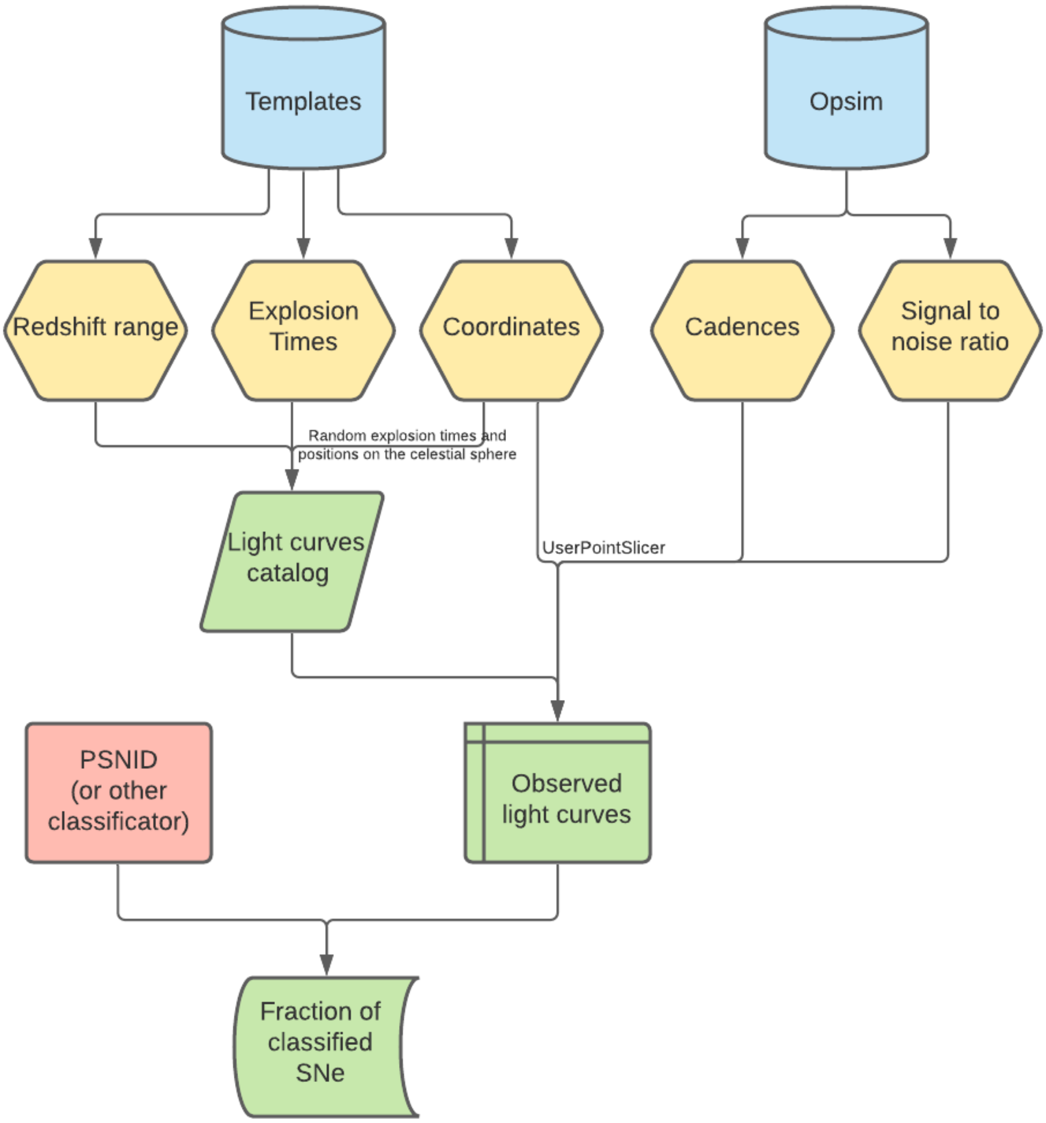
Supernova Rate Metric

Flow chart of the project:



The Supernova Rate Matric is a class object that gives as derivable the SN detection efficiency and classification efficiency from a given survey.

