Unfolding - Systematic Testing

January 29, 2019

1 Systematic testing of Bayesian unfolding using simulated spectra from LaBr3, Plastic, and PIPS detector

Created by Andrei R. Hanu

```
In [1]: # Libraries to handle ROOT files
        import ROOT
        import root_numpy

        # Theano
        import theano
        import theano.tensor

        # Copy function
        import copy

        # NumPy
        import numpy as np

        # PyMC3
        import pymc3 as pm

        # Texttable
        from texttable import Texttable

        # Color palette library for Python
        # How to choose a colour scheme for your data:
        # http://earthobservatory.nasa.gov/blogs/elegantfigures/2013/08/05/subtleties-of-color-p
        import palettable

        # Matplotlib - 2D plotting library
        %matplotlib inline
        import matplotlib.pyplot as plt
        from matplotlib import colors
        from matplotlib import gridspec
        #import seaborn.apionly as sns
        from matplotlib import rcParams
        from mpl_toolkits.axes_grid1.inset_locator import inset_axes
        from mpl_toolkits.axes_grid1 import Grid, AxesGrid

        # Python garbabe collector
        import gc
```

```
Welcome to JupyROOT 6.11/01


WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS functions.
/opt/root_build/lib/ROOT.py:318: FutureWarning: Conversion of the second argument of issubdtype
  return _orig_ihook( name, *args, **kwds )
```

```
In [2]: ##########################################################################################
        # Setting rcParams for publication quality graphs
        fig_size =  np.array([7.3,4.2])*1.5
        params = {'backend': 'pdf',
                'axes.labelsize': 12,
                'legend.fontsize': 12,
                'xtick.labelsize': 12,
                'ytick.labelsize': 12,
                'xtick.major.size': 7,
                'xtick.major.width': 1,
                'xtick.minor.size': 3.5,
                'xtick.minor.width': 1.25,
                'ytick.major.size': 7,
                'ytick.major.width': 1.25,
                'ytick.minor.size': 3.5,
                'ytick.minor.width': 1.25,
                'font.family': 'sans-serif',
                'font.sans-serif': 'Bitstream Vera Sans',
                'font.size': 11,
                'figure.figsize': fig_size}

        # Update rcParams
        rcParams.update(params)
```

```
In [3]: # Threshold Energy, in keV, above which unfolding will occur
        thld_e = 0.

        # Configuration
        det1 = 'Saint Gobain B380 LaBr3'
        det2 = 'Eljen Plastic Detector'
        det3 = 'Canberra PD450-15-500AM'

        #isotope = 'Cl36'
        isotope = 'Sr90Y90'
        #isotope = 'Cs137'
        f_data = isotope + '_R_25_cm_Nr_100000000_ISO.root'
```

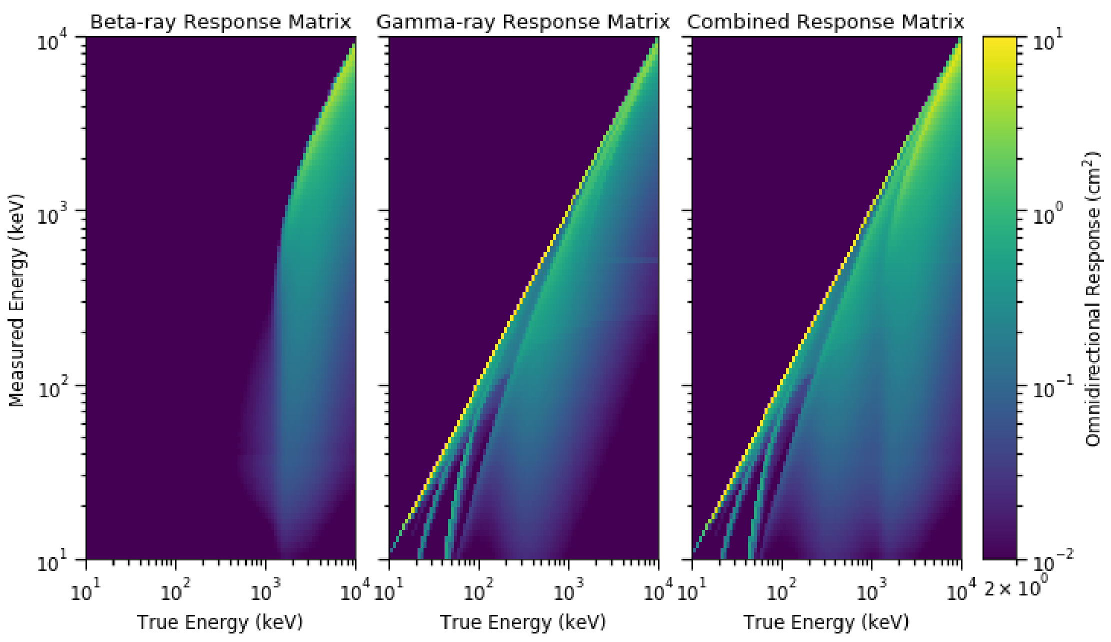
1.1 STEP 1 - Import the detector response matrices

```
In [4]: # Load the ROOT file containing the response matrix for the detector
        f_rspns = ROOT.TFile.Open('./TestData/'+det1+'/Response Matrix/'+det1+'.root')

        # Retrieve the electron and gamma-ray energy migration matrices and source vectors (i.e.
        # NOTE: Index 0 contains the bin values
        #       Index 1 contains the bin edges
        src_vec_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Electron)'),
                                                     include_overflow=False, copy=True, return_e

        src_vec_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Gamma)'),
                                                       include_overflow=False, copy=True, return

        mig_mat_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Elect
                                                     include_overflow=False, copy=True, return_e

        mig_mat_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Gam
                                                       include_overflow=False, copy=True, return

        # Calculate the response matrices by normalizing the energy migration matrices by the so
        rspns_mat_det1_e = copy.deepcopy(mig_mat_e)
        rspns_mat_det1_e[0] = np.nan_to_num(rspns_mat_det1_e[0]/src_vec_e[0])
        rspns_mat_det1_gam = copy.deepcopy(mig_mat_gam)
        rspns_mat_det1_gam[0] = np.nan_to_num(rspns_mat_det1_gam[0]/src_vec_e[0])

        # Remove response matrix elements below threshold energy
        rspns_mat_det1_e[0] = np.delete(rspns_mat_det1_e[0], np.where(rspns_mat_det1_e[1][0] < t
        rspns_mat_det1_e[0] = np.delete(rspns_mat_det1_e[0], np.where(rspns_mat_det1_e[1][0] < t
        rspns_mat_det1_e[1] = np.delete(rspns_mat_det1_e[1], np.where(rspns_mat_det1_e[1][0] < t
        rspns_mat_det1_gam[0] = np.delete(rspns_mat_det1_gam[0], np.where(rspns_mat_det1_gam[1][
        rspns_mat_det1_gam[0] = np.delete(rspns_mat_det1_gam[0], np.where(rspns_mat_det1_gam[1][
        rspns_mat_det1_gam[1] = np.delete(rspns_mat_det1_gam[1], np.where(rspns_mat_det1_gam[1][

        # Create a combined response matrix
        rspns_mat_det1_comb = copy.deepcopy(rspns_mat_det1_e)
        rspns_mat_det1_comb[0] += rspns_mat_det1_gam[0]

In [5]: # Load the ROOT file containing the response matrix for the detector
        f_rspns = ROOT.TFile.Open('./TestData/'+det2+'/Response Matrix/'+det2+'.root')

        # Retrieve the electron and gamma-ray energy migration matrices and source vectors (i.e.
        # NOTE: Index 0 contains the bin values
        #       Index 1 contains the bin edges
        src_vec_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Electron)'),
                                                     include_overflow=False, copy=True, return_e

        src_vec_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Gamma)'),
                                                       include_overflow=False, copy=True, return
```

