```
    1: # -*- coding: utf-8 -*-
    2: """
    3: Created on Tue Feb 23 11:52:39 2016
    4:
    5: @author: nknezek
    6: """
    7:
    8: from scipy.misc import factorial as _factorial
    9: from scipy.special import lpmv as _lpmv
   10: from numpy import sin as _sin
   11: from numpy import cos as _cos
   12: from numpy import zeros as _zeros
   13: from numpy import sum as _sum
   14: from numpy import exp as _exp
   15: import numpy as _np
   16: from .bspline2 import Bspline as _Bspline
   17: import os as _os
   18: import pyshtools as _sht
   19: import warnings as _warnings
   20:
   21: _gufm1_data_file = _os.path.dirname(_os.path.abspath(__file__)) + '/data/gufm1_dat
a.txt'
   22: _gufmsatE3_data_file = _os.path.dirname(_os.path.abspath(__file__)) + '/data/gufm-
sat-E3.txt'
   23: _gufmsatQ2_data_file = _os.path.dirname(_os.path.abspath(__file__)) + '/data/gufm-
sat-Q2.txt'
   24: _gufmsatQ3_data_file = _os.path.dirname(_os.path.abspath(__file__)) + '/data/gufm-
sat-Q3.txt'
   25: _chaos6_data_file = _os.path.dirname(_os.path.abspath(__file__)) + '/data/chaos6_d
ata.txt'
   26:
   27: class SphereHarmBase:
   28:     def __init__(self):
   29:         self.l_max = None
   30:
   31:     def SHrcmb2vs(self, Clm_rcmb, r_cmb=3480., r_s=6371.2, l_max=None):
   32:         """converts radial 4pm SH at CMB to potential SH at surface
   33:
   34:         Converts a radial field measurement in 4pi normalized spherical harmonics
at the CMB to the potential field
   35:         spherical harmonics at the surface.
   36:
   37:         Parameters
   38:         ----------
   39:         Clm_rcmb
   40:         r_cmb
   41:         r_s
   42:         l_max
   43:
   44:         Returns
   45:         -------
   46:         Clm_s = coefficients array with 4pi, csphase=1 normalization
   47:         """
   48:         if l_max is None:
   49:             if self.l_max is None:
   50:                 l_max = 14
   51:             else:
   52:                 l_max = self.l_max
   53:         Clm_in = self._convert_SHin(Clm_rcmb, l_max=l_max)
   54:         Clm_s = _np.zeros_like(Clm_in)
   55:         for l in range(Clm_in.shape[1]):
   56:             vs2rc = (r_s / r_cmb) ** (l + 2) * (l + 1)
   57:             Clm_s[:, l, :] = Clm_rcmb[:, l, :] / vs2rc
   58:         return Clm_s
   59:
   60:     def _convert_SHin(self, SHin, l_max=None):
   61:         """helper function to convert sht classes and arrays for functions
   62:
   63:         Parameters
   64:         ----------
```

```
   65:         SHin: _sht.SHCoeffs class or numpy array
   66:         l_max: l_max
   67:
   68:         Returns
   69:         -------
   70:             np.array of SH coefficients with normalization 4pi, csphase=1
   71:         """
   72:         if type(SHin) is _np.ndarray:
   73:             if SHin.shape[0] == 2 and len(SHin.shape) == 3:
   74:                 if l_max is None:
   75:                     l_max = SHin.shape[1] - 1
   76:                 if SHin.shape[1] >= l_max + 1:
   77:                     SHout = SHin[:, :l_max + 1, :l_max + 1]
   78:                 else:
   79:                     raise TypeError('SHin does not contain enough coefficients for
 requested l_max')
   80:             elif len(SHin.shape) < 3:
   81:                 if l_max is None:
   82:                     l_max = int(SHin.shape[0] ** 0.5 - 1)
   83:                 if SHin.shape[0] >= (l_max + 1) ** 2:
   84:                     SHout = _sht.shtools.SHVectorToCilm(SHin[:(l_max + 1) ** 2])
   85:                 else:
   86:                     raise TypeError('SHin does not contain enough coefficients for
 requested l_max')
   87:             else:
   88:                 raise TypeError('SHin not in a recognizable format or the wrong si
ze for l_max')
   89:         elif (type(SHin) is _sht.shclasses.SHRealCoeffs) or (type(SHin) is _sht.sh
classes.SHComplexCoeffs):
   90:             SHout = SHin.to_array(normalization='4pi', csphase=1, lmax=SHin.lmax)
   91:         else:
   92:             raise TypeError('SHin not in a recognizable format or the wrong size f
or l_max')
   93:         return SHout
   94:
   95:     def get_thvec_phvec_DH(self, Nth=None, l_max=None):
   96:         """returns theta and ph coordinate vectors for DH grid
   97:
   98:         :param Nth:
   99:         :param l_max:
  100:         :return:
  101:         """
  102:         if l_max is None:
  103:             l_max = self.l_max
  104:         if Nth is None:
  105:             Nth = l_max * 2 + 2
  106:         Nph = 2 * Nth
  107:         dth = 180 / Nth
  108:         th = _np.linspace(dth / 2, 180 - dth / 2, Nth)
  109:         ph = _np.linspace(dth / 2, 360 - dth / 2, Nph)
  110:         return th, ph
  111:
  112:     def _get_lmax(self, SHin):
  113:         """helper function to get the l_max of a set of spherical harmonics
  114:
  115:         :param SHin:
  116:         :return:
  117:         """
  118:         if type(SHin) is _np.ndarray:
  119:             if SHin.shape[0] == 2 and len(SHin.shape) == 3:
  120:                 l_max = int(SHin.shape[1]-1)
  121:                 if not SHin.shape == (2,l_max + 1,l_max+1):
  122:                     raise TypeError('SH is not the right shape l_max')
  123:             elif len(SHin.shape) < 3:
  124:                 l_max = int(SHin.shape[0]**0.5-1)
  125:                 if not SHin.shape[0] == (l_max+1)**2:
  126:                     raise TypeError('SHin does not contain enough coefficients for
 requested l_max')
  127:             else:
  128:                 raise TypeError('SHin not in a recognizable format or the wrong si
```

```
ze for l_max')
        elif (type(SHin) is _sht.shclasses.SHRealCoeffs) or (type(SHin) is _sht.sh
classes.SHComplexCoeffs):
            l_max = int(SHin.lmax)
        else:
            raise TypeError('SHin not in a recognizable format or the wrong size f
or l_max')
        return l_max

class MagModel(SphereHarmBase):
    def __init__(self, data_file):
        super(MagModel,self).__init__()
        self.data_file = data_file
        self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
le=self.data_file)
        self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
        self.bspline = self._make_bspline_basis(self.tknts)
        self.T_start = self.tknts[self.bspl_order-1]
        self.T_end = self.tknts[-self.bspl_order]

    def _read_data(self, data_file=None):
        """ read data from datafile

