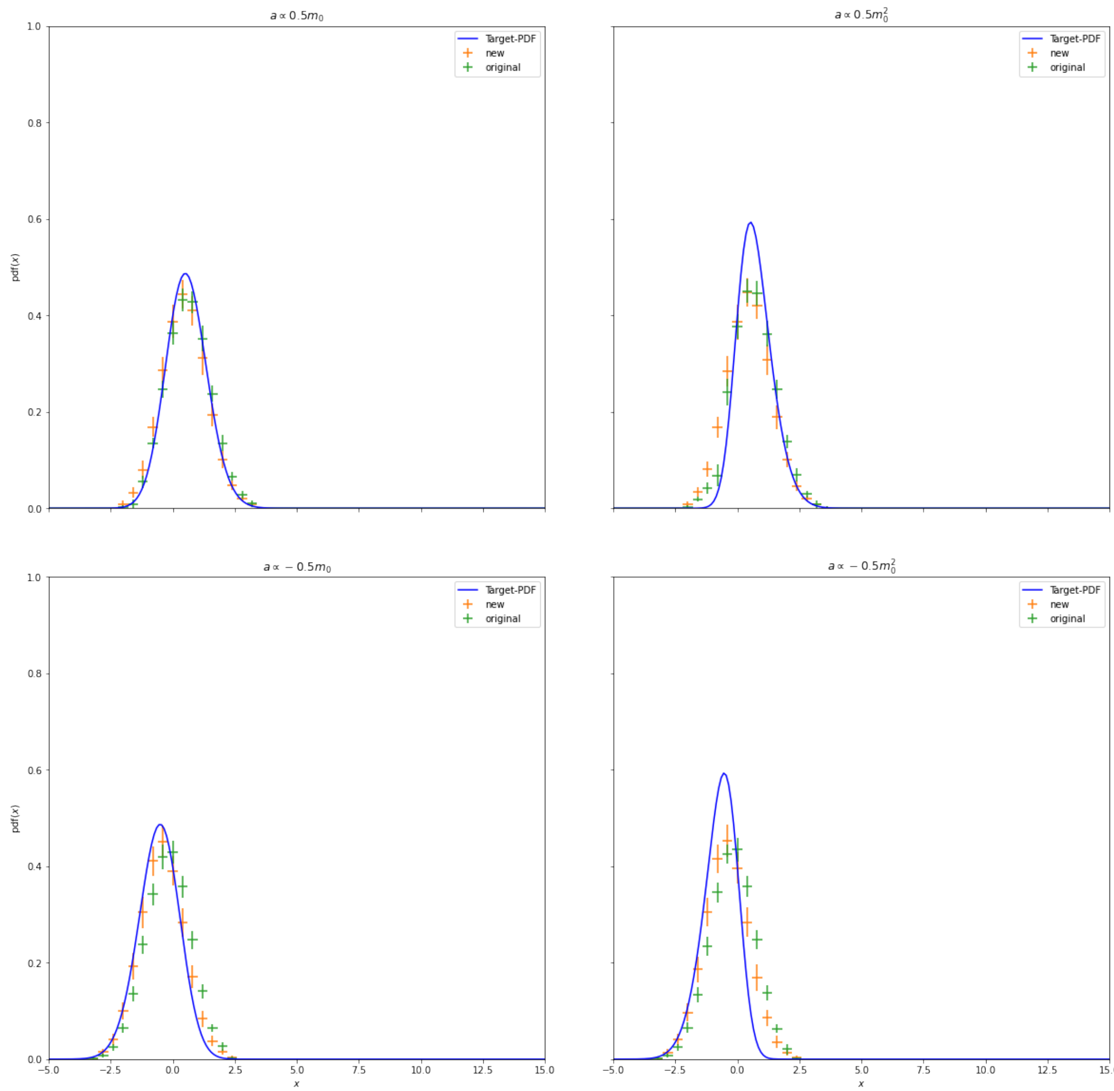


Changes in distribution std more problematic: - Methods perform comparable, new method again better. - Resonable results for linear changes. Bad results for non-linear scaling.