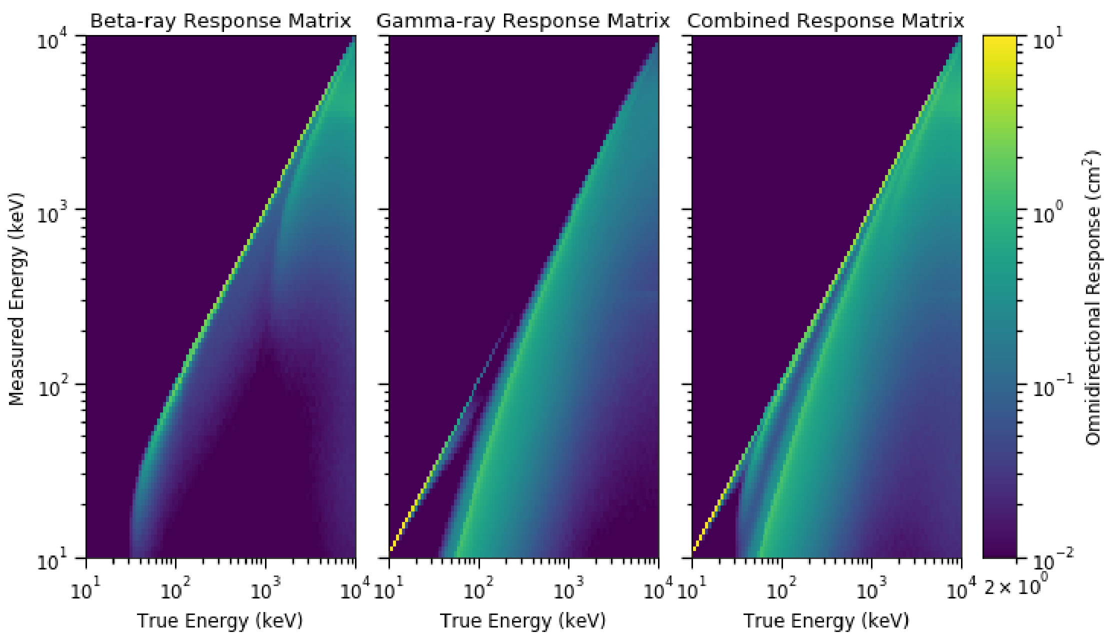
```
mig_mat_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Elect
                                              include_overflow=False, copy=True, return_e

mig_mat_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Gam
                                                include_overflow=False, copy=True, return

# Calculate the response matrices by normalizing the energy migration matrices by the so
rspns_mat_det2_e = copy.deepcopy(mig_mat_e)
rspns_mat_det2_e[0] = np.nan_to_num(rspns_mat_det2_e[0]/src_vec_e[0])
rspns_mat_det2_gam = copy.deepcopy(mig_mat_gam)
rspns_mat_det2_gam[0] = np.nan_to_num(rspns_mat_det2_gam[0]/src_vec_e[0])

# Remove response matrix elements below threshold energy
rspns_mat_det2_e[0] = np.delete(rspns_mat_det2_e[0], np.where(rspns_mat_det2_e[1][0] < t
rspns_mat_det2_e[0] = np.delete(rspns_mat_det2_e[0], np.where(rspns_mat_det2_e[1][0] < t
rspns_mat_det2_e[1] = np.delete(rspns_mat_det2_e[1], np.where(rspns_mat_det2_e[1][0] < t
rspns_mat_det2_gam[0] = np.delete(rspns_mat_det2_gam[0], np.where(rspns_mat_det2_gam[1][
rspns_mat_det2_gam[0] = np.delete(rspns_mat_det2_gam[0], np.where(rspns_mat_det2_gam[1][
rspns_mat_det2_gam[1] = np.delete(rspns_mat_det2_gam[1], np.where(rspns_mat_det2_gam[1][

# Create a combined response matrix
rspns_mat_det2_comb = copy.deepcopy(rspns_mat_det2_e)
rspns_mat_det2_comb[0] += rspns_mat_det2_gam[0]
```

```
In [6]: # Load the ROOT file containing the response matrix for the detector
        f_rspns = ROOT.TFile.Open('./TestData/'+det3+'/Response Matrix/'+det3+'.root')

        # Retrieve the electron and gamma-ray energy migration matrices and source vectors (i.e.
        # NOTE: Index 0 contains the bin values
        #       Index 1 contains the bin edges
        src_vec_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Electron)'),
                                                      include_overflow=False, copy=True, return_e

        src_vec_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Source Spectrum (Gamma)'),
                                                        include_overflow=False, copy=True, return

        mig_mat_e = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Elect
                                                      include_overflow=False, copy=True, return_e

        mig_mat_gam = np.asarray(root_numpy.hist2array(f_rspns.Get('Energy Migration Matrix (Gam
                                                        include_overflow=False, copy=True, return

        # Calculate the response matrices by normalizing the energy migration matrices by the so
        rspns_mat_det3_e = copy.deepcopy(mig_mat_e)
        rspns_mat_det3_e[0] = np.nan_to_num(rspns_mat_det3_e[0]/src_vec_e[0])
        rspns_mat_det3_gam = copy.deepcopy(mig_mat_gam)
        rspns_mat_det3_gam[0] = np.nan_to_num(rspns_mat_det3_gam[0]/src_vec_e[0])
        # Remove response matrix elements below threshold energy
        rspns_mat_det3_e[0] = np.delete(rspns_mat_det3_e[0], np.where(rspns_mat_det3_e[1][0] < t
        rspns_mat_det3_e[0] = np.delete(rspns_mat_det3_e[0], np.where(rspns_mat_det3_e[1][0] < t
        rspns_mat_det3_e[1] = np.delete(rspns_mat_det3_e[1], np.where(rspns_mat_det3_e[1][0] < t
        rspns_mat_det3_gam[0] = np.delete(rspns_mat_det3_gam[0], np.where(rspns_mat_det3_gam[1][
        rspns_mat_det3_gam[0] = np.delete(rspns_mat_det3_gam[0], np.where(rspns_mat_det3_gam[1][
        rspns_mat_det3_gam[1] = np.delete(rspns_mat_det3_gam[1], np.where(rspns_mat_det3_gam[1][

        # Create a combined response matrix
        rspns_mat_det3_comb = copy.deepcopy(rspns_mat_det3_e)
        rspns_mat_det3_comb[0] += rspns_mat_det3_gam[0]

In [7]: def plotResponseMatrix(rspns_mat_e, rspns_mat_gam, rspns_mat_comb, filename = 'Response
            # Plot the energy migration matrix
            fig_mig_mat = plt.figure()

            ax_mig_mat = AxesGrid(fig_mig_mat, 111,
                                  nrows_ncols=(1, 3),
                                  axes_pad=0.3,
                                  aspect=False,
                                  #label_mode = 'L',
                                  cbar_mode='single',
                                  cbar_location='right',
                                  cbar_pad=0.2,
                                  cbar_size = 0.3)

            # Color map
            cmap = palettable.matplotlib.Viridis_20.mpl_colormap
            cmap.set_bad(cmap(0.))
            cmap.set_over(cmap(1.))

            # Response Limits
            rLimUp = np.ceil(np.abs(np.log10(np.maximum(rspns_mat_e[0].max(), rspns_mat_gam[0].m
            rLimUp = 1E1
            rLimLow = rLimUp/1E3

            # Plot the response matrices
            X, Y = np.meshgrid(rspns_mat_e[1][0], rspns_mat_e[1][1])
            H0 = ax_mig_mat[0].pcolormesh(X, Y, rspns_mat_e[0].T, norm = colors.LogNorm(), cmap

            X, Y = np.meshgrid(rspns_mat_gam[1][0], rspns_mat_gam[1][1])
            H1 = ax_mig_mat[1].pcolormesh(X, Y, rspns_mat_gam[0].T, norm = colors.LogNorm(), cma

            X, Y = np.meshgrid(rspns_mat_comb[1][0], rspns_mat_comb[1][1])
            H2 = ax_mig_mat[2].pcolormesh(X, Y, rspns_mat_comb[0].T, norm = colors.LogNorm(), cm

            # Color limits for the plot
            H0.set_clim(rLimLow, rLimUp)
            H1.set_clim(rLimLow, rLimUp)
            H2.set_clim(rLimLow, rLimUp)

            # Colorbar
            from matplotlib.ticker import LogLocator
            ax_mig_mat.cbar_axes[0].colorbar(H2, spacing = 'uniform')
            ax_mig_mat.cbar_axes[0].set_yscale('log')
            ax_mig_mat.cbar_axes[0].axis[ax_mig_mat.cbar_axes[0].orientation].set_label('Omnidir

            # Figure Properties
            ax_mig_mat[0].set_xscale('log')
            ax_mig_mat[0].set_yscale('log')
            ax_mig_mat[0].set_ylabel('Measured Energy (keV)')
            ax_mig_mat[0].set_xlabel('True Energy (keV)')
            ax_mig_mat[0].set_title('Beta-ray Response Matrix')

            ax_mig_mat[1].set_xscale('log')
            ax_mig_mat[1].set_yscale('log')
            ax_mig_mat[1].set_xlabel('True Energy (keV)')
            ax_mig_mat[1].set_title('Gamma-ray Response Matrix')

            ax_mig_mat[2].set_xscale('log')
            ax_mig_mat[2].set_yscale('log')
            ax_mig_mat[2].set_xlabel('True Energy (keV)')
            ax_mig_mat[2].set_title('Combined Response Matrix')

            # Fine-tune figure
            fig_mig_mat.set_tight_layout(False)

            # Save the figure
            plt.savefig(filename, bbox_inches="tight")

            # Show the figure
            plt.show(fig_mig_mat)
            plt.close(fig_mig_mat)

        print('Response Matrix - Detector 1 - ' + det1)
        plotResponseMatrix(rspns_mat_det1_e, rspns_mat_det1_gam, rspns_mat_det1_comb, det1 + ' R
        print('Response Matrix - Detector 2 - ' + det2)
        plotResponseMatrix(rspns_mat_det2_e, rspns_mat_det2_gam, rspns_mat_det2_comb, det2 + ' R
        print('Response Matrix - Detector 3 - ' + det3)
        plotResponseMatrix(rspns_mat_det3_e, rspns_mat_det3_gam, rspns_mat_det3_comb, det3 + ' R
```

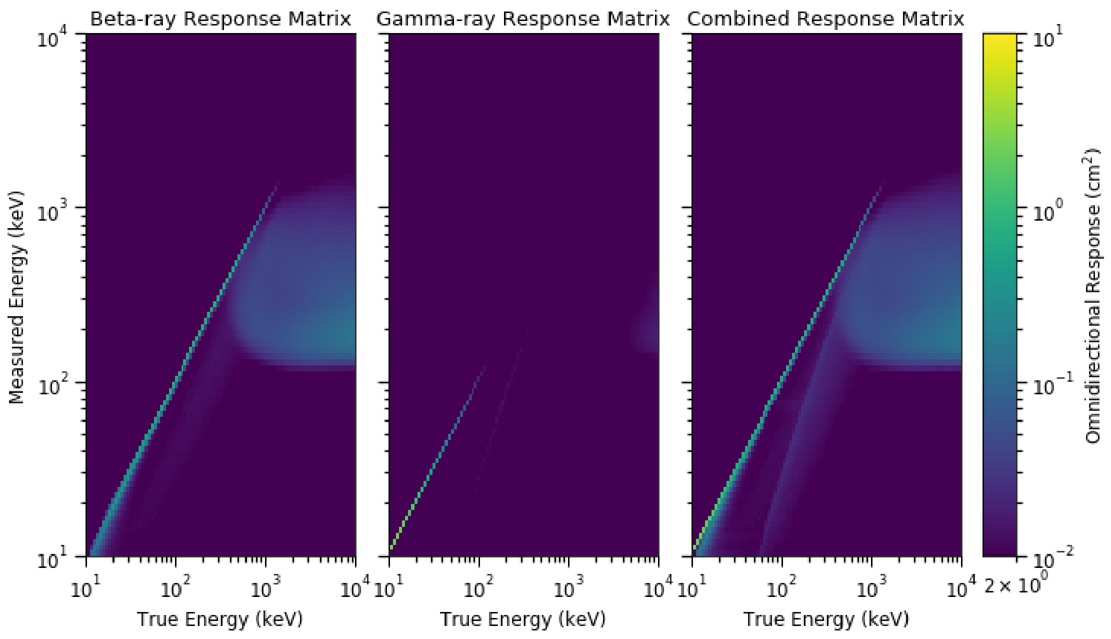
```
Response Matrix - Detector 1 - Saint Gobain B380 LaBr3
```