The metric is composed by three classes and a method:

- **template_lc**: class, it gives the k-corrected supernova light curve in *g, r* and *i* bands for a given redshift range;
- **generateSNPopSlicer**: method, Generate a population of SNe events, and put the info about them into a UserPointSlicer object

- **SNclassification_metric**: class, it simulates observed points of the observed light curves from LSST and gives the fraction of corrected classified SNe in function of type or redshift ;

template_lc:

GLOBAL PARAMETER:

- **templates**: dict, it is a dictionary where they are listed the templates type, subtype and fractions of each template per subgroup;
- ***z_min, z_max, ,z_step*** : float, those parameter define the range and the step in redshift to perform the simulation;
- ***extinction:*** float, it is a value for the extinction, default =0;
- ***path:*** string, it is the path where to save the output file if requested, default ='$HOME';
- ***dataout:*** boolean, if True the metric return a .txt file with the lc information, if False return a dictionary with magnitude epoch (respect the epoch of the maximum) for each filter. Default dataout=False.

All the coefficients needed for the transformation from a passband to another are stored in the dictionary parband:

```
self.band_label= ['bandw','fwhm','avgwv','equvw','zpoint','abmag for vega']
#landolt & johnson buser and kurucz 78
self.bandpar['U']=[205.79,484.6,3652,542.62,4.327e-9,  0.76]
self.bandpar['B']=[352.94,831.11,4448,1010.3,6.09e-9, -0.11]
self.bandpar['V']=[351.01,826.57,5505,870.65,3.53e-9, 0.]
#landolt & cousin bessel 83
self.bandpar['R']=[589.71,1388.7,6555,1452.2,2.104e-9, 0.18]
self.bandpar['I']=[381.67,898.77,7900.4,1226,1.157e-9, 0.42]
# HAWK-I filter
self.bandpar['Y']=[0,1019.4,10226.9,0,5.74e-10, '?']
#bessel in bessel and brett 88
self.bandpar['J']=[747.1,1759.3,12370,2034,3.05e-10, '?']
self.bandpar['H']=[866.55,2040.6,16471,2882.8,1.11e-10, '?']
self.bandpar['K']=[1188.9,2799.6,22126,3664.3,3.83e-11, '?']
# ASIAGO PHOTOMETRIC DATABASE
self.bandpar['L']=[0.,9000.,34000,0.,8.1e-12, '?']
self.bandpar['M']=[0.,11000.,50000,0.,2.2e-12, '?']
self.bandpar['N']=[0.,60000.,102000,0.,1.23e-13, '?']
# sloan
self.bandpar['u']=[194.41,457.79, 3561.8, 60.587,3.622e-9, 0.92]
self.bandpar['g']=[394.17,928.19, 4718.9, 418.52,5.414e-9, -0.11]
self.bandpar['r']=[344.67,811.65, 6185.2, 546.14,2.472e-9, 0.14]
self.bandpar['i']=[379.57,893.82, 7499.8, 442.15,1.383e-9, 0.36]
self.bandpar['z']=[502.45,1183.2, 8961.5, 88.505,8.15e-10, 0.52]
# SWIFT A-> UW1, D --> UM2, S -> UW2
#self.bandpar['A']=[348.43,820.5,2650.6,770.45,3.818e-9]
#self.bandpar['D']=[189.46,446.14,2269.2,519.03,4.321e-9]
#self.bandpar['S']=[285.12,671.4,2136.7,639.45,4.825e-9]
self.bandpar['A']=[0.,693.,2634,0.,4.3e-9, '?'] # Poole et al. 2008 383, 627
self.bandpar['D']=[0.,498.,2231,0.,4.0e-9, '?'] # corrected for z-point
self.bandpar['S']=[0.,657.,2030,0.,5.2e-9, '?']
```