        :param data_file:
        :return:
        """
        if data_file is None:
            data_file = self.data_file
        with open(data_file,'rb') as f:
            f.readline()
            line1 = f.readline().split()

            l_max = int(line1[0])
            nspl = int(line1[1])
            if float(line1[2]) < 1000:
                bspl_order = int(line1[2])
                l1_tknt_loc = 3
            else:
                bspl_order = 4
                l1_tknt_loc = 2

            n = l_max*(l_max+2)

            gt = _zeros(n*nspl)
            tknts = _zeros(nspl+bspl_order)
            tknt_l1 = [float(x) for x in line1[l1_tknt_loc:]]
            tknts[:len(tknt_l1)] = tknt_l1
            ti = len(tknt_l1)
            gi = 0
            for line in f:
                l_tmp = [float(x) for x in line.split()]
                nl = len(l_tmp)
                if ti+nl <= len(tknts):
                    tknts[ti:ti+nl] = l_tmp
                    ti += nl
                else:
                    gt[gi:gi+nl] = l_tmp
                    gi += nl
        gt_out = gt.reshape(n, nspl, order='F')
        return gt_out, tknts, l_max, bspl_order

    def _read_coeffs(self, data_file=None):
        """ read raw coefficients from data file

        :param data_file:
        :return:
        """
        if data_file is None:
            data_file = self.data_file
        with open(data_file,'rb') as f:
```

```
195:             f.readline()
196:             line1 = f.readline().split()
197:
198:             l_max = int(line1[0])
199:
200:             raw = []
201:             for line in f:
202:                 l_tmp = [float(x) for x in line.split()]
203:                 for l in l_tmp:
204:                     raw.append(l)
205:
206:         g_raw = _np.array(raw)
207:         return g_raw, l_max
208:
209:     def _make_bspline_basis(self, tknts, order=None):
210:         """ create the bspline basis function for interpolting vs time
211:
212:         :param tknts:
213:         :param order:
214:         :return:
215:         """
216:         if order is None:
217:             order = self.bspl_order
218:         bspline = _Bspline(tknts, order)
219:         return bspline
220:
221:     def _interval(self, time):
222:         """Calculates nleft: the index of the timeknot on the left of the interval
223:
224:             tknts[nleft] < tknts[nleft+1]
225:             tknts[nleft] <= time <= tknts[nleft+1]
226:
227:         Parameters
228:         ----------
229:         time:
230:             the time to calculate the field
231:
232:         Returns
233:         -------
234:         the index of the time knot on the left of the interval
235:         """
236:         tknts = self.tknts
237:         if (time >= tknts[self.bspl_order-1] and time <= tknts[-self.bspl_order]):
238:             for n in range(self.bspl_order-1,len(tknts)-self.bspl_order+1):
239:                 if time >= tknts[n]:
240:                     nleft = n-self.bspl_order+1
241:                 else:
242:                     break
243:         else:
244:             raise IndexError("The time you've chosen is outside this model")
245:         return nleft
246:
247:     def _Pml(self, x, l, m):
248:         """Associated Legendre Polynomial - Schmidt Quasi-Normalization
249:
250:         ============================================================
251:         Returns the evaulated Associated Legendre Polynomial of degree n and order
 m at location x.
252:
253:         This function evaluates the Associated Legendre Polynomials with Schmidt Q
uasi Normalization as defined in Schmidt (1917, p281).
254:         It uses the scipy built in associated legendre polynomials which have Ferr
er's normalization and converts the normalization.
255:
256:         Inputs
257:         -------
258:         x:
259:             Location of evaluation
260:         l:
261:             Degree of associated legendre polynomial
```

```
  262:         m:
  263:             Order of associated legendre polynomial
  264:
  265:         Returns
  266:         -------
  267:         The value of the polynomial at location specified. (float)
  268:
  269:         Associated Legendre Polynomial Normalizations:
  270:         ------
  271:
  272:         Schmidt Quasi-Normalized:
  273:             P^m_l(x) = sqrt{2*(l-m)!/(l+m)!}(1-x^2)^{m/2}(d/dx)^2 P_l(x)
  274:
  275:         Ferrer's (only for reference):
  276:             P^m_n(x) = (-1)^m(1-x^2)^{m/2}(d/dx)^2 P_n(x)
  277:
  278:         """
  279:         if m == 0:
  280:             return (_factorial(l-m)/_factorial(l+m))**0.5/(-1)**m*_lpmv(m,l,x)
  281:         else:
  282:             return (2*_factorial(l-m)/_factorial(l+m))**0.5/(-1)**m*_lpmv(m,l,x)
  283:
  284:     def _dtheta_Pml(self, x, l, m):
  285:         """
  286:         Theta derivative of Associated Legendre Polynomial - Schmidt Quasi-Normali
zation
  287:         ============================================================
  288:         Returns the theta derivative of the Associated Legendre Polynomial of degr
ee n and order m at location x=cos(theta).
  289:
  290:         Inputs
  291:         -------
  292:         x:
  293:             Location of evaluation ( cos(theta) )
  294:         l:
  295:             Degree of associated legendre polynomial
  296:         m:
  297:             Order of associated legendre polynomial
  298:
  299:         Returns
  300:         -------
  301:         The theta derivative of the polynomial at location specified. (float)
  302:
  303:         Associated Legendre Polynomial Normalizations:
  304:         ------
  305:
  306:         Schmidt Quasi-Normalized:
  307:             P^m_l(x) = sqrt{2*(l-m)!/(l+m)!}(1-x^2)^{m/2}(d/dx)^2 P_l(x)
  308:
  309:         Theta derivative:
  310:             d_theta P^m_l( x=cos(th) ) = -1/sqrt{1-x^2} * [ (l+1)*x*P^m_l(x) + (l+
1-m)*P^m_(l+1)(x) ]
  311:         """
  312:         if m == 0:
  313:             return 1/(1-x**2)**0.5 * ( -(l+1)*x*self._Pml(x, l, m) + (l+1-m)*(_fac
torial(l-m)/_factorial(l+m))**0.5/(-1)**m*_lpmv(m,l+1,x))
  314:         else:
  315:             return 1/(1-x**2)**0.5 * ( -(l+1)*x*self._Pml(x, l, m) + (l+1-m)*(2*_f
actorial(l-m)/_factorial(l+m))**0.5/(-1)**m*_lpmv(m,l+1,x))
  316:
  317:     def _calculate_g_raw_at_t(self, time):
  318:         """
  319:         Calculates the Gauss Coefficients in raw ordering given the parameters cal
culated by inverval() and _bspline().