Response Matrix - Detector 2 - Eljen Plastic Detector



Response Matrix - Detector 3 - Canberra PD450-15-500AM



1.2 STEP 2 - Import the measured spectra from each detector

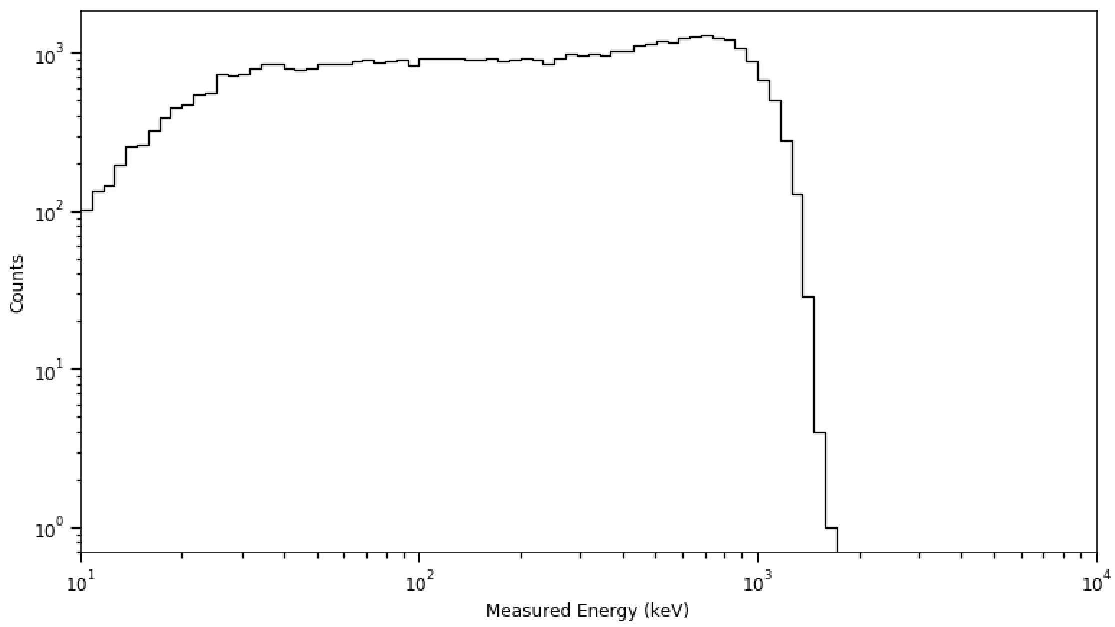
```
In [8]: # Load the ROOT file containing the measured spectrum
        f_meas = ROOT.TFile.Open('./TestData/'+det1+'/'+isotope+'/'+f_data)

        # Retrieve the measured spectrum
        # NOTE: Index 0 contains the bin values
        #       Index 1 contains the bin edges
        meas_vec_det1 = np.asarray(root_numpy.hist2array(f_meas.Get('Detector Measured Spectrum'
                                                   include_overflow=False, copy=True, return_ed

        truth_vec_det1_e = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Electro
                                                      include_overflow=False, copy=True, return

        truth_vec_det1_gam = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Gamma
                                                        include_overflow=False, copy=True, retu

        # Remove elements below threshold energy
        meas_vec_det1[0] = np.delete(meas_vec_det1[0], np.where(meas_vec_det1[1][0] < thld_e), a
        meas_vec_det1[1] = np.delete(meas_vec_det1[1], np.where(meas_vec_det1[1][0] < thld_e), a
        truth_vec_det1_e[0] = np.delete(truth_vec_det1_e[0], np.where(truth_vec_det1_e[1][0] < t
        truth_vec_det1_e[1] = np.delete(truth_vec_det1_e[1], np.where(truth_vec_det1_e[1][0] < t
        truth_vec_det1_gam[0] = np.delete(truth_vec_det1_gam[0], np.where(truth_vec_det1_gam[1][
        truth_vec_det1_gam[1] = np.delete(truth_vec_det1_gam[1], np.where(truth_vec_det1_gam[1][

In [9]: # Load the ROOT file containing the measured spectrum
        f_meas = ROOT.TFile.Open('./TestData/'+det2+'/'+isotope+'/'+f_data)
        # Retrieve the measured spectrum
        # NOTE: Index 0 contains the bin values
        #       Index 1 contains the bin edges
        meas_vec_det2 = np.asarray(root_numpy.hist2array(f_meas.Get('Detector Measured Spectrum'
                                                      include_overflow=False, copy=True, return_ed

        truth_vec_det2_e = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Electro
                                                         include_overflow=False, copy=True, return

        truth_vec_det2_gam = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Gamma
                                                           include_overflow=False, copy=True, retu

        # Remove elements below threshold energy
        meas_vec_det2[0] = np.delete(meas_vec_det2[0], np.where(meas_vec_det2[1][0] < thld_e), a
        meas_vec_det2[1] = np.delete(meas_vec_det2[1], np.where(meas_vec_det2[1][0] < thld_e), a
        truth_vec_det2_e[0] = np.delete(truth_vec_det2_e[0], np.where(truth_vec_det2_e[1][0] < t
        truth_vec_det2_e[1] = np.delete(truth_vec_det2_e[1], np.where(truth_vec_det2_e[1][0] < t
        truth_vec_det2_gam[0] = np.delete(truth_vec_det2_gam[0], np.where(truth_vec_det2_gam[1][
        truth_vec_det2_gam[1] = np.delete(truth_vec_det2_gam[1], np.where(truth_vec_det2_gam[1][
```

```
In [10]: # Load the ROOT file containing the measured spectrum
         f_meas = ROOT.TFile.Open('./TestData/'+det3+'/'+isotope+'/'+f_data)

         # Retrieve the measured spectrum
         # NOTE: Index 0 contains the bin values
         #       Index 1 contains the bin edges
         meas_vec_det3 = np.asarray(root_numpy.hist2array(f_meas.Get('Detector Measured Spectrum
                                                       include_overflow=False, copy=True, return_e

         truth_vec_det3_e = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Electr
                                                          include_overflow=False, copy=True, retur

         truth_vec_det3_gam = np.asarray(root_numpy.hist2array(f_meas.Get('Source Spectrum (Gamm
                                                            include_overflow=False, copy=True, ret

         # Remove elements below threshold energy
         meas_vec_det3[0] = np.delete(meas_vec_det3[0], np.where(meas_vec_det3[1][0] < thld_e),
         meas_vec_det3[1] = np.delete(meas_vec_det3[1], np.where(meas_vec_det3[1][0] < thld_e),
         truth_vec_det3_e[0] = np.delete(truth_vec_det3_e[0], np.where(truth_vec_det3_e[1][0] <
         truth_vec_det3_e[1] = np.delete(truth_vec_det3_e[1], np.where(truth_vec_det3_e[1][0] <
         truth_vec_det3_gam[0] = np.delete(truth_vec_det3_gam[0], np.where(truth_vec_det3_gam[1]
         truth_vec_det3_gam[1] = np.delete(truth_vec_det3_gam[1], np.where(truth_vec_det3_gam[1]
```

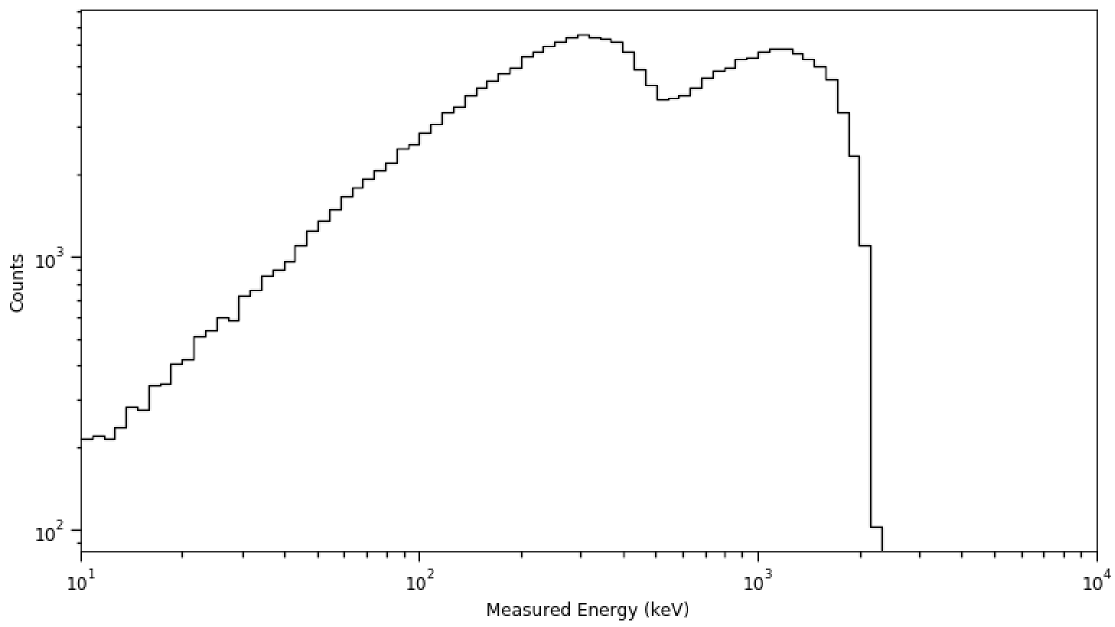
```
In [11]: # Plot the measured spectrum
         def plotMeasuredSpectrum(meas_vec, filename = 'Measured Spectrum.jpg'):
             # Plot the measured spectrum
             fig_meas_vec, ax_meas_vec = plt.subplots()
```

```
            # Plot the raw spectrum
            ax_meas_vec.plot(sorted(np.append(meas_vec[1][0][:-1], meas_vec[1][0][1:])),
                            np.repeat(meas_vec[0], 2),
                            lw=1.25,
                            color='black',
                            linestyle="-",
                            drawstyle='steps')

            # Figure properties
            ax_meas_vec.set_xlabel('Measured Energy (keV)')
            ax_meas_vec.set_ylabel('Counts')
            ax_meas_vec.set_xlim(min(meas_vec[1][0]),max(meas_vec[1][0]))
            ax_meas_vec.set_xscale('log')
            ax_meas_vec.set_yscale('log')

            # Fine-tune figure
            fig_meas_vec.set_tight_layout(True)

            # Save the figure
            plt.savefig(filename, bbox_inches="tight")

            # Show the figure
            plt.show(fig_meas_vec)
            plt.close(fig_meas_vec)

        print('Measured Spectrum - Detector 1 - ' + det1)
        plotMeasuredSpectrum(meas_vec_det1, isotope + ' - ' + det1 + ' - Measured Spectrum.jpg'
        print('Measured Spectrum - Detector 2 - ' + det2)
        plotMeasuredSpectrum(meas_vec_det2, isotope + ' - ' + det2 + ' - Measured Spectrum.jpg'
        print('Measured Spectrum - Detector 3 - ' + det3)
        plotMeasuredSpectrum(meas_vec_det3, isotope + ' - ' + det3 + ' - Measured Spectrum.jpg'
```

