Neutrino Oscillations and Detector Response

Authors: Jurjan Bootsma, Bouke Jung

This example notebook contains a class which can do calculations for all kinds of neutrino oscillations and detector responses of the ORCA detector (like fluxes, oscillation probabilities, cross sections, effective volume and interaction rates). After the class it makes plots using the most important functions, mainly in 2D with the logarithmic energy in GeV on the x-axis and the cos θ on the y-axis.

In [35]:
```
import matplotlib.pyplot as plt
import numpy as np

import km3pipe as kp
import km3modules as km
import km3io
from km3net_testdata import data_path
import jppy
import km3flux
from matplotlib import colors
```

In [36]:
```
class DetectorResponseModeller:
    """
    This class helps calculating neutrino fluxes, neutrino oscillations and d
    """
    def __init__(self, flavour, CParity):
        """
        Creating the object

        Parameters
        ----------
        flavour: int
            This should contain the pdgid of the neutrino (12,14,16)
        CParity: int
            This should contain the CParity of the incoming neutrino (1 for n
            anti-neutrinos).

        Member Variables
        ----------------
        flavour: int
            This should contain the pdgid of the neutrino (12,14,16)
        CParity: int
            This should contain the CParity of the incoming neutrino (1 for n
            anti-neutrinos).
        E: numpy.ndarray
            This contains all the energies from the Monte Carlo file for this
        costheta: numpy.ndarray
            This contains all the zenith angles from the Monte Carlo file for
        cs: numpy.ndarray
            This contains all the cross sections from the Monte Carlo file fo
        Emin: float
            This is the lowest energy from the data.
        Emax: float
            This is the highest energy from the data.
        costheta_min: float
            This is the lowest zenith angle from the data.
        costheta_max: float
            This is the highest zenith angle from the data.
        Ebreak: float
```

```
            This is the boundary energy between the 2 used files.
        alpha_low: float
            This is the generated energy slope of the file with the lower ene
        alpha_high: float
            This is the generated energy slope of the file with the higher en
        Ngen: int
            This is the total amount of neutrinos produced in both files.
        Vgen: int
            This is the volume of the entire detector.
        weights: numpy.ndarray
            These are the weights per event which contain information about r
            cross section and effective volume and can be used to get the rec
            interaction rate.
        flux_calc: km3flux.flux.HondaFlux
            This object calculates the atmospheric flux from Honda tables (fr
            solar minimum).
        interpolator: jppy.oscprob.JppyOscProbInterpolator
            This object calculates the oscillation probabilities from the ene
            oscillation parameters.

        """

        self.flavour = flavour
        self.CParity = CParity

        if flavour == 12:
            file_low = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9m
            file_high = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9
        elif flavour == 14:
            file_low = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9m
            file_high = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9
        elif flavour == 16:
            file_low = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9m
            file_high = "/dcache/km3net/mc/atm_neutrino/KM3NeT_ORCA_115_20m_9

        f_low = km3io.OfflineReader(data_path(file_low)) #Read the file
        f_high = km3io.OfflineReader(data_path(file_high))
        pdgid_nu = np.concatenate((np.array(f_low.mc_tracks.pdgid[:,0]),np.ar

        if CParity == 1:
            boolean = (pdgid_nu > 0)
        elif CParity == -1:
            boolean = (pdgid_nu < 0)

        #Extract some specific data
        self.E = np.concatenate((np.array(f_low.mc_tracks.E[:,0]),np.array(f_

        dir_x = np.concatenate((np.array(f_low.mc_tracks.dir_x[:,0]),np.array
        dir_y = np.concatenate((np.array(f_low.mc_tracks.dir_y[:,0]),np.array
        dir_z = np.concatenate((np.array(f_low.mc_tracks.dir_z[:,0]),np.array
        neutrino_dir = np.array([dir_x,dir_y,dir_z])
        x_len,y_len = np.shape(neutrino_dir)
        neutrino_dir = neutrino_dir.reshape((y_len,x_len))
        self.costheta = np.cos(kp.math.theta(neutrino_dir))

        self.cs = np.concatenate((np.array(f_low.w2list[:,2]),np.array(f_high

        self.Emin = f_low.header.cut_nu.Emin
        self.Emax = f_high.header.cut_nu.Emax
        self.costheta_min = f_low.header.cut_nu.cosTmin
        self.costheta_max = f_low.header.cut_nu.cosTmax
        self.Ebreak = f_low.header.cut_nu.Emax
        self.alpha_low = f_low.header.spectrum.alpha
        self.alpha_high = f_high.header.spectrum.alpha
```

```
        self.Ngen = f_low.header.genvol.numberOfEvents+f_high.header.genvol.n
        self.Vgen = f_low.header.genvol.volume

        weights_low = np.array(f_low.w[:,1])/(f_low.header.genvol.numberOfEve
        weights_high = np.array(f_high.w[:,1])/(f_high.header.genvol.numberOf
        self.weights = np.concatenate((weights_low, weights_high))[boolean]

        honda = km3flux.flux.Honda()
        self.flux_calc = honda.flux(2014, "Frejus", solar="min", averaged="az

        input_file = "/data/km3net/users/jurjanbootsma/joscprobtableDRM.dat"
        self.interpolator = jppy.oscprob.JppyOscProbInterpolator(input_file)

    def atmospheric_flux(self, E, costheta):
        """
        This function calculates the atmospheric neutrino flux (without takin
        account).

        Parameters
        ----------
        E: float, numpy.ndarray
            The energy or energies from which you want to know the flux.
        costheta: float, numpy.ndarray
            The zenith angle from which you want to know the flux.
        """

        #First identify the type of neutrino in a string
        if self.flavour == 12:
            if self.CParity == 1:
                neutrino = "nue"
            elif self.CParity == -1:
                neutrino = "anue"
        elif self.flavour == 14:
            if self.CParity == 1:
                neutrino = "numu"
            elif self.CParity == -1:
                neutrino = "anumu"
        elif self.flavour == 16:
            print("There is no tau neutrino flux.")
            return None

        E_grid, costheta_grid = np.meshgrid(E, costheta)
        flux_nu = self.flux_calc[neutrino](E_grid.flatten(),costheta_grid.fla
        flux_nu = flux_nu.reshape(np.shape(E_grid))

        return flux_nu

    def cs_bin(self, quantity, bound1, bound2):
        """
        This function makes a mask of values between 2 boundaries and then ca
        of the cross section within that bin. The quantity is mostly energy a
        of course be used for anything.

        Parameters:
        -----------
        quantity: numpy.ndarray
            The quantity in which you want to bin (normally E or costheta).
        bound1: float
            The lower boundary of the bin of the specified quantity.
        bound2: float
            The upper boundary of the bin of the specified quantity.
        """

        boolean = (quantity>bound1)*(quantity<bound2)
```

```
        return np.mean(self.cs[boolean])

    def cross_section_bin(self,E=False,bins=1):
        """
        This function calculates the cross section and the corresponding ener

        Parameters
        ----------
        E: bool, float
            This is the energy for which you want to know the cross section. 
            when looking at the cross section as a function of cosine of thet
            want to look at a certain energy).
        bins: int
            This parameter determines how many bins you want to look at the c
        """

        Es = np.logspace(np.log10(self.Emin),np.log10(self.Emax),num=bins+1,b
        i = 0
        cs_array = np.zeros(bins)
        while i < bins:
            cs_array[i] = self.cs_bin(self.E,Es[i],Es[i+1])
            if E >= Es[i] and E <= Es[i+1]:
                E_index = i
            i=i+1

        if E == False:
            Es_log = np.log10(Es)
            step = (Es_log[1] - Es_log[0])/2
            Es_log_center = (Es_log + step)[:-1]
            E_center = 10**Es_log_center