  320:
  321:         Parameters
  322:         ----------
  323:         time:
  324:
  325:         Returns
```

```
326:         -------
327:             Gauss Coefficients for a particular time in raw ordering.
328:         """
329:         gt = self.gt
330:         b = self.bspline(time)
331:         i = self._interval(time)
332:         bo = self.bspl_order
333:         g_raw = _sum(b[i:i+bo]*gt[:, i:i+bo], axis=1)
334:         return g_raw
335:
336:     def _convert_g_raw_to_shtarray(self, g_raw, l_max=None):
337:         """ BROKEN -- Converts g_raw computed for a time to shtools formatted arra
y
338:
339:         Inputs
340:         ------
341:         g_raw:
342:             numpy array of g_raw, standard ordering as on single-time g_raw files
from website.
343:         l_max:
344:             spherical harmonic degree included in model (automatically taken from
data_file)
345:         Returns
346:         -------
347:         coeffs:
348:             (2,l_max+1, l_max+1) size array of Gauss coefficients where e.g. coeff
s[0,2,1] = g(l=2, m=1), coeffs[1,2,0] = h(l=2, m=0)
349:
350:         """
351:         # if not l_max:
352:         #     l_max = self.l_max
353:         # coeffs = _np.zeros((2,l_max+1, l_max+1))
354:         # coeffs[0,1,0] = g_raw[0]
355:         # coeffs[0,1,1] = g_raw[1]
356:         # coeffs[1,1,1] = g_raw[2]
357:         # i = 3
358:         # for l in range(2,l_max+1):
359:         #     coeffs[0,l,0] = g_raw[i]
360:         #     i += 1
361:         #     for m in range(1,l+1):
362:         #         coeffs[0,l,m] = g_raw[i]
363:         #         i += 1
364:         #         coeffs[1,l,m] = g_raw[i]
365:         #         i += 1
366:         # return coeffs
367:         raise NotImplementedError("this function doesn't work yet")
368:
369:     def read_SH_from_file_gufm_form(self, file):
370:         """
371:         file:
372:             filename of file in gufm single-epoch form. First line is title line,
second line is l_max, then coefficients
373:
374:         Returns
375:         -------
376:             SH_gufm, l_max : _np.array of the raw coefficients and l_max
377:         """
378:         with open(file, 'rb') as f:
379:             f.readline()
380:             line1 = f.readline().split()
381:
382:             l_max = int(line1[0])
383:
384:             raw = []
385:             for line in f:
386:                 l_tmp = [float(x) for x in line.split()]
387:                 for l in l_tmp:
388:                     raw.append(l)
389:         SH_gufm = _np.array(raw)
390:         return SH_gufm, l_max
```

```
391:
392:     def write_SH_to_file_gufm_form(self, SH, file, heading, l_max, num_per_line=4)
:
393:         """
394:
395:         Parameters
396:         ----------
397:         SH: _sht.SHCoeffs class
398:         file: filename to write to
399:         heading: heading to write on line #1
400:         l_max: l_max to write on line #2
401:         num_per_line: number of coefficients to list per line
402:
403:         Returns
404:         -------
405:             none
406:         """
407:         Bschmidt = SH.to_array(normalization='schmidt', csphase=-1)
408:         SHv = self._convert_shtarray_to_gufm_form(Bschmidt, l_max=SH.lmax)
409:         with open(file, 'w') as f:
410:             f.write(heading + '\n')
411:             f.write('{0:.0f}\t0\n'.format(l_max))
412:             i = 0
413:             while (i < len(SHv)):
414:                 f.write('{0:.8e}'.format(SHv[i]))
415:                 if (i+1) % num_per_line == 0:
416:                     f.write('\n')
417:                 else:
418:                     f.write('\t')
419:                 i += 1
420:
421:     def _convert_gufm_form_to_shtarray(self, g_raw, l_max=None):
422:         """
423:         Converts g_raw computed for a time to shtools formatted array
424:
425:         Inputs
426:         ------
427:         g_raw:
428:             numpy array of g_raw, standard ordering as on single-time g_raw files
from website.
429:         l_max:
430:             spherical harmonic degree included in model (automatically taken from
data_file)
431:         Returns
432:         -------
433:         coeffs:
434:             _sht.SHCoeffs class: (2,l_max+1, l_max+1) size array of Gauss coeffici
ents where e.g. coeffs[0,2,1] = g(l=2, m=1), coeffs[1,2,0] = h(l=2, m=0)
435:
436:         """
437:         if not l_max:
438:             l_max = self.l_max
439:         coeffs = _np.zeros((2, l_max + 1, l_max + 1))
440:         coeffs[0, 1, 0] = g_raw[0]
441:         coeffs[0, 1, 1] = g_raw[1]
442:         coeffs[1, 1, 1] = g_raw[2]
443:         i = 3
444:         for l in range(2, l_max + 1):
445:             coeffs[0, l, 0] = g_raw[i]
446:             i += 1
447:             for m in range(1, l + 1):
448:                 coeffs[0, l, m] = g_raw[i]
449:                 i += 1
450:                 coeffs[1, l, m] = g_raw[i]
451:                 i += 1
452:         return _sht.SHCoeffs.from_array(coeffs, normalization='schmidt', csphase=-
1)
453:
454:     def _convert_shtarray_to_gufm_form(self, shtarray, l_max=None):
455:         """
```

```
456:         Converts g_raw computed for a time to shtools formatted array
457:
458:         Inputs
459:         ------
460:         shtarray:
461:             (2,l_max+1, l_max+1) size array of Gauss coefficients where e.g. coeff
s[0,2,1] = g(l=2, m=1), coeffs[1,2,0] = h(l=2, m=0)
462:         l_max:
463:             spherical harmonic degree included in model (automatically taken from
data_file)
464:         Returns
465:         -------
466:         gufm_raw:
467:             _np.array of length (l_max+1)**2-1 ordered according to the gufm stand
ard
468:         """
469:         shtarray = self._convert_SHin(shtarray, l_max=l_max)
470:         if l_max is None:
471:             l_max = shtarray.lmax
472:         raw = []
473:         raw.append(shtarray[0, 1, 0])
474:         raw.append(shtarray[0, 1, 1])
475:         raw.append(shtarray[1, 1, 1])
476:         for l in range(2, l_max + 1):
477:             raw.append(shtarray[0, l, 0])
478:             for m in range(1, l + 1):
479:                 raw.append(shtarray[0, l, m])
480:                 raw.append(shtarray[1, l, m])
481:         return _np.array(raw)
482:
483:     def _calculate_SVgh_raw_at_t(self, time):
484:         """
485:         Calculates the time derivatives of Gauss Coefficients in raw ordering give
n the parameters calculated by inverval() and _bspline().