6.0.3 Skewed distributions

[23]:
```
fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
                 )

titles = [r'$a \propto 0.5 m_0$', r'$a \propto 0.5 m_0^2$',
```

```
          r'$a \propto -0.5 m_0$', r'$a \propto -0.5 m_0^2$']

for ax, func, title in zip(grid, [skewnorm_skew1, skewnorm_skew2,␣
↪skewnorm_skewneg1, skewnorm_skewneg2], titles):
    new_interp , orig_interp = bootstrap_interpolation(func, m, mprime, bins)

    new_mean = np.mean(new_interp, axis=0)
    new_std = np.std(new_interp, ddof=1, axis=0)

    orig_mean = np.mean(orig_interp, axis=0)
    orig_std = np.std(orig_interp, ddof=1, axis=0)

    ax.errorbar(binmids, new_mean, xerr=binwidth/2, yerr=new_std, fmt='none',␣
↪color='C1', label="new")
    ax.errorbar(binmids, orig_mean, xerr=binwidth/2, yerr=orig_std, fmt='none',␣
↪color='C2', label="original")
    ax.plot(xrange, func(mprime).pdf(xrange), 'b-', label='Target-PDF')
    ax.legend(loc='best')
    ax.set_ylim(0,1)
    ax.set_xlim(-5, 15)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'pdf($x$)')
    ax.set_title(title)
```

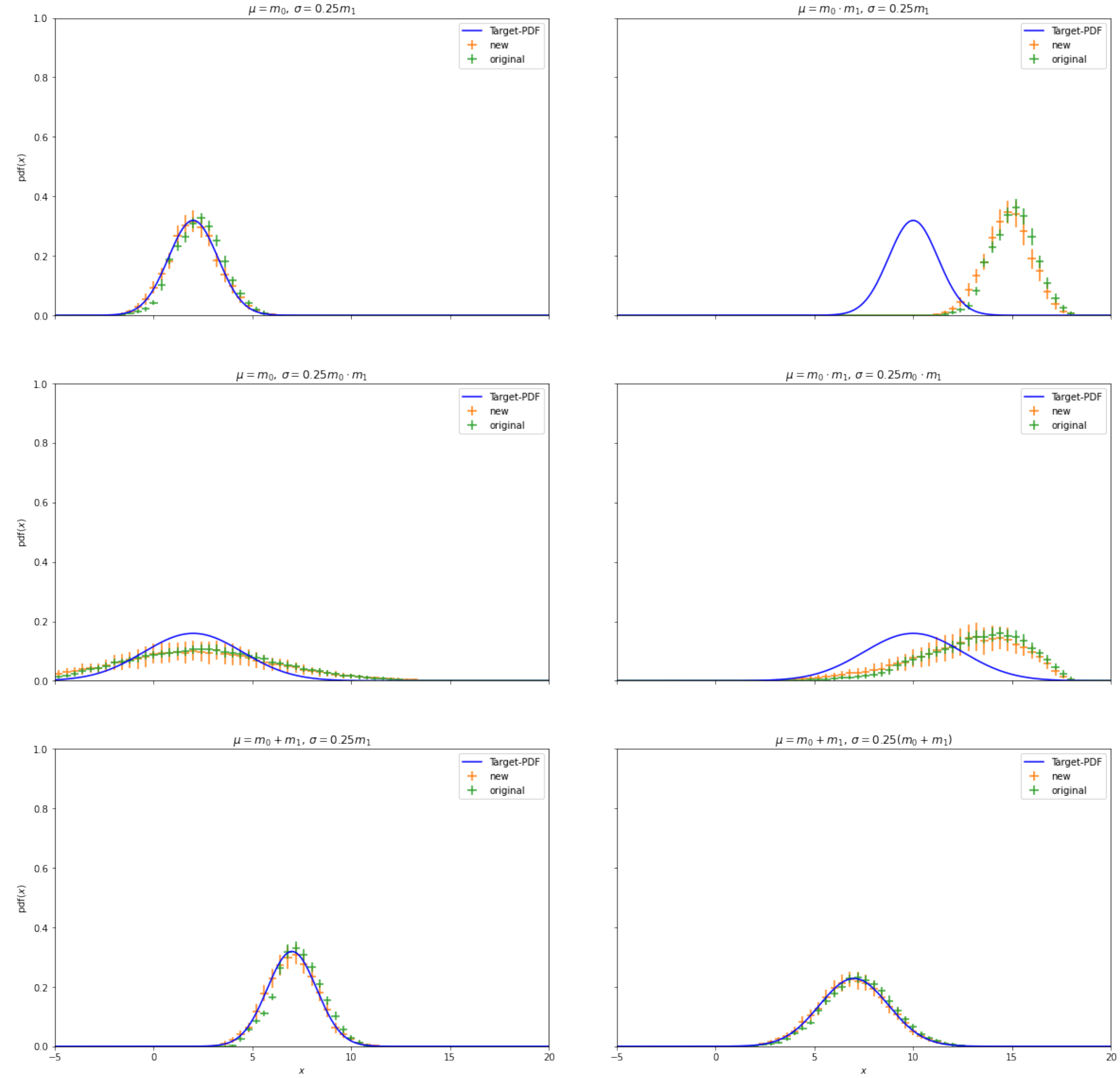


Both Methods have problems with (increasingly) skewed distributions.