            return E_center, cs_array

        elif E != False:
            costhetas = np.linspace(self.costheta_min,self.costheta_max,num=b
            costheta = costhetas[1:] - (costhetas[1]-costhetas[0])/2
            cs_costheta = cs_array[E_index] * np.ones(bins)

            return costheta, cs_costheta


    def oscillation_probability(self, E, costheta, flavour_in, flavour_out, C
        """
        This function calculates the oscillation probability of a flavour neu
        using joscprob from the jppy library as a function of energy and the 

        Parameters
        ----------
        E: int, float, numpy.ndarray
            This is the energy from which you want to know the oscillation pr
        costheta: int, float, numpy.ndarray
            This is the cosine of theta, or the zenith angle, from which you 
            oscillation probability.
        flavour_in: int
            This is the flavour of the neutrino before the oscillation starts
        flavour_out: int
            This is the flavour of neutrino after the oscillation (12,14,16).
        CParity: int
            This determines if the neutrinos you are looking at are normal or
        """

        channel = jppy.oscprob.JOscChannel(flavour_in, flavour_out, CParity)
        parameters = jppy.oscprob.JOscParameters(dM21sq=7.42*10**-5,dM31sq=2.
                                                  sinsqTh12=0.304,sinsqTh13=0.
```

```
        if type(costheta) == int or type(costheta) == float and type(E) == np
            costhetas = np.tile(costheta,len(E))
            Es = E
            probs = self.interpolator(parameters, channel, Es, costhetas)
        elif type(E) == int or type(E) == float and type(costheta) == np.ndar
            Es = np.tile(E,len(costheta))
            costhetas = costheta
            probs = self.interpolator(parameters, channel, Es, costhetas)
        elif type(E) == np.ndarray and type(costheta) == np.ndarray and len(E
            Es, costhetas = np.meshgrid(E,costheta)
            probs_1D = self.interpolator(parameters, channel, Es.flatten(), c
            probs = np.reshape(probs_1D,(len(costheta),len(E)))
        elif type(E) == np.ndarray and type(costheta) == np.ndarray and len(E
            probs = self.interpolator(parameters, channel, E, costheta)

        return probs

    def flux_oscillated(self, E, costheta):
        """
        This is the atmospheric neutrino flux after the oscillations arriving

        Parameters:
        ----------
        E: int, float, numpy.ndarray
            This is the energy from which you want to know the atmospheric os
        costheta: int, float, numpy.ndarray
            This is the cosine of theta, or the zenith angle, from which you
            atmospheric oscillated flux.
        """
        if self.CParity == 1:
            flux_electron = self.flux_calc["nue"](E,costheta)
            flux_muon = self.flux_calc["numu"](E,costheta)
        elif self.CParity == -1:
            flux_electron = self.flux_calc["anue"](E,costheta)
            flux_muon = self.flux_calc["anumu"](E,costheta)

        probs_e = self.oscillation_probability(E, costheta, 12, self.flavour,
        probs_m = self.oscillation_probability(E, costheta, 14, self.flavour,

        flux_osc = probs_e*flux_electron + probs_m*flux_muon

        return flux_osc

    def TrueInteractionRate_per_mass(self,E=0,costheta=0):
        """
        This calculates the true interaction rate (per second per radian per
        gives the interaction rate as a function of E for cosine of theta of

        Parameters:
        -----------
        E: int,float
            This is the energy from which you want to know the true interacti
            False, the function takes the energies from the Monte Carlo file
            function of energy. If it has a value, then it will be calculated
            of theta for that particular energy.
        costheta: int,float
            This is the cosine of theta from which you want to know the true
            mass. If False, the function takes the zenith angles from the Mon
            calculates it as a function of zenith angle. If it has a value, t
            as a function of energy for that particular zenith angle.
        """
        nucleons_number = 1/((10*1.6726231*10**-27 + 8*1.67493*10**-27)/18)
```

```
        if E == 0 and costheta != 0:
            flux = np.array(self.flux_oscillated(self.E,costheta))
            interaction_rate = nucleons_number * self.cs * flux
        elif E == 0 and costheta == 0:
            flux = np.array(self.flux_oscillated(self.E,costheta))
            interaction_rate = nucleons_number * self.cs * flux
        elif E != 0 and costheta == 0:
            flux = np.array(self.flux_oscillated(E,self.costheta))
            interaction_rate = nucleons_number * self.cs * flux

        return interaction_rate

    def TrueInteractionRate_per_mass_bin(self,E=None,costheta=None,bintype="2
        """
        This calculates the true interaction rate (per second per radian per 
        bins.

        Parameters:
        -----------
        E: bool, float
            This is the energy from which you want to know the true interacti
            False, the function takes the energies from the Monte Carlo file 
            function of energy. If it has a value, then it will be calculated
            of theta for that particular energy.
        costheta: bool, float
            This is the cosine of theta from which you want to know the true 
            mass. If False, the function takes the zenith angles from the Mon
            calculates it as a function of zenith angle. If it has a value, t
            as a function of energy for that particular zenith angle.
        bintype: string
            This specifies as a function of what you want to know the true in
            as a function of E, 'costheta' means as a function of costheta an
            of both E and costheta which gives a 2D array.
        bins: int
            The amount of bins you want. If E = False and costheta =! False, 
            bins, if E =! False and costheta = False, this is about cosine bi
            this is about 2D bins of energy and cosine of theta.
        """
        nucleons_number = 1/((10*1.6726231*10**-27 + 8*1.67493*10**-27)/18)

        if bintype == "E":
            E_bin, cs = self.cross_section_bin(bins=bins)
            flux = np.array(self.flux_oscillated(E_bin,costheta))
            interaction_rate = nucleons_number * cs * flux

            return E_bin, interaction_rate

        elif bintype == "costheta":
            costheta_bin, cs = self.cross_section_bin(E=E,bins=bins)
            flux = np.array(self.flux_oscillated(E,costheta_bin))
            interaction_rate = nucleons_number * cs * flux

            return costheta_bin, interaction_rate

        elif bintype == "2D":
            E_wanted = np.logspace(0,2,num=bins)
            costheta_wanted = np.linspace(-1,1,num=bins)
            flux_all = self.flux_oscillated(self.E,self.costheta)

            i=0
            indexes = np.zeros(bins*bins)
            flux_array = np.zeros(bins*bins)
            cs_array = np.zeros(bins*bins)
```

```
            while i < bins:
                boolean_theta = (self.costheta>costheta_wanted[i]-1/bins)*(se
                j=0
                while j < bins:
                    boolean_E = (np.log10(self.E) > np.log10(E_wanted[j])-1/b
                    boolean_res = boolean_E * boolean_theta
                    if np.sum(boolean_res) == 0:
                        flux_array[i*bins+j] = np.nan
                        cs_array[i*bins+j] = np.nan
                    else:
                        flux = flux_all[boolean_res]
                        flux_array[i*bins+j] = np.sum(flux)/len(flux)

                        cs_bin = self.cs[boolean_res]
                        cs_array[i*bins+j] = np.sum(cs_bin)/len(cs_bin)

                    j=j+1
                i=i+1

            interaction_rate = nucleons_number * cs_array * flux_array
            interaction_rate = np.reshape(interaction_rate, (bins,bins))
            interaction_rate = interaction_rate[::-1,:]
            return interaction_rate

    def Ndet_bin(self,quantity, bound1, bound2):
        """
        This function makes a mask of values between 2 boundaries and then ca
        are between those boundaries. This is then the amount of detected eve
        cosine of theta bin.

        Parameters:
        -----------
        quantity: numpy.ndarray