486:
487:         Parameters
488:         ----------
489:         time:
490:             date in years to calculate SVg_raw
491:         Returns
492:         -------
493:         SVg_raw:
494:             Time derivatives of Gauss Coefficients for a particular time in raw or
dering.
495:         """
496:         gt = self.gt
497:         SVb = self.bspline.d(time)
498:         i = self._interval(time)
499:         bo = self.bspl_order
500:         SVg_raw = _sum(SVb[i:i+bo]*gt[:, i:i+bo], axis=1)
501:         return SVg_raw
502:
503:     def _calculate_SAgh_raw_at_t(self, time):
504:         """
505:         Calculates the second time derivatives of Gauss Coefficients in raw orderi
ng given the parameters calculated by inverval() and _bspline().
506:
507:         Parameters
508:         ----------
509:         time:
510:             date in years to calculate SAg_raw
511:         Returns
512:         -------
513:         SAg_raw:
514:             Second time derivatives of Gauss Coefficients for a particular time in
 raw ordering.
515:         """
516:         gt = self.gt
517:         SAb = self.bspline.d2(time)
518:         i = self._interval(time)
```

```
519:         bo = self.bspl_order
520:         SAg_raw = _sum(SAb[i:i+bo]*gt[:, i:i+bo], axis=1)
521:         return SAg_raw
522:
523:     def _convert_g_raw_to_gh(self, g_raw, l_max=None):
524:         """
525:         Converts g_raw computed for a time to g, h dictionaries
526:
527:         Inputs
528:         ------
529:         g_raw:
530:             numpy array of g_raw, standard ordering as on single-time g_raw files
from website.
531:         l_max:
532:             spherical harmonic degree included in model (automatically taken from
data_file)
533:         Returns
534:         -------
535:         g, h:
536:             dictionaries of Gauss coefficients ordered as g[l][m] and h[l][m]
537:
538:         """
539:         if not l_max:
540:             l_max = self.l_max
541:         g = {}
542:         h = {}
543:         g[1] = {0:g_raw[0]}
544:         g[1][1] = g_raw[1]
545:         h[1] = {0:0, 1:g_raw[2]}
546:         i = 3
547:         for l in range(2,l_max+1):
548:             g[l] = {}
549:             h[l] = {}
550:             g[l][0] = g_raw[i]
551:             i += 1
552:             h[l][0] = 0.
553:             for m in range(1,l+1):
554:                 g[l][m] = g_raw[i]
555:                 i += 1
556:                 h[l][m] = g_raw[i]
557:                 i += 1
558:         return g, h
559:
560:     def convert_gh_to_complex(self, g, h, l_max=None):
561:         """
562:         converts g,h real spherical harmonics to A complex spherical harmonic.
563:
564:         V(r,th,ph) = a sum_m,l{ \norm(a/r)**(l+1) * ( g_m,l * cos(m*ph) + h_m,l *
sin(m*ph) ) * P_m,l(cos(th)) }
565:         c(th,ph) = sum_m,l{ c_m,l * exp(i*m*ph) * P_m,l(cos(th)) }
566:
567:         Parameters
568:         ----------
569:         g: cos(m\phi) term
570:         h: sin(m\phi) term
571:
572:         Returns
573:         -------
574:         c: complex spherical harmonics coefficients
575:         """
576:         if not l_max:
577:             l_max = max(g.keys())
578:         c = {}
579:         for l in range(1,l_max+1):
580:             c[l] = {}
581:             for m in range(0,l+1):
582:                 c[l][m] = g[l][m] - 1j*h[l][m]
583:         return c
584:
585:     def convert_complex_to_gh(self, c, l_max=None):
```

```
   586:         """
   587:         converts c complex spherical harmonic to g,h real spherical harmonics.
   588:
   589:         V(r,th,ph) = a sum_m,l{ \norm(a/r)**(l+1) * ( g_m,l * cos(m*ph) + h_m,l *
sin(m*ph) ) * P_m,l(cos(th)) }
   590:         c(th,ph) = sum_m,l{ c_m,l * exp(i*m*ph) * P_m,l(cos(th)) }
   591:
   592:         Parameters
   593:         ----------
   594:         c: complex spherical harmonics coefficients
   595:
   596:         Returns
   597:         -------
   598:         g,h: real spherical harmonics coefficients
   599:         """
   600:         if not l_max:
   601:             l_max = max(c.keys())
   602:         g = {}
   603:         h = {}
   604:         for l in range(1,l_max+1):
   605:             g[l] = {}
   606:             h[l] = {}
   607:             for m in range(0,l+1):
   608:                 g[l][m] = _np.real(c[l][m])
   609:                 h[l][m] = -_np.imag(c[l][m])
   610:         return g,h
   611:
   612:     def get_shtcoeffs_at_t(self, time, l_max=None):
   613:         """
   614:         Calculates Gauss coefficients at time T
   615:
   616:         Parameters
   617:         ----------
   618:         time:
   619:             time to calculate parameters
   620:         l_max:
   621:             spherical harmonic degree included in model (14)
   622:         Returns
   623:         -------
   624:         coeffs:
   625:             array containing Gauss coefficients at time time
   626:         """
   627:         g_raw = self._calculate_g_raw_at_t(time)
   628:         return self._convert_gufm_form_to_shtarray(g_raw, l_max=l_max)
   629:
   630:     def get_sht_allT(self, T, l_max=None):
   631:         """
   632:         Calculates Gauss coefficients of secular acceleration at a list or _np.arr
ay of times T
   633:
   634:         :param T: list of _np.array of times (yr)
   635:         :param l_max:
   636:         :return:
   637:         """
   638:         sht_list = []
   639:         for t in T:
   640:             sht_list.append(self.get_shtcoeffs_at_t(t,l_max=l_max))
   641:         return sht_list
   642:
   643:     def get_SVshtcoeffs_at_t(self, time, l_max=None):
   644:         """
   645:         Calculates Gauss coefficients of secular variation at time T
   646:
   647:         Parameters
   648:         ----------
   649:         time:
   650:             time to calculate parameters
   651:         l_max:
   652:             spherical harmonic degree included in model (14)
   653:         Returns
```

```
654:         -------
655:         coeffs:
656:             array containing Gauss coefficients of secular variation at time time
657:         """
658:         g_raw = self._calculate_SVgh_raw_at_t(time)
659:         return self._convert_gufm_form_to_shtarray(g_raw, l_max=l_max)
660:
661:     def get_SVsht_allT(self, T, l_max=None):
662:         """
663:         Calculates Gauss coefficients of secular variation at a list or _np.array
of times T
664:
665:         :param T: list of _np.array of times (yr)
666:         :param l_max:
667:         :return:
668:         """
669:         sht_list = []
670:         for t in T:
671:             sht_list.append(self.get_SVshtcoeffs_at_t(t,l_max=l_max))
672:         return sht_list
673:
674:     def get_SAshtcoeffs_at_t(self, time, l_max=None):
675:         """
676:         Calculates Gauss coefficients of secular acceleration at time T
677:
678:         Parameters
679:         ----------
680:         time:
681:             time to calculate parameters
682:         l_max:
683:             spherical harmonic degree included in model (14)
684:         Returns
685:         -------
686:         coeffs:
687:             array containing Gauss coefficients of secular acceleration at time ti
me
688:         """
689:         g_raw = self._calculate_SAgh_raw_at_t(time)
690:         return self._convert_gufm_form_to_shtarray(g_raw, l_max=l_max)
691:
692:     def get_SAsht_allT(self, T, l_max=None):
693:         """
694:         Calculates Gauss coefficients of secular acceleration at a list or _np.arr
ay of times T
695:
696:         :param T: list of _np.array of times (yr)
697:         :param l_max:
698:         :return:
699:         """
700:         sht_list = []
701:         for t in T:
702:             sht_list.append(self.get_SAshtcoeffs_at_t(t,l_max=l_max))
703:         return sht_list
704:
705:     def B_sht(self, shtcoeffs, r=3480, Nth=None, l_max=None, a=6371.2):
706:         """
707:         Calculates the radial magnetic field on a Driscoll-Healy grid given spheri
cal harmonics coefficients, using shtools
708:
709:         :param shtcoeffs: spherical harmonics coefficients in SHT format
710:         :param r: radius of caculation (km)
711:         :param Nth: Number of latitudinal grid point points to output (number of l
ongitudinal points Nph = 2*Nth)
712:         :param l_max: maximum spherical harmonic degree to use in computation
713:         :param a: radius of data (6371.2 km by default)
714:         :return:
715:             data on a Nth x Nph grid
716:         """
717:
718:         if l_max is None:
```

```
719:             l_max = shtcoeffs.lmax
720:
721:         # compute parameters for SHTools
722:         if Nth is None:
723:             lm = l_max
724:         else:
725:             lm = (Nth-2)//2
726:
727:         # catch bad inputs
728:         if l_max > shtcoeffs.lmax:
729:             l_max = shtcoeffs.lmax
730:             _warnings.warn('l_max set to {} as maximum degree available in provide
d coefficients'.format(l_max), UserWarning)
731:         if l_max > lm:
732:             lm = l_max
733:             _warnings.warn('grid size increased to Nth={} as must have Nth >= 2*l_
max+2'.format(lm*2+2), UserWarning)
734:         coeffs = self._convert_SHin(shtcoeffs, l_max=l_max)
735:         out = _sht.shtools.MakeGravGridDH(coeffs, a**2, a, a=r, sampling=2, lmax=l
m, lmax_calc=l_max)
736:         return -out[0]
737:
738:     def B_sht_allT(self, sht_allT, r=3480, Nth=None, l_max=None, a=6371.2):
739:         """
740:         Calculates the radial magnetic field on a Driscoll-Healy grid given a list
 of spherical harmonics coefficients, using shtools
741:
742:         :param sht_allT:
743:         :param r:
744:         :param Nth:
745:         :param l_max:
746:         :param a:
747:         :return:
748:         """
749:         B0 = self.B_sht(sht_allT[0], r=r, Nth=Nth, l_max=l_max, a=a)
750:         B_t = _np.empty((len(sht_allT), B0.shape[0], B0.shape[1]))
751:         for i,sh in enumerate(sht_allT):
752:             B_t[i,:,:] = self.B_sht(sh, r=r, Nth=Nth, l_max=l_max, a=a)
753:         return B_t
754:
755:     def gradB_sht(self, shtcoeffs, r=3480, Nth=None, l_max=None, a=6371.2):
756:         """
757:         Find the gradient of the B field, given a list of spherical harmonics coef
ficients
758:
759:         :param shtcoeffs: spherical harmonics of the field in [nT]
760:         :param r:
761:         :param Nth:
762:         :param l_max:
763:         :param a:
764:         :return: drB, dthB, dphB
765:             drB : radial gradient of the field in [nT/km]
766:             dthB : latitudinal gradient of the field [nT/km]
767:             dphB : longitudinal gradient of the field [nT/km]