the transient template is then read from a file, e.g. the format of the template is:

```
1987A     IIP jd_expl= 46849.8
LMC     ABg= 0.79 .00 ABi= 0.00 .00 mu= 18.49 .00
________________________________________________________________________________
UBVRI  JD_max  46849.69 .00 46951.69 .00 46849.69 .00  46829.69 .00 46829.69 .00
      m_max        9999 .00  9999 .00  9999 .00   9999 .00  9999 .00
 Date       JD          U        B         V         R          I         S      notes
 xxxxx    46849.      .00 .00  30.  .00  30.  .00  30.  .00  9999 .00  111
#
  51.3    46851.32   3.99 .00  4.74 .00  4.62 .00  4.40 .00  4.34 .00  1
  52.3    46852.32   4.19 .00  4.74 .00  4.52 .00  4.21 .00  4.13 .00  1
  53.2    46853.27   4.52 .00  4.87 .00  4.52 .00  4.20 .00  4.06 .00  1
  53.2    46853.28   4.49 .00  4.82 .00  4.46 .00  4.12 .00  4.02 .00  1
  53.3    46853.34   4.50 .00  4.80 .00  4.44 .00  4.11 .00  4.00 .00  1
  54.3    46854.39   5.18 .00  4.94 .00  4.43 .00  4.05 .00  3.94 .00  1
  55.2    46855.29   5.71 .00  5.08 .00  4.44 .00  4.05 .00  3.92 .00  1
  56.2    46856.29   6.18 .00  5.26 .00  4.49 .00  4.05 .00  3.89 .00  1
  58.2    46858.29   6.74 .00  5.41 .00  4.45 .00  4.00 .00  3.79 .00  1
  59.4    46859.44   6.93 .00  5.48 .00  4.41 .00  3.96 .00  3.72 .00  1
  60.2    46860.27   7.09 .00  5.55 .00  4.40 .00  3.94 .00  3.68 .00  1
  61.2    46861.25   7.24 .00  5.57 .00  4.36 .00  3.89 .00  3.62 .00  1
  62.2    46862.27   7.32 .00  5.60 .00  4.32 .00  3.85 .00  3.56 .00  1
  63.3    46863.39   7.39 .00  5.63 .00  4.30 .00  3.81 .00  3.50 .00  1
  64.2    46864.25   7.47 .00  5.65 .00  4.29 .00  3.78 .00  3.47 .00  1
  65.2    46865.25   7.53 .00  5.66 .00  4.27 .00  3.75 .00  3.41 .00  1
  66.2    46866.24   7.58 .00  5.67 .00  4.25 .00  3.72 .00  3.36 .00  1
  67.2    46867.25   7.61 .00  5.68 .00  4.22 .00  3.66 .00  3.30 .00  1
  68.2    46868.25   7.67 .00  5.68 .00  4.22 .00  3.66 .00  3.27 .00  1
  71.2    46871.27   7.74 .00  5.69 .00  4.18 .00  3.56 .00  3.16 .00  1
  72.2    46872.25   7.76 .00  5.68 .00  4.16 .00  3.53 .00  3.14 .00  1
  73.2    46873.25   7.78 .00  5.67 .00  4.14 .00  3.50 .00  3.11 .00  1
  74.2    46874.24   7.78 .00  5.66 .00  4.11 .00  3.45 .00  3.05 .00  1
  75.2    46875.25   7.80 .00  5.65 .00  4.09 .00  3.42 .00  3.03 .00  1
  76.2    46876.25   7.80 .00  5.65 .00  4.07 .00  3.40 .00  3.01 .00  1
  77.2    46877.24   7.80 .00  5.64 .00  4.06 .00  3.39 .00  2.98 .00  1
  78.2    46878.24   7.80 .00  5.61 .00  4.05 .00  3.35 .00  2.95 .00  1
  79.2    46879.25   7.78 .00  5.59 .00  4.02 .00  3.31 .00  2.92 .00  1
  80.2    46880.24   7.81 .00  5.58 .00  4.00 .00  3.30 .00  2.90 .00  1
  81.2    46881.24   7.80 .00  5.56 .00  3.97 .00  3.27 .00  2.87 .00  1
  82.2    46882.24   7.81 .00  5.54 .00  3.94 .00  3.23 .00  2.84 .00  1
  84.3    46884.30   7.79 .00  5.48 .00  3.88 .00  3.17 .00  2.79 .00  1
  85.2    46885.24   7.79 .00  5.46 .00  3.86 .00  3.15 .00  2.76 .00  1
  86.3    46886.33   7.74 .00  5.45 .00  3.82 .00  3.11 .00  2.74 .00  1
  87.2    46887.25   7.77 .00  5.40 .00  3.79 .00  3.08 .00  2.70 .00  1
  88.2    46888.24   7.78 .00  5.39 .00  3.76 .00  3.04 .00  2.68 .00  1
  90.3    46890.31   7.75 .00  5.32 .00  3.71 .00  2.98 .00  2.61 .00  1
  91.3    46891.39   7.71 .00  5.26 .00  3.66 .00  2.94 .00  2.57 .00  1
  92.2    46892.24   7.77 .00  5.26 .00  3.63 .00  2.93 .00  2.54 .00  1
  94.2    46894.25   7.76 .00  5.22 .00  3.58 .00  2.88 .00  2.50 .00  1
  95.2    46895.25   7.75 .00  5.19 .00  3.54 .00  2.85 .00  2.48 .00  1
  97.2    46897.25   7.71 .00  5.13 .00  3.50 .00  2.80 .00  2.42 .00  1
```