            The quantity in which you want to bin (normally E or costheta).
        bound1: float
            The lower boundary of the bin of the specified quantity.
        bound2: float
            The upper boundary of the bin of the specified quantity.
        """
        bool1 = (quantity > bound1)
        bool2 = (quantity < bound2)
        N = np.count_nonzero((quantity*bool1*bool2))
        return N

    def Ndet_bin_2D(self,Edet,E1,E2,costheta,costheta1,costheta2):
        """
        This function makes a mask of values between 2 boundaries of one quan
        another qunaitity and then calculates how many events are between tho
        then the amount of detected events within an energy and cosine of the

        Parameters:
        -----------
        Edet: numpy.ndarray
            The energy in which you want to bin.
        E1: float
            The lower boundary of the energy bin of the specified quantity.
        E2: float
            The upper boundary of the energy bin of the specified quantity.
        costheta: numpy.ndarray
            The cosine of theta in which you want to bin.
        costheta1: float
            The lower boundary of the cosine of theta bin of the specified qu
        costheta2: float
            The upper boundary of the cosine of theta bin of the specified qu
```

```
        """
        E_bool1 = (Edet > E1)
        E_bool2 = (Edet < E2)
        E_bool = (Edet*E_bool1*E_bool2)
        costheta_bool1 = (costheta > costheta1)
        costheta_bool2 = (costheta < costheta2)
        costheta_bool = (costheta*costheta_bool1*costheta_bool2)
        N = np.count_nonzero(E_bool*costheta_bool)
        return N

    def Ngen_bin(self,E1, E2):
        """
        This function calculates the amount of events within an energy bin ge

        Parameters:
        -----------
        E1: float
            The lower boundary of the energy bin.
        E2: float
            The upper boundary of the energy bin.
        """
        if E1>=self.Ebreak:
            alpha = self.alpha_high
        else:
            alpha = self.alpha_low

        return self.Ngen * ((E2**(alpha+1)-E1**(alpha+1))/((self.Ebreak**alph

    def Effective_Volume(self,bintype,bins):
        """
        This function calculates the effective volume of the ORCA detector.

        Parameters:
        -----------
        bintype: string
            This specifies as a function of what you want to know the effecti
            function of E, 'costheta' means as a function of costheta and '2D
            of both E and costheta which gives a 2D array.
        bins: int
            The amount of bins you want for the effective volume.
        """
        #Values from the header

        if bintype == "E":
            Es = np.logspace(np.log10(self.Emin),np.log10(self.Emax),num=bins
            volume_array = np.zeros(bins)
            i = 0
            while i < bins:
                volume_eff = self.Vgen * (self.Ndet_bin(self.E,Es[i],Es[i+1])
                volume_array[i] = volume_eff
                i=i+1

            Es_log = np.log10(Es)
            step = (Es_log[1] - Es_log[0])/2
            Es_log_center = (Es_log + step)[:-1]
            E_bin = 10**Es_log_center

            return E_bin, volume_array

        elif bintype == "costheta":
            costhetas = np.linspace(self.costheta_min,self.costheta_max,num=b
            volume_array = np.zeros(bins)
            i = 0
            while i < bins:
```

```
                volume_eff = self.Vgen * (self.Ndet_bin(self.costheta,costhet
                volume_array[i] = volume_eff
                i=i+1

            costheta_bin = costhetas[1:]-(costhetas[1]-costhetas[0])/2

            return costheta_bin, volume_array

        elif bintype == "2D":
            Es = np.logspace(np.log10(self.Emin),np.log10(self.Emax),num=bins
            costhetas = np.linspace(self.costheta_min,self.costheta_max,num=b

            volume_array = np.zeros((bins,bins))
            i = 0
            while i < bins:
                j = 0
                while j < bins:
                    volume_array[j,i] = self.Vgen * (self.Ndet_bin_2D(self.E,
                    j = j+1
                i=i+1
            return volume_array

    def TrueInteractionRate_detector(self,E=False,costheta=False,bintype="2D"
        """
        This function calculates the true interaction rate (per second per ra
        true interaction rate per unit mass multiplied with the effective mas

        Parameters:
        -----------
        E: bool,float
            The energy for which you want to know the true interaction rate o
        costheta: bool,float
            The cosine of theta for which you want to know the true interacti
        bintype: string
            This specifies as a function of what you want to know the true in
            as a function of E, 'costheta' means as a function of costheta an
            of both E and costheta which gives a 2D array.
        bins: int
            The amount of bins you want for the true interaction rate of this
        """
        water_density = 997 #kg per cubic meters

        if bintype == "E":
            E_bin, ir_per_mass = self.TrueInteractionRate_per_mass_bin(costhe
            E_bin, eff_vol = self.Effective_Volume(bintype=bintype,bins=bins)

            ir_detector = ir_per_mass*eff_vol*water_density

            return E_bin, ir_detector

        elif bintype == "costheta":
            costheta_bin, ir_per_mass = self.TrueInteractionRate_per_mass_bin
            costheta_bin, eff_vol = self.Effective_Volume(bintype=bintype,bin

            ir_detector = ir_per_mass*eff_vol*water_density

            return costheta_bin, ir_detector

        elif bintype == "2D":
            ir_per_mass = self.TrueInteractionRate_per_mass_bin(bintype=binty
            eff_vol = self.Effective_Volume(bintype=bintype,bins=bins)

            ir_detector = ir_per_mass*eff_vol*water_density
```

```
            return ir_detector

    def InteractionRate_rec(self,bintype,bins):
        """
        This function calculates the detector response using event weights, s
        weight in a bin multiplied with the oscillated flux in a bin.

        Parameters:
        -----------
        bintype: string
            This specifies as a function of what you want to know the true in
            as a function of E, 'costheta' means as a function of costheta an
            of both E and costheta which gives a 2D array.
        bins: int
            The amount of bins you want for the true interaction rate of this

        """

        flux_osc = self.flux_oscillated(self.E,self.costheta)
        weighted_flux = self.weights*flux_osc

        E_wanted = np.logspace(0,2,num=bins)
        costheta_wanted = np.linspace(self.costheta_min,self.costheta_max,num

        if bintype == "E":
            i=0
            ir_reco = np.zeros(bins)

            while i < bins:
                boolean_E = (np.log10(self.E) > np.log10(E_wanted[i])-1/bins)
                if np.sum(boolean_E) == 0:
                    ir_reco[i] = np.nan
                else:
                    ir_reco[i] = np.sum(weighted_flux[boolean_E])
                i=i+1

            return E_wanted, ir_reco

        elif bintype == "costheta":
            i=0
            ir_reco = np.zeros(bins)

            while i < bins:
                boolean_costheta = (self.costheta>costheta_wanted[i]-1/bins)*
                if np.sum(boolean_costheta) == 0:
                    ir_reco[i] = np.nan
                else:
                    ir_reco[i] = np.sum(weighted_flux[boolean_costheta])
                i=i+1

            return costheta_wanted, ir_reco

        elif bintype == "2D":
            i=0
            indexes = np.zeros(bins*bins)
            ir_reco = np.zeros(bins*bins)

            while i < bins:
                boolean_theta = (self.costheta>costheta_wanted[i]-1/bins)*(se
                j=0
                while j < bins:
                    boolean_E = (np.log10(self.E) > np.log10(E_wanted[j])-1/b
                    boolean_res = boolean_E * boolean_theta
                    if np.sum(boolean_res) == 0:
```

```
                    ir_reco[i*bins+j] = np.nan
                else:
                    weighted_flux_filtered = weighted_flux[boolean_res]
                    ir_reco[i*bins+j] = np.sum(weighted_flux_filtered)

                j=j+1
            i=i+1

        ir_reco = np.reshape(ir_reco, (bins,bins))
        ir_reco = ir_reco[::-1,:]

        return ir_reco
```