6.0.4 Multiple and non-seperated dependencies from hidden variables

[24]:
```
fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(3,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
                 )

titles = [r'$\mu = m_0,\, \sigma = 0.25 m_1$',
          r'$\mu = m_0 \cdot m_1,\, \sigma = 0.25 m_1$',
```

```
          r'$\mu = m_0,\, \sigma = 0.25 m_0 \cdot m_1$',
          r'$\mu = m_0 \cdot m_1,\, \sigma = 0.25 m_0 \cdot m_1$',
          r'$\mu = m_0 + m_1,\, \sigma = 0.25 m_1$',
          r'$\mu = m_0 + m_1,\, \sigma = 0.25 (m_0 + m_1)$']

for ax, func, title in zip(grid, [norm_both, norm_mu, norm_sig, norm_mu_sig,␣
↪norm_mu_add, norm_mu_sig_add], titles):
    new_interp , orig_interp = bootstrap_interpolation(func, m, mprime, bins)

    new_mean = np.mean(new_interp, axis=0)
    new_std = np.std(new_interp, ddof=1, axis=0)

    orig_mean = np.mean(orig_interp, axis=0)
    orig_std = np.std(orig_interp, ddof=1, axis=0)

    ax.errorbar(binmids, new_mean, xerr=binwidth/2, yerr=new_std, fmt='none',␣
↪color='C1', label="new")
    ax.errorbar(binmids, orig_mean, xerr=binwidth/2, yerr=orig_std, fmt='none',␣
↪color='C2', label="original")
    ax.plot(xrange, func(mprime).pdf(xrange), 'b-', label='Target-PDF')
    ax.legend(loc='best')
    ax.set_ylim(0,1)
    ax.set_xlim(-5, 20)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'pdf($x$)')
    ax.set_title(title)
```



Having μ and σ depending on combinations of the hidden varaibles combines the problems found above. Best case would be a scenario, where μ and σ are independent and vary only linear. Linear combinations of hidden variables are also unproblematic (bottom row). Non-linear combinations of the hidden variables have the worst results.

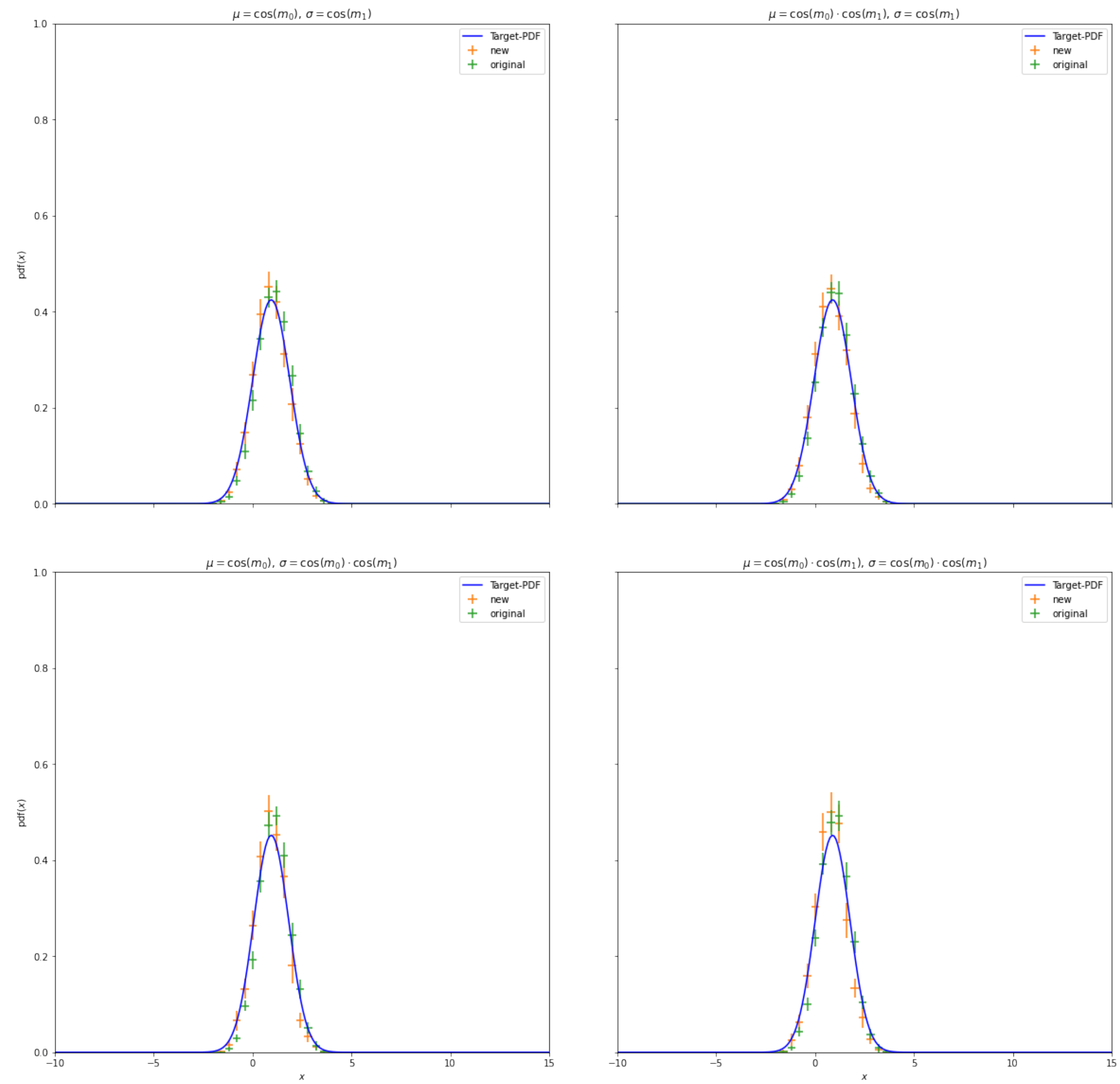
6.0.5 Non-Linear interpolation on cos-scales

Now interpolating between points $[0, 0]$, $[45, 0]$, $[0, 45]$, $[45, 45]$ deg to estimate distribution at $[20, 20]$ deg but having the pdf depending on cos()-expressions.

[25]:
```
m = np.array([[ 0,  0],
              [45,  0],
              [ 0, 45],
              [45, 45]])
```

```
mprime = np.array([20, 20])
```

[26]:
```
fig = plt.figure(figsize=(20, 20))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(2,2),
                 axes_pad=1,
                 share_all=False,
                 aspect=False
                 )

titles = [r'$\mu = $cos$(m_0),\, \sigma = $cos$(m_1)$',
          r'$\mu = $cos$(m_0) \cdot $cos$(m_1),\, \sigma = $cos$(m_1)$',
          r'$\mu = $cos$(m_0),\, \sigma = $cos$(m_0) \cdot $cos$(m_1)$',
          r'$\mu = $cos$(m_0) \cdot $cos$(m_1),\, \sigma = $cos$(m_0) \cdot␣
↪$cos$(m_1)$']

for ax, func, title in zip(grid, [norm_both_cos, norm_mu_cos, norm_sig_cos,␣
↪norm_mu_sig_cos], titles):
    new_interp , orig_interp = bootstrap_interpolation(func, m, mprime, bins)

    new_mean = np.mean(new_interp, axis=0)
    new_std = np.std(new_interp, ddof=1, axis=0)

    orig_mean = np.mean(orig_interp, axis=0)
    orig_std = np.std(orig_interp, ddof=1, axis=0)

    ax.errorbar(binmids, new_mean, xerr=binwidth/2, yerr=new_std, fmt='none',␣
↪color='C1', label="new")
    ax.errorbar(binmids, orig_mean, xerr=binwidth/2, yerr=orig_std, fmt='none',␣
↪color='C2', label="original")
    ax.plot(xrange, func(mprime).pdf(xrange), 'b-', label='Target-PDF')
    ax.legend(loc='best')
    ax.set_ylim(0,1)
    ax.set_xlim(-10, 15)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'pdf($x$)')
    ax.set_title(title)
```



Previous problems barely visible if changes are on small scales due to dependencies from e.g. $\cos(m_0)$, even though it is a non-linear transformation and the sample pdfs are chosen on a a wide grid $[0, 0]$, $[45, 0]$, $[0, 45]$, $[45, 45]$.

6.1 Gist

Not possible: Both methods have huge problems with strong non-linear behaviour, but since the core of both methods are linear interpolations this is not surprising. Both methods cannot handle highly skewed distributions.

Possible: Both methods can handle linear morphing, both methods seem to handle non-linear morphing if it happens on small scales.

The new implementation seems to be slightly more stable.