768:         """
769:
770:         if type(shtcoeffs) is not _np.ndarray:
771:             coeffs = shtcoeffs.to_array(normalization='4pi', csphase=1)
772:         else:
773:             coeffs = shtcoeffs
774:         if l_max is None:
775:             l_max = self.l_max
776:         if Nth is None:
777:             lm = l_max
778:         else:
779:             lm = (Nth-2)/2
780:         out = _sht.shtools.MakeGravGradGridDH(coeffs, a ** 2, a, a=r, sampling=2,
lmax=lm, lmax_calc=l_max)
781:         drB = out[2]
782:         dthB = out[4]
```

```
   783:         dphB = out[5]
   784:         return drB, dthB, dphB
   785:
   786:     def gradB_sht_allT(self, sht_allT, r=3480, Nth=None, l_max=None, a=6371.2):
   787:         drB0, dthB0, dphB0 = self.gradB_sht(sht_allT[0], r=r, Nth=Nth, l_max=l_max
, a=a)
   788:         drB_t = _np.empty((len(sht_allT), drB0.shape[0], drB0.shape[1]))
   789:         dthB_t = _np.empty((len(sht_allT), dthB0.shape[0], dthB0.shape[1]))
   790:         dphB_t = _np.empty((len(sht_allT), dphB0.shape[0], dphB0.shape[1]))
   791:         for i,sh in enumerate(sht_allT):
   792:             drB_t[i,:,:], dthB_t[i,:,:], dphB_t[i,:,:] = self.gradB_sht(sh, r=r, N
th=Nth, l_max=l_max, a=a)
   793:         return drB_t, dthB_t, dphB_t
   794:
   795:     def get_gh_at_t(self, time, l_max=None):
   796:         """
   797:         Calculates Gauss coefficients at time T
   798:
   799:         Parameters
   800:         ----------
   801:         time:
   802:             time to calculate parameters
   803:         l_max:
   804:             spherical harmonic degree included in model (14)
   805:
   806:         Returns
   807:         -------
   808:         g_dict, h_dict:
   809:             dictionaries containing Gauss coefficients at time time
   810:         """
   811:         g_raw = self._calculate_g_raw_at_t(time)
   812:         g_dict, h_dict = self._convert_g_raw_to_gh(g_raw, l_max=l_max)
   813:         return g_dict, h_dict
   814:
   815:     def get_SVgh_at_t(self, time, l_max=None):
   816:         """
   817:         Calculates Gauss coefficients at time T
   818:
   819:         Parameters
   820:         ----------
   821:         time:
   822:             time to calculate parameters
   823:         l_max:
   824:             spherical harmonic degree included in model (14)
   825:
   826:         Returns
   827:         -------
   828:         g_dict, h_dict:
   829:             dictionaries containing Gauss coefficients at time time
   830:         """
   831:         SVg_raw = self._calculate_SVgh_raw_at_t(time)
   832:         SVg_dict, SVh_dict = self._convert_g_raw_to_gh(SVg_raw, l_max=l_max)
   833:         return SVg_dict, SVh_dict
   834:
   835:     def get_SAgh_at_t(self, time, l_max=None):
   836:         """
   837:         Calculates Gauss coefficients at time T
   838:
   839:         Parameters
   840:         ----------
   841:         time:
   842:             time to calculate parameters
   843:         l_max:
   844:             spherical harmonic degree included in model (14)
   845:
   846:         Returns
   847:         -------
   848:         g_dict, h_dict:
   849:             dictionaries containing Gauss coefficients at time time
   850:         """
```

```
851:         SAg_raw = self._calculate_SAgh_raw_at_t(time)
852:         SAg_dict, SAh_dict = self._convert_g_raw_to_gh(SAg_raw, l_max=l_max)
853:         return SAg_dict, SAh_dict
854:
855:     def _Br_for_ml(self,r,th,ph,g,h,m,l, a=6371.2):
856:         """
857:         Calculates the Br contribution for one set of m,l, using the potential fie
ld.
858:
859:         Inputs
860:         ------
861:         r:
862:             radius location (km)
863:         th:
864:             latitude location (radians)
865:         ph:
866:             longitude location (radians)
867:         g:
868:             Gauss coefficient (cos term)
869:         h:
870:             Gauss coefficient (sin term)
871:         m:
872:             Order of calculation
873:         l:
874:             Degree of calculation
875:         a:
876:             Radius (km) at which Gauss coefficients are calculated
877:
878:         Returns
879:         -------
880:         Br contribution in Tesla at a particular point from a particular degree an
d order.
881:         """
882:         return (l+1.)*a**(l+2.)/abs(r)**(l+2.)*(g*_cos(m*ph) + h*_sin(m*ph))*self.
_Pml(_cos(th), l, m)
883:
884:     def _Br_for_ml_complex(self,r,th,ph,c,m,l, a=6371.2):
885:         """
886:         Calculates the Br contribution for one set of m,l, using the potential fie
ld.
887:
888:         Inputs
889:         ------
890:         r:
891:             radius location (km)
892:         th:
893:             latitude location (radians)
894:         ph:
895:             longitude location (radians)
896:         c:
897:             complex gauss coefficient
898:         m:
899:             Order of calculation
900:         l:
901:             Degree of calculation
902:         a:
903:             Radius (km) at which Gauss coefficients are calculated
904:
905:         Returns
906:         -------
907:         Br contribution in Tesla at a particular point from a particular degree an
d order.
908:         """
909:         return (l+1.)*a**(l+2.)/abs(r)**(l+2.)*c*_exp(1j*m*ph)*self._Pml(_cos(th),
 l, m)
910:
911:     def _dphi_Br_for_ml_complex(self, r, th, ph, c, m, l, a=6371.2):
912:         """
913:         Calculates the d_phi(Br)/(R*sin(th)) contribution for one set of m,l, usin
g the potential field.
```

```
914:
915:         Inputs
916:         ------
917:         r:
918:             radius location (km)
919:         th:
920:             latitude location (radians)
921:         ph:
922:             longitude location (radians)
923:         c:
924:             complex gauss coefficient
925:         m:
926:             Order of calculation
927:         l:
928:             Degree of calculation
929:         a:
930:             Radius (km) at which Gauss coefficients are calculated
931:
932:         Returns
933:         -------
934:         d_phi(Br)/(R*sin(th)) in Tesla/m at a particular point from a particular d
egree and order.
935:         """
936:         return 1j*m/(r*_sin(th))*self._Br_for_ml_complex(r, th, ph, c, m, l, a=a)
937:
938:     def _dtheta_Br_for_ml_complex(self, r, th, ph, c, m, l, a=6371.2):
939:         """
940:         Calculates the d_theta(Br)/R contribution for one set of m,l, using the po
tential field.
941:
942:         Inputs
943:         ------
944:         r:
945:             radius location (km)
946:         th:
947:             latitude location (radians)
948:         ph:
949:             longitude location (radians)
950:         c:
951:             complex gauss coefficient
952:         m:
953:             Order of calculation
954:         l:
955:             Degree of calculation
956:         a:
957:             Radius (km) at which Gauss coefficients are calculated
958:
959:         Returns
960:         -------
961:         d_theta(Br)/R in Tesla at a particular point from a particular degree and
order.
962:         """
963:         return (l+1.)*a**(l+2.)/abs(r)**(l+2.)*c*_exp(1j*m*ph)*self._dtheta_Pml(_c
os(th), l, m)/r
964:
965:     def _SVBr_for_ml(self,r,th,ph,SVg,SVh,m,l, a=6371.2):
966:         """
967:         Calculates the SVBr contribution for one set of m,l, using the potential f
ield.
968:
969:         Inputs
970:         ------
971:         r:
972:             radius location (km)
973:         th:
974:             latitude location (radians)
975:         ph:
976:             longitude location (radians)
977:         g:
978:             Gauss coefficient (cos term)
```

```
   979:         h:
   980:             Gauss coefficient (sin term)
   981:         m:
   982:             Order of calculation
   983:         l:
   984:             Degree of calculation
   985:         a:
   986:             Radius (km) at which Gauss coefficients are calculated
   987:
   988:         Returns
   989:         -------
   990:         SVBr contribution in Tesla at a particular point from a particular degree
and order.
   991:         """
   992:         return self._Br_for_ml(r,th,ph,SVg,SVh,m,l, a=a)
   993:
   994:     def _SABr_for_ml(self,r,th,ph,SAg,SAh,m,l, a=6371.2):
   995:         """
   996:         Calculates the SABr contribution for one set of m,l, using the potential f
ield.
   997:
   998:         Inputs
   999:         ------
  1000:         r:
  1001:             radius location (km)
  1002:         th:
  1003:             latitude location (radians)
  1004:         ph:
  1005:             longitude location (radians)
  1006:         g:
  1007:             Gauss coefficient (cos term)
  1008:         h:
  1009:             Gauss coefficient (sin term)
  1010:         m:
  1011:             Order of calculation
  1012:         l:
  1013:             Degree of calculation
  1014:         a:
  1015:             Radius (km) at which Gauss coefficients are calculated
  1016:
  1017:         Returns
  1018:         -------
  1019:         SVBr contribution in Tesla at a particular point from a particular degree
and order.
  1020:         """
  1021:         return self._Br_for_ml(r,th,ph,SAg,SAh,m,l, a=a)
  1022:
  1023:     def Br(self,r,th,ph, g_dict, h_dict, l_max=None):
  1024:         """
  1025:         Calculates the total radial magnetic field at a particular location, given
 a dictionary of gauss coefficients.
  1026:
  1027:         Inputs
  1028:         ------
  1029:         r:
  1030:             radius location (km)
  1031:         th:
  1032:             latitude location (radians)
  1033:         ph:
  1034:             longitude location (radians)
  1035:         g_dict:
  1036:             dictionary of g (cos) Gauss coefficients, ordered as g[l][m].
  1037:         h_dict:
  1038:             dictionary of h (sin) Gauss coefficients, ordered as h[l][m]. h coeffi
cients for m=0 should be explicitly included as 0.0
  1039:         l_max:
  1040:             maximum degree to use in calculation. By default uses all supplied deg
rees.
  1041:
  1042:         Returns
```

```
1043:         -------
1044:         Total Br at a particular point (Tesla)
1045:         """
1046:         if l_max is None:
1047:             l_max = max(g_dict.keys())
1048:         Br_sum = 0
1049:         for l in range(1,l_max+1):
1050:             for m in range(l+1):
1051:                 Br_sum += self._Br_for_ml(r,th,ph, g_dict[l][m], h_dict[l][m], m,
l)
1052:         return Br_sum
1053:
1054:     def Br_complex(self, r, th, ph, c_dict, l_max=None):
1055:         """
1056:         Calculates the total radial magnetic field at a particular location, give
a dictionary of complex gauss coefficients.
1057:
1058:         Inputs
1059:         ------
1060:         r:
1061:             radius location (km)
1062:         th:
1063:             latitude location (radians)
1064:         ph:
1065:             longitude location (radians)
1066:         c_dict:
1067:             dictionary of complex Gauss coefficients, ordered as c[l][m].
1068:         l_max:
1069:             maximum degree to use in calculation. By default uses all supplied deg
rees.
1070:
1071:         Returns
1072:         -------
1073:         Total Br at a particular point (Tesla, complex)
1074:         """
1075:         if l_max is None:
1076:             l_max = max(c_dict.keys())
1077:         Br_sum = 0
1078:         for l in range(1,l_max+1):
1079:             for m in range(l+1):
1080:                 Br_sum += self._Br_for_ml_complex(r,th,ph, c_dict[l][m], m, l)
1081:         return Br_sum
1082:
1083:     def grad_Br_complex(self,r,th,ph, c_dict, l_max=None):
1084:         """
1085:         Calculates the hoizontal gradient of the total radial magnetic field at a
particular location, give a dictionary of complex gauss coefficients.
1086:
1087:         Inputs
1088:         ------
1089:         r:
1090:             radius location (km)
1091:         th:
1092:             latitude location (radians)
1093:         ph:
1094:             longitude location (radians)
1095:         c_dict:
1096:             dictionary of complex Gauss coefficients, ordered as c[l][m].
1097:         l_max:
1098:             maximum degree to use in calculation. By default uses all supplied deg
rees.
1099:
1100:         Returns
1101:         -------
1102:         grad_theta(Br), grad_phi(Br) at a particular point (Tesla, complex)
1103:         """
1104:         if l_max is None:
1105:             l_max = max(c_dict.keys())
1106:         dth_Br_sum = 0
1107:         dph_Br_sum = 0
```

```
 1108:         for l in range(1,l_max+1):
 1109:             for m in range(l+1):
 1110:                 dth_Br_sum += self._dtheta_Br_for_ml_complex(r,th,ph, c_dict[l][m]
, m, l)
 1111:                 dph_Br_sum += self._dphi_Br_for_ml_complex(r,th,ph, c_dict[l][m],
m, l)
 1112:         return dth_Br_sum, dph_Br_sum
 1113:
 1114:     def SVBr(self,r,th,ph, SVg_dict, SVh_dict, l_max=None):
 1115:         """
 1116:         Calculates the total radial magnetic field at a particular location, give
a dictionary of gauss coefficients.
 1117:
 1118:         Inputs
 1119:         ------
 1120:         r:
 1121:             radius location (km)
 1122:         th:
 1123:             latitude location (radians)
 1124:         ph:
 1125:             longitude location (radians)
 1126:         g_dict:
 1127:             dictionary of g (cos) Gauss coefficients, ordered as g[l][m].