In [37]:

```
#Creating the objects (e,m,t,ae,am,at are respectively the neutrino flavours
#anti-electron, anti-muon, anti-tau

nu_e = DetectorResponseModeller(12,1)
nu_m = DetectorResponseModeller(14,1)
nu_t = DetectorResponseModeller(16,1)
nu_ae = DetectorResponseModeller(12,-1)
nu_am = DetectorResponseModeller(14,-1)
nu_at = DetectorResponseModeller(16,-1)
```

In [38]:

```
#Ticks needed for styling the x-axis of 2D plots
ticks = [1,50,100]
tick_labels = ["1","10","100"]

#Range where we are looking at
E = np.logspace(0,2,base=10,num=200)
costheta = np.linspace(-1,1,num=200)
```

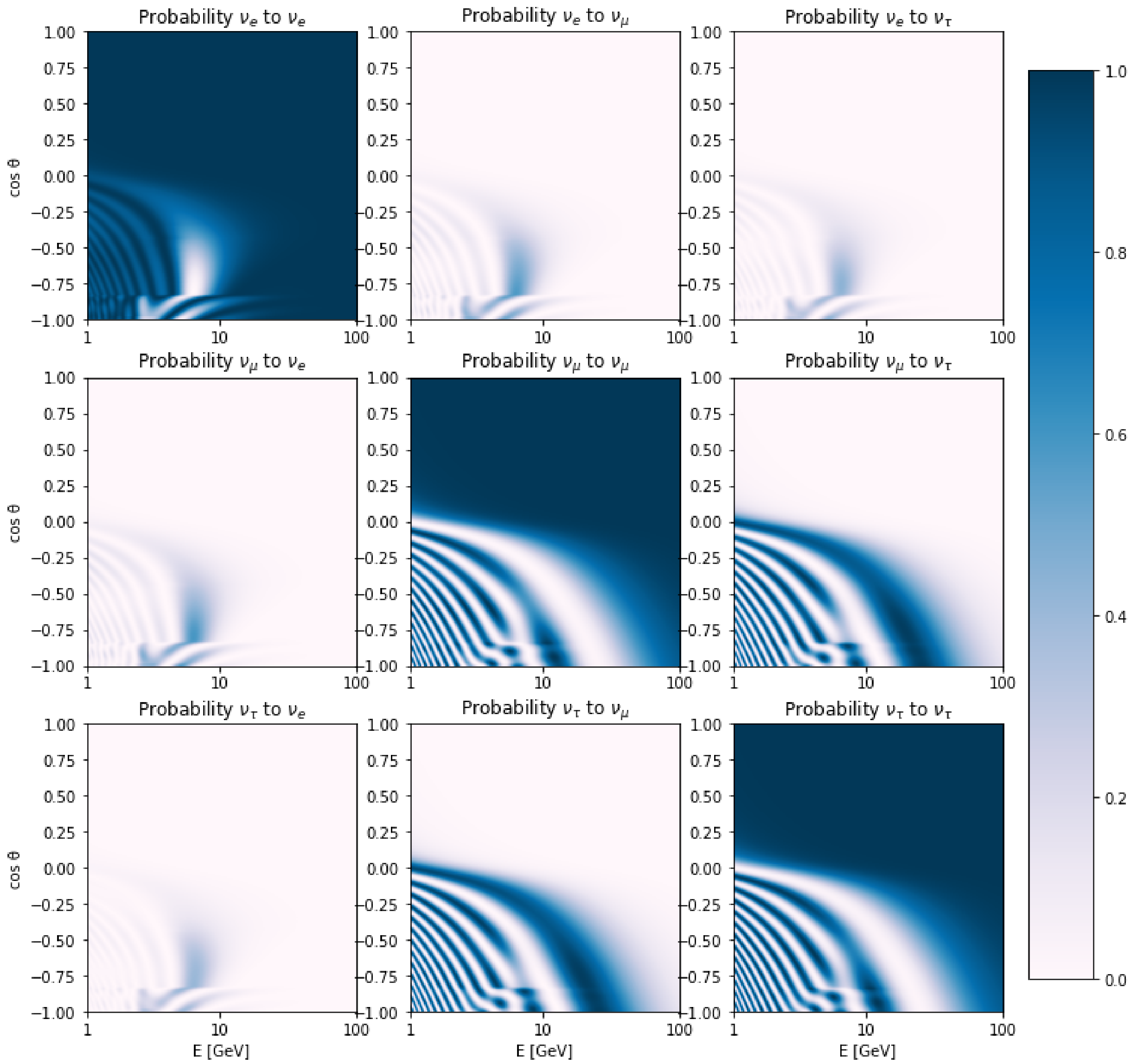
Oscillation Probabilities

In [39]:

```
ticks = [1,50,100]
tick_labels = ["1","10","100"]

fig, axs = plt.subplots(3,3,figsize=(12,12))
axs[0,0].imshow(nu_e.oscillation_probability(E,costheta,12,12,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"Probability $\nu_{e}$ to $\nu_e$")
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[0,1].imshow(nu_e.oscillation_probability(E,costheta,12,14,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,1].set_title(r"Probability $\nu_{e}$ to $\nu_{\mu}$")
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
axs[0,2].imshow(nu_e.oscillation_probability(E,costheta,12,16,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,2].set_title(r"Probability $\nu_{e}$ to $\nu_{\tau}$")
axs[0,2].set_xticks(ticks)
axs[0,2].set_xticklabels(tick_labels)
axs[1,0].imshow(nu_e.oscillation_probability(E,costheta,14,12,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[1,0].set_ylabel("cos θ")
axs[1,0].set_title(r"Probability $\nu_{\mu}$ to $\nu_e$")
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,1].imshow(nu_e.oscillation_probability(E,costheta,14,14,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[1,1].set_title(r"Probability $\nu_{\mu}$ to $\nu_{\mu}$")
```

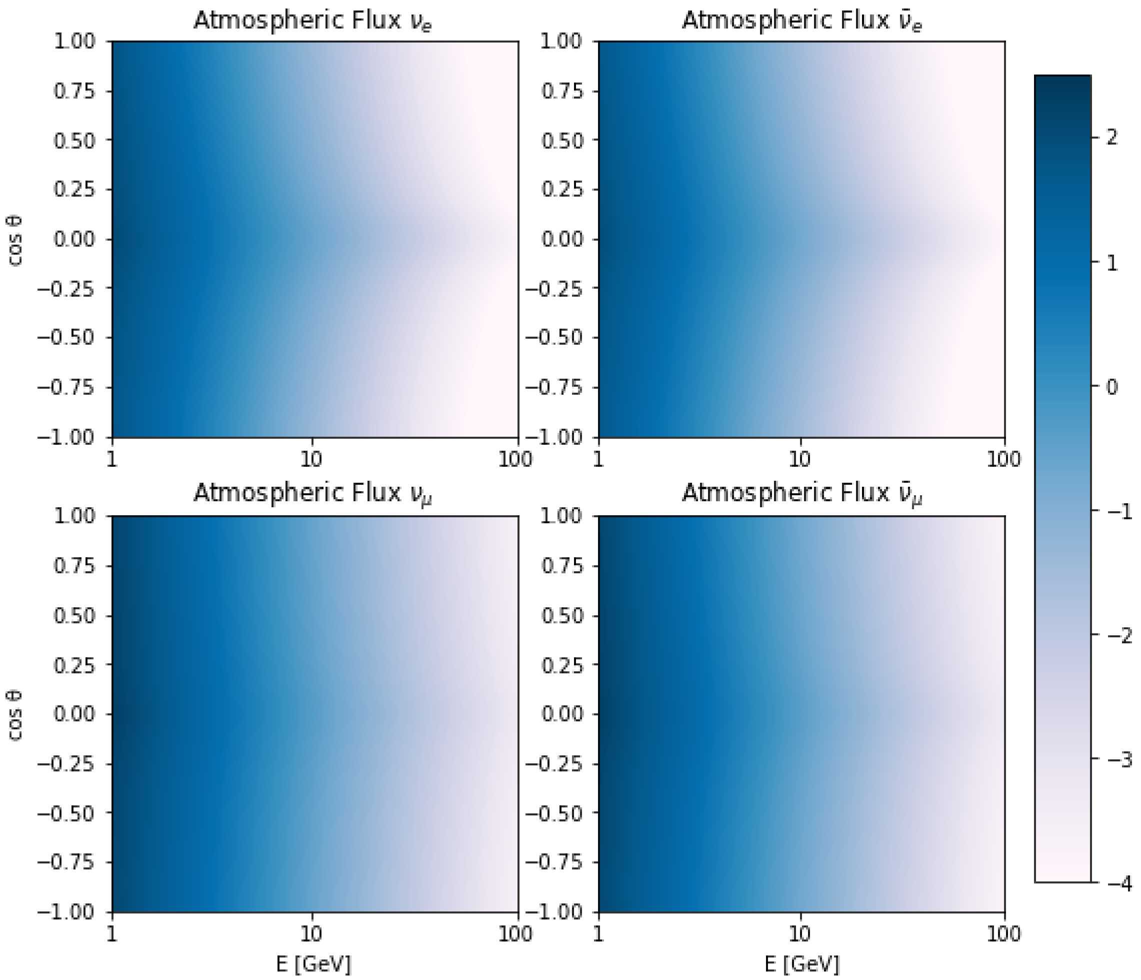
```
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
axs[1,2].imshow(nu_e.oscillation_probability(E,costheta,14,16,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[1,2].set_title(r"Probability $\nu_{\mu}$ to $\nu_{\tau}$")
axs[1,2].set_xticks(ticks)
axs[1,2].set_xticklabels(tick_labels)
axs[2,0].imshow(nu_e.oscillation_probability(E,costheta,16,12,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[2,0].set_ylabel("cos θ")
axs[2,0].set_xlabel("E [GeV]")
axs[2,0].set_title(r"Probability $\nu_{\tau}$ to $\nu_e$")
axs[2,0].set_xticks(ticks)
axs[2,0].set_xticklabels(tick_labels)
axs[2,1].imshow(nu_e.oscillation_probability(E,costheta,16,14,1),cmap='PuBu',
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[2,1].set_xlabel("E [GeV]")
axs[2,1].set_title(r"Probability $\nu_{\tau}$ to $\nu_{\mu}$")
axs[2,1].set_xticks(ticks)
axs[2,1].set_xticklabels(tick_labels)
cb=axs[2,2].imshow(nu_e.oscillation_probability(E,costheta,16,16,1),cmap='PuB
                   extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax
axs[2,2].set_xlabel("E [GeV]")
axs[2,2].set_title(r"Probability $\nu_{\tau}$ to $\nu_{\tau}$")
axs[2,2].set_xticks(ticks)
axs[2,2].set_xticklabels(tick_labels)
fig.subplots_adjust(right=0.83)
cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
plt.show()
```