the output is a dictionary:

```
sndata = {'sn':sn, 'sntype':sntype, 'galaxy':galaxy, 'bands':bands,
          'ABg':abg, 'ABg_err':abg_err, 'ABi':abi, 'ABi_err':abi_err,\
          'mu':mu, 'mu_err':mu_err, 'jd_expl':jd_expl, \
          'jdmax':jdmax, 'jdmax_err':jdmax_err, 'magmax':magmax,\
          'magmax_err':magmax_err, 'format':'OLD',\
          'jd':jd, 'mag':mag, 'mag_err':mag_err, 'source':source}
```

The Cardelli law is used to estimate the absorption for each passband:

```
def AX(self, band='',AB='',R_V=3.1):

    if not band: band = raw_input('<<  Band ? ')
    if AB=='': AB = float(raw_input('<<  Ab absorption ? '))

    abfac = self.cardelli(self.bandpar[band][self.band_label.index('avgwv')],R_V)\
            /self.cardelli(self.bandpar['B'][self.band_label.index('avgwv')],R_V)
    return AB*abfac
```

Then the kcor is read from template table for given redshift ranges and given bands:

```
def kcor_read(self, sn,tsn,): #########

    kph,kz,kcor,kiko = {},{},{},{}

    ff = open(kcor_dir+'/kcor/kk/'+sn+'.kcor')
    righe = ff.readlines()
    kkph = [float(x) for x in righe[1].split()]

    ign = []      # read correction for specific reshift template
    for i,r in enumerate(righe):
        if '***' in r: ign.append(i)
    for i in ign:
        gn = righe[i].split()[0]
        kiko[gn] = np.array([float(k) for k in righe[i+1].split()])

    ff = open(kcor_dir+'/kcor/kk/'+tsn['kkclass']+'.kk')
    righe = ff.readlines()

    igf = []                               #   read kcor for SN class
    for i,r in enumerate(righe):
        if '---' in r: igf.append(i-1)

    for n,i in enumerate(igf):
        gf = righe[i].split()[0]
        kz[gf],kcor[gf] = [],[]
        il = len(righe)
        if n<len(igf)-1: il = igf[n+1]
        kph[gf] = np.array([float(p) for p in righe[i+1].split()[1:]])

        for r in righe[i+2:il]:
            kz[gf].append(float(r.split()[0]))
            kcor[gf].append(np.array([float(k) for k in r.split()[1:]]))
    ff.close()

    return kkph,kiko,kph,kz,kcor
```

and finally we estimate the kcorr at the redshift we decided through an interpolation:

```
def kcor_interpolate(self, tph,z,kph,kz,kcor):

    if len(kz)>1:
        jz = np.searchsorted(kz,z)-1
        if jz < len(kz)-1:
            il = np.where(kcor[jz]<100)
            _kl = np.interp(tph,kph[il],kcor[jz][il])
            iu = np.where(kcor[jz+1]<100)
            _ku = np.interp(tph,kph[iu],kcor[jz+1][iu])
            _kcor = _kl+(z-kz[jz])*(_ku-_kl)/(kz[jz+1]-kz[jz])
        else:
            il = np.where(kcor[jz]<100)
            _kcor = np.interp(tph,kph[il],kcor[jz][il])
    else:
        il = np.where(kcor[0]<100)
        _kcor = np.interp(tph,kph[il],kcor[0][il])

    return _kcor
```

and we correct the template magnitude and phase for the relative kcor to have the rest frame magnitude:

```
def mag_observer_frame(self, tph,tabsmag,tzed,z,kcor,kiko,tEXT):

    phobs = tph*(1-tzed+float(z))
    magobs = tabsmag+kcor+kiko+cosmo.mu(float(z))+tEXT
    return phobs,magobs
```

generateSNPopSlicer:

GLOBAL PARAMETER:

- **templates=** dict, it is a dictionary where they are listed the templates type, subtype and fractions of each template per subgroup;
- **t_start :** float, first epoch of the epochs range from which the explosion times are drawn
- **t_end :** float, final epoch of the epochs range from which the explosion times are drawn
- **n_events :** int The number of SNe events to generate
- **seed :** float The seed passed to np.random
- **z_min :** float or int Minimum reshift
- **z_max :** float or int Maximum redshift
- **zstep:** float or int redshift step
- **coo**: dict, specifics coordinate where to perform the simulation

The supernova templates are simulated at each redshift in the redshift range defined as [zmin : zstep : zmax] calling the class template_lc. Than from uniform distribution within the range [t_start, t_end] are drawn the epochs when the SNe explode. Finally the coordinates where to simulate the SNe are given by the dictionary coo, or are drawn by a uniform distribution over the celestial sphere.

```
# The SN template light curves are simulated at the redshifts in the redshift range
    # K-correction is applied at each redshift z
if not os.path.exists('./template'):
    os.makedirs('./template')
zmin = zmin
zmax = zmax
zstep = zstep
temp = template_lc(sn_group= templates, z_min=zmin,z_max= zmax,z_step=zstep)
obs_template = temp.run()
zrange = temp.zrange
filtri = temp.filtri
for j, z in enumerate(zrange):
        for ty in templates:
            for sty in templates[ty]:
                for sn in templates[ty][sty][0]:
                    asciifile = './template/snlc_{}_z={}_DDF.ascii'.format(sn,str(z))
                    ff = open(asciifile,'w')
                    if ty in ['Ia','Ibc']:    endTime = 50.*(1+z)
                    else:                     endTime =100.*(1+z)
                    for f in filtri:
                        for i,p in enumerate(obs_template['phobs'][sn][z][f]):
                            if obs_template['phobs'][sn][z][f][i] > endTime: break
                            ff.write('{:.2f} {:.3f} {}\n'.format(p,obs_template['magobs'][sn][z][f][i],f))
                    ff.close()


explosion_times = np.random.uniform(low=t_start, high=t_end, size=n_events)

# Set up the slicer to evaluate the catalog we just made
if coo:
    slicer = slicers.UserPointsSlicer(coo['ra'], coo['dec'], latLonDeg=True, badval=0)
else:
    ra, dec = sample_sphere(n_events, seed=seed)
    slicer = slicers.UserPointsSlicer(ra, dec, latLonDeg=True, badval=0)
# Add any additional information about each object to the slicer
slicer.slicePoints['explosion_times'] = explosion_times
slicer.slicePoints['zrange'] = zrange
return slicer
```

SNclassification_metric:

GLOBAL PARAMETER:

Survey Parameters:

- ***mjdCol***= MJD observations column name from Opsim database (DEFAULT = observationStartMJD)
- ***m5Col***= Magnitude limit column name from Opsim database (DEFAULT = fiveSigmaDepth)
- ***filterCol***= Filters column name from Opsim database (DEFAULT = filter)
- ***exptimeCol*** = Column name for the total exposure time of the visit(DEFAULT = visitExposureTime)

- ***nightCol* =** The night's column of the survey (starting at 1) (DEFAULT = night)
- ***vistimeCol* =** Column name for the total time of the visit (DEFAULT = visitTime)
- ***RACol*=** RA column name from Opsim database (DEFAULT = fieldRA)
- ***DecCol*=** Dec column name from Opsim database (DEFAULT = fieldDec)
- ***surveyDuration*=** Survey Duration (DEFAULT = 10)
- ***surveyStart*=** Survey start date (DEFAULT = None)

Detection parameters:
- ***detectSNR*=** dictionary with SNR threshold for each filter (DEFAULT = {'u': 5, 'g': 5, 'r': 5, 'i': 5, 'z': 5, 'y': 5})
- ***nFilters*=** None or list of filters constrained for the classification threshold (DEFAULT = None)
- ***ndetect*=** integer, threshold number of points detected on the lightcurve (aside from the filters) (DEFAULT = 3)

Classification parameters
- ***nclass*=** integer, threshold number of points detected on the lightcurve (aside from the filters) to be classified (DEFAULT = 3)

OUTPUT:
- ***dataout* =** True, Dictionary containing the coordinates of all the SNe detected, the time of explosions and the number of detected and no-detected along with the number of classifies, un-classified and mis-classified SNe for each type
- ***dataout* =** False, fraction of correctly classified SNe

Methods from the class are :

1. *read_lightCurve*:
   a. input: ascii file with epoch, mag and filter for each template;
   b. output: two deliverables, an array with epoch, mag and filter; the transient duration. both are given as input to the second method.
2. *make_lightCurve*:
   a. input- the outputs of the first method,
   b. output- interpolation of the template with the survey epochs.
3. sim_mag_noise:
   a. input- magnitude and snr
   b. output- simulated observations with errors
4. coadd:
   a. input- opsim's metadata
   b. output- coadd of exposures within the same night and relative magnitude limit.

```
    def read_lightCurve(self, asciifile):
        """Reads in an ascii file, 3 columns: epoch, magnitude, filter

        Returns
        -------
        numpy.ndarray
            The data read from the ascii text file, in a numpy structured array with columns
            'ph' (phase / epoch, in days), 'mag' (magnitude), 'flt' (filter for the magnitude).
        """
        if not os.path.isfile(asciifile):
            raise IOError('Could not find lightcurve ascii file %s' % (asciifile))
        self.lcv_template = np.genfromtxt(asciifile, dtype=[('ph', 'f8'), ('mag', 'f8'), ('flt', 'S1')])
        self.transDuration = self.lcv_template['ph'].max() - self.lcv_template['ph'].min()

    def make_lightCurve(self, time, filters):
        """Turn lightcurve definition into magnitudes at a series of times.

        Parameters
        ----------
        time : numpy.ndarray
            The times of the observations.
        filters : numpy.ndarray
            The filters of the observations.

        Returns
        -------
        numpy.ndarray
             The magnitudes of the object at the times and in the filters of the observations.
        """
        lcMags = np.zeros(time.size, dtype=float)
        for key in set(self.lcv_template['flt']):
            fMatch_ascii = np.where(np.array(self.lcv_template['flt']) == key)[0]
            # Interpolate the lightcurve template to the times of the observations, in this filter.
            lc_ascii_filter = np.interp(time, np.array(self.lcv_template['ph'], float)[fMatch_ascii],
                                        np.array(self.lcv_template['mag'], float)[fMatch_ascii])
            lcMags[filters == key.decode("utf-8")] = lc_ascii_filter[filters == key.decode("utf-8")]
        lcMags += self.peakOffset
        return lcMags
```

```
def sim_mag_noise(self,mag, snr):
    noise = (snr)**(-1)
    mag_from_dist = np.random.normal(mag, noise,1)
    return mag_from_dist

def coadd(self, data):
    """
    Method to coadd data per band and per night
    Parameters
    ------------
    data : `pd.DataFrame`
        pandas df of observations
    Returns
    -------
    coadded data : `pd.DataFrame`
    """

    keygroup = [self.filterCol, self.nightCol]

    data.sort_values(by=keygroup, ascending=[
                     True, True], inplace=True)

    coadd_df = data.groupby(keygroup).agg({self.vistimeCol: ['sum'],
                                           self.exptimeCol: ['sum'],
                                           self.mjdCol: ['mean'],
                                           self.RACol: ['mean'],
                                           self.DecCol: ['mean'],
                                           self.m5Col: ['mean']}).reset_index()

    coadd_df.columns = [self.filterCol, self.nightCol,
                        self.vistimeCol, self.exptimeCol, self.mjdCol,
                        self.RACol, self.DecCol, self.m5Col]

    coadd_df.loc[:, self.m5Col] += 1.25 * \
        np.log10(coadd_df[self.vistimeCol]/30.)

    coadd_df.sort_values(by=[self.filterCol, self.nightCol], ascending=[
                         True, True], inplace=True)

    return coadd_df.to_records(index=False)
```

For each template we at a given coordinate we estimate for all the explosion times we simulate the transient explode:

- the observed epochs
- the magnitude at those epochs

- the snr of the magnitudes at each epoch

Finally we compare the snr with the threshold defined at the beginning to have the epochs at which a point on the light curve is detected.

If the number of point overcome the minimum requested for the set of filters for the light curve to be detected or also classified we count it for the estimation of the fraction of the SN detection efficiency.

```
        for k,times in enumerate(slicePoints['explosion_times']):
            sn_list+=1
            expldist.append(times)
            indexlc = np.where((obs>= times) & (obs<=times+self.transDuration)) # we create a mask for all the
observation whitin the transient duration
            lcEpoch = (obs[indexlc] - times) # define the dates of the phases from the explosion time

            if np.size(indexlc)>0:
                lcMags = self.make_lightCurve(lcEpoch, obs_filter[indexlc]) # Generate the observed light curve
magnitudes
                lcSNR = m52snr(lcMags, obs_m5[indexlc])
                lcpoints_AboveThresh = np.zeros(len(lcSNR), dtype=bool)
                sim_mag_noise = np.vectorize(self.sim_mag_noise)
                nfilt_det = []
                nfilt_class = []
                for f in self.observedFilter:
                    filtermatch = np.where(obs_filter[indexlc] == f)
                    lcpoints_AboveThresh[filtermatch] = np.where(lcSNR[filtermatch] >= self.detectSNR[f],True,False) #
we define a mask for the detected points on the light curve

                Dpoints = np.sum(lcpoints_AboveThresh) #counts the number of detected points
                if Dpoints>=self.ndetect:
                    nDetected+=1
                else:
                    nUnDetected+=1

                if self.nFilters:
                    Dpoints =0
                    nfilt_class=[]
                    for f in self.nFilters:
                        filtermatch = np.where(obs_filter[indexlc] == f)
                        epoch_filt=lcEpoch[filtermatch]
                        mask_class = epoch_filt<30
                        if any(mask_class):
                            Dpoint_class += np.sum(lcpoints_AboveThresh[filtermatch][mask_class])
                        else:
                            Dpoint_class += 0
                    if Dpoints>=self.nclass: nfilt_class.append(True)
                else:

                    mask_class = lcEpoch<30
                    if any(mask_class):

                        Dpoint_class = np.sum(lcpoints_AboveThresh[mask_class])
                    else:

                        Dpoint_class = 0
```

The light curves that satisfy the requirements for the classification are stored in a text file to pass to PSNID.

```
snname='LSST_{}_{}_{}_DDF.dat'.format(sn,z,np.round(times,2))
if self.nFilters:
    if np.sum(nfilt_class) == np.size(self.nFilters):
        listout.append(snname+'\n')
        classifiable += 1


else:
    if Dpoint_class>=self.nclass:
        classifiable += 1
        listout.append(snname+'\n')



if snname+'\n' in listout:
    lc = {}
    lc["Mags"] = lcMags
    lc["filter"] = obs_filter[indexlc]
    lc["SNR"] = lcSNR
    lc["Epoch"] = obs[indexlc]
    lc['detect'] = lcpoints_AboveThresh
    # producing a file to pass to PSNID for the classification

    mag = {}
    jd = {}
    merr = {}
    snr={}

    output  = 'SURVEY:  LSST \n'
    output += 'SNID: {}_{}_{} \n'.format(sn,z,k)
    output += 'IAUC:    UNKNOWN \n'
    output +=  'RA:      '+str(fieldRA)+'  deg \n'
    output +=  'DECL:    '+str(fieldDec)+'  deg \n'
    output +=  'MWEBV:     0.0  MW E(B-V) \n'
    output +=  'REDSHIFT_FINAL:  '+z+' +- '+'%5.3f' % self.z[2]+' (CMB)\n'
    output +=  'FILTERS:  {}   \n'.format(filterN)
    output +=  ' \n'
    output += '# ======================================\n'
    output += '# TERSE LIGHT CURVE OUTPUT\n'
    output += '#\n'
    output += 'NOBS: {} \n'.format(np.size(lcMags[lcpoints_AboveThresh]))
    output += 'NVAR: 8 \n'
    output += 'VARLIST:  MJD  FLT FIELD   FLUXCAL   FLUXCALERR   SNR    MAG     MAGERR \n'
    lcMags = sim_mag_noise(lcMags_temp[lcpoints_AboveThresh],lcSNR_temp[lcpoints_AboveThresh])
    lcSNR = m52snr(lcMags,obs_m5[indexlc][lcpoints_AboveThresh])
    for f in filterNames:
        filtermatch = np.where(obs_filter[indexlc] == f)
        detect= np.array(lc['detect'][filtermatch])
        mag[f] = lcMags[filtermatch][detect]
        jd[f] = obs[indexlc][filtermatch][detect]
        snr[f] = lcSNR[filtermatch][detect]
        merr[f] = 2.5*np.log10(1+1/snr[f])
        for h,j in enumerate(jd[f]):

            fl = 10**(-0.4*(mag[f][h]))*1e11
            if snr[f][h]>1:

                flerr = fl/snr[f][h]/1.3
            else:

                flerr = fl/1.1
            output += 'OBS: %9.3f   %s NULL  %7.3f  %7.3f  %7.3f  %7.3f  %7.3f \n' %
(j,f,fl,flerr,snr[f][h],mag[f][h],merr[f][h])
    output +='END: '


    ofile =
open(os.path.join(self.LCfolder,'LSST_{}_{}_{}_DDF.dat'.format(sn,z,np.round(times,2))),'w')
    ofile.write(output)
    ofile.close()
```