 1128:         h_dict:
 1129:             dictionary of h (sin) Gauss coefficients, ordered as h[l][m]. h coeffi
cients for m=0 should be explicitly included as 0.0
 1130:         l_max:
 1131:             maximum degree to use in calculation. By default uses all supplied deg
rees.
 1132:
 1133:         Returns
 1134:         -------
 1135:         Total Br at a particular point (Tesla)
 1136:         """
 1137:         if l_max is None:
 1138:             l_max = max(SVg_dict.keys())
 1139:         SVBr_sum = 0
 1140:         for l in range(1,l_max+1):
 1141:             for m in range(l+1):
 1142:                 SVBr_sum += self._SVBr_for_ml(r,th,ph, SVg_dict[l][m], SVh_dict[l]
[m], m, l)
 1143:         return SVBr_sum
 1144:
 1145:     def SABr(self,r,th,ph, SAg_dict, SAh_dict, l_max=None):
 1146:         """
 1147:         Calculates the total radial magnetic field at a particular location, give
a dictionary of gauss coefficients.
 1148:
 1149:         Inputs
 1150:         ------
 1151:         r:
 1152:             radius location (km)
 1153:         th:
 1154:             latitude location (radians)
 1155:         ph:
 1156:             longitude location (radians)
 1157:         g_dict:
 1158:             dictionary of g (cos) Gauss coefficients, ordered as g[l][m].
 1159:         h_dict:
 1160:             dictionary of h (sin) Gauss coefficients, ordered as h[l][m]. h coeffi
cients for m=0 should be explicitly included as 0.0
 1161:         l_max:
 1162:             maximum degree to use in calculation. By default uses all supplied deg
rees.
 1163:
 1164:         Returns
 1165:         -------
 1166:         Total Br at a particular point (Tesla)
 1167:         """
 1168:         if l_max is None:
```

```
 1169:             l_max = max(SAg_dict.keys())
 1170:         SABr_sum = 0
 1171:         for l in range(1,l_max+1):
 1172:             for m in range(l+1):
 1173:                 SABr_sum += self._SABr_for_ml(r,th,ph, SAg_dict[l][m], SAh_dict[l]
[m], m, l)
 1174:         return SABr_sum
 1175:
 1176:     def compute_Bdata_allT(self, T, Nth, l_max=None, B_lmax=None, SV_lmax=None, SA
_lmax=None):
 1177:         if l_max is None:
 1178:             l_max = self.l_max
 1179:         if B_lmax is None:
 1180:             B_lmax = l_max
 1181:         if SV_lmax is None:
 1182:             SV_lmax = l_max
 1183:         if SA_lmax is None:
 1184:             SA_lmax = l_max
 1185:         Bsh = self.get_sht_allT(T, l_max=B_lmax)
 1186:         B = self.B_sht_allT(Bsh, Nth=Nth, l_max=B_lmax)
 1187:         _, dthB, dphB = self.gradB_sht_allT(Bsh, Nth=Nth, l_max=B_lmax)
 1188:
 1189:         SVsh = self.get_SVsht_allT(T, l_max=SV_lmax)
 1190:         SV = self.B_sht_allT(SVsh, Nth=Nth, l_max=SV_lmax)
 1191:         _, dthSV, dphSV = self.gradB_sht_allT(SVsh, Nth=Nth, l_max=SV_lmax)
 1192:
 1193:         SAsh = self.get_SAsht_allT(T, l_max=SA_lmax)
 1194:         SA = self.B_sht_allT(SAsh, Nth=Nth, l_max=SA_lmax)
 1195:         _, dthSA, dphSA = self.gradB_sht_allT(SAsh, Nth=Nth, l_max=SA_lmax)
 1196:
 1197:         return (B, dthB, dphB, Bsh, SV, dthSV, dphSV, SVsh, SA, dthSA, dphSA, SAsh
)
 1198:
 1199: class Gufm1(MagModel):
 1200:     def __init__(self, data_file = _gufm1_data_file):
 1201:         self.data_file = data_file
 1202:         self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
le=self.data_file)
 1203:         self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
 1204:         self.bspline = self._make_bspline_basis(self.tknts)
 1205:         self.T_start = self.tknts[self.bspl_order-1]
 1206:         self.T_end = self.tknts[-self.bspl_order]
 1207:         self.name = 'GUFM-1'
 1208:
 1209: class GufmSatE3(MagModel):
 1210:     def __init__(self, data_file = _gufmsatE3_data_file):
 1211:         self.data_file = data_file
 1212:         self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
le=self.data_file)
 1213:         self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
 1214:         self.bspline = self._make_bspline_basis(self.tknts)
 1215:         self.T_start = self.tknts[self.bspl_order-1]
 1216:         self.T_end = self.tknts[-self.bspl_order]
 1217:         self.name = 'GUFM-SAT-E3'
 1218:
 1219: class GufmSatQ2(MagModel):
 1220:     def __init__(self, data_file = _gufmsatQ2_data_file):
 1221:         self.data_file = data_file
 1222:         self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
le=self.data_file)
 1223:         self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
 1224:         self.bspline = self._make_bspline_basis(self.tknts)
 1225:         self.T_start = self.tknts[self.bspl_order-1]
 1226:         self.T_end = self.tknts[-self.bspl_order]
 1227:         self.name = 'GUFM-SAT-Q2'
 1228:
 1229: class GufmSatQ3(MagModel):
 1230:     def __init__(self, data_file = _gufmsatQ3_data_file):
 1231:         self.data_file = data_file
 1232:         self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
```

```
le=self.data_file)
 1233:         self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
 1234:         self.bspline = self._make_bspline_basis(self.tknts)
 1235:         self.T_start = self.tknts[self.bspl_order-1]
 1236:         self.T_end = self.tknts[-self.bspl_order]
 1237:         self.name = 'GUFM-SAT-Q3'
 1238:
 1239: class Chaos6(MagModel):
 1240:     def __init__(self, data_file = _chaos6_data_file):
 1241:         self.data_file = data_file
 1242:         self.gt, self.tknts, self.l_max, self.bspl_order = self._read_data(data_fi
le=self.data_file)
 1243:         self.dT = self.tknts[len(self.tknts)//2+1]-self.tknts[len(self.tknts)//2]
 1244:         self.bspline = self._make_bspline_basis(self.tknts)
 1245:         self.T_start = self.tknts[self.bspl_order-1]
 1246:         self.T_end = self.tknts[-self.bspl_order]
 1247:         self.name = 'CHAOS-6'
 1248:
```