Atmospheric Neutrinos

In [40]:
```
fig, axs = plt.subplots(2,2,figsize=(8,8))
axs[0,0].imshow(np.log10(nu_e.atmospheric_flux(E,costheta)),cmap='PuBu',vmin=
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"Atmospheric Flux $\nu_{e}$")
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[1,0].imshow(np.log10(nu_m.atmospheric_flux(E,costheta)),cmap='PuBu',vmin=
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[1,0].set_xlabel("E [GeV]")
axs[1,0].set_title(r"Atmospheric Flux $\nu_{\mu}$")
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,0].set_ylabel("cos θ")
axs[0,1].imshow(np.log10(nu_ae.atmospheric_flux(E,costheta)),cmap='PuBu',vmin
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,1].set_title(r"Atmospheric Flux $\bar{\nu}_{e}$")
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
cb=axs[1,1].imshow(np.log10(nu_am.atmospheric_flux(E,costheta)),cmap='PuBu',v
                   extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax
axs[1,1].set_title(r"Atmospheric Flux $\bar{\nu}_{\mu}$")
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
```

```
axs[1,1].set_xlabel("E [GeV]")
cbar_ax = fig.add_axes([0.925, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
plt.show()
```

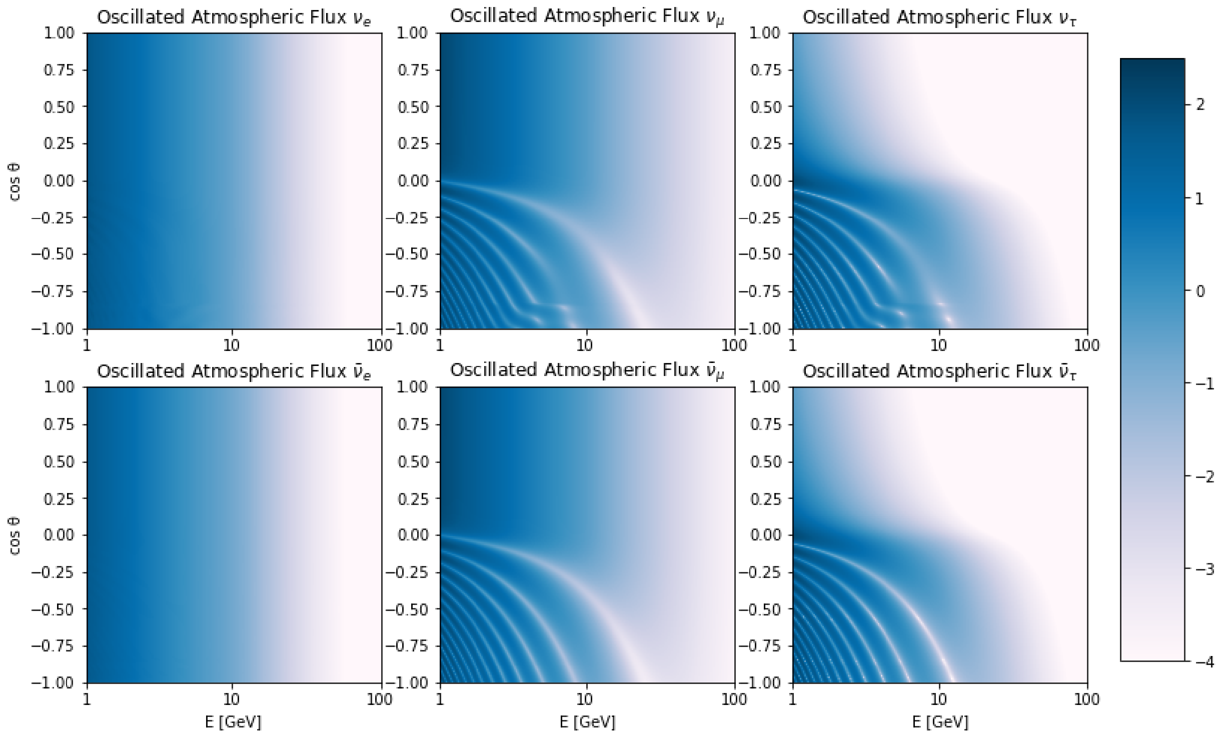


Oscillated Atmospheric Neutrinos

In [41]:

```
fig, axs = plt.subplots(2,3,figsize=(12,8))
axs[0,0].imshow(np.log10(nu_e.flux_oscillated(E,costheta)),cmap='PuBu',vmin=-
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"Oscillated Atmospheric Flux $\nu_{e}$")
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[1,0].imshow(np.log10(nu_ae.flux_oscillated(E,costheta)),cmap='PuBu',vmin=
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[1,0].set_xlabel("E [GeV]")
axs[1,0].set_title(r"Oscillated Atmospheric Flux $\bar{\nu}_{e}$")
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,0].set_ylabel("cos θ")
axs[0,1].imshow(np.log10(nu_m.flux_oscillated(E,costheta)),cmap='PuBu',vmin=-
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,1].set_title(r"Oscillated Atmospheric Flux $\nu_{\mu}$")
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
axs[1,1].imshow(np.log10(nu_am.flux_oscillated(E,costheta)),cmap='PuBu',vmin=
                   extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax
axs[1,1].set_title(r"Oscillated Atmospheric Flux $\bar{\nu}_{\mu}$")
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
```

```
axs[1,1].set_xlabel("E [GeV]")
axs[0,2].imshow(np.log10(nu_t.flux_oscillated(E,costheta)),cmap='PuBu',vmin=-
                extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax(co
axs[0,2].set_title(r"Oscillated Atmospheric Flux $\nu_{\tau}$")
axs[0,2].set_xticks(ticks)
axs[0,2].set_xticklabels(tick_labels)
cb=axs[1,2].imshow(np.log10(nu_at.flux_oscillated(E,costheta)),cmap='PuBu',vm
                   extent=(np.amin(E), np.amax(E), np.amin(costheta), np.amax
axs[1,2].set_title(r"Oscillated Atmospheric Flux $\bar{\nu}_{\tau}$")
axs[1,2].set_xticks(ticks)
axs[1,2].set_xticklabels(tick_labels)
axs[1,2].set_xlabel("E [GeV]")
cbar_ax = fig.add_axes([0.925, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
plt.show()
```