Finally it is called PSNID for the classification and the output is then analysed to check the fraction of corrected classified, mis-classified and unclassified SNe. Ultimately a confusion matrix is estimated to store the measured fractions.

```
            r = subprocess.check_output([os.environ['SNANA_DIR']+'/bin/psnid.exe',
os.environ['LSST_DIR']+'/PSNID_LSST_'+self.name+'.nml'], stderr=subprocess.STDOUT)

            # we search for classification flags in the variable r
            line= np.array(r.split())
            custom_split = np.vectorize(self.custom_split)
            types = np.where(line==b'type')
            sn_t = custom_split(x=line[np.where(line==b'Done')[0]+3],c='_',index=0)
            z_t = custom_split(x=line[np.where(line==b'Done')[0]+3],c='_',index=1)
            float_z = np.vectorize(float)
            z_t = float_z(z_t)
            z = np.unique(z_t)

            # confusion matrix
            for zz in z:
                nClassified=0
                CM[zz]['Ia']['Ia']=0
                CM[zz]['Ibc']['Ia']=0
                CM[zz]['II']['Ia']=0
                CM[zz]['UKNOWN']['Ia']=0
                CM[zz]['Ia']['Ibc']=0
                CM[zz]['Ibc']['Ibc']=0
                CM[zz]['II']['Ibc']=0
                CM[zz]['UKNOWN']['Ibc']=0
                CM[zz]['Ia']['II']=0
                CM[zz]['Ibc']['II']=0
                CM[zz]['II']['II']=0
                CM[zz]['UKNOWN']['II']=0
                z_index = np.in1d(z_t,[zz])
                type_z= line[types[0]+2][z_index]
                sn_Ia= np.in1d(sn_t,Ia)[z_index]
                sn_Ibc= np.in1d(sn_t,Ibc)[z_index]
                sn_II= np.in1d(sn_t,II)[z_index]
                CM[zz]['Ia']['Ia']+=np.nansum(type_z[sn_Ia]==b'Ia')/(np.nansum(sn_Ia))
                CM[zz]['Ibc']['Ia']+=np.nansum(type_z[sn_Ia]==b'Ibc')/(np.nansum(sn_Ia))
                CM[zz]['II']['Ia']+=np.nansum(type_z[sn_Ia]==b'II')/(np.nansum(sn_Ia))
                CM[zz]['UKNOWN']['Ia']+=np.nansum(type_z[sn_Ia]==b'UNKNOWN')/(np.nansum(sn_Ia))
                CM[zz]['Ia']['Ibc']+=np.nansum(type_z[sn_Ibc]==b'Ia')/(np.nansum(sn_Ibc))
                CM[zz]['Ibc']['Ibc']+=np.nansum(type_z[sn_Ibc]==b'Ibc')/(np.nansum(sn_Ibc))
                CM[zz]['II']['Ibc']+=np.nansum(type_z[sn_Ibc]==b'II')/(np.nansum(sn_Ibc))
                CM[zz]['UKNOWN']['Ibc']+=np.nansum(type_z[sn_Ibc]==b'UNKNOWN')/(np.nansum(sn_Ibc))
                CM[zz]['Ia']['II']+=np.nansum(type_z[sn_II]==b'Ia')/(np.nansum(sn_II))
                CM[zz]['Ibc']['II']+=np.nansum(type_z[sn_II]==b'Ibc')/(np.nansum(sn_II))
                CM[zz]['II']['II']+=np.nansum(type_z[sn_II]==b'II')/(np.nansum(sn_II))
                CM[zz]['UKNOWN']['II']+=np.nansum(type_z[sn_II]==b'UNKNOWN')/(np.nansum(sn_II))
                CM[zz]=CM[zz].fillna(0)
                nClassified+=
np.nansum(type_z[sn_Ia]==b'Ia')+np.nansum(type_z[sn_Ia]==b'Ibc')+np.nansum(type_z[sn_Ia]==b'II')
                +np.nansum(type_z[sn_Ibc]==b'Ia')+np.nansum(type_z[sn_Ibc]==b'Ibc')+np.nansum(type_z[sn_Ibc]==b'II')
                +np.nansum(type_z[sn_II]==b'Ia')+np.nansum(type_z[sn_II]==b'Ibc')+np.nansum(type_z[sn_II]==b'II')
                nUnClassified = classify['nFiltered'][zz]['filtered_class']-nClassified
                classify['nClassified'][zz]['classified']+= nClassified
                classify['nClassified'][zz]['unclassified']+= nUnClassified
```