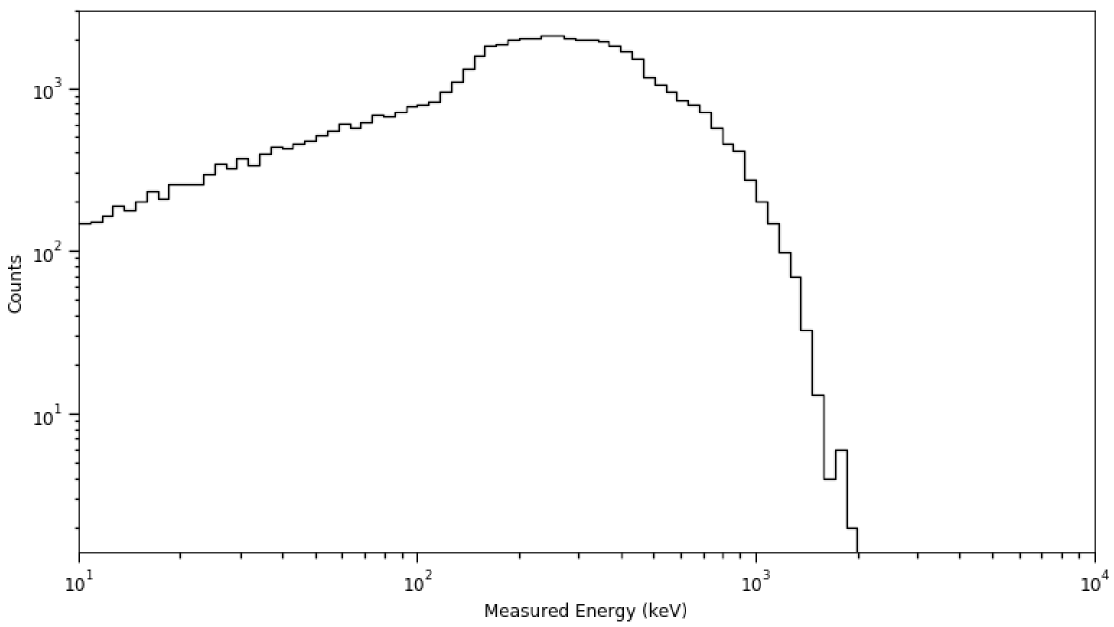
```
Measured Spectrum - Detector 1 - Saint Gobain B380 LaBr3


/usr/local/lib/python2.7/dist-packages/matplotlib/figure.py:2299: UserWarning: This figure inclu
  warnings.warn("This figure includes Axes that are not compatible "
```



Measured Spectrum - Detector 2 - Eljen Plastic Detector



Measured Spectrum - Detector 3 - Canberra PD450-15-500AM



1.3 STEP 3 - Build the generative models

Generally, when a detector is exposed to a homogeneous radiation field, the relationship between the incoming particle fluence spectrum and the measured energy spectrum, $D(E)$, can be described by the following Fredholm integral equation of the first kind:

$$D\left(E\right) = \int_0^\infty R\left(E,E^{'}\right)\Phi\left(E^{'}\right)dE^{'}\,,\, 0 \le E \le \infty$$

where $R\left(E,E^{'}\right)$ is a kernel describing the detector response in terms of the measured energy, E, and the true energy, $E^{'}$, of the incoming particle and $\Phi\left(E^{'}\right)$ is the incoming particle fluence spectrum.

Within the context of Bayesian inference, the above equation is often refered to as the generative model that describes how the measured data was generated when the detector was exposed to the radiation field.

For this systematic testing, the following generative models are available: - **model_det1** - this model uses **only** the response matrix and measured spectra from Detector 1 (det1)

```
In [12]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)

         with pm.Model() as model_det1:
             '''
             Define the upper and lower bounds of the uniform prior based on the measured data a
```

```
    For an ideal radiation detector, the response matrix would a diagonal meaning that
    '''

    lb_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    ub_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    lb_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)
    ub_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)

    # Get the upper and lower bounds from the combined response matrix
    t = Texttable()
    t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
    for i in np.arange(rspns_mat_det1_comb[1][0].size - 1):
        # Find the minimum and maximum non-zero response elements for each true energy
        index_det1_min = np.argwhere(rspns_mat_det1_comb[0][i,:] == np.min(rspns_mat_de
        index_det1_max = np.argwhere(rspns_mat_det1_comb[0][i,:] == rspns_mat_det1_comb

        # Calculate the lower and upper bounds on the prior based on the measured count
        # indeces and response elements
        min_phi = np.min([meas_vec_det1[0][index_det1_min], meas_vec_det1[0][index_det1

        max_phi = np.max([meas_vec_det1[0][index_det1_min], meas_vec_det1[0][index_det1

        # Update the bounds
        #lb_phi_e[i] = min_phi
        ub_phi_e[i] = max_phi
        #lb_phi_gam[i] = min_phi
        ub_phi_gam[i] = max_phi

        # Add it to the table for printout
        t.add_row(['{:.1f} kev'.format(rspns_mat_det1_comb[1][0][i]),
                   '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                   '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])

    #print t.draw()

    # Define the prior probability densities
    phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
    phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

    # Define the generative models
    M_det1 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_gam[0].T)), phi_gam)

    # Define the likelihood (aka. posterior probability function)
    PPF_det1 = pm.Poisson('PPF_det1', mu = M_det1, observed = theano.shared(meas_vec_de
```

- **model_det2** - this model uses **only** the response matrix and measured spectra from Detector

2 (det2)

```
In [13]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)

         with pm.Model() as model_det2:
             '''
             Define the upper and lower bounds of the uniform prior based on the measured data a

             For an ideal radiation detector, the response matrix would a diagonal meaning that
             '''

             lb_phi_e = np.zeros(rspns_mat_det2_e[1][0].size - 1)
             ub_phi_e = np.zeros(rspns_mat_det2_e[1][0].size - 1)
             lb_phi_gam = np.zeros(rspns_mat_det2_gam[1][0].size - 1)
             ub_phi_gam = np.zeros(rspns_mat_det2_gam[1][0].size - 1)

             # Get the upper and lower bounds from the combined response matrix
             t = Texttable()
             t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
             for i in np.arange(rspns_mat_det2_comb[1][0].size - 1):
                 # Find the minimum and maximum non-zero response elements for each true energy
                 index_det2_min = np.argwhere(rspns_mat_det2_comb[0][i,:] == np.min(rspns_mat_de
                 index_det2_max = np.argwhere(rspns_mat_det2_comb[0][i,:] == rspns_mat_det2_comb

                 # Calculate the lower and upper bounds on the prior based on the measured count
                 # indeces and response elements
                 min_phi = np.min([meas_vec_det2[0][index_det2_min], meas_vec_det2[0][index_det2

                 max_phi = np.max([meas_vec_det2[0][index_det2_min], meas_vec_det2[0][index_det2

                 # Update the bounds
                 #lb_phi_e[i] = min_phi
                 ub_phi_e[i] = max_phi
                 #lb_phi_gam[i] = min_phi
                 ub_phi_gam[i] = max_phi

                 # Add it to the table for printout
                 t.add_row(['{:.1f} kev'.format(rspns_mat_det2_comb[1][0][i]),
                            '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                            '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])

             #print t.draw()

             # Define the prior probability densities
```

```
    phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
    phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

    # Define the generative models
    M_det2 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_gam[0].T)), phi_gam)

    # Define the likelihood (aka. posterior probability function)
    PPF_det2 = pm.Poisson('PPF_det2', mu = M_det2, observed = theano.shared(meas_vec_de
```

- **model_det3** - this model uses **only** the response matrix and measured spectra from Detector 3 (det3)

```
In [14]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)

         with pm.Model() as model_det3:
             '''
             Define the upper and lower bounds of the uniform prior based on the measured data a

             For an ideal radiation detector, the response matrix would a diagonal meaning that
             '''

             lb_phi_e = np.zeros(rspns_mat_det3_e[1][0].size - 1)
             ub_phi_e = np.zeros(rspns_mat_det3_e[1][0].size - 1)
             lb_phi_gam = np.zeros(rspns_mat_det3_gam[1][0].size - 1)
             ub_phi_gam = np.zeros(rspns_mat_det3_gam[1][0].size - 1)

             # Get the upper and lower bounds from the combined response matrix
             t = Texttable()
             t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
             for i in np.arange(rspns_mat_det3_comb[1][0].size - 1):
                 # Find the minimum and maximum non-zero response elements for each true energy
                 index_det3_min = np.argwhere(rspns_mat_det3_comb[0][i,:] == np.min(rspns_mat_de
                 index_det3_max = np.argwhere(rspns_mat_det3_comb[0][i,:] == rspns_mat_det3_comb

                 # Calculate the lower and upper bounds on the prior based on the measured count
                 # indeces and response elements
                 min_phi = np.min([meas_vec_det3[0][index_det3_min], meas_vec_det3[0][index_det3

                 max_phi = np.max([meas_vec_det3[0][index_det3_min], meas_vec_det3[0][index_det3

                 # Update the bounds
                 #lb_phi_e[i] = min_phi
                 ub_phi_e[i] = max_phi
```

```
            #lb_phi_gam[i] = min_phi
            ub_phi_gam[i] = max_phi

            # Add it to the table for printout
            t.add_row(['{:.1f} kev'.format(rspns_mat_det3_comb[1][0][i]),
                       '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                       '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])

        #print t.draw()

        # Define the prior probability densities
        phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
        phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

        # Define the generative models
        M_det3 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_e[0].T)), phi_e) + \
                 theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_gam[0].T)), phi_gam)

        # Define the likelihood (aka. posterior probability function)
        PPF_det3 = pm.Poisson('PPF_det3', mu = M_det3, observed = theano.shared(meas_vec_de
```

- **model_det1_det2** - this model uses the response matrix and measured spectra from Detectors 1 and 2

```
In [15]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)

         with pm.Model() as model_det1_det2:
             '''
             Define the upper and lower bounds of the uniform prior based on the measured data a

             For an ideal radiation detector, the response matrix would a diagonal meaning that
             '''

             lb_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
             ub_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
             lb_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)
             ub_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)

             # Get the upper and lower bounds from the combined response matrix
             t = Texttable()
             t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
             for i in np.arange(rspns_mat_det1_comb[1][0].size - 1):
                 # Find the minimum and maximum non-zero response elements for each true energy
                 index_det1_min = np.argwhere(rspns_mat_det1_comb[0][i,:] == np.min(rspns_mat_de
                index_det2_min = np.argwhere(rspns_mat_det2_comb[0][i,:] == np.min(rspns_mat_de
                index_det1_max = np.argwhere(rspns_mat_det1_comb[0][i,:] == rspns_mat_det1_comb
                index_det2_max = np.argwhere(rspns_mat_det2_comb[0][i,:] == rspns_mat_det2_comb

                # Calculate the lower and upper bounds on the prior based on the measured count
                # indeces and response elements
                min_phi = np.minimum(np.min([meas_vec_det1[0][index_det1_min],
                                             meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_
                                     np.min([meas_vec_det2[0][index_det2_min],
                                             meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_

                max_phi = np.maximum(np.max([meas_vec_det1[0][index_det1_min],
                                             meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_
                                     np.max([meas_vec_det2[0][index_det2_min],
                                             meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_

                # Update the bounds
                #lb_phi_e[i] = min_phi
                ub_phi_e[i] = max_phi
                #lb_phi_gam[i] = min_phi
                ub_phi_gam[i] = max_phi

                # Add it to the table for printout
                t.add_row(['{:.1f} kev'.format(rspns_mat_det1_comb[1][0][i]),
                           '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                           '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])

            #print t.draw()

            # Define the prior probability densities
            phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
            phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

            # Define the generative models
            M_det1 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_e[0].T)), phi_e) + \
                     theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_gam[0].T)), phi_gam)
            M_det2 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_e[0].T)), phi_e) + \
                     theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_gam[0].T)), phi_gam)

            # Define the likelihood (aka. posterior probability function)
            PPF_det1 = pm.Poisson('PPF_det1', mu = M_det1, observed = theano.shared(meas_vec_de
            PPF_det2 = pm.Poisson('PPF_det2', mu = M_det2, observed = theano.shared(meas_vec_de
```