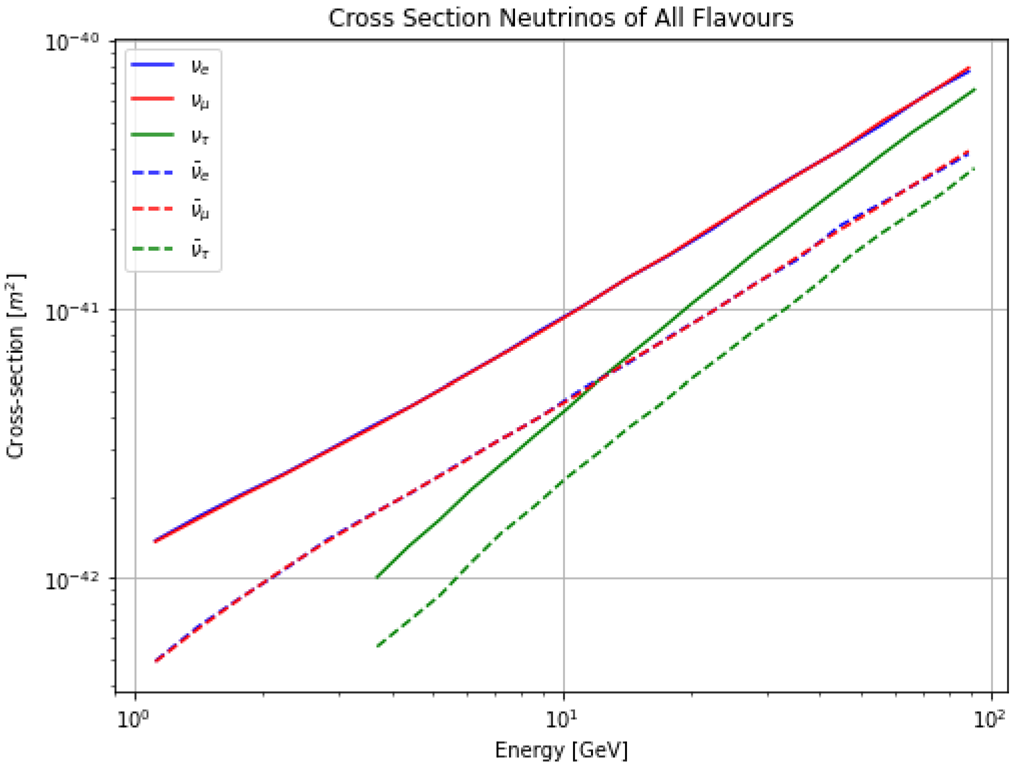


Cross Sections

In [42]:
```
bins = 20
E_e, cs_e = nu_e.cross_section_bin(bins=bins)
E_m, cs_m = nu_m.cross_section_bin(bins=bins)
E_t, cs_t = nu_t.cross_section_bin(bins=bins)
E_ae, cs_ae = nu_ae.cross_section_bin(bins=bins)
E_am, cs_am = nu_am.cross_section_bin(bins=bins)
E_at, cs_at = nu_at.cross_section_bin(bins=bins)
```

In [43]:
```
fig=plt.figure(figsize=(8,6))
plt.title("Cross Section Neutrinos of All Flavours")
plt.loglog(E_e, cs_e, 'blue', ls='-', label=r'$\nu_e$')
plt.loglog(E_m, cs_m, 'red', ls='-', label=r'$\nu_{\mu}$')
plt.loglog(E_t, cs_t, 'green', ls='-', label=r'$\nu_{\tau}$')
plt.loglog(E_ae, cs_ae, 'blue', ls='--', label=r'$\bar{\nu}_e$')
plt.loglog(E_am, cs_am, 'red', ls='--', label=r'$\bar{\nu}_{\mu}$')
plt.loglog(E_at, cs_at, 'green', ls='--', label=r'$\bar{\nu}_{\tau}$')
plt.xlabel("Energy [GeV]")
plt.ylabel("Cross-section [$m^2$]")
plt.xlim(0.9,110)
plt.grid()
```

```
plt.legend()
```



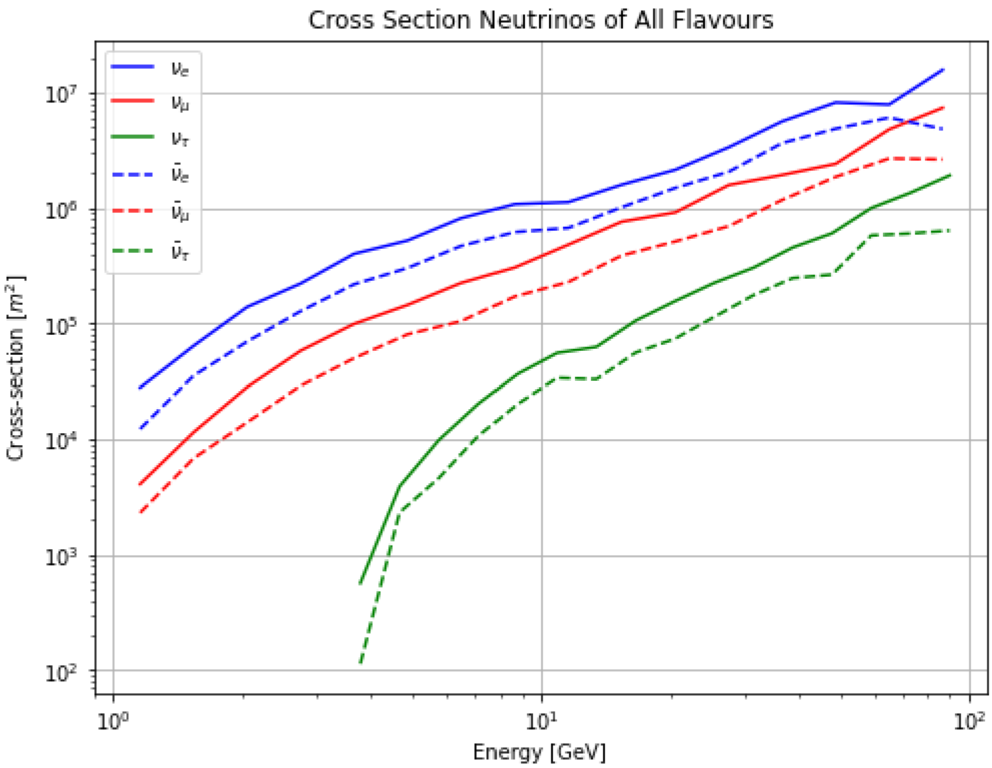
Effective Volume

In [44]:

```
bins = 16
E_e, effvol_e = nu_e.Effective_Volume(bintype='E',bins=bins)
E_m, effvol_m = nu_m.Effective_Volume(bintype='E',bins=bins)
E_t, effvol_t = nu_t.Effective_Volume(bintype='E',bins=bins)
E_ae, effvol_ae = nu_ae.Effective_Volume(bintype='E',bins=bins)
E_am, effvol_am = nu_am.Effective_Volume(bintype='E',bins=bins)
E_at, effvol_at = nu_at.Effective_Volume(bintype='E',bins=bins)
```

In [45]:

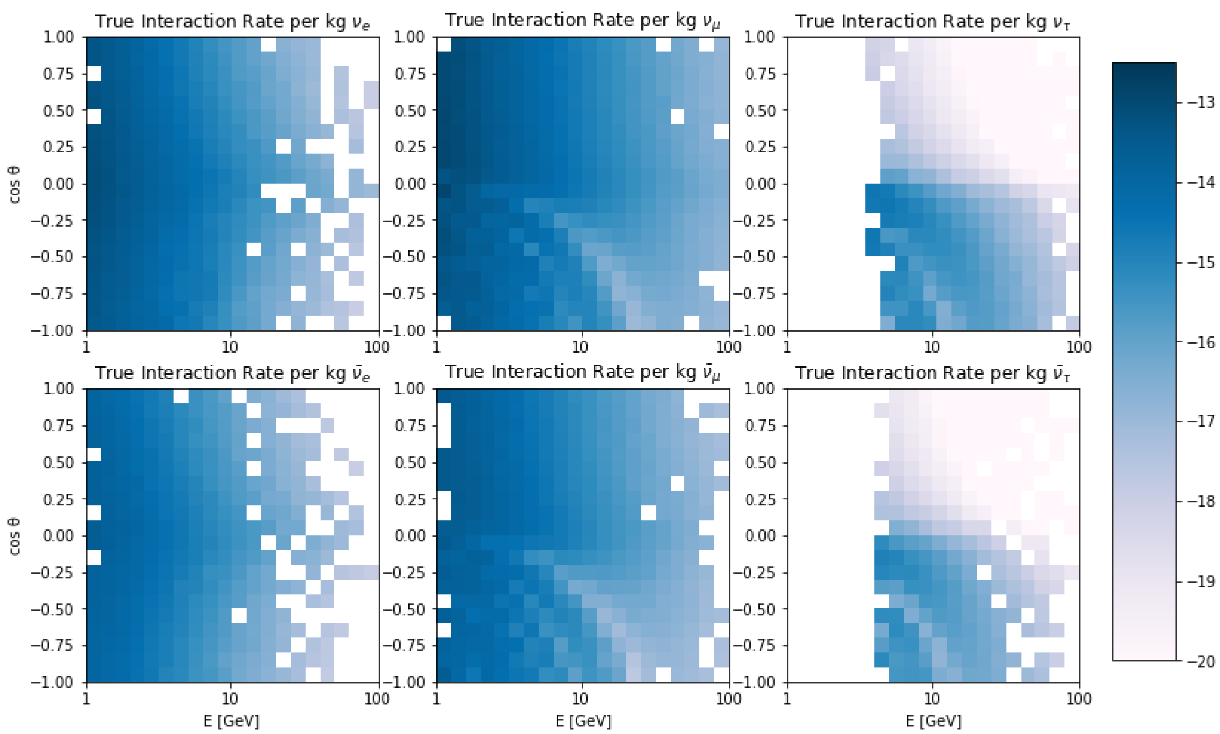
```
fig=plt.figure(figsize=(8,6))
plt.title("Cross Section Neutrinos of All Flavours")
plt.loglog(E_e, effvol_e, 'blue', ls='-', label=r'$\nu_e$')
plt.loglog(E_m, effvol_m, 'red', ls='-', label=r'$\nu_{\mu}$')
plt.loglog(E_t, effvol_t, 'green', ls='-', label=r'$\nu_{\tau}$')
plt.loglog(E_ae, effvol_ae, 'blue', ls='--', label=r'$\bar{\nu}_e$')
plt.loglog(E_am, effvol_am, 'red', ls='--', label=r'$\bar{\nu}_{\mu}$')
plt.loglog(E_at, effvol_at, 'green', ls='--', label=r'$\bar{\nu}_{\tau}$')
plt.xlabel("Energy [GeV]")
plt.ylabel("Cross-section [$m^2$]")
plt.xlim(0.9,110)
plt.grid()
plt.legend()
plt.show()
```



Interaction Rate per kg

In [46]:
```
fig, axs = plt.subplots(2,3,figsize=(12,8))
axs[0,0].imshow(np.log10(nu_e.TrueInteractionRate_per_mass_bin(bins=20)),cmap
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"True Interaction Rate per kg $\nu_{e}$")
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[1,0].imshow(np.log10(nu_ae.TrueInteractionRate_per_mass_bin(bins=20)),cma
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,0].set_xlabel("E [GeV]")
axs[1,0].set_title(r"True Interaction Rate per kg $\bar{\nu}_{e}$")
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,0].set_ylabel("cos θ")
axs[0,1].imshow(np.log10(nu_m.TrueInteractionRate_per_mass_bin(bins=20)),cmap
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[0,1].set_title(r"True Interaction Rate per kg $\nu_{\mu}$")
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
axs[1,1].imshow(np.log10(nu_am.TrueInteractionRate_per_mass_bin(bins=20)),cma
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,1].set_title(r"True Interaction Rate per kg $\bar{\nu}_{\mu}$")
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
axs[1,1].set_xlabel("E [GeV]")
axs[0,2].imshow(np.log10(nu_t.TrueInteractionRate_per_mass_bin(bins=20)),cmap
                extent=(nu_t.Emin, nu_t.Emax, nu_t.costheta_min, nu_t.costhet
axs[0,2].set_title(r"True Interaction Rate per kg $\nu_{\tau}$")
axs[0,2].set_xticks(ticks)
axs[0,2].set_xticklabels(tick_labels)
cb=axs[1,2].imshow(np.log10(nu_at.TrueInteractionRate_per_mass_bin(bins=20)),
                   extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.cost
axs[1,2].set_title(r"True Interaction Rate per kg $\bar{\nu}_{\tau}$")
axs[1,2].set_xticks(ticks)
axs[1,2].set_xticklabels(tick_labels)
```

```
axs[1,2].set_xlabel("E [GeV]")
cbar_ax = fig.add_axes([0.925, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
plt.show()
```