- **model_det1_det3** - this model uses the response matrix and measured spectra from Detectors 1 and 3

```
In [16]: def asMat(x):
             '''
```

```
    Transform an array of doubles into a Theano-type array so that it can be used in th
    '''
    return np.asarray(x,dtype=theano.config.floatX)

with pm.Model() as model_det1_det3:
    '''
    Define the upper and lower bounds of the uniform prior based on the measured data a

    For an ideal radiation detector, the response matrix would a diagonal meaning that
    '''

    lb_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    ub_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    lb_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)
    ub_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)

    # Get the upper and lower bounds from the combined response matrix
    t = Texttable()
    t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
    for i in np.arange(rspns_mat_det1_comb[1][0].size - 1):
        # Find the minimum and maximum non-zero response elements for each true energy
        index_det1_min = np.argwhere(rspns_mat_det1_comb[0][i,:] == np.min(rspns_mat_de
        index_det3_min = np.argwhere(rspns_mat_det3_comb[0][i,:] == np.min(rspns_mat_de
        index_det1_max = np.argwhere(rspns_mat_det1_comb[0][i,:] == rspns_mat_det1_comb
        index_det3_max = np.argwhere(rspns_mat_det3_comb[0][i,:] == rspns_mat_det3_comb

        # Calculate the lower and upper bounds on the prior based on the measured count
        # indeces and response elements
        min_phi = np.minimum(np.min([meas_vec_det1[0][index_det1_min],
                                     meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_
                             np.min([meas_vec_det3[0][index_det3_min],
                                     meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_

        max_phi = np.maximum(np.max([meas_vec_det1[0][index_det1_min],
                                     meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_
                             np.max([meas_vec_det3[0][index_det3_min],
                                     meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_

        # Update the bounds
        #lb_phi_e[i] = min_phi
        ub_phi_e[i] = max_phi
        #lb_phi_gam[i] = min_phi
        ub_phi_gam[i] = max_phi

        # Add it to the table for printout
        t.add_row(['{:.1f} kev'.format(rspns_mat_det1_comb[1][0][i]),
                   '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                   '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])
    #print t.draw()

    # Define the prior probability densities
    phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
    phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

    # Define the generative models
    M_det1 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_gam[0].T)), phi_gam)
    M_det3 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_gam[0].T)), phi_gam)

    # Define the likelihood (aka. posterior probability function)
    PPF_det1 = pm.Poisson('PPF_det1', mu = M_det1, observed = theano.shared(meas_vec_de
    PPF_det3 = pm.Poisson('PPF_det3', mu = M_det3, observed = theano.shared(meas_vec_de
```

- **model_det2_det3** - this model uses the response matrix and measured spectra from Detectors 2 and 3

```
In [17]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)

         with pm.Model() as model_det2_det3:
             '''
             Define the upper and lower bounds of the uniform prior based on the measured data a

             For an ideal radiation detector, the response matrix would a diagonal meaning that
             '''

             lb_phi_e = np.zeros(rspns_mat_det2_e[1][0].size - 1)
             ub_phi_e = np.zeros(rspns_mat_det2_e[1][0].size - 1)
             lb_phi_gam = np.zeros(rspns_mat_det2_gam[1][0].size - 1)
             ub_phi_gam = np.zeros(rspns_mat_det2_gam[1][0].size - 1)

             # Get the upper and lower bounds from the combined response matrix
             t = Texttable()
             t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
             for i in np.arange(rspns_mat_det2_comb[1][0].size - 1):
                 # Find the minimum and maximum non-zero response elements for each true energy
                 index_det2_min = np.argwhere(rspns_mat_det2_comb[0][i,:] == np.min(rspns_mat_de
                 index_det3_min = np.argwhere(rspns_mat_det3_comb[0][i,:] == np.min(rspns_mat_de
                 index_det2_max = np.argwhere(rspns_mat_det2_comb[0][i,:] == rspns_mat_det2_comb
                 index_det3_max = np.argwhere(rspns_mat_det3_comb[0][i,:] == rspns_mat_det3_comb
        # Calculate the lower and upper bounds on the prior based on the measured count
        # indeces and response elements
        min_phi = np.minimum(np.min([meas_vec_det2[0][index_det2_min],
                                     meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_
                             np.min([meas_vec_det3[0][index_det3_min],
                                     meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_

        max_phi = np.maximum(np.max([meas_vec_det2[0][index_det2_min],
                                     meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_
                             np.max([meas_vec_det3[0][index_det3_min],
                                     meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_

        # Update the bounds
        #lb_phi_e[i] = min_phi
        ub_phi_e[i] = max_phi
        #lb_phi_gam[i] = min_phi
        ub_phi_gam[i] = max_phi

        # Add it to the table for printout
        t.add_row(['{:.1f} kev'.format(rspns_mat_det2_comb[1][0][i]),
                   '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                   '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])

    #print t.draw()

    # Define the prior probability densities
    phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
    phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

    # Define the generative models
    M_det2 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_gam[0].T)), phi_gam)
    M_det3 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_e[0].T)), phi_e) + \
             theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_gam[0].T)), phi_gam)

    # Define the likelihood (aka. posterior probability function)
    PPF_det2 = pm.Poisson('PPF_det2', mu = M_det2, observed = theano.shared(meas_vec_de
    PPF_det3 = pm.Poisson('PPF_det3', mu = M_det3, observed = theano.shared(meas_vec_de
```

- **model_det1_det2_det3** - this model uses the response matrix and measured spectra from Detectors 1, 2, and 3

```
In [18]: def asMat(x):
             '''
             Transform an array of doubles into a Theano-type array so that it can be used in th
             '''
             return np.asarray(x,dtype=theano.config.floatX)
```

```
with pm.Model() as model_det1_det2_det3:
    '''
    Define the upper and lower bounds of the uniform prior based on the measured data a

    For an ideal radiation detector, the response matrix would a diagonal meaning that
    '''

    lb_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    ub_phi_e = np.zeros(rspns_mat_det1_e[1][0].size - 1)
    lb_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)
    ub_phi_gam = np.zeros(rspns_mat_det1_gam[1][0].size - 1)

    # Get the upper and lower bounds from the combined response matrix
    t = Texttable()
    t.add_row(['True Energy', 'Min Phi (cm-2)', 'Max Phi (cm-2)'])
    for i in np.arange(rspns_mat_det1_comb[1][0].size - 1):
        # Find the minimum and maximum non-zero response elements for each true energy
        index_det1_min = np.argwhere(rspns_mat_det1_comb[0][i,:] == np.min(rspns_mat_de
        index_det2_min = np.argwhere(rspns_mat_det2_comb[0][i,:] == np.min(rspns_mat_de
        index_det3_min = np.argwhere(rspns_mat_det3_comb[0][i,:] == np.min(rspns_mat_de
        index_det1_max = np.argwhere(rspns_mat_det1_comb[0][i,:] == rspns_mat_det1_comb
        index_det2_max = np.argwhere(rspns_mat_det2_comb[0][i,:] == rspns_mat_det2_comb
        index_det3_max = np.argwhere(rspns_mat_det3_comb[0][i,:] == rspns_mat_det3_comb

        # Calculate the lower and upper bounds on the prior based on the measured count
        # indeces and response elements
        min_phi = np.min([np.min([meas_vec_det1[0][index_det1_min],
                                 meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_comb
                          np.min([meas_vec_det2[0][index_det2_min],
                                  meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_com
                          np.min([meas_vec_det3[0][index_det3_min],
                                  meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_com

        max_phi = np.max([np.max([meas_vec_det1[0][index_det1_min],
                                  meas_vec_det1[0][index_det1_max]])/rspns_mat_det1_com
                          np.max([meas_vec_det2[0][index_det2_min],
                                  meas_vec_det2[0][index_det2_max]])/rspns_mat_det2_com
                          np.max([meas_vec_det3[0][index_det3_min],
                                  meas_vec_det3[0][index_det3_max]])/rspns_mat_det3_com

        # Update the bounds
        ub_phi_e[i] = max_phi
        ub_phi_gam[i] = max_phi

        # Add it to the table for printout
        t.add_row(['{:.1f} kev'.format(rspns_mat_det1_comb[1][0][i]),
                   '{:.3e} e-\n{:.3e} gam'.format(lb_phi_e[i], lb_phi_gam[i]),
                   '{:.3e} e-\n{:.3e} gam'.format(ub_phi_e[i], ub_phi_gam[i])])
        #print t.draw()

        # Define the prior probability densities
        phi_e = pm.Uniform('phi_e', lower = lb_phi_e, upper = ub_phi_e, shape = (ub_phi_e.s
        phi_gam = pm.Uniform('phi_gam', lower = lb_phi_gam, upper = ub_phi_gam, shape = (ub

        # Define the generative models
        M_det1 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_e[0].T)), phi_e) + \
                 theano.tensor.dot(theano.shared(asMat(rspns_mat_det1_gam[0].T)), phi_gam)
        M_det2 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_e[0].T)), phi_e) + \
                 theano.tensor.dot(theano.shared(asMat(rspns_mat_det2_gam[0].T)), phi_gam)
        M_det3 = theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_e[0].T)), phi_e) + \
                 theano.tensor.dot(theano.shared(asMat(rspns_mat_det3_gam[0].T)), phi_gam)

        # Define the likelihood (aka. posterior probability function)
        PPF_det1 = pm.Poisson('PPF_det1', mu = M_det1, observed = theano.shared(meas_vec_de
        PPF_det2 = pm.Poisson('PPF_det2', mu = M_det2, observed = theano.shared(meas_vec_de
        PPF_det3 = pm.Poisson('PPF_det3', mu = M_det3, observed = theano.shared(meas_vec_de
```

1.4 STEP 4 - Sample the posterior for each model

Using the MCMC sampling algorithm

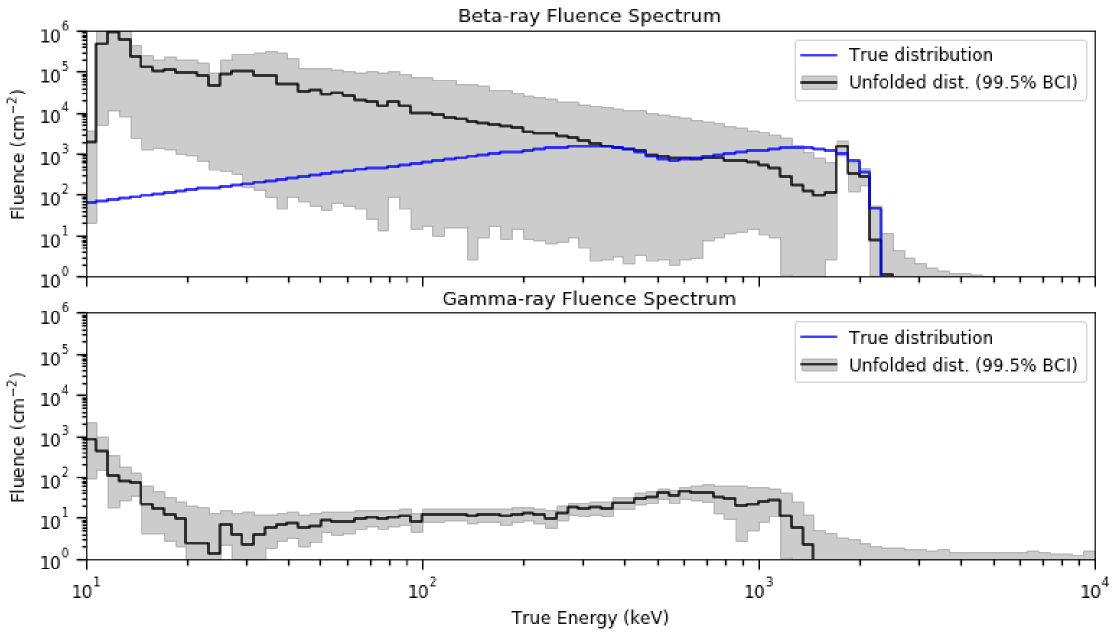
```
In [19]: def plotReconstructedSpectrum(trace, filename = 'Unfolded Fluence Spectrum.jpg'):
             # Create a Pandas dataframe of summary information from the sampling
             df_reco = pm.summary(trace, alpha=0.005)

             # Create a figure and axis to plot the unfolded (aka. reconstructed) beta-ray and g
             fig_reco_vec = plt.figure()

             ax_reco_vec = Grid(fig_reco_vec,
                                111,
                                nrows_ncols=(2, 1),
                                axes_pad=(0.35, 0.35),
                                add_all=True,
                                label_mode = 'L')

             # Plot the unfolded spectrum
             pMeanBeta, = ax_reco_vec[0].plot(sorted(np.append(rspns_mat_det1_e[1][0][:-1], rspn
                                              np.repeat(df_reco[df_reco.index.str.startswith('ph
                                              lw=1.5,
                                              color='black',
                                              linestyle="-",
                                              drawstyle='steps')

             pBCIBeta = ax_reco_vec[0].fill_between(sorted(np.append(rspns_mat_det1_e[1][0][:-1]
                                                    np.repeat(df_reco[df_reco.index.str.startswi
                                    np.repeat(df_reco[df_reco.index.str.startswi
                                    color='black',
                                    alpha=0.2)

pMeanGamma, = ax_reco_vec[1].plot(sorted(np.append(rspns_mat_det1_gam[1][0][:-1], r
                                 np.repeat(df_reco[df_reco.index.str.startswith('p
                                 lw=1.5,
                                 color='black',
                                 linestyle="-",
                                 drawstyle='steps')

pBCIGamma = ax_reco_vec[1].fill_between(sorted(np.append(rspns_mat_det1_gam[1][0][:
                                        np.repeat(df_reco[df_reco.index.str.startsw
                                        np.repeat(df_reco[df_reco.index.str.startsw
                                        color='black',
                                        alpha=0.2)

# Plot the truth spectrum (if known)
pTruthBeta, = ax_reco_vec[0].plot(sorted(np.append(truth_vec_det1_e[1][0][:-1], tru
                                 np.repeat(truth_vec_det1_e[0], 2),
                                 lw=1.5,
                                 color='blue',
                                 linestyle="-",
                                 drawstyle='steps')

pTruthGamma, = ax_reco_vec[1].plot(sorted(np.append(truth_vec_det1_gam[1][0][:-1],
                                  np.repeat(truth_vec_det1_gam[0], 2),
                                  lw=1.5,
                                  color='blue',
                                  linestyle="-",
                                  drawstyle='steps')

# Find min and max y value for scaling the plot
y_lim_up = np.max([truth_vec_det1_e[0].max(),
                   truth_vec_det1_gam[0].max(),
                   df_reco[df_reco.index.str.startswith('phi_e')]['hpd_99.75'].max(
                   df_reco[df_reco.index.str.startswith('phi_gam')]['hpd_99.75'].ma
y_lim_up = 10**np.ceil(np.abs(np.log10(y_lim_up)))
y_lim_up = 1E6
y_lim_down = y_lim_up/1E6

# Plot statistics text
print('\nStatistics from reconstructed Beta-ray Fluence Spectrum \
       \n-------------------------------------------------------- \
       \nRMSE \t{:.2E} ({:.2E} - {:.2E}) \
       \nMAE \t{:.2E} ({:.2E} - {:.2E})'
      .format(np.sqrt(((df_reco[df_reco.index.str.startswith('phi_e')]['mean'] - tr
              np.sqrt(((df_reco[df_reco.index.str.startswith('phi_e')]['hpd_0.25']
                      np.sqrt(((df_reco[df_reco.index.str.startswith('phi_e')]['hpd_99.75']
                      np.abs(truth_vec_det1_e[0] - df_reco[df_reco.index.str.startswith('ph
                      np.abs(truth_vec_det1_e[0] - df_reco[df_reco.index.str.startswith('ph
                      np.abs(truth_vec_det1_e[0] - df_reco[df_reco.index.str.startswith('ph

        print('\nStatistics from reconstructed Beta-ray Fluence Spectrum \
               \n------------------------------------------------------ \
               \nRMSE \t{:.2E} ({:.2E} - {:.2E}) \
               \nMAE \t{:.2E} ({:.2E} - {:.2E})'
              .format(np.sqrt(((df_reco[df_reco.index.str.startswith('phi_gam')]['mean'] -
                      np.sqrt(((df_reco[df_reco.index.str.startswith('phi_gam')]['hpd_0.25'
                      np.sqrt(((df_reco[df_reco.index.str.startswith('phi_gam')]['hpd_99.75
                      np.abs(truth_vec_det1_gam[0] - df_reco[df_reco.index.str.startswith('
                      np.abs(truth_vec_det1_gam[0] - df_reco[df_reco.index.str.startswith('
                      np.abs(truth_vec_det1_gam[0] - df_reco[df_reco.index.str.startswith('

        # Figure properties
        ax_reco_vec[0].set_xlabel('True Energy (keV)')
        ax_reco_vec[0].set_ylabel('Fluence (cm$^{-2}$)')
        ax_reco_vec[0].set_xlim(min(rspns_mat_det1_e[1][0]), max(rspns_mat_det1_e[1][0]))
        ax_reco_vec[0].set_ylim(y_lim_down, y_lim_up)
        ax_reco_vec[0].set_xscale('log')
        ax_reco_vec[0].set_yscale('log')
        ax_reco_vec[0].set_title('Beta-ray Fluence Spectrum')
        ax_reco_vec[0].legend([pTruthBeta, (pBCIBeta, pMeanBeta)], ['True distribution','Un

        ax_reco_vec[1].set_xlabel('True Energy (keV)')
        ax_reco_vec[1].set_ylabel('Fluence (cm$^{-2}$)')
        ax_reco_vec[1].set_xlim(min(rspns_mat_det1_gam[1][0]),max(rspns_mat_det1_gam[1][0])
        ax_reco_vec[1].set_ylim(y_lim_down, y_lim_up)
        ax_reco_vec[1].set_xscale('log')
        ax_reco_vec[1].set_yscale('log')
        ax_reco_vec[1].set_title('Gamma-ray Fluence Spectrum')
        ax_reco_vec[1].legend([pTruthGamma, (pBCIGamma, pMeanGamma)], ['True distribution',

        # Fine-tune figure
        fig_reco_vec.set_tight_layout(True)

        # Save the figure
        plt.savefig(filename, bbox_inches="tight")

        # Show the figure
        plt.show(fig_reco_vec)
        plt.close(fig_reco_vec)
```

```
In [20]: DRAWS = 100000
         TUNE = 250000
```

Sampling the posterior from model_det1

```
In [21]: with model_det1:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
             approxADVI = pm.fit(n=TUNE,
                                 method='fullrank_advi',
                                 start=pm.find_MAP(model = model_det1),
                                 callbacks=[CheckParametersConvergence(every=1000,
                                                                       diff='absolute',
                                                                       tolerance = 5E-2)])

             # Draw samples from ADVI fit
             trace = approxADVI.sample(draws=DRAWS)

             plotReconstructedSpectrum(trace, isotope + ' - ' + det1 + ' - Unfolded Fluence Spec

         # Free up memory after sampling
         gc.collect()
```

```
Sampling the posterior distribution using ADVI ...


logp = -2,241.4, ||grad|| = 0.051689: 100%|| 574/574 [00:00<00:00, 1233.65it/s]
Average Loss = 1,630.2:  34%|      | 84977/250000 [04:43<09:10, 300.02it/s]
Convergence archived at 85000
Interrupted at 84,999 [33%]: Average Loss = 1,919
/usr/local/lib/python2.7/dist-packages/pandas/core/computation/check.py:17: UserWarning: The ins
The minimum supported version is 2.4.6

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         1.36E+05 (1.75E+03 - 2.79E+05)
MAE         4.21E+04 (8.79E+02 - 1.06E+05)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         1.01E+02 (2.13E+01 - 2.62E+02)
MAE         2.71E+01 (9.06E+00 - 6.16E+01)
```



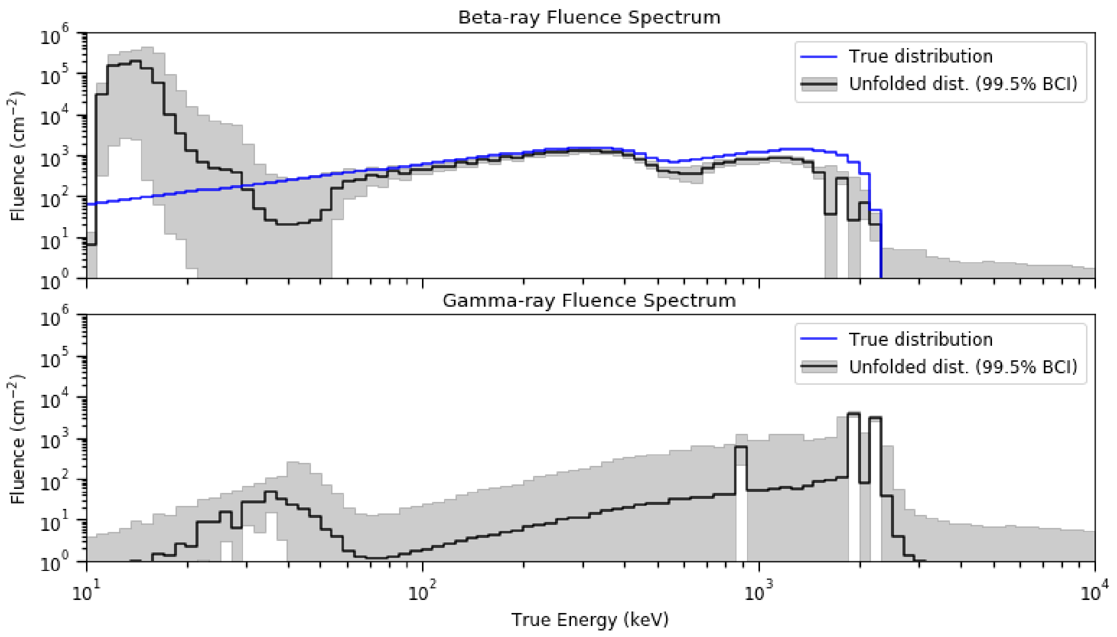
```
Out[21]: 58924
```

Sampling the posterior from model_det2

```
In [22]: with model_det2:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
             approxADVI = pm.fit(n=TUNE,
                                 method='fullrank_advi',
                                 start=pm.find_MAP(model = model_det2),
                                 callbacks=[CheckParametersConvergence(every=1000,
                                                                       diff='absolute',
                                                                       tolerance = 5E-2)])

             # Draw samples from ADVI fit
             trace = approxADVI.sample(draws=DRAWS)

             plotReconstructedSpectrum(trace, isotope + ' - ' + det2 + ' - Unfolded Fluence Spec

         # Free up memory after sampling
         gc.collect()
```

```
Sampling the posterior distribution using ADVI ...
```

```
logp = -2,300.7, ||grad|| = 0.093117: 100%|| 3350/3350 [00:02<00:00, 1206.41it/s]
Average Loss = 2,151:  20%|        | 48971/250000 [02:38<10:51, 308.75it/s]
Convergence archived at 49000
Interrupted at 48,999 [19%]: Average Loss = 3,691.6



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          3.51E+04 (5.57E+02 - 8.65E+04)
MAE          8.52E+03 (3.50E+02 - 2.27E+04)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          5.18E+02 (4.36E+02 - 8.16E+02)
MAE          9.91E+01 (6.75E+01 - 3.54E+02)
```

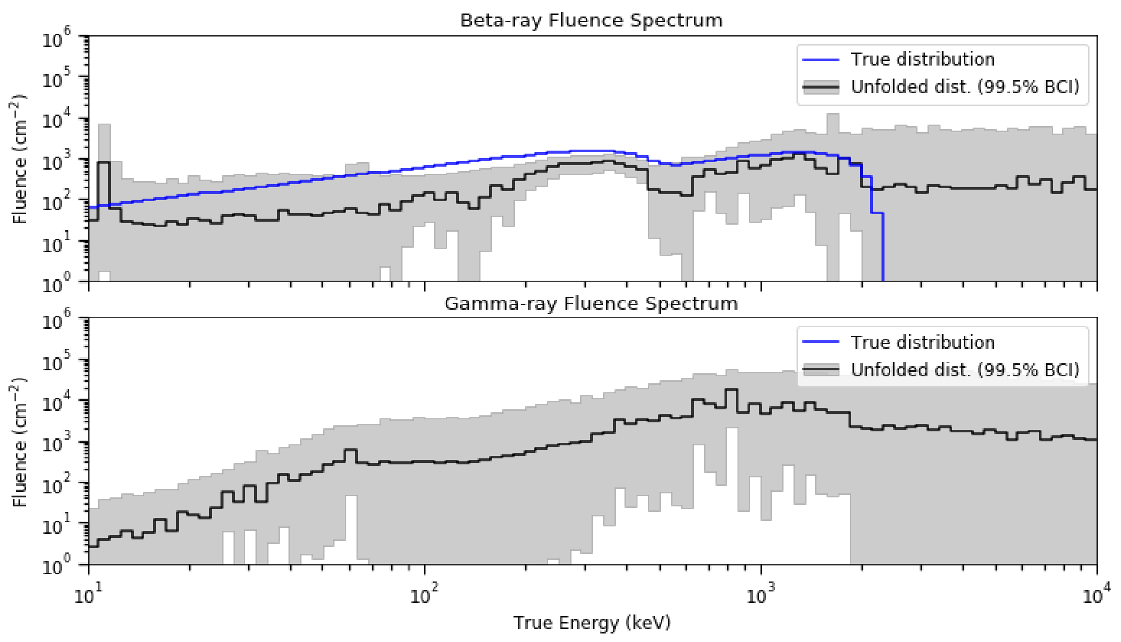


```
Out[22]: 36804
```

Sampling the posterior from model_det3

```
In [23]: with model_det3:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
```

```
            approxADVI = pm.fit(n=TUNE,
                                method='fullrank_advi',
                                start=pm.find_MAP(model = model_det3),
                                callbacks=[CheckParametersConvergence(every=1000,
                                                                      diff='absolute',
                                                                      tolerance = 5E-2)])

            # Draw samples from ADVI fit
            trace = approxADVI.sample(draws=DRAWS)

            # Plot the reconstructed spectrum
            plotReconstructedSpectrum(trace, isotope + ' - ' + det3 + ' - Unfolded Fluence Spec

        # Free up memory after sampling
        gc.collect()

Sampling the posterior distribution using ADVI ...


logp = -2,696.5, ||grad|| = 0.075616: 100%|| 601/601 [00:00<00:00, 1295.77it/s]
Average Loss = 9,767.6:  22%|       | 54993/250000 [02:54<10:20, 314.40it/s]
Convergence archived at 55000
Interrupted at 54,999 [21%]: Average Loss = 4,810.1



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         4.51E+02 (6.66E+02 - 3.08E+03)
MAE         3.74E+02 (5.00E+02 - 1.89E+03)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         3.49E+03 (2.57E+02 - 3.07E+04)
MAE         2.00E+03 (5.10E+01 - 2.20E+04)
```

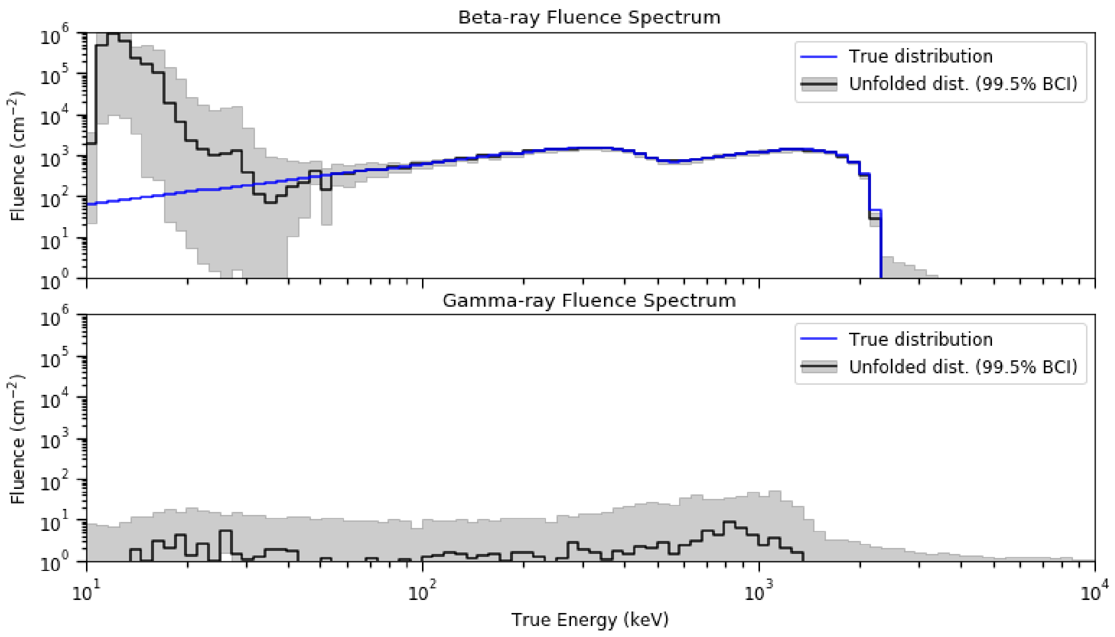


```
Out[23]: 35962
```

Sampling the posterior from model_det1_det2

```
In [24]: with model_det1_det2:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
             approxADVI = pm.fit(n=TUNE,
                                 method='fullrank_advi',
                                 start=pm.find_MAP(model = model_det1_det2),
                                 callbacks=[CheckParametersConvergence(every=1000,
                                                                       diff='absolute',
                                                                       tolerance = 5E-2)])


             # Draw samples from ADVI fit
             trace = approxADVI.sample(draws=DRAWS)

             # Plot the reconstructed spectrum
             plotReconstructedSpectrum(trace, isotope + ' - ' + det1 + ' - ' + det2 + ' - Unfold

         # Free up memory after sampling
         gc.collect()

Sampling the posterior distribution using ADVI ...
```

```
logp = -2,718.4, ||grad|| = 0.31967: 100%|| 2394/2394 [00:02<00:00, 1076.86it/s]
Average Loss = 2,538.3:  17%|         | 41988/250000 [02:13<11:03, 313.74it/s]
Convergence archived at 42000
Interrupted at 41,999 [16%]: Average Loss = 5,427.7



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          1.32E+05 (1.53E+03 - 2.68E+05)
MAE          2.85E+04 (4.06E+02 - 6.30E+04)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          2.09E+00 (1.97E-01 - 1.69E+01)
MAE          1.44E+00 (4.50E-02 - 1.27E+01)
```



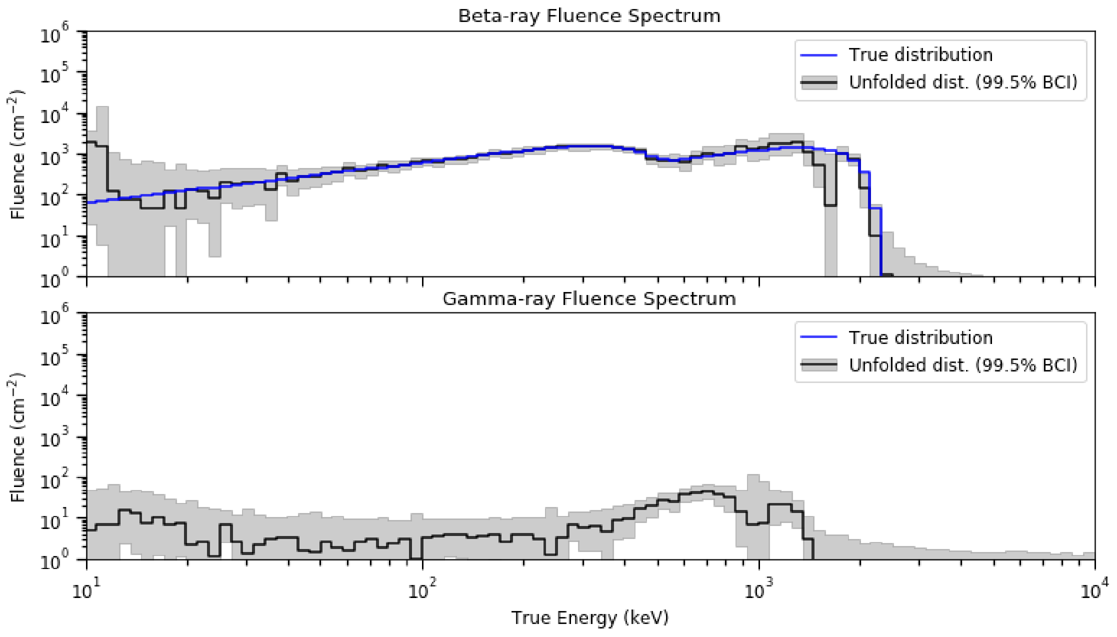
```
Out[24]: 38099
```

Sampling the posterior from model_det1_det3

```
In [25]: with model_det1_det3:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
```

```
            approxADVI = pm.fit(n=TUNE,
                                method='fullrank_advi',
                                start=pm.find_MAP(model = model_det1_det3),
                                callbacks=[CheckParametersConvergence(every=1000,
                                                                      diff='absolute',
                                                                      tolerance = 5E-2)])

            # Draw samples from ADVI fit
            trace = approxADVI.sample(draws=DRAWS)

            # Plot the reconstructed spectrum
            plotReconstructedSpectrum(trace, isotope + ' - ' + det1 + ' - ' + det3 + ' - Unfold

        # Free up memory after sampling
        gc.collect()
```

```
Sampling the posterior distribution using ADVI ...


logp = -3,109.1, ||grad|| = 0.18467: 100%|| 1742/1742 [00:01<00:00, 1147.27it/s]
Average Loss = 2,836.8:  18%|        | 45968/250000 [02:22<10:34, 321.68it/s]
Convergence archived at 46000
Interrupted at 45,999 [18%]: Average Loss = 3,607.5



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         3.14E+02 (2.76E+02 - 1.69E+03)
MAE         1.14E+02 (1.79E+02 - 5.08E+02)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         1.19E+01 (6.49E+00 - 2.92E+01)
MAE         6.63E+00 (2.51E+00 - 1.98E+01)
```

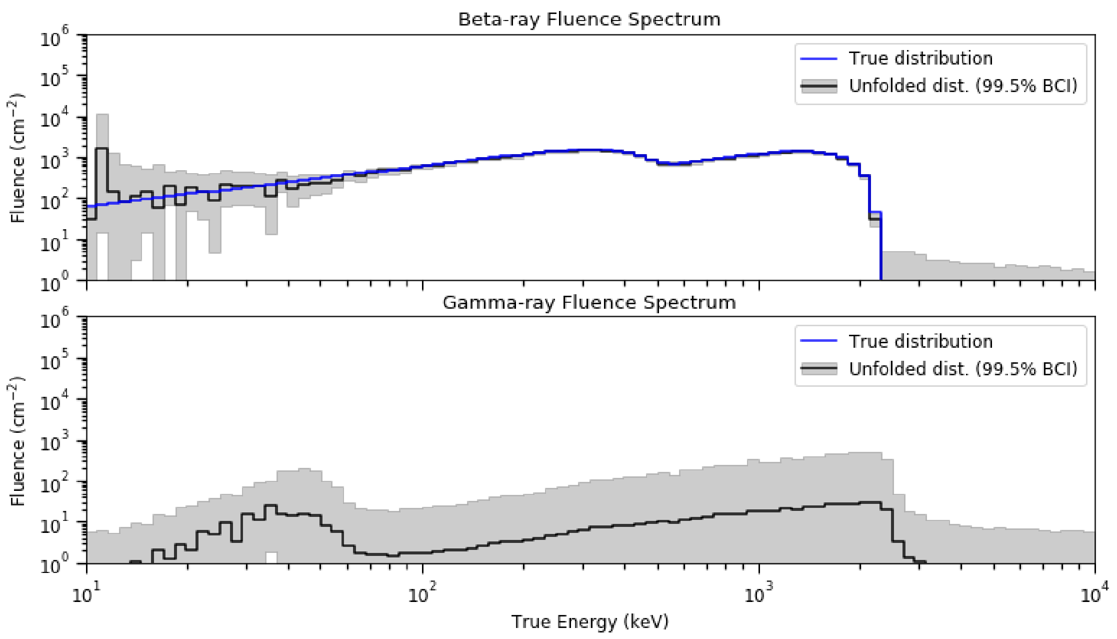


```
Out[25]: 38917
```

Sampling the posterior from model_det2_det3

```
In [26]: with model_det2_det3:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
             approxADVI = pm.fit(n=TUNE,
                                 method='fullrank_advi',
                                 start=pm.find_MAP(model = model_det2_det3),
                                 callbacks=[CheckParametersConvergence(every=1000,
                                                                       diff='absolute',
                                                                       tolerance = 5E-2)])

             # Draw samples from ADVI fit
             trace = approxADVI.sample(draws=DRAWS)

             # Plot the reconstructed spectrum
             plotReconstructedSpectrum(trace, isotope + ' - ' + det2 + ' - ' + det2 + ' - Unfold

         # Free up memory after sampling
         gc.collect()

Sampling the posterior distribution using ADVI ...
```

```
logp = -3,150.6, ||grad|| = 0.15489: 100%|| 2059/2059 [00:01<00:00, 1099.08it/s]
Average Loss = 2,836.5:  18%|        | 44983/250000 [02:27<11:13, 304.33it/s]
Convergence archived at 45000
Interrupted at 44,999 [17%]: Average Loss = 5,632.6



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          1.67E+02 (1.01E+02 - 1.27E+03)
MAE          4.18E+01 (8.53E+01 - 2.44E+02)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE          1.12E+01 (2.22E-01 - 1.76E+02)
MAE          7.68E+00 (4.30E-02 - 1.12E+02)
```



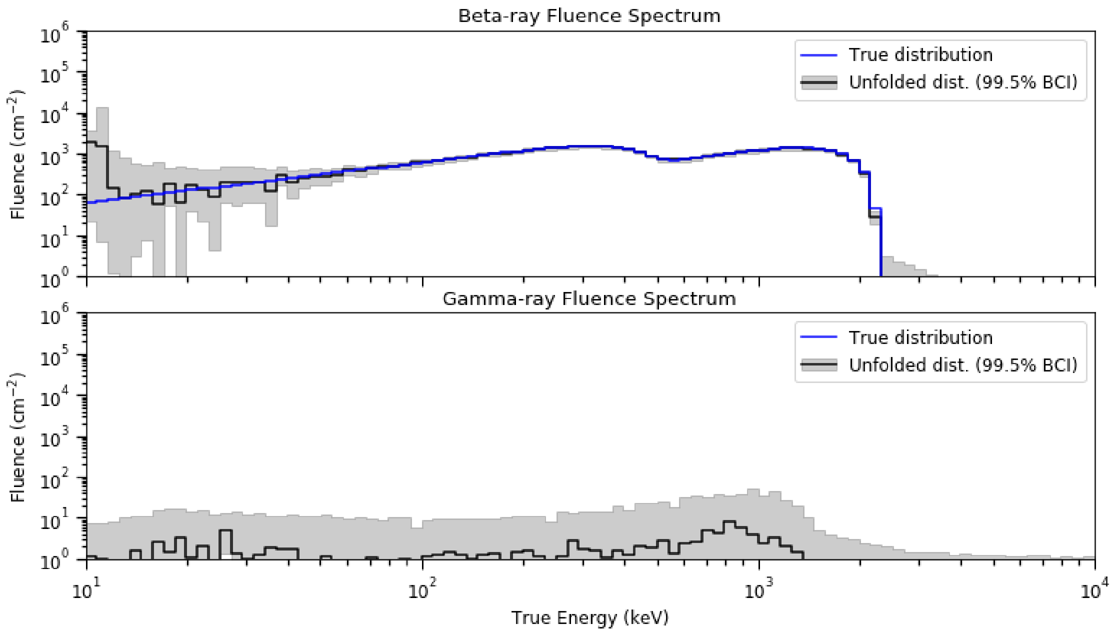
```
Out[26]: 38935
```

Sampling the posterior from model_det1_det2_det3

```
In [27]: with model_det1_det2_det3:
             print 'Sampling the posterior distribution using ADVI ...'
             from pymc3.variational.callbacks import CheckParametersConvergence

             # Fit the model using ADVI
```

```
            approxADVI = pm.fit(n=TUNE,
                                method='fullrank_advi',
                                start=pm.find_MAP(model = model_det1_det2_det3),
                                callbacks=[CheckParametersConvergence(every=1000,
                                                                      diff='absolute',
                                                                      tolerance = 5E-2)])

            # Draw samples from ADVI fit
            trace = approxADVI.sample(draws=DRAWS)

            # Plot the reconstructed spectrum
            plotReconstructedSpectrum(trace, isotope + ' - ' + det1 + ' - ' + det2 + ' - ' + de

        # Free up memory after sampling
        gc.collect()
```

```
Sampling the posterior distribution using ADVI ...


logp = -3,506.7, ||grad|| = 0.43646: 100%|| 1564/1564 [00:01<00:00, 996.62it/s]
Average Loss = 3,423.2:  17%|        | 42992/250000 [02:22<11:25, 301.97it/s]
Convergence archived at 43000
Interrupted at 42,999 [17%]: Average Loss = 6,697.6



Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         2.51E+02 (1.04E+02 - 1.54E+03)
MAE         5.69E+01 (8.88E+01 - 3.31E+02)

Statistics from reconstructed Beta-ray Fluence Spectrum
-------------------------------------------------------
RMSE         1.91E+00 (1.54E-01 - 1.62E+01)
MAE         1.33E+00 (3.15E-02 - 1.22E+01)
```


Beta-ray Fluence Spectrum
True distribution
Unfolded dist. (99.5% BCI)
Fluence (cm^{-2})
Gamma-ray Fluence Spectrum
True Energy (keV)
10^1
10^2
10^3
10^4

```
Out[27]: 76995
```