True Interaction Rate of ORCA-115

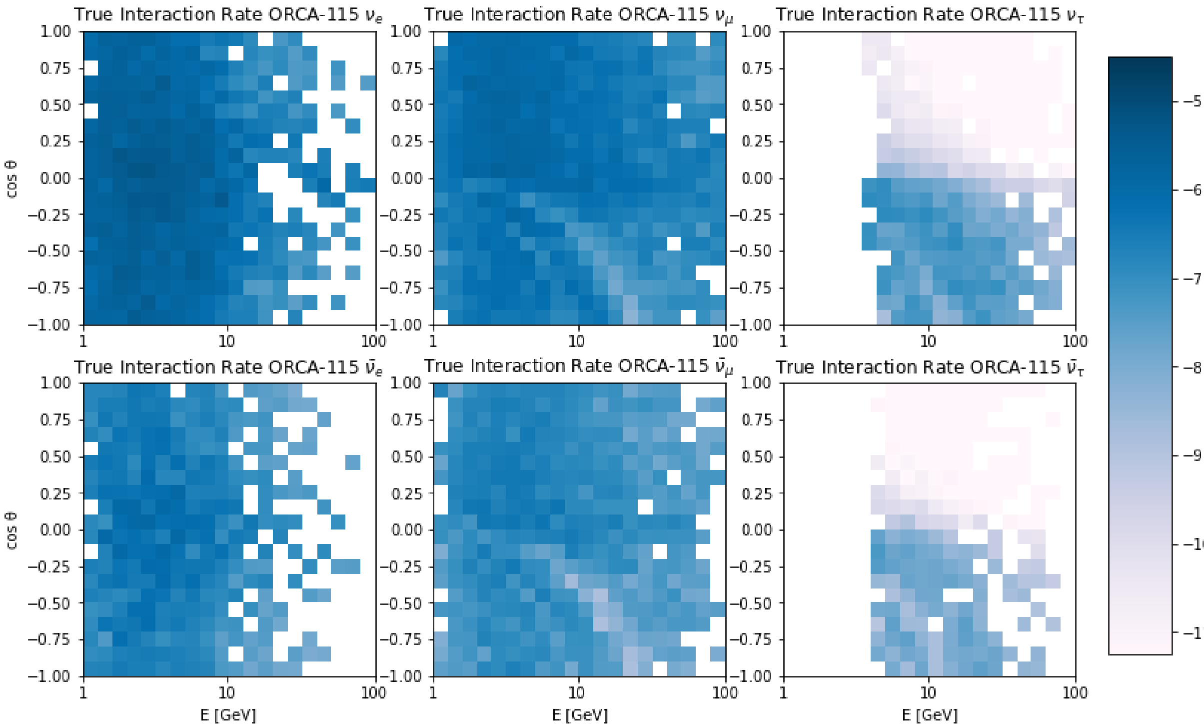
In [47]:
```
fig, axs = plt.subplots(2,3,figsize=(12,8))
axs[0,0].imshow(np.log10(nu_e.TrueInteractionRate_detector(bintype="2D",bins=
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"True Interaction Rate ORCA-115 $\nu_{e}$")
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[1,0].imshow(np.log10(nu_ae.TrueInteractionRate_detector(bintype="2D",bins
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,0].set_xlabel("E [GeV]")
axs[1,0].set_title(r"True Interaction Rate ORCA-115 $\bar{\nu}_{e}$")
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,0].set_ylabel("cos θ")
axs[0,1].imshow(np.log10(nu_m.TrueInteractionRate_detector(bintype="2D",bins=
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[0,1].set_title(r"True Interaction Rate ORCA-115 $\nu_{\mu}$")
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
axs[1,1].imshow(np.log10(nu_am.TrueInteractionRate_detector(bintype="2D",bins
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,1].set_title(r"True Interaction Rate ORCA-115 $\bar{\nu}_{\mu}$")
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
axs[1,1].set_xlabel("E [GeV]")
axs[0,2].imshow(np.log10(nu_t.TrueInteractionRate_detector(bintype="2D",bins=
                extent=(nu_t.Emin, nu_t.Emax, nu_t.costheta_min, nu_t.costhet
axs[0,2].set_title(r"True Interaction Rate ORCA-115 $\nu_{\tau}$")
axs[0,2].set_xticks(ticks)
axs[0,2].set_xticklabels(tick_labels)
cb=axs[1,2].imshow(np.log10(nu_at.TrueInteractionRate_detector(bintype="2D",b
                   extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.cost
```

```
axs[1,2].set_title(r"True Interaction Rate ORCA-115 $\bar{\nu}_{\tau}$")
axs[1,2].set_xticks(ticks)
axs[1,2].set_xticklabels(tick_labels)
axs[1,2].set_xlabel("E [GeV]")
cbar_ax = fig.add_axes([0.925, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
```

```
/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:2: RuntimeWarning: divide by zero encountered in log10

/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:8: RuntimeWarning: divide by zero encountered in log10

/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:15: RuntimeWarning: divide by zero encountered in log10
  from ipykernel import kernelapp as app
/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:20: RuntimeWarning: divide by zero encountered in log10
/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:26: RuntimeWarning: divide by zero encountered in log10
/project/antares/jurjanbootsma/venv/lib64/python3.6/site-packages/ipykernel_la
uncher.py:31: RuntimeWarning: divide by zero encountered in log10
```



Reconstructed Interaction Rate

In [48]:

```
fig, axs = plt.subplots(2,3,figsize=(12,8))
axs[0,0].imshow(nu_e.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu', 
                   extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.cost
axs[0,0].set_xticks(ticks)
axs[0,0].set_xticklabels(tick_labels)
axs[0,0].set_ylabel("cos θ")
axs[0,0].set_title(r"Reconstructed Interaction Rate $\nu_{e}$")

axs[1,0].imshow(nu_ae.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu',
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,0].set_xticks(ticks)
axs[1,0].set_xticklabels(tick_labels)
axs[1,0].set_ylabel("cos θ")
axs[1,0].set_xlabel("E [GeV]")
axs[1,0].set_title(r"Reconstructed Interaction Rate $\bar{\nu}_{e}$")
```

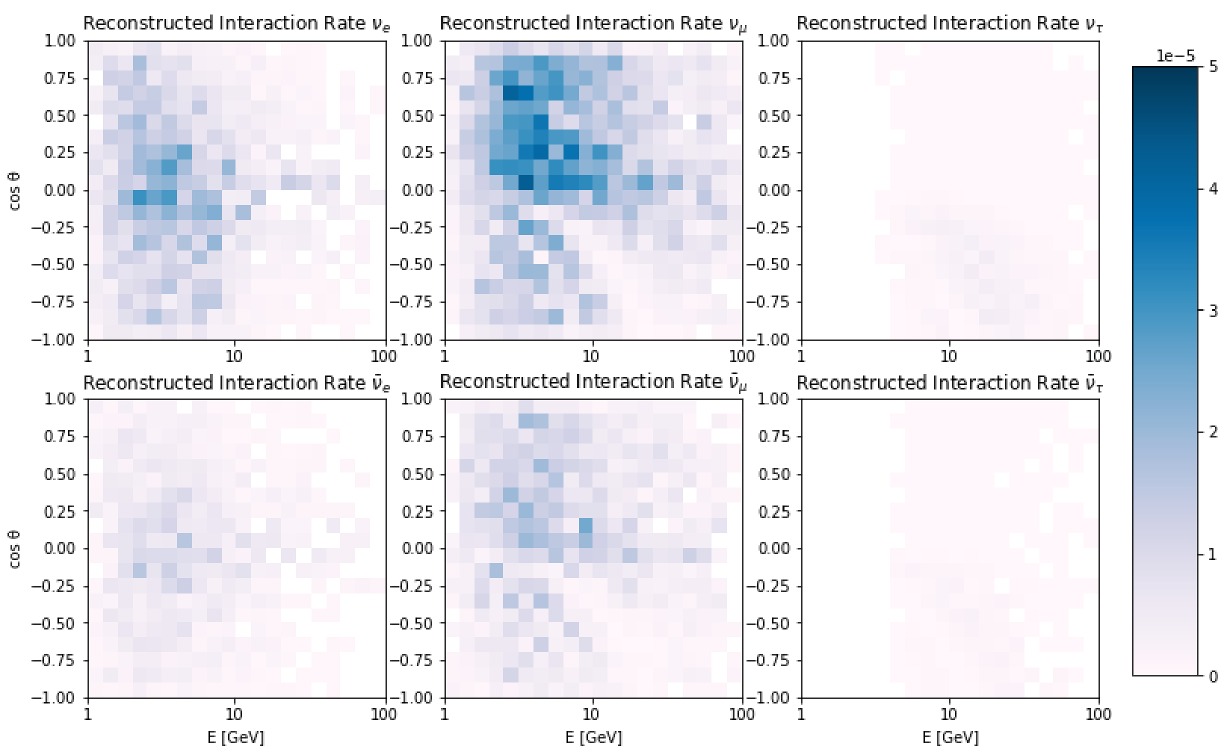
```
cb=axs[0,1].imshow(nu_m.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu
                   extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.cost
axs[0,1].set_xticks(ticks)
axs[0,1].set_xticklabels(tick_labels)
axs[0,1].set_title(r"Reconstructed Interaction Rate $\nu_{\mu}$")

axs[1,1].imshow(nu_am.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu',
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,1].set_xticks(ticks)
axs[1,1].set_xticklabels(tick_labels)
axs[1,1].set_xlabel("E [GeV]")
axs[1,1].set_title(r"Reconstructed Interaction Rate $\bar{\nu}_{\mu}$")

axs[0,2].imshow(nu_t.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu',
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[0,2].set_xticks(ticks)
axs[0,2].set_xticklabels(tick_labels)
axs[0,2].set_title(r"Reconstructed Interaction Rate $\nu_{\tau}$")

axs[1,2].imshow(nu_at.InteractionRate_rec(bintype="2D",bins=20), cmap='PuBu',
                extent=(nu_m.Emin, nu_m.Emax, nu_m.costheta_min, nu_m.costhet
axs[1,2].set_xticks(ticks)
axs[1,2].set_xticklabels(tick_labels)
axs[1,2].set_xlabel("E [GeV]")
axs[1,2].set_title(r"Reconstructed Interaction Rate $\bar{\nu}_{\tau}$")

cbar_ax = fig.add_axes([0.925, 0.15, 0.05, 0.7])
fig.colorbar(cb, cax=cbar_ax)
```



In [ ]:

In [ ]: