MSG

Rich Townsend & The MSG Team

Sep 19, 2023

USER GUIDE

Multidimensional Spectral Grids (MSG) is an open-source software package that synthesizes stellar spectra and photometric colors via interpolation in pre-calculated grids. MSG is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, version 3.

CHAPTER

ONE

PRELIMINARIES

1.1 Why use MSG?

Synthesizing stellar spectra from first principles is a complicated endeavor, requiring a detailed understanding of radiative transfer and atomic physics, together with significant computational resources. Therefore, in most circumstances its better to use one of the many pre-calculated grids of spectra published in the astrophysical literature (see, e.g., Lanz & Hubeny, 2003; Lanz & Hubeny, 2007; Kirby, 2011; de Laverny et al., 2012; Husser et al., 2013; Allende Prieto et al., 2018; Chiavassa et al., 2018; Zsargó et al., 2020). However, even with these grids a significant obstacle remains: when photospheric parameters fall between the grid nodes, some kind of interpolation is necessary in order to evaluate a spectrum.

MSG is designed to solve this problem. It's not the first software package that offers stellar spectral interpolation (see, e.g., FERRE, Starfish and stsynphot); however, with spectral interpolation as its *sole* focus, it offers a combination of features unmatched by other packages:

- scalability — MSG handles grids that are much larger (on disk) than available computer memory.
- extensibility — MSG handles grids with an arbitrary number of dimensions.
- portability — MSG is platform-agnostic and provide APIs for the programming languages (Fortran, C, Python) most commonly used in Astronomy.
- performance — MSG provides smooth and accurate interpolates with minimal computational cost.
- robustness — MSG gracefully handles missing data caused by holes and/or ragged boundaries in the grid.

Together, these features mean that MSG is flexible and powerful while remaining straightforward to use: it's the perfect condiment to add flavor to your science!.

1.2 Obtaining MSG

The source code for MSG is hosted in the https://github.com/rhdtownsend/msg git repository on GitHub. Instructions for downloading and installing the software can be found in the *Quick Start* chapter.

1.3 Citing MSG

If you use MSG in your research, please cite the following papers:

- Townsend Rich, Lopez Aaron, 2022, *Journal of Open-Source Software*, 8(81), 4602, https://doi.org/10.21105/joss.04

Be sure to also cite the source of the grid data that you're using with MSG. For instance, if you're working with one of the CAP18 grids, you should cite Allende Prieto et al. (2018).

1.4 Development Team

MSG remains under active development by the following team:

- Rich Townsend (University of Wisconsin-Madison); project leader

If you are interested in contributing toward further development of MSG, please see the *Contributing* chapter.

1.5 Related Links

- The MESA Software Development Kit (SDK), which provides the compilers and supporting libraries needed to build MSG.

1.6 Acknowledgments

MSG has been developed with financial support from the following grants:

- NSF awards ACI-1663696 and AST-1716436;
- NASA award 80NSSC20K0515.

CHAPTER

TWO

QUICK START

To get started with MSG, follow these simple steps:

- Install the MESA Software Development Kit.
- Download the MSG source code.
- Unpack the source code using the **tar** utility.
- Set the `MSG_DIR` environment variable to point to the newly created source directory.
- Compile MSG using the command **make -C $MSG_DIR**.
- Test MSG using the command **make -C $MSG_DIR test**.
- Install the *pymsg* Python module using the command **pip install $MSG_DIR/python**.

(the last step can be skipped if you don't plan to use the Python interface).

For a more in-depth installation guide that covers alternative use-cases, refer to the *Installation* chapter. If the code doesn't compile properly, consult the *Troubleshooting* chapter. Otherwise, proceed to the next chapter where you'll undertake your first MSG calculations.

CHAPTER

THREE

WALKTHROUGHS

The best way to familiarize yourself with MSG is use it. This chapter provides example walkthroughs in Python, Fortran, and C, each focused on calculating the spectrum and colors of https://en.wikipedia.org/wiki/Sirius (α Canis Majoris A).

3.1 Python Walkthrough

This section walks through using the MSG Python interface to evaluate spectra and photometric colors for a model of Sirius. The code fragments are presented as a sequence of Jupyter notebook cells, but pasting all of them into a Python script should work also. Alternatively, the entire notebook (including this text) is available in the source distribution at `$MSG_DIR/docs/source/user-guide/walkthroughs/python-walkthrough.ipynb`.

3.1.1 Preparation

Before starting Jupyter, make sure that you have installed the *pymsg* package as described in the *Quick Start* and/or *Installation* chapters. Also, ensure that the NumPy, Matplotlib and Astropy packages are installed on your system. Finally, confirm that the `MSG_DIR` environment variable is set.

3.1.2 Initialization

Launch Jupyter, create a new Python3 notebook, and initialize the notebook with the following statements:

[1]:
```
# Import standard modules

import os
import numpy as np
import astropy.constants as con
import astropy.units as unt
import matplotlib.pyplot as plt

# Import pymsg

import pymsg

# Set plot parameters

%matplotlib inline
plt.rcParams.update({'font.size': 16})
```

3.1.3 Loading & Inspecting the Demo Grid

For demonstration purposes MSG includes a low-resolution specgrid (spectroscopic grid) file, `$MSG_DIR/data/grids/sg-demo.h5`[1]. Load this demo grid into memory[2] by creating a new *pymsg.SpecGrid* object:

[2]:
```
# Load the SpecGrid

MSG_DIR = os.environ['MSG_DIR']
GRID_DIR = os.path.join(MSG_DIR, 'data', 'grids')

specgrid_file_name = os.path.join(GRID_DIR, 'sg-demo.h5')

specgrid = pymsg.SpecGrid(specgrid_file_name)
```

This object has a number of (read-only) properties that inform us about the parameter space spanned by the grid:

[3]:
```
# Inspect grid parameters

print('Grid parameters:')

for label in specgrid.axis_labels:
    print(f'  {label} [{specgrid.axis_x_min[label]} -> {specgrid.axis_x_max[label]}]')

print(f'  lam [{specgrid.lam_min} -> {specgrid.lam_max}]')
```

```
Grid parameters:
  Teff [3500.0 -> 50000.0]
  log(g) [0.0 -> 5.0]
  lam [2999.9999999999977 -> 9003.490078514782]
```

Here, `Teff` and `log(g)` correspond (respectively) to the $T_{\rm eff}/{\rm K}$ and $\log_{10}(g/{\rm cm\,s^{-2}})$ photospheric parameters, while `lam` is wavelength $\lambda/\mathring{\rm A}$.

3.1.4 Plotting the Flux

With the grid loaded, let's evaluate and plot a flux spectrum for Sirius A. First, store the photospheric parameters for the star in a dict:

[4]:
```
# Set Sirius A's fundamental parameters

M = 2.02 *con.M_sun
R = 1.71 * con.R_sun
L = 25.4 * con.L_sun

p = 0.379
d = 1/p * con.pc

# Set photospheric parameters dict

Teff = (L/(4*np.pi*R**2*con.sigma_sb))**0.25/unt.K
logg = np.log10(con.G*M/R**2/(unt.cm/unt.s**2))
```

(continues on next page)

[1] Larger grids can be downloaded separately; see the grid files web page.
[2] Behind the scenes, the data in the specgrid file are loaded on demand; see the *Data Caching* chapter for further details.

(continued from previous page)

```
x = {'Teff': Teff, 'log(g)': logg}
print(Teff, logg)
```

```
9909.16258994498 4.277426775965449
```

(the mass, radius and luminosity data are taken from Liebert et al., 2005, while the parallax comes from Simbad). Then set up a uniformly-spaced wavelength abcissa for a spectrum spanning the visible range, 3 000 Å to 7 000 Å.

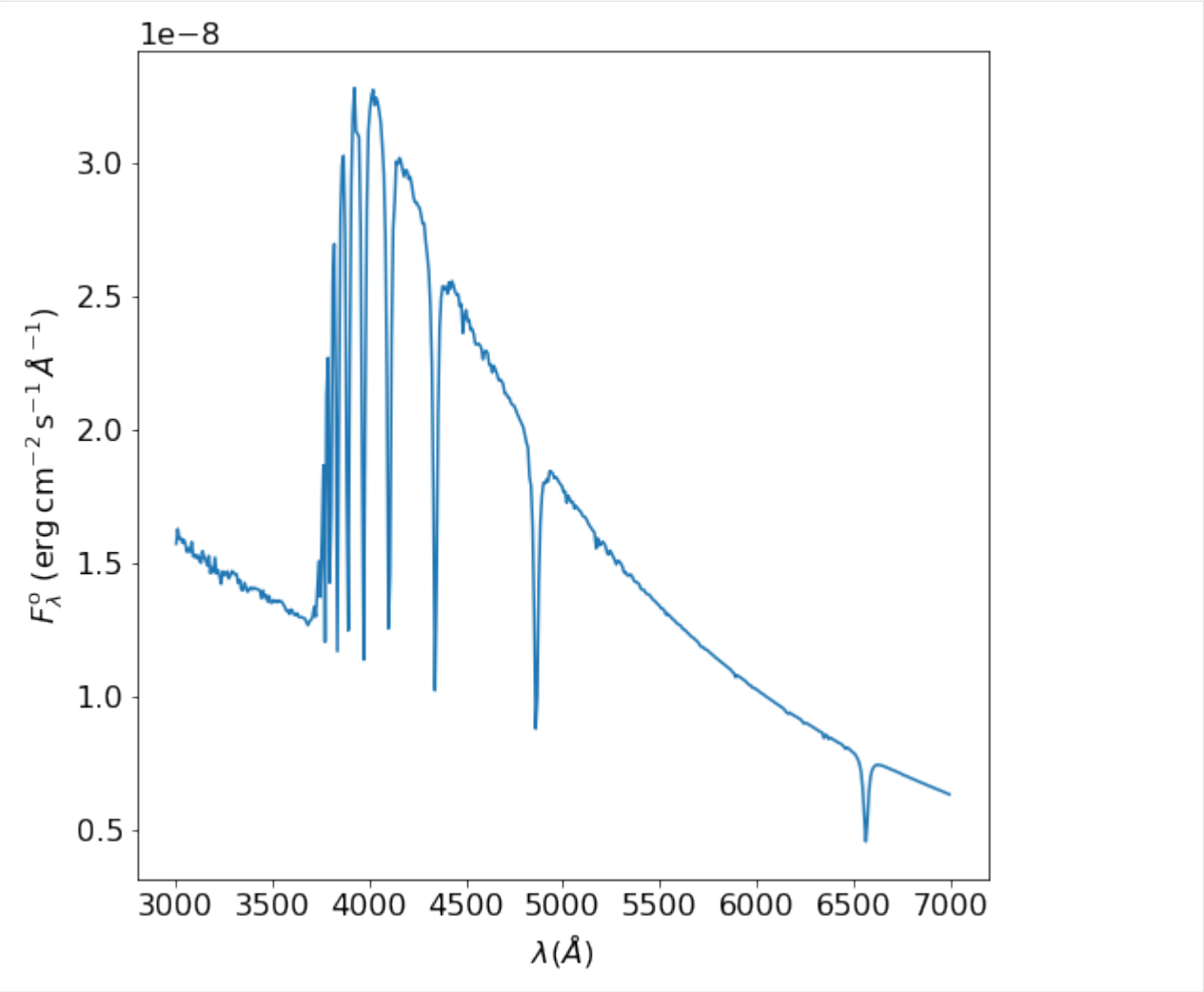
[5]:
```
# Set up the wavelength abscissa

lam_min = 3000.
lam_max = 7000.

lam = np.linspace(lam_min, lam_max, 501)

lam_c = 0.5*(lam[1:] + lam[:-1])
```

Here, the array `lam` defines the boundaries of 500 wavelength bins $\{\lambda_i, \lambda_{i+1}\}$ $(i = 1, \ldots, 500)$ and the array `lam_c` stores the central wavelength of each bin.

With all our parameters defined, evaluate the observed flux using a call to the *pymsg.SpecGrid.flux()* method, and then plot it:

[6]:
```
# Evaluate the observed flux

F_obs = (R/d)**2*specgrid.flux(x, lam)

# Plot

plt.figure(figsize=[8,8])
plt.plot(lam_c, F_obs)

plt.xlabel(r'$\lambda ({\AA})$')
plt.ylabel(r'$F^{\mathrm{o}}_{\lambda}\ ({\rm erg\,cm^{-2}\,s^{-1}}\,\AA^{-1})$')
```

[6]:
```
Text(0, 0.5, '$F^{\\mathrm{o}}_{\\lambda}\\ ({\\rm erg\\,cm^{-2}\\,s^{-1}}\\,\\AA^{-1})$
↪')
```



(See equation (4.7) of the *MSG Fundamentals* chapter to understand the origin of the expression for `F_obs`). The plot looks about right — we can clearly see the Balmer series, starting with Hα at 6563 Å.

3.1.5 Plotting the Intensity

Sometimes we want to know the specific intensity of the radiation emerging from each element of a star's surface; an example might be when we're modeling a rotationally distorted star, whose surface geometry and properties vary with latitude. For this, we can use the *`pymsg.SpecGrid.intensity()`* method.

Here's a demonstration of this method in action, plotting the specific intensity across the Hα line profile for ten different values of the emergence direction cosine $\mu = 0.1, 0.2, \ldots, 1.0$.

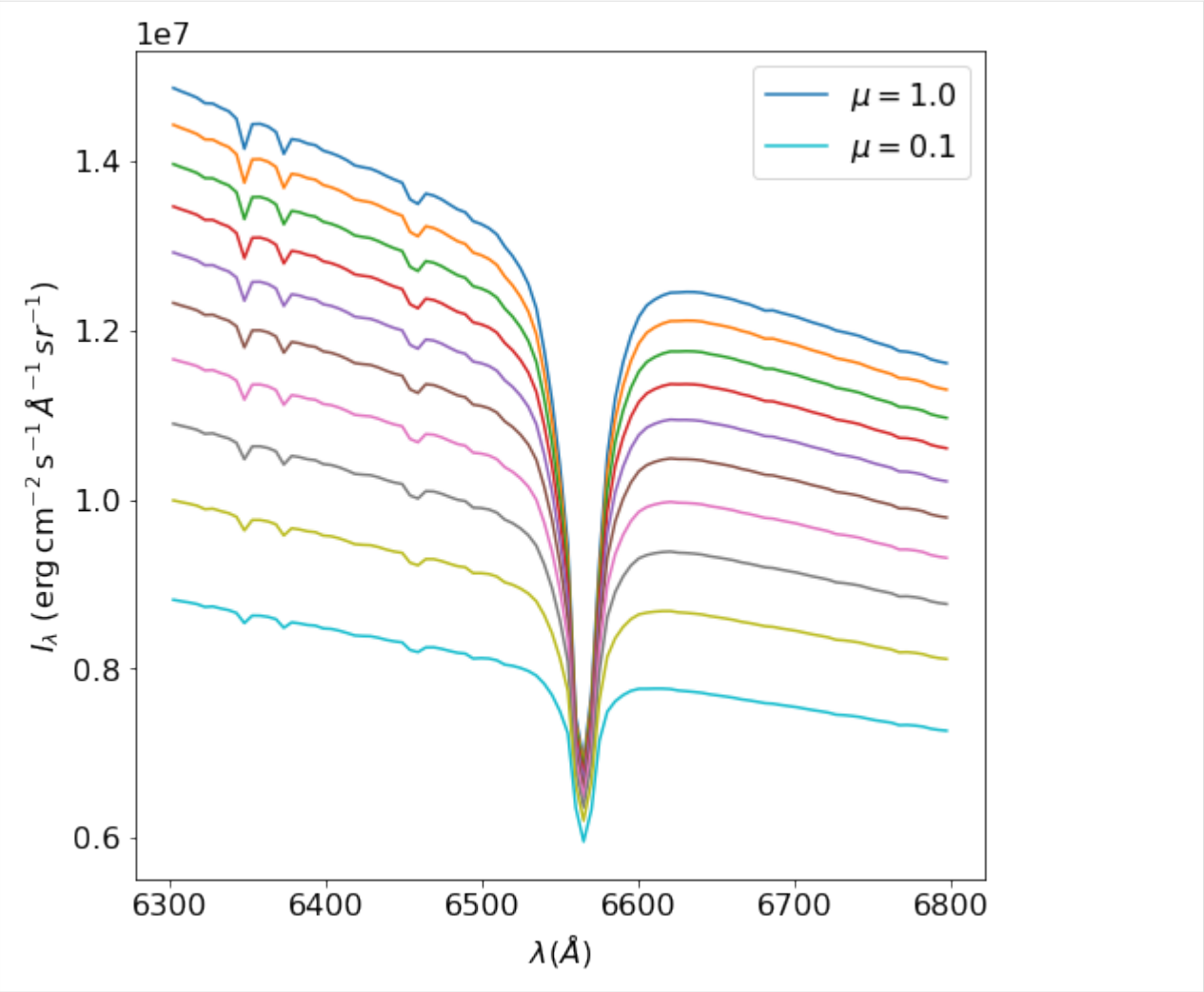
[7]:
```
# Set up the wavelength abscissa

lam_min = 6300.
lam_max = 6800.

lam = np.linspace(lam_min, lam_max, 100)
```

(continues on next page)

(continued from previous page)

```
lam_c = 0.5*(lam[1:] + lam[:-1])

# Loop over mu

plt.figure(figsize=[8,8])

for mu in np.linspace(1.0, 0.1, 10):

    # Evaluate the intensity

    I = specgrid.intensity(x, mu, lam)

    # Plot

    if mu==0.1 or mu==1.0:
        label=r'$\mu={:3.1f}$'.format(mu)
    else:
        label=None

    plt.plot(lam_c, I, label=label)

plt.xlabel(r'$\lambda ({\AA})$')
plt.ylabel(r'$I_{\lambda}\ ({\rm erg\,cm^{-2}\,s^{-1}}\,\AA^{-1}\,sr^{-1})$')

plt.legend()
```

[7]:
```
<matplotlib.legend.Legend at 0x7ff643465670>
```



We can clearly see that limb-darkening in the line core is much weaker than in the continuum — exactly what we expect from such a strong line.

3.1.6 Evaluating Magnitudes & Colors

As a final step in this walkthrough, let's evaluate the magnitude and colors of Sirius A in the Johnson photometric system. We can do this by creating a new *pymsg.PhotGrid* object for each passband:

[8]:
```
# Load the PhotGrids

PASS_DIR = os.path.join(MSG_DIR, 'data', 'passbands')
filters = ['U', 'B', 'V']

photgrids = {}

for filter in filters:
    passband_file_name = os.path.join(PASS_DIR, f'pb-Generic-Johnson.{filter}-Vega.h5')
    photgrids[filter] = pymsg.PhotGrid(specgrid_file_name, passband_file_name)
```

(for convenience, we store the *pymsg.PhotGrid* objects in a dict, keyed by filter name). In the calls to the object

constructor *pymsg.PhotGrid()*, the first argument is the name of a specgrid file (here, the demo grid), and the second argument is the name of a passband file; a limited set of these files is provided in the `$MSG_DIR/data/passbands` subdirectory[3]. The observed fluxes of Sirius A are then found using the *pymsg.PhotGrid.flux()* method:

[9]:
```
# Evaluate the observed fluxes (each normalized to the passband
# zero-point flux)

F_obs = {}

for filter in filters:
    F_obs[filter] = (R/d)**2*photgrids[filter].flux(x)
```

To convert these into apparent magnitudes, we apply Pogson's logarithmic formula:

[10]:
```
# Evaluate apparent magnitudes and print out magnitude & colors

mags = {}

for filter in filters:
    mags[filter] = -2.5*np.log10(F_obs[filter])

print(f"V={mags['V']}, U-B={mags['U']-mags['B']}, B-V={mags['B']-mags['V']}")
```

```
V=-1.4466779855804992, U-B=-0.05964681835037444, B-V=0.001121934559611848
```

Reassuringly, the resulting values are within 10 millimag of Sirius A's apparent V-band magnitude and colors, as given by Simbad.

3.2 Fortran Walkthrough

This section reprises the steps of the *Python Walkthrough* — evaluating spectra and photometric colors for Sirius A — but now using the MSG Fortran interface.

3.2.1 Preparation

In your working directory, create a new file `fortran-walkthrough.f90` with the following source code:

```
program fortran_walkthrough

   ! Uses

   use forum_m
   use fmsg_m

   ! No implicit typing

   implicit none

   ! Parameters
```

(continues on next page)

[3] Passband definition files for other instruments/photometric systems can be downloaded separately; see the passband files web page.

(continued from previous page)

```
   real(RD), parameter :: lam_min = 3000._RD
   real(RD), parameter :: lam_max = 7000._RD
   integer, parameter  :: n_lam = 501

   ! Variables

   type(specgrid_t)     :: specgrid
   type(axis_t)         :: axis
   character(LABEL_LEN) :: label
   integer              :: i
   real(RD)             :: R2_d2
   real(RD)             :: lam(n_lam)
   real(RD)             :: lam_c(n_lam-1)
   real(RD)             :: x_vec(2)
   real(RD)             :: F(n_lam-1)
   real(RD)             :: F_obs(n_lam-1)
   integer              :: unit
   type(photgrid_t)     :: photgrid_U
   type(photgrid_t)     :: photgrid_B
   type(photgrid_t)     :: photgrid_V
   real(RD)             :: F_U
   real(RD)             :: F_B
   real(RD)             :: F_V
   real(RD)             :: F_U_obs
   real(RD)             :: F_B_obs
   real(RD)             :: F_V_obs
   real(RD)             :: U
   real(RD)             :: B
   real(RD)             :: V

   ! Load the specgrid

   call load_specgrid('sg-demo.h5', specgrid)

   ! Set photospheric parameters to correspond to Sirius A

   do i = 1, 2

      call specgrid%get_axis(i, axis)
      call axis%get_label(label)

      select case(label)
      case('log(g)')
         x_vec(i) = 4.2774_RD
      case('Teff')
         x_vec(i) = 9909.2_RD
      case default
         stop 'unrecognized axis label'
      end select

   end do
```

(continues on next page)

(continued from previous page)

```
   ! Set the dilution factor R2_d2 = R**2/d**2, where R is Sirius A's
   ! radius and d its distance

   R2_d2 = 2.1351E-16_RD

   ! Set up the wavelength abscissa

   lam = [((lam_min*(n_lam-i) + lam_max*(i-1))/(n_lam-1), i=1,n_lam)]

   lam_c = 0.5_RD*(lam(:n_lam-1) + lam(2:))

   ! Evaluate the observed flux

   call specgrid%interp_flux(x_vec, lam, F)

   F_obs = R2_d2*F

   ! Write it to a file

   open(NEWUNIT=unit, FILE='spectrum.dat', STATUS='REPLACE')

   do i = 1, n_lam-1
      write(unit, *) lam_c(i), F_obs(i)
   end do

   close(unit)

   ! Load the photgrids

   call load_photgrid_from_specgrid('sg-demo.h5', 'pb-Generic-Johnson.U-Vega.h5',␣
↪photgrid_U)
   call load_photgrid_from_specgrid('sg-demo.h5', 'pb-Generic-Johnson.B-Vega.h5',␣
↪photgrid_B)
   call load_photgrid_from_specgrid('sg-demo.h5', 'pb-Generic-Johnson.V-Vega.h5',␣
↪photgrid_V)

   ! Evaluate fluxes

   call photgrid_U%interp_flux(x_vec, F_U)
   call photgrid_B%interp_flux(x_vec, F_B)
   call photgrid_V%interp_flux(x_vec, F_V)

   F_U_obs = R2_d2*F_U
   F_B_obs = R2_d2*F_B
   F_V_obs = R2_d2*F_V

   ! Evaluate apparent magnitudes

   U = -2.5_RD*LOG10(F_U_obs)
   B = -2.5_RD*LOG10(F_B_obs)
   V = -2.5_RD*LOG10(F_V_obs)
```

(continues on next page)

(continued from previous page)

```
   print *, '  V=', V
   print *, 'U-B=', U-B
   print *, 'B-V=', B-V

   ! Finish

end program fortran_walkthrough
```

A few brief comments on the code:

- The `use forum_m` statement provides access to the Fortran Utility Module (ForUM). For the purposes of the demo program, this module defines the *RD* kind type parameter for double precision real variables.
- The `use fmsg_m` statement provides access to the MSG Fortran interface. Primarily, this interface serves to define the *specgrid_t* and *photgrid_t* datatypes.
- Because Fortran doesn't have `dict` datatypes, the photospheric parameters must be passed to MSG as a plain array (here, stored in the variable `x_vec`). A `select case` construct is used to make sure the correct values are stored in each array element.

3.2.2 Compiling

The next step is to compile the demo program. Make sure the `MSG_DIR` environment variable is set, as described in the *Quick Start* chapter. Then, enter the following on the command line:

```
$ gfortran -o fortran-walkthrough fortran-walkthrough.f90 -I$MSG_DIR/include `$MSG_DIR/scripts/fmsg_lin
```

The `-I$MSG_DIR/include` option tells the compiler where to find the module definition (`.mod`) files, while the `$MSG_DIR/scripts/fmsg_link` clause (note the enclosing backticks) runs a link script that returns the compiler flags necessary to link the program against the appropriate libraries.

3.2.3 Running

To run the code, first create a symbolic link to the demo grid:

```
$ ln -s $MSG_DIR/data/grids/sg-demo.h5
```

Then, execute the command

```
$ ./fortran-walkthrough
```

The code will create a file `spectrum.dat` containing the flux spectrum for Sirius A (as an ASCII table), and print out the apparent V-band magnitude and colors of the star.

3.3 C Walkthrough

This section reprises the steps of the *Python Walkthrough* — evaluating spectra and photometric colors for Sirius A — but now using the MSG C interface.

3.3.1 Preparation

In your working directory, create a new file `c-walkthrough.c` with the following source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "cmsg.h"

#define N_LAM 501
#define LAM_MIN 3000.
#define LAM_MAX 7000.

int main(int argc, char *argv[]) {

  SpecGrid specgrid;
  PhotGrid photgrid_U;
  PhotGrid photgrid_B;
  PhotGrid photgrid_V;

  char label[17];

  double x_vec[2];

  double R2_d2;

  double lam[N_LAM];
  double lam_c[N_LAM-1];
  double F[N_LAM-1];
  double F_obs[N_LAM-1];
  double F_U;
  double F_B;
  double F_V;
  double F_U_obs;
  double F_B_obs;
  double F_V_obs;
  double U;
  double B;
  double V;

  FILE *fptr;

  // Load the specgrid

  load_specgrid("sg-demo.h5", &specgrid, NULL);

  // Set photospheric parameters to correspond to Sirius A

  for(int i=0; i < 2; i++) {

    get_specgrid_axis_label(specgrid, i, label);
```

(continues on next page)

(continued from previous page)

```
  if (strcmp(label, "log(g)") == 0) {
    x_vec[i] = 4.2774;
  }
  else if (strcmp(label, "Teff") == 0) {
    x_vec[i] = 9909.2;
  }
  else {
    printf("invalid label\n");
    exit(1);
  }

}

// Set the dilution factor R2_d2 = R**2/d**2, where R is Sirius A's
// radius and d its distance

R2_d2 = 2.1351E-16;

// Set up the wavelength abscissa

for(int i=0; i < N_LAM; i++) {
  lam[i] = (LAM_MIN*(N_LAM-1-i) + LAM_MAX*i)/(N_LAM-1);
}

for(int i=0; i < N_LAM-1; i++) {
  lam_c[i] = 0.5*(lam[i] + lam[i+1]);
}

// Evaluate the flux

interp_specgrid_flux(specgrid, x_vec, N_LAM, lam, F, NULL, NULL);

for(int i=0; i < N_LAM-1; i++) {
  F_obs[i] = R2_d2*F[i];
}

// Write it to a file

fptr = fopen("spectrum.dat", "w");

for(int i=0; i < N_LAM-1; i++) {
  fprintf(fptr, "%.17e %.17e\n", lam_c[i], F_obs[i]);
}

fclose(fptr);

// Load the photgrids

load_photgrid_from_specgrid("sg-demo.h5", "pb-Generic-Johnson.U-Vega.h5", &photgrid_U,␣
↪NULL);
load_photgrid_from_specgrid("sg-demo.h5", "pb-Generic-Johnson.B-Vega.h5", &photgrid_B,␣
```

(continues on next page)

(continued from previous page)

```
↪NULL);
  load_photgrid_from_specgrid("sg-demo.h5", "pb-Generic-Johnson.V-Vega.h5", &photgrid_V,␣
↪NULL);

  // Evaluate fluxes

  interp_photgrid_flux(photgrid_U, x_vec, &F_U, NULL, NULL);
  interp_photgrid_flux(photgrid_B, x_vec, &F_B, NULL, NULL);
  interp_photgrid_flux(photgrid_V, x_vec, &F_V, NULL, NULL);

  F_U_obs = R2_d2*F_U;
  F_B_obs = R2_d2*F_B;
  F_V_obs = R2_d2*F_V;

  // Evaluate apparent magnitudes

  U = -2.5*log10(F_U_obs);
  B = -2.5*log10(F_B_obs);
  V = -2.5*log10(F_V_obs);

  printf("  V=  %.17e\n", V);
  printf("U-B=  %.17e\n", U-B);
  printf("B-V=  %.17e\n", B-V);

  // Clean up

  unload_specgrid(specgrid);

  unload_photgrid(photgrid_U);
  unload_photgrid(photgrid_B);
  unload_photgrid(photgrid_V);

  // Finish

  exit(0);

}
```

A few brief comments on the code:

- The `#include "cmsg.h"` statement includes the header definitions for the MSG C interface.
- Because C doesn't have `dict` datatypes, the photospheric parameters must be passed to MSG as a plain array (here, stored in the variable `x_vec`). A loop with `strcmp()` calls is used to make sure the correct values are stored in each array element.
- Many of the calls to MSG routines (e.g., *load_specgrid()*, *interp_specgrid_flux()*) contain `NULL` trailing arguments; these correspond to omitted optional arguments.

3.3.2 Compiling

The next step is to compile the demo program. Make sure the `MSG_DIR` environment variable is set, as described in the *Quick Start* chapter. Then, enter the following on the command line:

```
$ gcc -o c-walkthrough c-walkthrough.c -I$MSG_DIR/include `$MSG_DIR/scripts/cmsg_link`
```

The `-I$MSG_DIR/include` option tells the compiler where to find the header file, while the `$MSG_DIR/scripts/cmsg_link` clause (note the enclosing backticks) runs a link script that returns the compiler flags necessary to link the program against the appropriate libraries.

3.3.3 Running

To run the code, first create a symbolic link to the demo grid:

```
$ ln -s $MSG_DIR/data/grids/sg-demo.h5
```

Then, execute the command

```
$ ./c-walkthrough
```

The code will create a file `spectrum.dat` containing the flux spectrum for Sirius A (as an ASCII table), and print out the apparent V-band magnitude and colors of the star.

CHAPTER

FOUR

MSG FUNDAMENTALS

This chapter expands on the *Walkthroughs*, by describing in detail how MSG evaluates stellar spectra and photometric colors. It also aims to clarify the variety of different concepts that are often lumped together under the name 'spectrum'.

4.1 Elemental Spectra

The radiation emitted by a small element of a star's surface is most completely characterized by the specific intensity $I_\lambda(\lambda; \hat{\mathbf{s}}; \mathbf{x})$. This quantity is defined such that energy passing through the element into a solid angle $\mathrm{d}\Omega$ oriented along the unit direction vector $\hat{\mathbf{s}}$, and within wavelength interval $[\lambda, \lambda + \mathrm{d}\lambda]$ and time interval $\mathrm{d}t$, is

$$\mathrm{d}E = I_\lambda(\lambda; \hat{\mathbf{s}}; \mathbf{x})\, \hat{\mathbf{s}} \cdot \hat{\mathbf{n}}\, \mathrm{d}\Omega\, \mathrm{d}\lambda\, \mathrm{d}t\, \mathrm{d}A.$$

Here, $\mathrm{d}A$ is the area of the surface element, $\hat{\mathbf{n}}$ is the unit surface normal vector, and $\mathbf{x}$ is a vector specifying the photospheric parameters of the element — for instance, its effective temperature T_eff and gravity g. Integrating this equation over all solid angles yields the net energy passing through the element in the wavelength and time intervals:

$$E = F_\lambda(\lambda; \mathbf{x})\, \mathrm{d}t\, \mathrm{d}A, \tag{4.1}$$

where the flux is introduced as

$$F_\lambda(\lambda; \mathbf{x}) \equiv \int I_\lambda(\lambda; \hat{\mathbf{s}}; \mathbf{x})\, \hat{\mathbf{s}} \cdot \hat{\mathbf{n}}\, \mathrm{d}\Omega. \tag{4.2}$$

Typically the radiation field is axisymmetric around $\hat{\mathbf{n}}$, and so I_λ depends on direction solely via the angle parameter $\mu \equiv \hat{\mathbf{s}} \cdot \hat{\mathbf{n}}$. Then, the flux simplifies to

$$F_\lambda(\lambda; \mathbf{x}) = 2\pi \int_0^1 I_\lambda(\lambda; \mu; \mathbf{x})\, \mu\, \mathrm{d}\mu \tag{4.3}$$

(the lower bound on μ is set to 0 rather than -1 under the assumption that there is no external radiation at the stellar surface).

Both I_λ and F_λ can reasonably be dubbed a 'spectrum', as they each represent the distribution of electromagnetic energy with respect to wavelength. However, one should be careful to distinguish these *elemental* spectra from the *observed spectrum* of a star; this distinction is further clarified below.

Evaluating an elemental spectrum requires solution of the radiative transfer equation throughout the atmospheric layers composing the surface element. This is often far too computationally expensive to do on-the-fly. An alternative approach is to pre-calculate spectra across a multi-dimensional grid spanning a range of photospheric parameters, and then interpolate within this grid when an elemental spectrum is required for a specific $\mathbf{x}$. *This is the fundamental purpose of MSG.*

4.2 Evaluating Elemental Spectra

To evaluate an elemental specific intensity spectrum, MSG models the dependence of I_λ on each of its three arguments as follows:

- Wavelength dependence is represented using piecewise-constant functions.
- Angle dependence is represented using limb-darkening laws.
- Photospheric parameter dependence is represented using tensor surface interpolation.

The following subsections discuss these in greater detail.

4.2.1 Wavelength Dependence

The λ dependence of the specific intensity is represented as a piecewise-constant function on a wavelength abscissa $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_M\}$:

$$I_\lambda(\lambda) = I_i \qquad \lambda_i \leq \lambda < \lambda_{i+1}.$$

(for brevity, the dependence of I_λ on μ and $\mathbf{x}$ has been suppressed). Mapping intensity data onto a new abscissa $\lambda' = \{\lambda'_1, \lambda'_2, \ldots\ \lambda'_{M'}\}$ is performed conservatively, according to the expression

$$I'_i = \frac{\int_{\lambda'_i}^{\lambda'_{i+1}} I_\lambda(\lambda)\mathrm{d}\lambda}{\lambda'_{i+1} - \lambda'_i}.$$

Beyond its simplicity, the advantage of this approach (as compared to higher-order interpolations) is that the equivalent width of line profiles is preserved.

4.2.2 Angle Dependence

The μ dependence of the specific intensity is represented using limb-darkening laws. Most familiar is the linear law

$$\frac{I_\lambda(\mu)}{I_\lambda(1)} = 1 - c\,[1 - \mu]\,, \tag{4.4}$$

where $I_\lambda(1)$ represents the normally emergent ($\mu = 1$) intensity and c is the linear limb-darkening coefficient (as before, the dependence of the intensity on other parameters has been suppressed). An improved characterization involves additional μ-dependent terms on the right-hand side; for instance, the four-coefficient law devised by Claret (2000) is

$$\frac{I_\lambda(\mu)}{I_\lambda(1)} = 1 - \sum_{k=1}^{4} c_k \left[1 - \mu^{k/2}\right], \tag{4.5}$$

where there are now four limb-darkening coefficients c_k.

The advantage of using limb-darkening laws is the ease with which other useful quantities can be calculated. For instance, the flux (4.3) can be evaluated analytically, as can any of the Eddington (1926) intensity moments (or *E-moments*, as MSG terms them):

$$\mathcal{E}^i_\lambda(\lambda; \mathbf{x}) = \frac{1}{2}\int_{-1}^{1} I_\lambda(\lambda; \mu; \mathbf{x})\, \mu^i \,\mathrm{d}\mu.$$

MSG supports the following limb-darkening laws:

CONST
: Constant law, where I_λ has no dependence on μ whatsoever. This is discussed further below.

LINEAR
Linear law given in equation (4.4) above.

SQRT
Square-root law introduced by Diaz-Cordoves & Gimenez (1992).

QUAD
Quadratic law introduced by Wade & Rucinski (1985).

CLARET
Four-coefficient law introduced by Claret (2000) and given in equation (4.5) above.

The choice of law is made during grid construction (see the *Grid Tools* appendix for more details). The coefficients appearing in the limb-darkening laws (e.g., c and c_k) are typically determined from least-squares fits to tabulations of the specific intensity. In cases where these tabulations include flux but not specific intensity data, the *CONST* law is used; the angle-independent specific intensity is determined so that it produces the correct flux when evaluated using equation (4.3).

4.2.3 Photospheric Parameter Dependence

The photospheric parameter dependence of the specific intensity is represented using cubic Hermite tensor product interpolation. The appendices provide a *(relatively) gentle introduction to tensor product interpolation*. The short version is that the intensity is modeled via piecewise-cubic functions of each component of $\mathbf{x}$, constructed to be continuous and smooth at the join between each piecewise region. The derivatives at these joins are estimated using second-order finite difference approximations involving neighboring points (or first-order at grid boundaries).

Grids often contain holes and/or ragged boundaries (the latter typically arising near the edge of the region of the $T_{\rm eff} - g$ plane corresponding to super-Eddington luminosity). When an interpolation tries to access such missing data, MSG either switches to a lower-order scheme, or (if there simply aren't sufficient data to interpolate) signals an exception (see the *Exception Handling* chapter for further details).

4.3 Observed Spectra

Suppose we observe a star from Earth, at a distance d along unit direction vector $\hat{\mathbf{d}}$. The energy measured by a detector of area $A^{\rm o}$, within the usual wavelength and time intervals, is

$$E^{\rm o} = F^{\rm o}_\lambda(\lambda)\,{\rm d}t\,A^{\rm o}$$

(here and subsequently the superscript $^{\rm o}$ should be read as 'observed'), where the observed flux is introduced as

$$F^{\rm o}_\lambda(\lambda) \equiv \frac{1}{d^2}\int_{\rm vis.} I_\lambda(\lambda; -\hat{\mathbf{d}}; \mathbf{x})\,[-\hat{\mathbf{d}}\cdot\hat{\mathbf{n}}]\,{\rm d}A. \tag{4.6}$$

The integral here is similar to that in equation (4.2), but $\hat{\mathbf{s}}$ has been replaced by $-\hat{\mathbf{d}}$, the solid angle element ${\rm d}\Omega$ has been replaced by ${\rm d}A/d^2$, and the bounds of the integral are limited to the parts of the stellar surface that are visible from Earth.

However, for stars that are spherical and have photospheric parameters that don't vary across their surface, further simplifications can be made. Let θ and ϕ be the colatitude and azimuth angles in a spherical coordinate system centered on the star and with polar axis antiparallel to $\hat{\mathbf{d}}$. Then, the observed flux becomes

$$F^{\rm o}_\lambda(\lambda) = \frac{R^2}{d^2}\int_0^{2\pi}\int_0^{\pi/2} I_\lambda(\lambda; -\hat{\mathbf{d}}; \mathbf{x})\,\cos\theta\,\sin\theta\,{\rm d}\theta\,{\rm d}\phi,$$

where R is the stellar radius. Assuming an axisymmetric radiation field, this further reduces to

$$F_\lambda^{\rm o}(\lambda) = 2\pi \frac{R^2}{d^2} \int_0^{\pi/2} I_\lambda(\lambda; \cos\theta; \mathbf{x}) \, \cos\theta \, \sin\theta \, {\rm d}\theta.$$

With the substitution $\mu = \cos\theta$ (replacing a spatial coordinate θ with a directional one μ) the result pops out that

$$F_\lambda^{\rm o}(\lambda) = \frac{R^2}{d^2} F_\lambda(\lambda; \mathbf{x}). \tag{4.7}$$

Don't be fooled by the apparent triviality of this result: it means that we need only the elemental flux spectrum, and not the specific intensity, to calculate the observed spectrum of a star. This is why many spectral grids in the literature include flux spectra instead of specific intensity spectra.

However, remember that equation (4.7) applies only for spherically symmetric and fixed-$\mathbf{x}$ stars. In more complex situations, for instance when a star is rotationally oblate, spotted, pulsating or even eclipsed, evaluation of $F_\lambda^{\rm o}$ must proceed via the visible-disk integration appearing in equation (4.6), which requires the specific intensity.

4.4 Photometric Colors

To evaluate a photometric color, MSG convolves stellar spectra with an appropriate passband response function $S'(\lambda)$. This function represents the combined sensitivity of the optical pathway, filter and detector. The passband-averaged specific intensity is defined as

$$\overline{I}(\hat{\mathbf{s}}; \mathbf{x}) = \int_0^\infty I_\lambda(\lambda; \hat{\mathbf{s}}; \mathbf{x}) S'(\lambda) {\rm d}\lambda \Big/ \int_0^\infty S'(\lambda) {\rm d}\lambda \ , \tag{4.8}$$

meaning that $S'(\lambda)$ is interpreted as an *energy* response function (see appendix A of Bessell & Murphy, 2012 for a discussion of the relationship between S' and the corresponding photon response function S). The passband-averaged observed flux follows from equation (4.6) as

$$\overline{F}^{\rm o} = \frac{1}{d^2} \int_{\rm vis.} \overline{I}(-\hat{\mathbf{d}}; \mathbf{x}) \, [-\hat{\mathbf{d}} \cdot \hat{\mathbf{n}}] \, {\rm d}A,$$

and the apparent magnitude of the star is

$$m = -2.5 \log_{10} \left(\frac{\overline{F}^{\rm o}}{F_0} \right),$$

where the normalizing flux F_0 is set by the zero-point of the photometric system.

The convolution in (4.8) can be performed before or after the interpolations discussed above:

- the 'before' option performs the convolution as a pre-processing step using the **`specgrid_to_photgrid`** tool to create a photgrid file from a specgrid file (as discussed in the *Importing Data* section). This is computationally more efficient, but requires a separate photgrid file to be created for each passband.
- the 'after' option loads data from a specgrid file, but performs the convolution on-the-fly after each spectrum is interpolated. This is computationally less efficient, but incurs no storage requirements beyond the specgrid file.

CHAPTER

FIVE

DATA FILES

This chapter discusses the files in which MSG stores its data. These files adopt the HDF5 data format, a platform-neutral binary storage format with advanced features such as transparent decompression.

5.1 File Types

There are five types of MSG data files, distinguished by their differing content:

- specint files store spectroscopic intensity data for a single combination of photospheric parameters.
- photint files store photometric intensity data for a single combination of photospheric parameters.
- specgrid files store spectroscopic intensity data over a grid of photospheric parameters.
- photgrid files store photometric intensity data over a grid of photospheric parameters.
- passband files store passband response functions, used to convert spectroscopic intensities into corresponding photometric intensities.

For a detailed description of each file type, refer to the *Data Schema* chapter.

5.2 Obtaining Data

MSG ships with a limited set of data files in the `$MSG_DIR/data` subdirectory, sufficient to enable the *walkthroughs*:

- `grids/sg-demo.h5` is a specgrid file based on a solar-metallicity Kurucz atmosphere grid (ap00k2odfnew), with intensity spectra synthesized using SYNSPEC (Hubeny & Lanz, 2017).
- `passbands/pb-Generic-Johnson.*-Vega.h5` are passband files for the U, B and V filters of the Johnson photometric system.

Additional files can be downloaded separately from the grid files and passband files web pages.

5.3 Importing Data

To import an existing spectroscopic grid into MSG, first convert the individual spectra into corresponding specint files. MSG provides a number of tools to assist with this conversion; see the *Grid Tools* appendix for further details.

The next step is to create a manifest (named, say, `manifest.txt`) listing all the specint files composing the grid. This is a simple text file with each line naming one file; for instance:

```
specint-0001.h5
specint-0002.h5
specint-0003.h5
```

Then, run the **specint_to_specgrid** tool to create a specgrid file:

```
$ $MSG_DIR/bin/specint_to_specgrid manifest.txt specgrid.h5
```

To build a photgrid file from the data in a specgrid file, run the **specgrid_to_photgrid** tool:

```
$ $MSG_DIR/bin/specgrid_to_photgrid specgrid.h5 passband.h5 photgrid.h5
```

...where `passband.h5` is the name of the passband file to use. Note that it's not always necessary to create a photgrid file, as MSG can convolve with passbands on the fly (as discussed in the *Photometric Colors* section).

CHAPTER

SIX

DATA CACHING

This chapter discusses MSG's data caching, a key component of its performance and scaleability. Each instance of a grid data type (e.g., *specgrid_t* in Fortran, or *pymsg.PhotGrid* in Python) has an associated cache that temporarily stores spectroscopic and/or photometric data for use in interpolations. This grid cache is initially empty when the instance is created, but grows in size as data are loaded on-demand from disk. Eventually, once a user-definable size limit is reached, old data are discarded from the cache to make room for new.

6.1 Caching Demo

With its caching functionality, MSG can in principle work with grids that are much larger on disk than available computer memory. The following Python code fragments showcase this capability, while also illustrating the basics of inspecting and controlling a grid cache.

To get started, download the sg-CAP18-coarse.h5 spectroscopic grid, storing it your working directory. Then, initialize MSG and load the grid:

[1]:
```
# Import standard modules

import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import random as rn
import time

# Import pymsg

import pymsg

# Set plot parameters

%matplotlib inline
plt.rcParams.update({'font.size': 16})

# Load the SpecGrid

specgrid = pymsg.SpecGrid('sg-CAP18-coarse.h5')
```

Next, inspect the state of the cache attached to the `specgrid` object:

[11]:
```
# Inspect the cache

print(f'cache usage: {specgrid.cache_usage} MB')
print(f'cache limit: {specgrid.cache_limit} MB')
```

```
cache usage: 128 MB
cache limit: 128 MB
```

The *pymsg.SpecGrid.cache_usage* property reports the current memory usage (in megabytes) of the cache. Because we've just loaded `specgrid`, this usage is zero — no data have (yet) been read into memory. The *pymsg.SpecGrid.cache_limit* property likewise specifies the maximum memory usage (again, in megabytes) of the cache. By default, this limit is set to 128 MB.

Let's define a function that interpolates the visible-spectrum flux for a randomly-chosen set of photospheric parameters. Since we're interested primarily in the behavior of the grid cache, the function returns its execution time rather than interpolation result.

[3]:
```
# Random flux routine

def random_flux():

    start_time = time.process_time()

    # Set up the wavelength abscissa (visible-spectrum)

    lam_min = 3000.
    lam_max = 7000.

    lam = np.linspace(lam_min, lam_max, 1000)

    # Loop until a valid set of photospheric parameters is found

    while True:

        # Choose random photospheric parameters

        x = {}

        for label in specgrid.axis_labels:
            x[label] = rn.uniform(specgrid.axis_x_min[label], specgrid.axis_x_max[label])

        # Interpolate the flux, allowing for the fact that the
        # photospheric parameters may fall in a grid void

        try:
            F = specgrid.flux(x, lam)
            break
        except LookupError:
            pass

    end_time = time.process_time()

    return end_time-start_time
```

Running this function a few times (for now ignoring the return value), we can watch the cache fill up to the 128-MB

limit.

[4]:
```
# Run random_flux three times

for i in range(3):
    random_flux()
    print(f'cache usage: {specgrid.cache_usage} MB')
```

```
cache usage: 86 MB
cache usage: 128 MB
cache usage: 128 MB
```

If needed this limit can be increased by changing the *pymsg.SpecGrid.cache_limit* property, as this example demonstrates:

[5]:
```
# Increase the cache limit to 256 MB

specgrid.cache_limit = 256

# Run random_flux three times

for i in range(3):
    random_flux()
    print(f'cache usage: {specgrid.cache_usage} MB')
```

```
cache usage: 215 MB
cache usage: 256 MB
cache usage: 256 MB
```

Finally, the contents of the cache can be flushed (and the memory freed up) with a call to the *pymsg.SpecGrid.flush_cache()* method:

[6]:
```
# Flush the cache

print(f'cache usage: {specgrid.cache_usage} MB')

specgrid.flush_cache()

print(f'cache usage: {specgrid.cache_usage} MB')
```

```
cache usage: 256 MB
cache usage: 0 MB
```

6.2 Wavelength Subsetting

When working with spectroscopic grids, it's often the case one is interested only in a subset of the wavelength range covered by the grid. As an example, the `random_flux()` function defined above focuses on just the visible part of the electromagnetic spectrum. In such cases, you can direct MSG to cache only the data within the subset by setting the *pymsg.SpecGrid.cache_lam_min* and *pymsg.SpecGrid.cache_lam_min* properties. This has the benefit of slowing the rate at which the cache fills up, as the following example demonstrates:

[7]:
```
# Subset to the visible part of the spectrum
```

(continues on next page)

(continued from previous page)

```
specgrid.cache_lam_min = 3000.
specgrid.cache_lam_max = 7000.

# Run random_flux three times

for i in range(3):
    random_flux()
    print(f'cache usage: {specgrid.cache_usage} MB')
```

```
cache usage: 18 MB
cache usage: 30 MB
cache usage: 55 MB
```

(compare this against the cache growth in the preceding examples).

6.3 Performance Impact

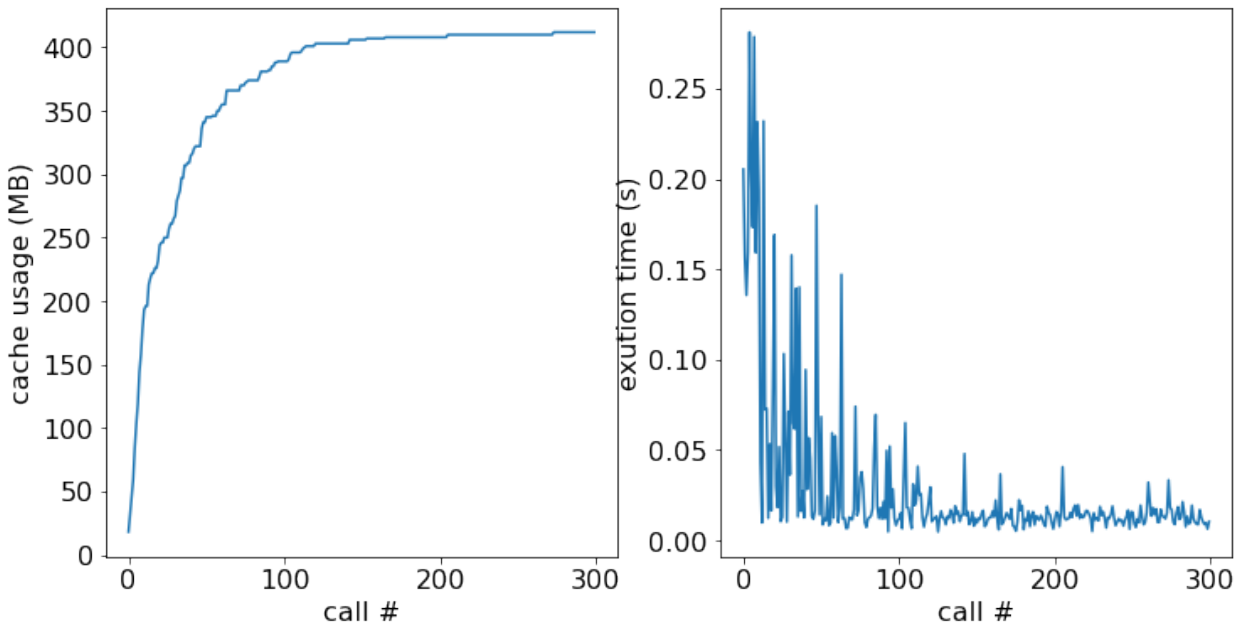
Let's now explore how caching can have a significant impact on MSG's performance. Run the following code, which expands the cache limit further to 512 MB, and then calls `random_spectrum()` 300 times, storing the usage and execution time after each call:

[8]:
```
# Flush the cache and set the limit to 512 MB

specgrid.flush_cache()
specgrid.cache_limit = 512

# Allocate usage & timing arrays

n = 300

cache_usages_512 = np.empty(n, dtype=int)
exec_timings_512 = np.empty(n)

# Call random_flux

for i in range(n):
    exec_timings_512[i] = random_flux()
    cache_usages_512[i] = specgrid.cache_usage
```

Then, plot the results:

[9]:
```
# Plot cache usage and execution time

fig, ax = plt.subplots(ncols=2, figsize=[12,6])

ax[0].plot(cache_usages_512)
ax[0].set_xlabel('call #')
ax[0].set_ylabel('cache usage (MB)')

ax[1].plot(exec_timings_512)
ax[1].set_xlabel('call #')
ax[1].set_ylabel('exution time (s)')
```

[9]:
```
Text(0, 0.5, 'exution time (s)')
```



In the left-hand panel we see the cache usage grow and eventually asymptote toward the point where the entire grid (around 400 MB) is resident in memory. In the right-hand panel the execution time rapidly drops from ~0.2s down to ~0.01s, as more and more data can be read from memory (fast) rather than disk (slow).

Let's now repeat the exercise, but for a cache limit reduced back down to 128 MB:

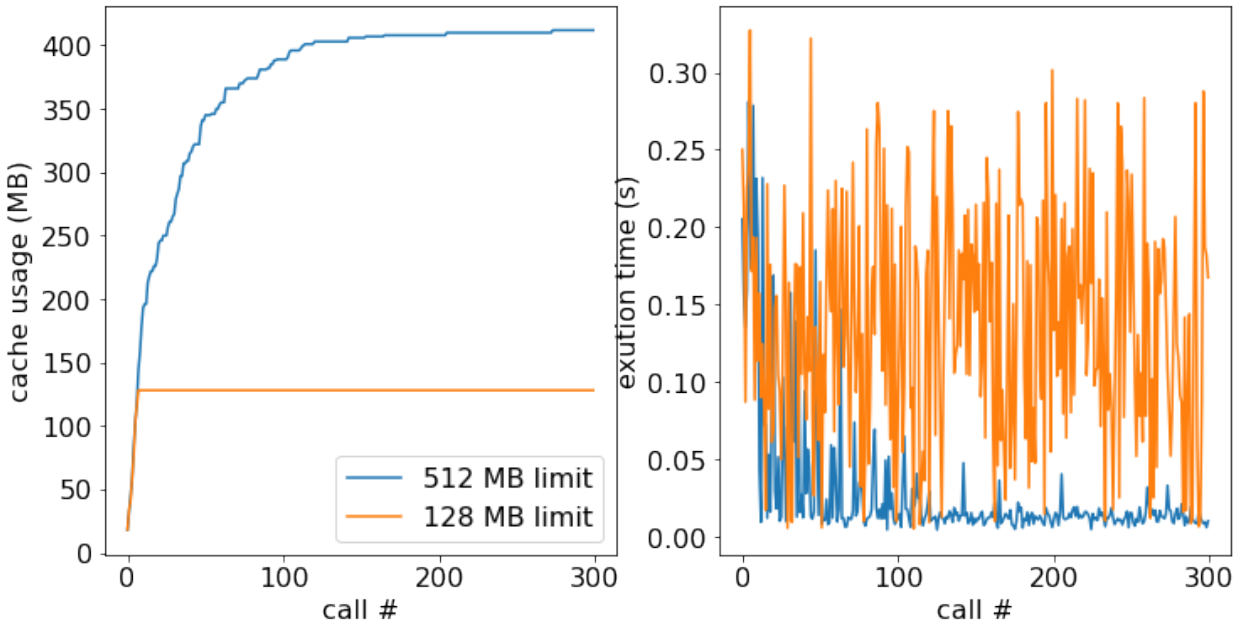
[10]:
```
# Flush the cache and set the limit to 128 MB

specgrid.flush_cache()
specgrid.cache_limit = 128

# Allocate usage & timing arrays

n = 300

cache_usages_128 = np.empty(n, dtype=int)
exec_timings_128 = np.empty(n)

# Call random_flux

for i in range(n):
    exec_timings_128[i] = random_flux()
    cache_usages_128[i] = specgrid.cache_usage

# Plot cache usage and execution time

fig, ax = plt.subplots(ncols=2, figsize=[12,6])

ax[0].plot(cache_usages_512, label='512 MB limit')
ax[0].plot(cache_usages_128, label='128 MB limit')
ax[0].set_xlabel('call #')
```

(continues on next page)

(continued from previous page)

```
ax[0].set_ylabel('cache usage (MB)')
ax[0].legend(loc=4)

ax[1].plot(exec_timings_512)
ax[1].plot(exec_timings_128)
ax[1].set_xlabel('call #')
ax[1].set_ylabel('exution time (s)')
```

[10]:
```
Text(0, 0.5, 'exution time (s)')
```



The left-hand panel reveals that, as expected, the growth of the cache usage stops once the limit is reached. Because not all the data can be held in memory, the performance gains from the cache are more modest than in the previous case, with the execution times in the right-hand panel averaging 0.15s.

6.4 Choosing Cache Settings

As the demonstration above makes clear, the choice of the *pymsg.SpecGrid.cache_usage* property can have a significant impact on the performance of MSG. Ideally, this property should be set to exceed the total size of the grid[1]; but if that's not possible due to limited computer memory, try some of the following measures:

- set the *pymsg.SpecGrid.cache_usage* property to at least 4^N times the size of a single node, where N is the number of dimensions in a grid (see the *Tensor Product Interpolation* appendix).
- if undertaking a sequence of interpolations, reorder them so that ones with similar photospheric parameters are grouped together.
- for spectroscopic grids, use the *pymsg.SpecGrid.cache_lam_min* and/or *pymsg.SpecGrid.cache_lam_max* properties, as discussed above in the *Wavelength Subsetting* section.

[1] Note that the in-memory size of a grid is typically larger than its on-disk size, due to compression.

6.5 Technical Details

The basic data unit of MSG's caches is a grid node, representing the spectroscopic or photometric intensity for a single set of photospheric parameters. During an interpolation the nodes required for the calculation are accessed through the cache, which reads them from disk as necessary. After every node access, the current cache usage is compared against the user-specified limit. If it exceeds this limit, then one or more nodes are evicted from the cache using a least-recently used (LRU) algorithm.

This implementation — with eviction taking place **after** cache access — means that in principle the cache limit can be zero. For performance reasons, however, this is not recommended.

CHAPTER

SEVEN

EXCEPTION HANDLING

This chapter discusses how MSG handles exceptions. These can arise in a variety of circumstances:

- Attempts to load grids from files that are missing or corrupt.
- Attempts to interpolate at locations outside the bounds of a grid.
- Attempts to interpolate at locations where grid data are missing (so-called *grid voids*).
- Attempts to interpolate with incomplete specification of photospheric parameters.
- Attempts to interpolate for invalid wavelength and/or angle parameters.

When an exception occurs, how it's signaled depends on the language interface being used.

7.1 Python

Using the Python interface, exceptions are signaled using the language's built-in exception handling capabilities. The list of exceptions that can be thrown by each function is provided in the *Python Interface* chapter.

7.2 Fortran

Using the Fortran interface, exceptions are signaled through the optional procedure argument `stat`. If this argument is present, then on return it is set to one of the status code values defined in the *Fortran parameters* section. The value `STAT_OK` indicates that no problem was encountered; other values signal an exception. If `stat` is not present when an exception occurs, then execution halts with an error message printed to standard output.

7.3 C

Using the C interface, exceptions are signaled through the pointer function argument `stat`. If this pointer is non-null, then on return the pointer target `*stat` is set to one of the status code values defined in the *C enums* section. The value *STAT_OK* indicates that no problem was encountered; other values signal an exception. If `stat` is null when an exception occurs, then execution halts with an error message printed to standard output.

CHAPTER

EIGHT

GRID VOIDS

This chapter explains the causes and workarounds for 'voids' — regions of the photospheric parameter space spanned by a grid where MSG is unable to perform an interpolation. Attempts to interpolate in a void result in a *STAT_UNAVAILABLE_DATA* status code (Fortran and C), or a *LookupError* exception (Python).

8.1 Causes of Voids

MSG's *Tensor Product Interpolation* algorithm divides an N-dimensional grid into subintervals. Each subinterval is an N-dimensional hyperrectangle; in one dimension, this is a line segment, in two a rectangle, and so on. To perform an interpolation in a given subinterval, MSG requires that all 2^N vertices (corners) of the subinterval have a spectral (or photometric) data associated with them. If this isn't the case, the subinterval cannot be used for interpolation, and is considered a void.

8.1.1 The Hydrostatic Limit

Most spectral synthesis codes require that the underlying model atmosphere be in hydrostatic equilibrium — that is, the outward radiative acceleration in the atmosphere,

$$g_{\mathrm{rad}} = \frac{\kappa_F F}{c},$$

be less than the inward gravitational acceleration g. Here, F is the bolometric flux, c is the speed of light, and

$$\kappa_F \equiv \frac{1}{F}\int_0^\infty \kappa F_\lambda \mathrm{d}\lambda$$

is the flux-weighted mean of the opacity κ. Recall that the effective temperature T_{eff} is defined in terms of the bolometric flux by

$$F = \sigma T_{\mathrm{eff}}^4,$$

where σ is the Stefan-Boltzmann constant. The condition for hydrostatic equilibrium can then be expressed as

$$g > g_{\mathrm{rad}} = \frac{\kappa_F \sigma T_{\mathrm{eff}}^4}{c}. \tag{8.1}$$

Violation of this inequality is a principal reason why grids lack data (and therefore have voids) at the high-T_{eff}, low-g region of photospheric parameter space.

8.1.2 Convergence Limits

Although the hydrostatic limit (8.1) places a hard lower bound on the surface gravity at a given effective temperature, in practice the actual lower bound is somewhat larger. This is because spectral synthesis codes tend to have difficulty converging to solutions that are close to the limit.

Convergence can also prove difficult in extreme parts of photospheric parameters space — for instance, at the highest or lowest metallicities. In some cases the resulting voids can be filled in with manual intervention (e.g., by tweaking numerical parameters of the synthesis code); but often this would be too time consuming, and the end-users of the grid (us!) have to cope as best we can.

8.2 Parameter Adjustment

One approach to mitigating the impact of voids is to adjust photospheric parameters until they fall within a defined part of the grid. Typically, one hopes that the adjustment has only a small effect on the resulting interpolated spectrum.

MSG provides routines to help with this adjustment; for spectral grids, these are *specgrid_t%adjust()* (Fortran), *adjust_specgrid_x_vec()* (C), and *pymsg.SpecGrid.adjust_x()* (Python). These all accept a starting set of photospheric parameters and a direction to adjust in. The following Python code, which uses the demo grid, demonstrates how one might handle grid voids by adjusting parameters in the direction of increasing g.

```
# Import standard modules

import os
import numpy as np
import astropy.constants as con
import astropy.units as unt
import matplotlib.pyplot as plt

# Import pymsg

import pymsg

# Load the SpecGrid

MSG_DIR = os.environ['MSG_DIR']
GRID_DIR = os.path.join(MSG_DIR, 'data', 'grids')

specgrid_file_name = os.path.join(GRID_DIR, 'sg-demo.h5')

specgrid = pymsg.SpecGrid(specgrid_file_name)

# Set up atmosphere parametets

x = {'Teff': 15000., 'log(g)': 2.3}

# Attemt an interpolation

lam = np.array([3000., 4000., 5000.])

try:

    flux = specgrid.flux(x, lam)
```

(continues on next page)

(continued from previous page)

```
except LookupError:

    # Adjust parameters

    dx = {'Teff': 0., 'log(g)': 1.0}

    x = specgrid.adjust_x(x, dx)

    # Attempt again

    flux = specgrid.flux(x,lam)
```

CHAPTER

NINE

TROUBLESHOOTING

9.1 Missing `forum.inc`

During compilation, if the error `include file 'forum.inc' not found` arises then it's likely you have an incomplete copy of the source code. Verify that when you *checked out* the code from GitHub, you used the **`--recurse-submodules`** option.

9.2 Other Problems

If you encounter something else that doesn't work as it should, please open an issue on GitHub. In your issue, please specify the following:

- The release of MSG you are using (e.g. 1.1).
- The operating system and architecture you are using (e.g., Mac OS 13.1 on Intel).
- A brief description of the problem.
- A narrative of the steps to reproduce the problem.

CHAPTER

TEN

CONTRIBUTING

10.1 Code

To contribute code to the MSG project, from bug fixes through to major new features, the preferred method is the standard fork—branch—pull-request paradigm described here. If your pull request languishes for more than a couple of weeks without receiving a response, consider opening an issue to give the developers a nudge.

10.2 Documentation

To contribute documentation toward the project, follow the same approach as above for code. (The ReStructured Text source files for the documentation reside in the `docs/source` subdirectory.)

10.3 Data

To contribute data toward the project (in the form of spectral intensity or flux grids), a number of options are available. The *Importing Data* section explains how to import an existing grid; but if this lies beyond your technical expertise, then the MSG team will be happy to assist. Either way, if you want to make your data publicly available and listed on the grid files web page, then please open an issue.

CHAPTER

ELEVEN

ISOCHRONE COLOR-MAGNITUDE DIAGRAM

This *Cookbook* recipe demonstrates how to create a color-magnitude diagram (CMD) for an isochrone, using the MSG Python interface.

11.1 Preparation

Download `isochrone-cmd.tar.bz2` and unpack it in your working directory. This archive contains the following items:

- `MIST.iso` — isochrone data[1]
- `pg-*.h5` — photgrid files for each passband[2]
- `read_mist_models.py` — Python module for reading `MIST.iso`

11.2 Initialization

To start, import modules and set other parameters:

[1]:
```
# Import modules

import numpy as np
import matplotlib.pyplot as plt

import astropy.constants as con

import pymsg
import read_mist_models

# Set plot parameters

%matplotlib inline
plt.rcParams.update({'font.size': 16})
```

[1] Generated using the MIST web interpolator for an age $\log(t/\mathrm{yr}) = 9.6$ (non-rotating, solar-metallicity).
[2] Generated by applying the **`specgrid_to_photgrid`** tool to the demo grid.

11.3 Create PhotGrids

Next, create a pair of *pymsg.PhotGrid* objects for interpolating fluxes in the Johnson B and V bands:

[2]:
```
# Create PhotGrid objects

photgrid_B = pymsg.PhotGrid('pg-Johnson-B.h5')
photgrid_V = pymsg.PhotGrid('pg-Johnson-V.h5')
```

11.4 Read Isochrone Data

Now read in the isochrone data file and extract the stellar parameters:

[3]:
```
# Read isochrone data file

iso = read_mist_models.ISO('MIST.iso')

# Extract stellar parameters

Teff = 10**iso.isos[0]['log_Teff']
logg = iso.isos[0]['log_g']

R = 10**iso.isos[0]['log_R']*con.R_sun.value
```

```
Reading in: MIST.iso
```

11.5 Evaluate Fluxes

Using these parameters, evaluate the observed flux at 10 parsecs in each band:

[4]:
```
# Evaluate fluxes

n = len(Teff)

F_B = np.empty(n)
F_V = np.empty(n)

for i in range(n):

    # Set up photospheric parameters dict

    x = {'Teff': Teff[i],
         'log(g)': logg[i]}

    # Evaluate fluxes. Use try/execpt clause to deal with
    # points that fall outside the grid

    try:
        F_B[i] = (R[i]/(10*con.pc.value))**2 * photgrid_B.flux(x)
```

(continues on next page)

(continued from previous page)

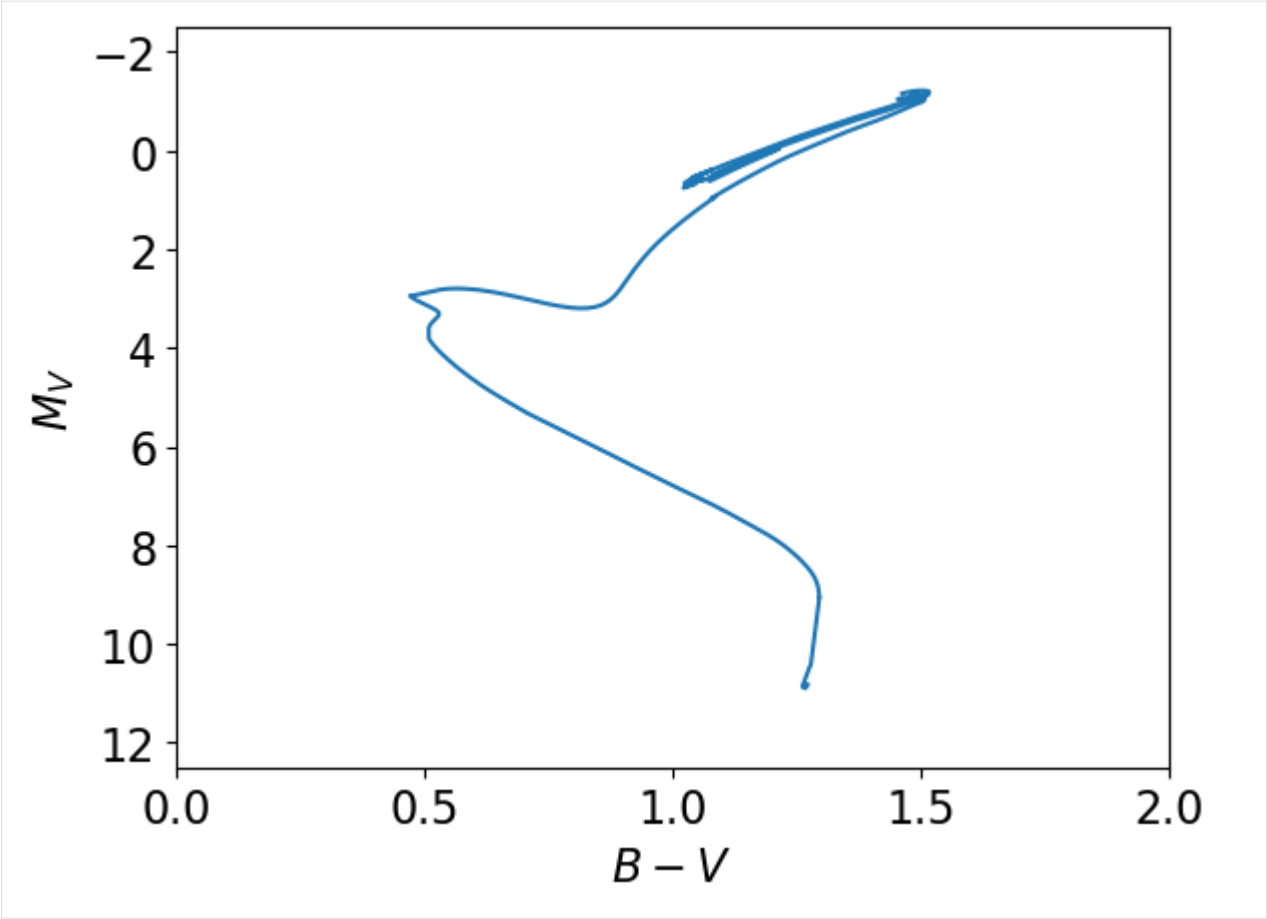
```
        F_V[i] = (R[i]/(10*con.pc.value))**2 * photgrid_V.flux(x)
    except (ValueError, LookupError):
        F_B[i] = np.NAN
        F_V[i] = np.NAN
```

11.6 Plot the CMD

As a final step, convert fluxes to (absolute) magnitudes and plot the CMD:

[5]:
```
plt.figure()

M_B = -2.5*np.log10(F_B)
M_V = -2.5*np.log10(F_V)

plt.plot(M_B-M_V, M_V)

plt.xlim(0, 2)
plt.ylim(12.5,-2.5)

plt.xlabel('$B-V$')
plt.ylabel('$M_V$')
```

[5]:
```
Text(0, 0.5, '$M_V$')
```


−2
0
2
4
6
8
10
12
M_V
0.0
0.5
1.0
1.5
2.0
$B - V$

CHAPTER

TWELVE

LIGHT CURVES FOR AN SPB STAR

This *Cookbook* recipe demonstrates how to synthesize light curves for a slowly pulsating B-type (SPB) star, using the MSG Python interace. The light curves are based on surface perturbation data for a $5\,\mathrm{M}_\odot$ model, produced by the GYRE stellar oscillation code.

12.1 Preparation

Download `spb-light-curve.tar.bz2` and unpack it in your working directory. This archive contains the following items:

- `summary.h5` — surface perturbation data produced by GYRE
- `pygyre` — Python module for reading `summary.h5`
- `pg-*.h5` — photgrid files for each passband[1]
- `gyre.in` — GYRE input file used to create `summary.h5`
- `spb.mesa` — stellar model file used to create `summary.h5`[2]

Note that `gyre.in` and `spb.mesa` aren't needed for the recipe; they're included for those who wish to recreate `summary.h5`, or are curious about its provenance.

12.2 Initialization

To start, import modules and set other parameters:

[1]:
```
# Import modules

import numpy as np
import matplotlib.pyplot as plt
import scipy.special as ss

import astropy.constants as con

import pymsg
import pygyre

# Set plot parameters
```

(continues on next page)

[1] Generated by applying the **specgrid_to_photgrid** tool to an extended-wavelength version of the demo grid.
[2] Generated using the MESA stellar evolution code.

(continued from previous page)

```
%matplotlib inline
plt.rcParams.update({'font.size': 16})
```

12.3 Define a Synthesis Function

Next, define a function that synthesizes a light curve given the following arguments:

- `phi` — oscillation phase array (radians)
- `i` — observer inclination angle (radians)
- `photgrid_name` — name of photgrid file

This function follows the semi-analytical approach described in Section 2 of Townsend (2003); the comments in the function definition highlight the specific equations from that paper.

[2]:
```
def synthesize_light_curve(phi, i, photgrid_name):

    # Create the PhotGrid object

    photgrid = pymsg.PhotGrid(photgrid_name)

    # Read the GYRE output file into a table

    tbl = pygyre.read_output('summary.h5')

    # Select the table row corresponding to the l=m=2, g_30 mode

    mask = (tbl['l'] == 2) & (tbl['m'] == 2) & (tbl['n_pg'] == -30)

    row = tbl[mask][0]

    # Extract stellar parameters

    M_star = row['M_star']
    R_star = row['R_star']
    L_star = row['L_star']

    Teff = (L_star/(4*np.pi*con.sigma_sb.cgs.value*R_star**2))**0.25
    g = con.G.cgs.value*M_star/R_star**2

    # Extract surface perturbation parameters

    l = row['l']
    m = row['m']

    omega = row['omega']

    dR_R = row['xi_r_ref']      # Delta_R (T03, eqn. 4)
    dL_L = row['lag_L_ref']
    dT_T = 0.25*(dL_L - 2*dR_R) # Delta_T (T03, eqn. 5)
```

(continues on next page)

(continued from previous page)

```
    dg_g = -(2 + omega**2)*dR_R # Delta_g (T03, eqn. 6)

    # Evaluate intensity moments & their partial derivatives (T03, eqn. 15)

    x = {'Teff': Teff, 'log(g)': np.log10(g)}

    I_0 = photgrid.D_moment(x, 0)
    I_x = photgrid.D_moment(x, l)

    dI_x_dlnT = photgrid.D_moment(x, l, deriv={'Teff': True}) * Teff
    dI_x_dlng = photgrid.D_moment(x, l, deriv={'log(g)': True}) * np.log(np.exp(1.))

    # Set up differential flux functions (T03, eqns. 12-14)

    R_lm = (2+l)*(1-l) * I_x/I_0 * ss.sph_harm(m, l, 0., i)
    T_lm = dI_x_dlnT/I_0 * ss.sph_harm(m, l, 0., i)
    G_lm = dI_x_dlng/I_0 * ss.sph_harm(m, l, 0., i)

    # Construct the light curve (T03, eqn. 11)

    dF_F = ((dR_R*R_lm + dT_T*T_lm + dg_g*G_lm) * np.exp(1j*phi)).real

    # Return it

    return dF_F
```

12.4 Evaluate Light Curves

Using this function, evaluate light curves in each passband:

[3]:
```
# Set up phase array

phi = np.linspace(0, 4*np.pi, 1001)

# Set observer inclination

i = np.radians(75)

# Evaluate light curves

dF_F_Kepler = synthesize_light_curve(phi, i, 'pg-Kepler.h5')
dF_F_TESS = synthesize_light_curve(phi, i, 'pg-TESS.h5')
dF_F_Gaia_B = synthesize_light_curve(phi, i, 'pg-Gaia-B.h5')
dF_F_Gaia_R = synthesize_light_curve(phi, i, 'pg-Gaia-R.h5')
dF_F_JWST = synthesize_light_curve(phi, i, 'pg-JWST.h5')
```

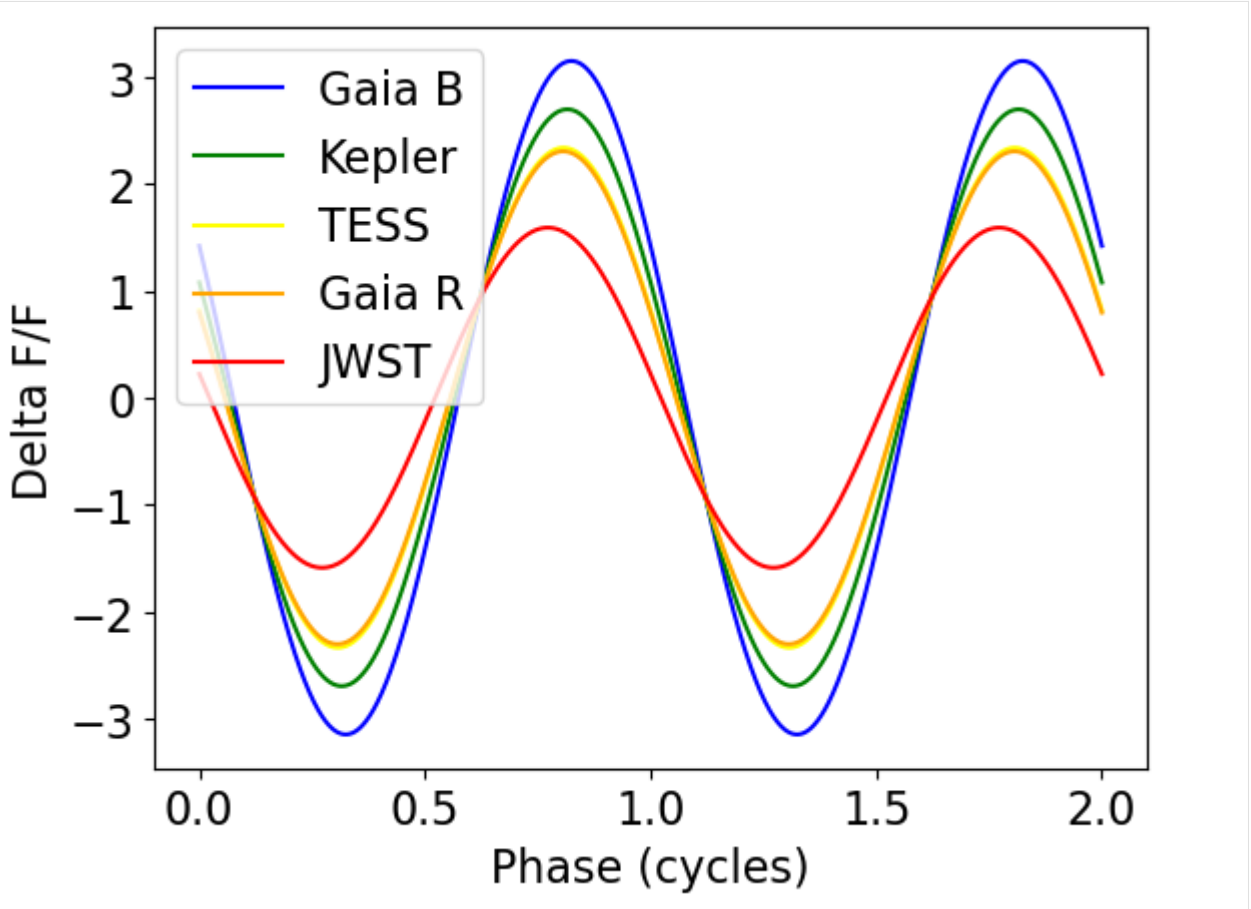
Note that the `pg-JWST.h5` file is based on the JWST NIRCam F150W filter — chosen to demonstrate what the light curve will look like in the near infrared, but with the recognition that it's very unlikely JWST will ever observe an SPB star!

12.5 Plot the Light Curves

As a final step, plot the light curve in each filter:

[4]:
```
# Plot light curves

plt.figure()

plt.plot(phi/(2*np.pi), dF_F_Gaia_B, label='Gaia B', color='blue')
plt.plot(phi/(2*np.pi), dF_F_Kepler, label='Kepler', color='green')
plt.plot(phi/(2*np.pi), dF_F_TESS, label='TESS', color='yellow')
plt.plot(phi/(2*np.pi), dF_F_Gaia_R, label='Gaia R', color='orange')
plt.plot(phi/(2*np.pi), dF_F_JWST, label='JWST', color='red')

plt.xlabel('Phase (cycles)')
plt.ylabel('Delta F/F')
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7f8a86fb86a0>
```



A clear trend can be seen in the light curves: from red wavelengths to blue (i.e., JWST → Gaia R → TESS → Kepler → Gaia B), the amplitude becomes larger and maximum/minimum light occur at later phases. The amplitude trend arises because the visible and NIR fall on the Rayleigh-Jeans tail of the star's spectral energy distribution, and therefore the flux variations due to temperature perturbations vary as λ^{-4}. The phase trend is a consequence of non-adiabatic effects that cause (for this particular mode) the radius perturbations to lag the temperature perturbations by $\sim 45°$.

Note that the overall scaling of the light curves is not physically meaningful; as with any linear oscillation calculation, the normalization of perturbations is arbitrary. In the case of GYRE, this normalization is chosen to yield a mode inertia of MR^2.

CHAPTER

THIRTEEN

LINE PROFILE VARIATIONS OF A BE STAR

This *Cookbook* recipe demonstrates how to simulate the spectral line-profile variations (lpv) of a pulsating Be star, using the MSG Python interface. Synthesizing a spectrum for a pulsating star involves constructing an ensemble of elements representing the stellar surface; subjecting these elements to perturbations from one or more modes; and then integrating the specific intensity over the elements visible to the observer. This process is similar to the disk integration in equation (4.6) of the *MSG Fundamentals* chapter, but a further complication is that one must correctly account for Doppler shifts arising from the motion of the elements along the line-of-sight.

The open-source BRUCE and KYLIE codes (Townsend, 1997) perform these tasks. BRUCE constructs and perturbs surface elements, while KYLIE then performs the specific intensity calculations and visible-disk integration. In this recipe the KYLIE functionality is re-implemented in Python.

13.1 Preparation

Download `be-star-lpv.tar.bz2` and unpack it in your working directory. This archive contains the following items:

- `elements-*` — surface element data produced by BRUCE
- `sg-6678.h5` — specgrid file covering the HeI λ6678 line profile[1]
- `input.bdf` — BRUCE input file used to create `elements-*`

Note that `input.bdf` isn't needed for the recipe; it's included for those who wish to recreate `elements-*`, or are curious about their provenance.

13.2 Initialization

To start, import modules and set other parameters:

[1]:
```
# Import modules

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import scipy.io as si

import astropy.constants as con
```

(continues on next page)

[1] Generated from a solar-metallicity Kurucz atmosphere grid (ap00k2odfnew), with intensity spectra synthesized using SYNSPEC (Hubeny & Lanz, 2017).

(continued from previous page)

```
import pymsg

# Set plot parameters

%matplotlib inline
plt.rcParams.update({'font.size': 16})
```

13.3 Create a SpecGrid

Next, create a *pymsg.SpecGrid* object for interpolating specific intensities:

[2]:
```
# Create SpecGrid object

specgrid = pymsg.SpecGrid('sg-6678.h5')
```

13.4 Define a Synthesis Function

Now define a function that synthesizes a light curve given the following parameters:

- *lam* — wavelength abscissa array (Angstroms)
- *d* — distance to star (meters)
- *bruce_file* — name of BRUCE output file
- *specgrid* — *pymsg.SpecGrid* object

This function is quite simple: it simply adds up the specific intensities from each surface element, appropriately Doppler-shifted and weighted by the elements's projected area and the observer solid angle.

[3]:
```
def synthesize_spectrum(lam, d, bruce_file, specgrid):

    # Initialize the flux array

    n_lam = len(lam)

    F = np.zeros(n_lam-1)

    # Open the BRUCE output file

    with si.FortranFile(bruce_file, 'r') as f:

        # Read header

        record = f.read_record('<i4', '<f4')

        n_vis = record[0][0]
        time = record[1][0]

        # Read elements and accumulate flux
```

(continues on next page)

(continued from previous page)

```
        for i in range(n_vis):

            Teff, V_proj, A_proj, g, mu = f.read_reals('<f4')

            # Evaluate specific intenensity

            x = {'Teff': Teff, 'log(g)': np.log10(g)+2.}

            dI = specgrid.intensity(x, mu, lam*(1+V_proj/con.c.value))

            # Accumulate flux

            F += A_proj * dI * (1./d)**2

    # Return the time and flux

    return time, F
```

13.5 Evaluate the Spectra

Using this function, evaluate a time-series of spectra:

[4]:
```
# Set up parameters

d = 10*con.pc.value

lam = np.linspace(6673, 6682, 101)
lam_mid = 0.5*(lam[:-1] + lam[1:])

n_time = 20

# Evaluate spectra

time = []
F = []

for i in range(n_time):

    bruce_file = f'elements-{i+1:03d}'

    result = synthesize_spectrum(lam, 10*con.pc.value, bruce_file, specgrid)

    time += [result[0]]
    F += [result[1]]
```

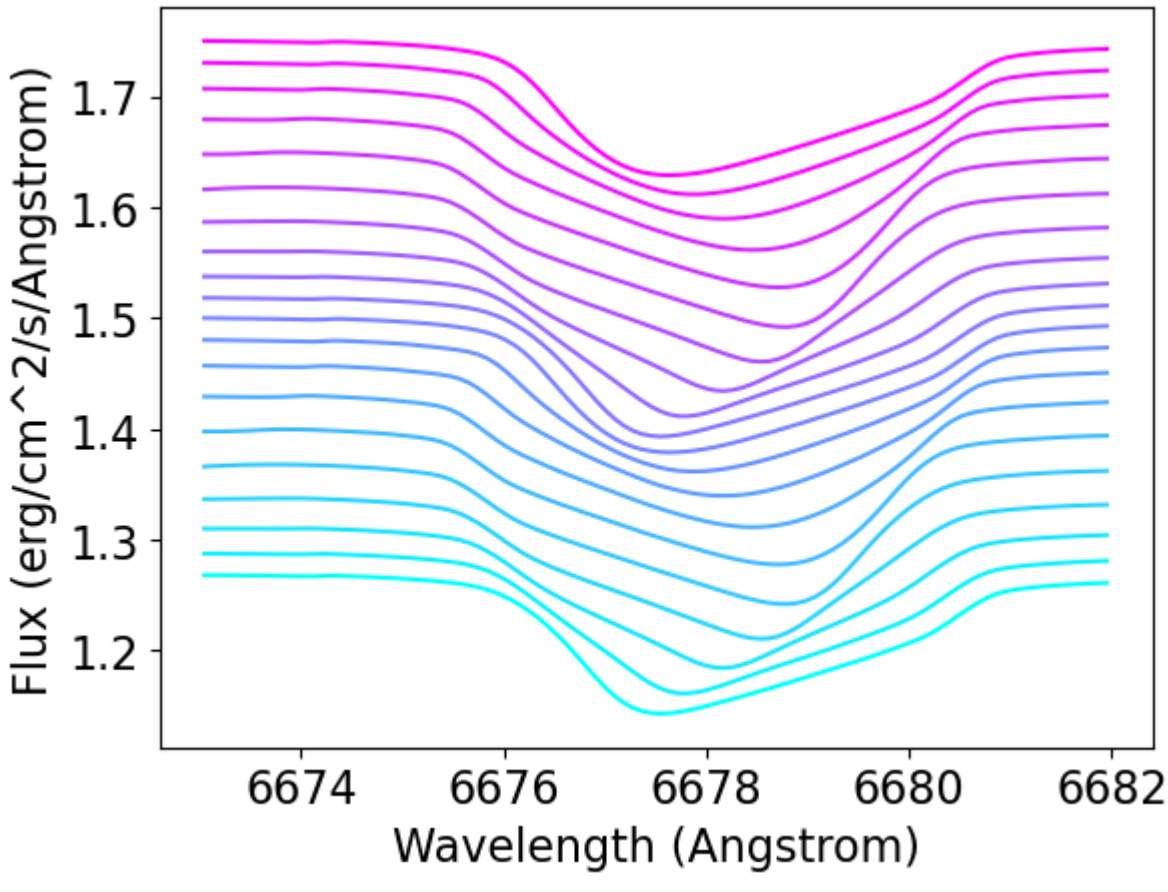
13.6 Plot the Spectra

As a final step, plot the spectra (with a vertical offset between times, to allow comparison):

[5]:
```
# Plot spectra

plt.figure()

cmap = mpl.colormaps['cool']

fig = plt.figure()

offset=0.2*(np.max(F[0]) - np.min(F[0]))

for i in range(n_time):

    plt.plot(lam_mid, 1E8*(F[i]+i*offset), color=cmap(i/(n_time-1)))

plt.xlabel('Wavelength (Angstrom)')
plt.ylabel('Flux (erg/cm^2/s/Angstrom)')
```

[5]:
```
Text(0, 0.5, 'Flux (erg/cm^2/s/Angstrom)')
```

```
<Figure size 640x480 with 0 Axes>
```



The lpv in the figure take the form of dips of enhanced absorption that travel across the line profile from blue to red.

This is similar to the variations observed in pulsating Be stars (see, e.g., Štefl et al. 2003).

CHAPTER

FOURTEEN

INSTALLATION

This chapter discusses MSG installation in detail. If you just want to get up and running, have a look at the *Quick Start* chapter.

14.1 Pre-Requisites

To compile and use MSG, you'll need the following software components:

- A modern (2008+) Fortran compiler
- The LAPACK linear algebra library
- The LAPACK95 Fortran 95 interface to LAPACK
- The fypp Fortran pre-processor
- The HDF5 data management library

On Linux and MacOS platforms, these components are bundled together in the MESA Software Development Kit (SDK). Using this SDK is *strongly* recommended.

If you're planning on using the *pymsg* Python module, then you'll also need the following components:

- Python 3.7 (or more recent)
- NumPy 1.15 (or more recent)

14.2 Building MSG

Download the MSG source code, and unpack it from the command line using the **tar** utility:

```
$ tar xf msg-dev.tar.gz
```

Set the `MSG_DIR` environment variable with the path to the newly created source directory; this can be achieved, e.g., using the **realpath** command[1]:

```
$ export MSG_DIR=$(realpath msg-dev)
```

Finally, compile MSG using the **make** utility:

```
$ make -j -C $MSG_DIR
```

(the **-j** flags tells **make** to use multiple cores, speeding up the build). If things go awry, consult the *Troubleshooting* chapter.

[1] The **realpath** command is included in the GNU CoreUtils package. Mac OS users can install CoreUtils using MacPorts or Homebrew.

14.3 Testing MSG

To test MSG, use the command

```
$ make -C $MSG_DIR test
```

This runs unit tests for the various Fortran modules that together compose the MSG library. At the end of the test sequence, a summary of the number of tests passed and failed is printed. All tests should pass; if one or more fails, then please open an issue to report the problem.

14.4 Installing the `pymsg` Module

To install the *pymsg* Python module, use the **pip** tool:

```
$ pip install $MSG_DIR/python
```

You can alternatively add the `$MSG_DIR/python/src` directory to the `PYTHONPATH` environment variable. Note that in order for *pymsg* to function correctly, the `MSG_DIR` environment variable must be set at Python runtime (this variable allows the module to find the Python extension that interfaces to the back-end).

14.5 Custom Builds

Custom builds of MSG can be created by setting certain environment variables, and/or variables in the file *$MSG_DIR/*`src/build/Makefile`, to the value `yes`. The following variables are currently supported:

DEBUG
: Enable debugging mode (default `no`)

FPE
: Enable floating point exception checks (default `yes`)

OMP
: Enable OpenMP parallelization (default `yes`)

PYTHON
: Enable building of the Python extension (default `yes`)

TEST
: Enable building of testing tools (default `yes`)

TOOLS
: Enable building of development tools (default `yes`)

If a variable is not set, then its default value is assumed.

14.6 GitHub Access

Sometimes, you'll want to try out new features in MSG that haven't yet made it into a formal release. In such cases, you can check out MSG directly from the https://github.com/rhdtownsend/msg git repository on GitHub:

```
$ git clone --recurse-submodules https://github.com/rhdtownsend/msg.git
```

However, a word of caution: MSG is under constant development, and features in the `main` branch can change without warning.

footnote

CHAPTER

FIFTEEN

PYTHON INTERFACE

The Python interface is provided through the *pymsg* module.

15.1 API Specification

15.1.1 Classes

class pymsg.**SpecGrid**(*file_name*)

The SpecGrid class represents a grid of spectroscopic intensity data.

This grid may be used to interpolate the intensity (or related quantities) across a wavelength abscissa and for a set of photospheric parameter values.

__init__(*file_name*)

SpecGrid constructor (via loading data from a *specgrid* file).

Parameters
file_name (*string*) – Name of the file

Returns
Constructed object.

Return type
pymsg.SpecGrid

Raises

- **FileNotFound** – If the the file cannot be found.
- **TypeError** – If the file contains an incorrect datatype.

property rank

Number of dimensions in grid.

Type
int

property axis_labels

Photospheric parameter axis labels.

Type
list

property axis_x_min

Photospheric parameter axis minima.

Type
dict

property axis_x_max

Photospheric parameter axis maxima.

Type
dict

property lam_min

Minimum wavelength of grid (Å).

Type
double

property lam_max

Maximum wavelength of grid (Å).

Type
double

property cache_lam_min

Minimum wavelength of grid cache (Å).

Type
double

property cache_lam_max

Maximum wavelength of grid cache (Å).

Type
double

property cache_usage

Current memory usage of grid cache (MB).

Type
int

property cache_limit

Maximum memory usage of grid cache (MB).

Type
int

flush_cache()

Flush the grid cache

intensity(*x*, *mu*, *lam*, *deriv=None*, *order=3*)

Evaluate the spectroscopic intensity.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.b
- **mu** (*double*) – Cosine of angle of emergence relative to surface normal.
- **lam** (*numpy.ndarray*) – Wavelength abscissa (Å).

- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns

Spectroscopic intensity (erg/cm^2/s/Å/sr) in bins delineated by lam; length len(lam)-1.

Return type

numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x*, *mu*, or any part of the wavelength abscissa falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

E_moment(*x*, *k*, *lam*, *deriv=None*, *order=3*)

Evaluate the spectroscopic intensity E-moment.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **k** (*int*) – Degree of moment.
- **lam** (*numpy.ndarray*) – Wavelength abscissa (Å).
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns

Spectroscopic intensity E-moment (erg/cm^2/s/Å) in bins delineated by lam; length len(lam)-1.

Return type

numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x*, *k*, or any part of the wavelength abscissa falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

D_moment(*x*, *l*, *lam*, *deriv=None*, *order=3*)

Evaluate the spectroscopic intensity D-moment.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **l** (*int*) – Harmonic degree of moment.
- **lam** (*numpy.ndarray*) – Wavelength abscissa (Å).

- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns

Spectroscopic intensity D-moment (erg/cm^2/s/Å) in bins delineated by lam; length len(lam)-1.

Return type

numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x*, *l*, or any part of the wavelength abscissa falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

flux(*x*, *lam*, *deriv=None*, *order=3*)

Evaluate the spectroscopic flux.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **lam** (*numpy.ndarray*) – Wavelength abscissa (Å)
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns

Spectroscopic flux (erg/cm^2/s/Å) in bins delineated by lam; length len(lam)-1.

Return type

numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x* or any part of the wavelength abscissa falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

adjust_x(*x*, *dx*)

Adjust photospheric parameters in a specified direction, until they fall within the valid part of the grid.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **dx** (*dict*) – Photospheric parameter adjustment direction; keys must match *axis_labels* property, values must be double. The overall scaling is unimportant, but at least one value must be non-zero.

Returns
Adjusted photospheric parameters.

Return type
numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If no valid *x* can be found, or if *dx* is invalid.

class pymsg.**PhotGrid**(*file_name*, *passband_file_name=None*)

The PhotGrid class represents a grid of photometric intensity data.

This grid may be used to interpolate the intensity (or related quantities) for a set of photospheric parameter values.

__init__(*file_name*, *passband_file_name=None*)

PhotGrid constructor (via loading data from a *photgrid* file, or
from a *specgrid* file together with a passband file).

Parameters

- **file_name** (*string*) – Name of grid file.
- **passband_file_name** (*string*) – Name of passband file (if file_name corresponds to a specgrid file)

Returns
Constructed object.

Return type
pymsg.PhotGrid

Raises

- **FileNotFound** – If either file cannot be found.
- **TypeError** – If either file contains an incorrect datatype.

property rank

Number of dimensions in grid.

Type
int

property axis_labels

Photospheric parameter axis labels.

Type
list

property axis_x_min

Photospheric parameter axis minima.

Type
dict

property axis_x_max

Photospheric parameter axis maxima.

Type
dict

property cache_usage

Current memory usage of grid cache (MB).

Type
int

property cache_limit

Maximum memory usage of grid cache (MB).

Type
int

flush_cache()

Flush the grid cache

intensity(*x*, *mu*, *deriv=None*, *order=3*)

Evaluate the photometric intensity, normalized to the zero- point flux.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **mu** (*double*) – Cosine of angle of emergence relative to surface normal.
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns
photometric intensity (/sr).

Return type
double

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x* or *mu* falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

E_moment(*x*, *k*, *deriv=None*, *order=3*)

Evaluate the photometric intensity E-moment, normalized to the zero-point flux.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **k** (*int*) – Degree of moment.
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.

- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns
photometric intensity E-moment.

Return type
double

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x* or *k* falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

D_moment(*x*, *l*, *deriv=None*, *order=3*)

Evaluate the photometric intensity D-moment, normalized to the zero-point flux.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **l** (*int*) – Harmonic degree of moment.
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns
photometric intensity D-moment.

Return type
double

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x* or *l* falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

flux(*x*, *deriv=None*, *order=3*)

Evaluate the photometric flux, normalized to the zero-point flux.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **deriv** (*dict, optional*) – Flags indicating whether to evaluate derivative with respect to each photospheric parameter; keys must match the *axis_labels* property, values must be boolean.
- **order** (*int, optional*) – Interpolation order; valid values are 1 or 3.

Returns
photometric flux.

Return type
double

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If *x* or *l* falls outside the bounds of the grid.
- **LookupError** – If *x* falls in a grid void.

adjust_x(*x*, *dx*)

Adjust photospheric parameters in a specified direction, until they fall within the valid part of the grid.

Parameters

- **x** (*dict*) – Photospheric parameters; keys must match *axis_labels* property, values must be double.
- **dx** (*dict*) – Photospheric parameter adjustment direction; keys must match *axis_labels* property, values must be double. The overall scaling is unimportant, but at least one value must be non-zero.

Returns

Adjusted photospheric parameters.

Return type

numpy.ndarray

Raises

- **KeyError** – If *x* does not define all keys appearing in the *axis_labels* property.
- **ValueError** – If no valid *x* can be found, or if *dx* is invalid.

CHAPTER

SIXTEEN

FORTRAN INTERFACE

The Fortran interface is provided through the *fmsg_m* module, which defines derived types together with supporting parameters and procedures.

16.1 API Specification

16.1.1 Derived Types

type specgrid_t

The specgrid_t type represents a grid of spectroscopic data.

This grid may be used to interpolate the intensity (or related quantities) across a wavelength abscissa and for a set of photospheric parameter values.

subroutine get_rank(*rank*)

Get the rank (dimension) of the grid.

Parameters
rank *[integer ,out]* :: Rank.

subroutine get_shape(*shape*)

Get the shape of the grid.

Parameters
shape (*) *[integer ,out]* :: Shape; length *rank*.

subroutine get_axis(*i*, *axis*)

Get an axis of the grid.

Parameters

- **i** *[integer ,in]* :: Index of axis (between 1 and *rank*).
- **axis** *[axis_t ,out]* :: Axis.

subroutine get_lam_min(*lam_min*)

Get the minimum wavelength of the grid.

Parameters
lam_min *[real(RD),out]* :: Minimum wavelength (Å).

subroutine get_lam_max(*lam_max*)

Get the maximum wavelength of the grid.

Parameters
lam_max *[real(RD),out]* :: Maximum wavelength (Å).

`subroutine get_cache_lam_min`(*cache_lam_min*)

Get the minimum wavelength of the grid cache.

Parameters
lam_min *[real(RD),out]* :: Minimum wavelength (Å).

`subroutine get_cache_lam_max`(*cache_lam_max*)

Get the maximum wavelength of the grid cache.

Parameters
cache_lam_max *[real(RD),out]* :: Maximum wavelength (Å).

`subroutine get_cache_limit`(*cache_limit*)

Get the maximum memory usage of the grid cache.

Parameters
cache_limit *[integer ,out]* :: Maximum memory usage (MB).

`subroutine get_cache_usage`(*cache_usage*)

Get the current memory usage of the grid cache.

Parameters
cache_usage *[integer ,out]* :: Current memory usage (MB)

`subroutine set_cache_lam_min`(*cache_lam_min*, *stat*)

Set the minimum wavelength of the grid cache.

Parameters
lam_min *[real(RD),in]* :: Minimum wavelength (Å).

Options
stat *[integer ,out]* :: Status code.

`subroutine set_cache_lam_max`(*cache_lam_max*, *stat*)

Set the maximum wavelength of the grid cache.

Parameters
cache_lam_max *[real(RD),in]* :: Maximum wavelength (Å).

Options
stat *[integer ,out]* :: Status code.

`subroutine set_cache_limit`(*cache_limit*, *stat*)

Set the maximum memory usage of the grid cache.

Parameters
cache_limit *[integer ,in]* :: Maximum memory usage (MB).

Options
stat *[integer ,out]* :: Status code.

`subroutine interp_intensity`(*x_vec*, *mu*, *lam*, *I*, *stat*, *deriv_vec*, *order*)

Interpolate the spectroscopic intensity.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **mu** *[real(RD),in]* :: Cosine of angle of emergence relative to surface normal.

- **lam** (*) *[real(RD),in]* :: Wavelength abscissa (Å).
- **I** (*) *[real(RD),out]* :: Spectroscopic intensity (erg/cm^2/s/Å/sr) in bins delineated by lam; length LEN(lam)-1.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_E_moment`(*x_vec*, *k*, *lam*, *E*, *stat*, *deriv_vec*, *order*)

Interpolate the spectroscopic intensity E-moment.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **k** *[integer ,in]* :: Degree of moment.
- **lam** (*) *[real(RD),in]* :: Wavelength abscissa (Å).
- **E** (*) *[real(RD),out]* :: Spectroscopic intensity E-moment (erg/cm^2/s/Å) in bins delineated by lam; length LEN(lam)-1.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_D_moment`(*x_vec*, *l*, *lam*, *D*, *stat*, *deriv_vec*, *order*)

Interpolate the spectroscopic intensity D-moment.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **l** *[integer ,in]* :: Harmonic degree of moment.
- **lam** (*) *[real(RD),in]* :: Wavelength abscissa (Å).
- **D** (*) *[real(RD),out]* :: Spectroscopic intensity D-moment (erg/cm^2/s/Å) in bins delineated by lam; length LEN(lam)-1.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_flux`(*x_vec*, *lam*, *I*, *stat*, *deriv_vec*, *order*)

Interpolate the spectroscopic flux.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **lam** (*) *[real(RD),in]* :: Wavelength abscissa (Å).
- **F** (*) *[real(RD),out]* :: Spectroscopic flux (erg/cm^2/s/Å) in bins delineated by lam; length LEN(lam)-1.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine adjust_x_vec(x_vec, dx_vec, x_adj, stat)`

Adjust photospheric parameters in a specified direction, until they fall within a valid part of the grid.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **dx_vec** (*) *[real(RD),in]* :: Photospheric parameter adjustment direction. The overall scaling is unimportant, but at least one element must be non-zero.
- **x_adj** (*) *[real(RD),out]* :: Adjusted photospheric parameter values.

Options

stat *[integer ,out]* :: Status code.

`type photgrid_t`

The photgrid_t type represents a grid of photometric data.

This grid may be used to interpolate the intensity (or related quantities) for a set of photospheric parameter values.

`subroutine get_rank(rank)`

Get the rank (dimension) of the grid.

Parameters

rank *[integer ,out]* :: Returned rank.

`subroutine get_shape(shape)`

Get the shape of the grid.

Parameters

shape (*) *[integer ,out]* :: Shape; length *rank*.

`subroutine get_axis(i, axis)`

Get an axis of the grid.

Parameters

- **i** *[integer ,in]* :: Index of axis (between 1 and *rank*)
- **axis** *[axis_t ,out]* :: Returned axis.

`subroutine get_cache_limit(cache_limit)`

Get the maximum memory usage of the cache.

Parameters

cache_limit *[integer ,out]* :: Maximum memory usage (MB).

`subroutine get_cache_usage(cache_usage)`

Get the current memory usage of the cache.

Parameters

cache_usage *[integer ,out]* :: Current memory usage (MB)

`subroutine set_cache_limit`(*cache_limit*, *stat*)

Set the maximum memory usage of the cache.

Parameters

cache_limit *[integer ,in]* :: Maximum memory usage (MB).

Options

stat *[integer ,out]* :: Status code.

`subroutine interp_intensity`(*x_vec*, *mu*, *I*, *stat*, *deriv_vec*, *order*)

Interpolate the photometric intensity, normalized to the zero-point flux.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **mu** *[real(RD),in]* :: Cosine of angle of emergence relative to surface normal.
- **I** *[real(RD),out]* :: Photometric intensity (/sr).

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_E_moment`(*x_vec*, *k*, *E*, *stat*, *deriv_vec*, *order*)

Interpolate the photometric E-moment, normalized to the zero-point flux.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **k** *[integer ,in]* :: Degree of of moment.
- **E** *[real(RD),out]* :: Photometric E-moment.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_D_moment`(*x_vec*, *l*, *D*, *stat*, *deriv_vec*, *order*)

Interpolate the photometric D-moment, normalized to the zero-point flux.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **l** *[integer ,in]* :: Harmonic degree of moment.
- **D** *[real(RD),out]* :: Photometric D-moment.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine interp_flux`(*x_vec*, *F*, *stat*, *deriv_vec*, *order*)

Interpolate the photometric flux, normalized to the zero-point flux.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **F** *[real(RD),out]* :: Photometric flux.

Options

- **stat** *[integer ,out]* :: Status code.
- **deriv_vec** (*) *[logical ,in]* :: Derivative flags.
- **order** *[integer ,in]* :: Interpolation order (1 or 3).

`subroutine adjust_x_vec`(*x_vec*, *dx_vec*, *x_adj*, *stat*)

Adjust photospheric parameters in a specified direction, until they fall within a valid part of the grid.

Parameters

- **x_vec** (*) *[real(RD),in]* :: Photospheric parameter values.
- **dx_vec** (*) *[real(RD),in]* :: Photospheric parameter adjustment direction. The overall scaling is unimportant, but at least one element must be non-zero.
- **x_adj** (*) *[real(RD),out]* :: Adjusted photospheric parameter values.

Options

stat *[integer ,out]* :: Status code.

`type axis_t`

The axis_t type represents a grid axis.

`subroutine get_n`(*n*)

Get the number of points making up the axis.

Parameters

rank *[integer ,out]* :: Number of points.

`subroutine get_x_min`(*x_min*)

Get the minimum value of the axis.

Parameters

x_min *[real(RD),out]* :: Minimum value.

`subroutine get_x_max`(*x_max*)

Get the maximum value of the axis.

Parameters

x_max *[real(RD),out]* :: Maximum value.

`subroutine get_label`(*label*)

Get the axis label.

Parameters

character (*) :: Label.

subroutine fetch(*i*, *x*, *stat*)

Fetch an axis value.

Parameters

- **i** *[integer ,in]* :: Index of value (from 1 to *n*).
- **x** *[real(RD),out]* :: Axis value.

Options

stat *[integer ,out]* :: Status code.

subroutine locate(*x*, *i*)

Locate where along the axis a value falls.

Parameters

- **x** *[real(RD),out]* :: Value to locate.
- **i** *[integer ,out]* :: Location index.

16.1.2 Parameters

int RD

Kind type parameter for reals.

int LABEL_LEN

Length of *axis_t* labels.

int ~fmsg_m/STAT_OK

Status code indicating procedure call completed without error.

int STAT_OUT_OF_BOUNDS_RANGE_LO

Status code indicating procedure call encountered an out-of-bounds reference, below the range minimum.

int STAT_OUT_OF_BOUNDS_RANGE_HI

Status code indicating procedure call encountered an out-of-bounds reference, above the range maximum.

int STAT_OUT_OF_BOUNDS_AXIS_LO

Status code indicating procedure call encountered an out-of-bounds reference, below the axis minimum.

int STAT_OUT_OF_BOUNDS_AXIS_HI

Status code indicating procedure call encountered an out-of-bounds reference, above the axis maximum.

int STAT_OUT_OF_BOUNDS_LAM_LO

Status code indicating procedure call encountered an out-of-bounds reference, below the wavelength minimum.

int STAT_OUT_OF_BOUNDS_LAM_HI

Status code indicating procedure call encountered an out-of-bounds reference, above the wavelength maximum.

int STAT_OUT_OF_BOUNDS_MU_LO

Status code indicating procedure call encountered an out-of-bounds reference, below the emergence cosine minimum.

int STAT_OUT_OF_BOUNDS_MU_HI

Status code indicating procedure call encountered an out-of-bounds reference, above the emergence cosine maximum.

int STAT_UNAVAILABLE_DATA

Status code indicating procedure call encountered unavailable data.

int STAT_INVALID_ARGUMENT

Status code indicating procedure call encountered an invalid argument.

int STAT_FILE_NOT_FOUND

Status code indicating procedure call encountered a file that could not be found.

int STAT_INVALID_FILE_TYPE

Status code indicating procedure call encountered a file with an invalid type.

int STAT_INVALID_GROUP_TYPE

Status code indicating procedure call encountered a file group with an invalid type.

int STAT_INVALID_GROUP_REVISION

Status code indicating procedure call encountered a file group with an invalid revision number.

16.1.3 Procedures

subroutine load_specgrid(*specgrid_file_name*, *specgrid*, *stat*)

Create a new *specgrid_t* by loading data from a specgrid file.

Parameters

- **specgrid_file_name** *[character(*),in]* :: Name of the *specgrid* file.
- **specgrid** *[specgrid_t ,out]* :: Grid object.

Options

stat *[integer ,out]* :: Status code.

subroutine load_photgrid(*photgrid_file_name*, *photgrid*, *stat*)

Create a new *photgrid_t* by loading data from a photgrid file.

Parameters

- **photgrid_file_name** *[character(*),in]* :: Name of the photgrid file.
- **photgrid** *[specgrid_t ,out]* :: Grid object.

Options

stat *[integer ,out]* :: Status code.

subroutine load_photgrid_from_specgrid(*specgrid_file_name*, *passband_file_name*, *photgrid*, *stat*)

Create a new *photgrid_t* by loading data from a specgrid file, and convolving on-the-fly with a response function loaded from a passband file.

Parameters

- **specgrid_file_name** *[character(*),in]* :: Name of the specgrid file.
- **passband_file_name** *[character(*),in]* :: Name of the passband file.
- **photgrid** *[specgrid_t ,out]* :: Grid object.

Options

stat *[integer ,out]* :: Status code.

subroutine get_version(*version*)

Get the version of the MSG library

Parameters

character (*) :: Version string.

16.2 Compiling/Linking

The module file `fmsg_m` for the Fortran interface is provided in the directory `$MSG_DIR/include`, and executables should be linked against `$MSG_DIR/lib/libfmsg.so` (Linux) or `$MSG_DIR/lib/libfmsg.dylib` (MacOS). To simplify this process, a script `$MSG_DIR/scripts/fmsg_link` is provided that writes the appropriate linker commands to standard output. This script can be used to compile/link a program with **gfortran** as follows:

```
$ gfortran -o myprogram myprogram.f90 -I $MSG_DIR/include `$MSG_DIR/scripts/fmsg_link`
```

CHAPTER

SEVENTEEN

C INTERFACE

The C interface is provided through the `cmsg.h` header file, which defines typedefs, enums and functions.

17.1 API Specification

17.1.1 Typedefs

type **SpecGrid**

A *void* * pointer to a Fortran *specgrid_t* spectroscopic grid object.

type **PhotGrid**

A *void* * pointer to a Fortran *specgrid_t* photometric grid object.

17.1.2 Enums

enum **Stat**

enumerator **STAT_OK**

Status code indicating function call completed without error.

enumerator **STAT_OUT_OF_BOUNDS_RANGE_LO**

Status code indicating function call encountered an out-of-bounds reference, below the range minimum.

enumerator **STAT_OUT_OF_BOUNDS_RANGE_HI**

Status code indicating function call encountered an out-of-bounds reference, above the range maximum.

enumerator **STAT_OUT_OF_BOUNDS_AXIS_LO**

Status code indicating function call encountered an out-of-bounds reference, below the axis minimum.

enumerator **STAT_OUT_OF_BOUNDS_AXIS_HI**

Status code indicating function call encountered an out-of-bounds reference, above the axis maximum.

enumerator **STAT_OUT_OF_BOUNDS_LAM_LO**

Status code indicating function call encountered an out-of-bounds reference, below the wavelength minimum.

enumerator **STAT_OUT_OF_BOUNDS_LAM_HI**

Status code indicating function call encountered an out-of-bounds reference, above the wavelength maximum.

enumerator **STAT_OUT_OF_BOUNDS_MU_LO**

Status code indicating function call encountered an out-of-bounds reference, below the emergence cosine minimum.

enumerator **STAT_OUT_OF_BOUNDS_MU_HI**

Status code indicating function call encountered an out-of-bounds reference, above the emergence cosine maximum.

enumerator **STAT_UNAVAILABLE_DATA**

Status code indicating function call encountered unavailable data.

enumerator **STAT_INVALID_ARGUMENT**

Status code indicating function call encountered an invalid argument.

enumerator **STAT_FILE_NOT_FOUND**

Status code indicating function call encountered a file that could not be found.

enumerator **STAT_INVALID_FILE_TYPE**

Status code indicating function call encountered a file with an invalid type.

enumerator **STAT_INVALID_GROUP_TYPE**

Status code indicating function call encountered a file group with an invalid type.

enumerator **STAT_INVALID_GROUP_REVISION**

Status code indicating function call encountered a file group with an invalid revision number.

17.1.3 Functions

SpecGrid

void **load_specgrid**(const char *specgrid_file_name, *SpecGrid* *specgrid, *Stat* *stat)

Create a new *SpecGrid* by loading data from a specgrid file.

Parameters

- **specgrid_file_name** – Name of the specgrid file.
- **specgrid** – Grid object.
- **stat** – Status code (set to NULL if not required).

void **unload_specgrid**(*SpecGrid* specgrid)

Unload a *specgrid* grid, freeing up memory.

Parameters

- **specgrid** – Grid object.

void **get_specgrid_rank**(*SpecGrid* specgrid, int *rank)

Get the rank (dimension) of the grid.

Parameters

- **specgrid** – Grid object.
- **rank** – Rank.

void **get_specgrid_shape**(*SpecGrid* specgrid, int shape[])

Get the shape of the grid.

Parameters

- **specgrid** – Grid object.
- **rank** – Shape.

void **get_specgrid_lam_min**(*SpecGrid* specgrid, double *lam_min)

Get the minimum wavelength of the grid.

Parameters

- **specgrid** – Grid object.
- **lam_min** – Minimum wavelength (Å).

void **get_specgrid_lam_max**(*SpecGrid* specgrid, double *lam_max)

Get the maximum wavelength of the grid.

Parameters

- **specgrid** – Grid object.
- **lam_max** – Maximum wavelength (Å).

void **get_specgrid_cache_lam_min**(*SpecGrid* specgrid, double *cache_lam_min)

Get the minimum wavelength of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_lam_min** – Minimum wavelength (Å).

void **get_specgrid_cache_lam_max**(*SpecGrid* specgrid, double *cache_lam_max)

Get the maximum wavelength of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_lam_max** – Maximum wavelength (Å).

void **get_specgrid_cache_limit**(*SpecGrid* specgrid, int *cache_limit)

Get the maximum memory usage of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_limit** – Maximum memory usage (MB).

void **get_specgrid_cache_usage**(*SpecGrid* specgrid, int *cache_usage)

Get the current memory usage of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_usage** – Current memory usage (MB).

void **get_specgrid_axis_x_min**(*SpecGrid* specgrid, int i, double *x_min)

Get the minimum value of the i'th grid axis.

Parameters

- **specgrid** – Grid object.
- **i** – Axis index (beginning at 0).
- **x_min** – Minimum value.

void **get_specgrid_axis_x_max**(*SpecGrid* specgrid, int i, double *x_max)

Get the maximum value of the i'th grid axis.

Parameters

- **specgrid** – Grid object.
- **i** – Axis index (beginning at 0).
- **x_max** – Maximum value.

void **get_specgrid_axis_label**(*SpecGrid* specgrid, int i, char *label)

Get the label of the i'th grid axis.

Parameters

- **photgrid** – Grid object.
- **i** – Index of the label (beginning at 0).
- **axis_label** – Buffer to store axis label (at least 17 bytes, to accomodate label plus null terminator).

void **set_specgrid_cache_lam_min**(*SpecGrid* specgrid, double cache_lam_min, *Stat* *stat)

Set the minimum wavelength of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_lam_min** – Minimum wavelength (Å).
- **stat** – Status code (set to NULL if not required).

void **set_specgrid_cache_lam_max**(*SpecGrid* specgrid, double cache_lam_max, *Stat* *stat)

Set the maximum wavelength of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_lam_max** – Maximum wavelength (Å).
- **stat** – Status code (set to NULL if not required).

void **set_specgrid_cache_limit**(*SpecGrid* specgrid, int cache_limit, *Stat* *stat)

Set the maximum memory usage of the grid cache.

Parameters

- **specgrid** – Grid object.
- **cache_limit** – Maximum memory usage (MB).
- **stat** – Status code (set to NULL if not required).

void **interp_specgrid_intensity**(*SpecGrid* specgrid, double x_vec[], double mu, int n, double lam[], double I[], *Stat* *stat, bool deriv_vec[], int *order)

Interpolate the spectroscopic intensity.

Parameters

- **specgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **mu** – Cosine of angle of emergence relative to surface normal.
- **n** – Number of points in wavelength abscissa.
- **lam[n]** – Wavelength abscissa (Å).
- **I[n-1]** – Spectroscopic intensity (erg/cm^2/s/Å/sr) in bins delineated by lam
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_specgrid_E_moment**(*SpecGrid* specgrid, double x_vec[], int k, int n, double lam[], double E[], *Stat* *stat, bool deriv_vec[], int *order)

Interpolate the spectroscopic intensity E-moment.

Parameters

- **specgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **k** – Degree of moment.
- **n** – Number of points in wavelength abscissa.
- **lam[n]** – Wavelength abscissa (Å).
- **D[n-1]** – Spectroscopic intensity E-moment (erg/cm^2/s/Å) in bins delineated by lam
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_specgrid_D_moment**(*SpecGrid* specgrid, double x_vec[], int l, int n, double lam[], double D[], *Stat* *stat, bool deriv_vec[], int *order)

Interpolate the spectroscopic intensity D-moment.

Parameters

- **specgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **l** – Harmonic degree of moment.
- **n** – Number of points in wavelength abscissa.
- **lam[n]** – Wavelength abscissa (Å).
- **D[n-1]** – Spectroscopic intensity D-moment (erg/cm^2/s/Å) in bins delineated by lam
- **stat** – Status code (set to NULL if not required).

- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_specgrid_flux**(*SpecGrid* specgrid, double x_vec[], int n, double lam[], double F[], *Stat* *stat, bool deriv_vec[], int *order)

Interpolate the spectroscopic flux.

Parameters

- **specgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **n** – Number of points in wavelength abscissa.
- **lam[n]** – Wavelength abscissa (Å).
- **F[n-1]** – Spectroscopic flux (erg/cm^2/s/Å) in bins delineated by lam
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **adjust_specgrid_x_vec**(*SpecGrid* specgrid, double x_vec[], double dx_vec[], double x_adj[], *Stat* *stat)

Adjust photospheric parameters in a specified direction, until they fall within a valid part of the grid.

Parameters

- **specgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **dx_vec** – Photospheric parameter adjustment direction. The overall scaling is unimportant, but at least one element must be non-zero.
- **x_adj** – Adjusted photospheric parameter values.
- **stat** – Status code (set to NULL if not required).

PhotGrid

void **load_photgrid**(const char *photgrid_file_name, *PhotGrid* *photgrid, int *stat)

Create a new *PhotGrid* by loading data from a photgrid file.

Parameters

- **photgrid_file_name** – Name of the photgrid file.
- **photgrid** – Grid object.
- **stat** – Status code (set to NULL if not required).

void **load_photgrid_from_specgrid**(const char *specgrid_file_name, const char *passband_file_name, *PhotGrid* *photgrid, int *stat)

Create a new *PhotGrid* by loading data from a specgrid file, and convolving on-the-fly with a response function loaded from a passband file.

Parameters

- **specgrid_file_name** – Name of the specgrid file.
- **passband_file_name** – Name of the passband file.

- **photgrid** – Grid object.
- **stat** – Status code (set to NULL if not required).

void **unload_photgrid**(*PhotGrid* photgrid)

Unload a photometric grid, freeing up memory.

Parameters

- **photgrid** – Grid object.

void **get_photgrid_rank**(Photgrid photgrid, int *rank)

Get the rank (dimension) of the grid.

Parameters

- **photgrid** – Grid object.
- **rank** – Rank.

void **get_photgrid_shape**(Photgrid photgrid, int shaope[])

Get the shape of the grid.

Parameters

- **photgrid** – Grid object.
- **rank** – Shape.

void **get_photgrid_cache_usage**(Photgrid photgrid, int *cache_usage)

Get the current memory usage of the grid cache.

Parameters

- **photgrid** – Grid object.
- **cache_usage** – Current memory usage (MB).

void **get_photgrid_cache_limit**(Photgrid photgrid, int *cache_limit)

Get the maximum memory usage of the grid cache.

Parameters

- **photgrid** – Grid object.
- **cache_limit** – Maximum memory usage (MB).

void **get_photgrid_axis_x_min**(Photgrid photgrid, int i, double *x_min)

Get the minimum value of the i'th grid axis.

Parameters

- **photgrid** – Grid object.
- **i** – Axis index (beginning at 0).
- **x_min** – Minimum value.

void **get_photgrid_axis_x_max**(Photgrid photgrid, int i, double *x_max)

Get the maximum value of the i'th grid axis.

Parameters

- **photgrid** – Grid object.
- **i** – Axis index (beginning at 0).

- **x_max** – Maximum value.

void **get_photgrid_axis_label**(*SpecGrid* specgrid, int i, char *label)

Get the label of the i'th grid axis.

Parameters

- **photgrid** – Grid object.
- **i** – Index of the label (beginning at 0).
- **axis_label** – Buffer to store axis label (at least 17 bytes, to accomodate label plus null terminator).

void **set_photgrid_cache_limit**(Photgrid photgrid, int cache_limit, int *stat)

Set the maximum memory usage of the grid cache.

Parameters

- **photgrid** – Grid object.
- **cache_limit** – Maximum memory usage (MB).
- **stat** – Status code (set to NULL if not required).

void **interp_photgrid_intensity**(*PhotGrid* photgrid, double x_vec[], double mu, double *I, int *stat, bool deriv_vec[], int *order)

Interpolate the photometric intensity, normalized to the zero-point flux.

Parameters

- **photgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **mu** – Cosine of angle of emergence relative to surface normal.
- **I** – Photometric intensity (/sr).
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_photgrid_E_moment**(*PhotGrid* photgrid, double x_vec[], int k, double *E, int *stat, bool deriv_vec[], int *order)

Interpolate the photometric intensity E-moment, normalized to the zero-point flux.

Parameters

- **photgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **k** – Degree of moment.
- **D** – Photometric intensity E-moment.
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_photgrid_D_moment**(*PhotGrid* photgrid, double x_vec[], int l, double *D, int *stat, bool deriv_vec[], int *order)

Interpolate the photometric intensity D-moment, normalized to the zero-point flux.

Parameters

- **photgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **l** – Harmonic degree of moment.
- **D** – Photometric intensity D-moment.
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **interp_photgrid_flux**(*PhotGrid* photgrid, double x_vec[], double *F, int *stat, bool deriv_vec[], int *order)

Interpolate the photometric flux, normalized to the zero-point flux.

Parameters

- **PhotGrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **F** – Photometric flux.
- **stat** – Status code (set to NULL if not required).
- **deriv_vec** – Derivative flags (set to NULL if not required).
- **order** – Interpolation order (1 or 3; set to NULL if not required).

void **adjust_photgrid_x_vec**(*PhotGrid* photgrid, double x_vec[], double dx_vec[], double x_adj[], *Stat* *stat)

Adjust photospheric parameters in a specified direction, until they fall within a valid part of the grid.

Parameters

- **photgrid** – Grid object.
- **x_vec** – Photospheric parameter values.
- **dx_vec** – Photospheric parameter adjustment direction. The overall scaling is unimportant, but at least one element must be non-zero.
- **x_adj** – Adjusted photospheric parameter values.
- **stat** – Status code (set to NULL if not required).

Other

void **get_msg_version**(char *version)

Get the version of the MSG library.

Parameters

- **version** – Buffer to store version (recommended at least 10 bytes).

17.2 Compiling/Linking

Headers for the C interface are provided in the header file `$MSG_DIR/include/cmsg.h`, and executables should be linked against `$MSG_DIR/lib/libcmsg.so` (Linux) or `$MSG_DIR/lib/libcmsg.dylib` (MacOS). To simplify this process, a script `$MSG_DIR/scripts/cmsg_link` is provided that writes the appropriate linker commands to standard output. This script can be used to compile/link a program with **gcc** as follows:

```
$ gcc -o myprogram myprogram.c -I $MSG_DIR/include `$MSG_DIR/scripts/cmsg_link`
```

CHAPTER

EIGHTEEN

DATA SCHEMA

This chapter specifies the data schema used by MSG data files. These files are stored on disk in HDF5 format, and the schema describes what data appear in each type of file.

18.1 Conventions

All MSG data files adhere to the following conventions:

- HDF5 groups are used to store nested data structures that correspond reasonably closely to the derived data types (e.g., *specgrid_t*) of the Fortran interface.
- Real values are written with HDF5 type *H5T_IEEE_F64LE*, or *H5T_IEEE_F32LE* when reduced precision is permitted.
- Integer values are written with HDF5 type *H5T_STD_I32LE*.
- Logical (boolean) values are written with HDF5 type *H5T_STD_I32LE*, with 1 corresponding to true, and 0 corresponding to false.
- Character values are written with HDF5 type *H5T_NATIVE_CHARACTER*.

18.2 Files

passband

Item	Object Type	Data Type	Definition
/	group	passband	passband.

photgrid

Item	Object Type	Data Type	Definition
/	group	photgrid	photgrid.

photint

Item	Object Type	Data Type	Definition
/	group	limb_photint	specint
/labels/<label>	attribute	real	label value.

specgrid

Item	Object Type	Data Type	Definition
/	group	specgrid	specgrid.

specint

Item	Object Type	Data Type	Definition
/	group	limb_specint	specint
/labels/<label>	attribute	real	label value.

18.3 Groups

axis

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'axis_t'.
REVISION	attribute	integer	literal 1.
label	attribute	character	label.
x	dataset	real(:)	abscissa values.

comp_range

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'comp_range_t'.
REVISION	attribute	integer	literal 1.
n_ranges	dataset	integer	number of sub-ranges.
ranges[i]	group	lin_range | log_range | tab_range | comp_range	sub-ranges (i = 1, ..., n_ranges).

cubint

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'cubint_t'.
REVISION	attribute	integer	literal 1.
n	attribute	integer	number of points.
x	dataset	real(n)	abscissa values.
f	dataset	real(n)	ordinate values.
df_dx	dataset	real(n)	derivative values.

limb

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'limb_t'.
REVISION	attribute	integer	literal 1.
law	attribute	character	limb-darkening law (see *here* for a list of options).

limb_photint

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'limb_photint_t'.
REVISION	attribute	integer	literal 1.
precise	attribute	logical	precision of data (.*TRUE.* for 64-bit, .*FALSE.* for 32-bit).
c	dataset	real(:)	intensity coefficients (/sr).
limb	group	limb	limb-darkening law.

limb_specint

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'limb_specint_t'.
REVISION	attribute	integer	literal 1.
lam_min	attribute	real	minimum wavelength (Å).
lam_max	attribute	real	maximum wavelength (Å).
precise	attribute	logical	storage precision of c (true for *H5T_IEEE_F64LE*, false for *H5T_IEEE_F32LE*).
c	dataset	real(:,:)	intensity coefficients (erg/cm^2/s/Å/sr).
range	group	lin_range | log_range | tab_range | comp_range	wavelength grid (Å).
limb	group	limb	limb-darkening law.

lin_range

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'lin_range_t'.
REVISION	attribute	integer	literal 1.
x_0	attribute	real	abscissa start value.
dx	attribute	real	abscissa spacing.
n	attribute	integer	number of points.

log_range

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'log_range_t'.
REVISION	attribute	integer	literal 1.
logx_0	attribute	real	logarithm of abscissa start value.
dlogx	attribute	real	abscissa logarithmic spacing.
n	attribute	integer	number of points.

passband

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'passband_t'.
REVISION	attribute	integer	literal 1.
F_0	attribute	real	normalizing flux (erg/cm^2/s).
cubint	group	cubint	filter interpolant.

photgrid

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'photgrid_t'.
REVISION	attribute	integer	literal 1.
photsource/n	attribute	integer	number of photints.
photsource/photints[i]	group	limb_photint	grid photints (i = 1, ..., n).
vgrid	group	vgrid	virtual grid.

specgrid

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'specgrid_t'.
REVISION	attribute	integer	literal 1.
specsource/n	attribute	integer	number of specints.
specsource/lam_min	attribute	real	minimum wavelength (Å).
specsource/lam_max	attribute	real	maximum wavelength (Å).
specsource/specints[i]	group	limb_specint	grid specints (i = 1, ..., n).
vgrid	group	vgrid	virtual grid.

tab_range

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'tab_range_t'.
REVISION	attribute	integer	literal 1.
x	dataset	real(:)	abscissa values.

vgrid

Item	Object Type	Data Type	Definition
TYPE	attribute	character	literal 'vgrid_t'.
REVISION	attribute	integer	literal 2.
shape	attribute	integer(rank)	shape of grid.
rank	attribute	integer	rank of grid.
axes[i]	group	axis	grid axes (i = 1, ..., rank).

CHAPTER

NINETEEN

GRID TOOLS

This appendix outlines the set of tools provided with MSG to assist in creating and managing custom grids. These tools are built by default during compilation, and can be found in the `$MSG_DIR/bin` directory.

19.1 Converting Spectra

The following tools convert existing spectra and/or spectroscopic grids into one or more specint files:

19.1.1 synspec_to_specint

The **synspec_to_specint** tool extracts an single intensity spectrum from a `fort.18` data file produced by the SYNSPEC spectral synthesis package (Lanz & Hubeny, 2003), and writes it to a specint file. It accepts the following command-line arguments:

<synspec_file_name>
: Name of input SYNSPEC file.

<n_mu>
: Number of μ values in input file (as specified by the `nmu` parameter in the `fort.55` SYNSPEC control file).

<mu_0>
: Minimum μ value in input file (as specified by the `ang0` parameter in the `fort.55` SYNSPEC control file).

<lam_min>
: Minimum wavelength of output file (Å).

<lam_max>
: Maximum wavelength in output file (Å).

<R>
: Resolution $\mathcal{R} = \lambda/\Delta\lambda$ in output file.

<law_str>
: Limb-darkening law in output file (see *here* for a list of options).

<specint_file_name>
: Name of output specint file.

<label> `(optional)`
: Label of atmosphere parameter (must be accompanied by a corresponding *<value>* argument).

<value> (optional)

Value of atmosphere parameter (must be accompanied by a corresponding *<label>* argument).

Note that *<label>* and *<value>* parameters must be paired; and that there can be multiple of these pairs. For the law selected by the *<law_str>* option, the tool calculates the limb-darkening coefficients at each wavelength via a least-squares fit to the function

$$y(\mu) = 1 - \frac{I_\lambda(\mu)}{I_\lambda(1)}.$$

19.1.2 ferre_to_specint

The **ferre_to_specint** tool extracts a series of flux spectra from a data file in FERRE format, and writes them to specint files. It accepts the following command-line arguments:

<ferre_file_name>

Name of input FERRE file.

<ferre_file_type>

Type of input file. This determines the mapping between photospheric parameters given in the input file, and photospheric parameters written to the output file. Supported options are `CAP18` (for the Allende Prieto et al., 2018 grids).

<specint_prefix>

Prefix of output specint files; these will have the name `<specint_prefix>-NNNNNNNN.h5`, where `NNNNNNNN` is the zero-padded index of the spectrum (starting at 1).

19.1.3 goettingen_to_specint

The **goettingen_to_specint** tool extracts a flux spectrum from a data file in FITS format (with the schema described by Husser et al., 2013), and writes it to a specint file. This tool accepts the following command-line arguments:

<fits_file_name>

Name of input FITS file.

<wave_type>

Type of wavelength abscissa. This determines the number and distribution of points to assume for the input file. Supported options, corresponding to the different grids described by Husser et al. (2013), are `HiRes` (high-resolution), `MedRes-A1` (medium-resolution, $\Delta\lambda = 1$ Å) and `MedRes-R10000` (medium resolution, $\mathcal{R} = 10\,000$).

<specint_file_name>

Name of output specint file.

Note: In order for **goettingen_to_specint** to build, you must first uncomment/edit the line in `$MSG_DIR/build/Makefile` that defines the *FITS_LDFLAGS* variable. This variable defines the flags used to link against your system's FITS library.

19.1.4 c3k_to_specint

The **c3k_to_specint** tool extracts a series of flux spectra from a data file in C3K format, and writes them to specint files. It accepts the following command-line arguments:

<c3k_file_name>

Name of input C3K file.

<specint_prefix>

Prefix of output specint files; these will have the name <specint_prefix>-NNNNNNNN.h5, where NNNNNNNN is the zero-padded index of the spectrum (starting at 1).

19.1.5 ascii_to_specint

The **ascii_to_specint** tool reads a generic flux spectrum from an ASCII text file, and writes it to a specint file. It accepts the following command-line arguments:

<ascii_file_name>

Name of input ASCII text file (see below).

<lam_units>

Units of wavelength data in input file. Possible choices are **'A'** (Å).

<flux_units>

Units of flux data in input file. Possible choices are **'erg/cm^2/s/A'** ($\mathrm{erg\,cm^{-2}\,s^{-1}\,\AA^{-1}}$).

<specint_file_name>

Name of output specint file.

<label> (optional)

Label of atmosphere parameter (must be accompanied by a corresponding *<value>* argument).

<value> (optional)

Value of atmosphere parameter (must be accompanied by a corresponding *<label>* argument).

Note that *<label>* and *<value>* parameters must be paired; and that there can be multiple of these pairs. The input ASCII text file should contain one or more lines, each consisting of a wavelength value followed by a flux value.

19.1.6 specint_to_specint

The **specint_to_specint** tool subsets and/or rebins data in an existing specint file It accepts the following command-line arguments:

<specint_file_name_in>

Name of input specint file.

<specint_file_name_out>

Name of output specint file.

lam_min=<value> (optional)

Subset to have a minimum wavelength of at least <*value*> (Å).

lam_max=<value> (optional)

Subset to have a maximum wavelength of at most <*value*> (Å).

R=<value> (optional)

Rebin to have a uniform resolution $\mathcal{R}$ of <*value*>.

dlam=<value> (optional)

Rebin to have a uniform wavelength spacing $\Delta\lambda$ of <*value*> (Å).

just=<L|R> (optional)

Justify the new wavelength abscissa to the left (L) or right (R).

If you would like to convert spectra from a format that isn't covered by these tools, please open an issue describing the format and/or pointing to relevant literature.

19.2 Packaging Grids

The following tools package multiple specint files to create specgrid and photgrid files:

19.2.1 specint_to_specgrid

The **specint_to_specgrid** tool packages together multiple specint files to make a spectroscopic grid, writing it to a specgrid. It accepts the following command-line arguments:

<manifest_file_name>

Name of input manifest file (see below).

<specgrid_file_name>

Name of output specgrid file.

The manifest file is a simple text file that lists all the specint files (one per line) that should be included in the grid. The axes and the topology of the grid are automatically determined by the labels attached to each specint file. If two files have identical labels, then the one appearing first in the manifest file is used.

19.2.2 specgrid_to_photgrid

The **specgrid_to_photgrid** tool convolves a spectroscopic grid with a passband to make a photometric grid, writing it to a photgrid file. It accepts the following command-line arguments:

<specgrid_file_name>

Name of input specgrid file.

<passband_file_name>

Name of input passband file.

<photgrid_file_name>

Name of output photgrid file.

Note that it's not always necessary to create photgrid files, as MSG can convolve with passbands on the fly (as discussed in the *Photometric Colors* section).

19.3 Managing Grids

The following tools help with managing grid files:

19.3.1 inspect_grid

The **inspect_grid** tool extracts metadata from a specgrid or photgrid files, and prints it to standard output. It accepts the following command-line arguments:

<grid_file_name>

Name of input grid file.

19.4 Creating Passband Files

The following tools create passband files:

19.4.1 make_passband

The **make_passband** tool reads response data, and writes them to a passband file. It accepts the following command-line arguments:

<input_file_name>

Name of input file (see below).

<F_0>

Normalizing flux F_0 ($\mathrm{erg\,cm^{-2}\,s^{-1}\,\AA^{-1}}$).

<passband_file_name>

Name of output passband file.

The input file is a text file tabulating wavelength λ (Å) and passband response function $S'(\lambda)$ (see the *Photometric Colors* section).

19.4.2 make_photon_passband

The **make_photon_passband** tool creates response data for evaluating photon fluxes, and writes them to a passband file. It accepts the following command-line arguments:

<lam_min>

Minimum wavelength λ_{min} (Å).

<lam_max>

Maximum wavelength λ_{max} (Å).

<passband_file_name>

Name of output passband file.

CHAPTER

TWENTY

TENSOR PRODUCT INTERPOLATION

Tensor product interpolation is a generalization of low-dimension (typically, 2-d or 3-d) multivariate interpolation schemes to an arbitrary number of dimensions. The literature on this topic is rather specialized (although see this Wikipedia section); therefore, this appendix provides a brief introduction. Interpolation in general is covered by many textbooks, so the focus here is reviewing how simple schemes can be generalized.

20.1 Univariate Interpolation

Suppose we are given a set of values $f(x_1), f(x_2), \ldots, f(x_n)$, representing some function $f(x)$ evaluated at a set of n discrete grid points. Then, a piecewise-linear interpolation scheme approximates $f(x)$ in each subinterval $x_i \leq x \leq x_{i+1}$ via

$$\tilde{f}(x) = f_i \, \ell_1(u) + f_{i+1} \, \ell_2(u)$$

Here $f_i \equiv f(x_i)$, while

$$u \equiv \frac{x - x_i}{x_{i+1} - x_i}$$

is a normalized coordinate that expresses where in the subinterval we are; $u = 0$ corresponds to $x = x_i$, and $u = 1$ to $x = x_{i+1}$. The linear basis functions are defined by

$$\ell_1(u) = 1 - u, \qquad \ell_2(u) = u.$$

This interpolation scheme is C^0 continuous, and reproduces $f(x)$ exactly at the grid points.

If we want a smoother representation of the function, we generally need more data. Suppose then that we're also provided the derivatives $\mathrm{d}f/\mathrm{d}x$ at the grid points. Then, a piecewise-cubic interpolation scheme is

$$\tilde{f}(x) = f_i \, c_1(u) + f_{i+1} \, c_2(u) + \partial_x f_i \, c_3(u) + \partial_x f_{i+1} \, c_4(u)$$

where u has the same definition as before, $\partial_x f_i \equiv (\mathrm{d}f/\mathrm{d}x)_{x=x_i}$, and the cubic basis functions are

$$c_1(u) = 2u^3 - 3u^2 + 1, \quad c_2(u) = u^3 - u^2, \quad c_3(u) = -2u^3 + 3u^2, \quad c_4(u) = u^3 - 2u^2 + u$$

(these can be recognized as the basis functions for cubic Hermite splines). This new definition is C^1 continuous, and reproduces $f(x)$ *and* its first derivative exactly at the grid points.

20.2 Bivariate Interpolation

Bivariate interpolation allows us to approximate a function $f(x, y)$ from its values on a rectilinear (but not necessarily equally spaced) grid described by the axis values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$. A piecewise-bilinear interpolating scheme approximates $f(x, y)$ in each subinterval $x_i \le x \le x_{i+1}, y_j \le y \le y_{j+1}$ via

$$\tilde{f}(x, y) = f_{i,j}\,\ell_1(u)\ell_1(v) + f_{i+1,j}\,\ell_2(u)\ell_2(v) + f_{i,j+1}\,\ell_1(u)\ell_1(v) + f_{i+1,j+1}\,\ell_2(u)\ell_2(v)$$

Here, u has the same definition as before, while

$$v \equiv \frac{y - y_j}{y_{j+1} - y_j},$$

and $f_{i,j} \equiv f(x_i, y_j)$. We can also write the scheme in the more-compact form

$$\tilde{f}(x, y) = \sum_{p,q=1}^{2} \mathcal{L}^{p,q}\,\ell_p(u)\ell_q(v),$$

where the coefficients $\mathcal{L}^{p,q}$ can be expressed as the matrix

$$\mathcal{L}^{p,q} \equiv \begin{bmatrix} f_{i,j} & f_{i,j+1} \\ f_{i+1,j} & f_{i+1,j+1} \end{bmatrix}.$$

A corresponding piecewise-bicubic interpolating scheme can be written in the form

$$\tilde{f}(x, y) = \sum_{p,q=1}^{4} \mathcal{C}^{p,q}\,c_p(u)c_q(v),$$

where the coefficients $\mathcal{C}^h$ can be expressed as the matrix

$$\mathcal{C}^{p,q} \equiv \begin{bmatrix} f_{i,j} & f_{i,j+1} & \partial_y f_{i,j} & \partial_y f_{i,j+1} \\ f_{i+1,j} & f_{i+1,j+1} & \partial_y f_{i+1,j} & \partial_y f_{i+1,j+1} \\ \partial_x f_{i,j} & \partial_x f_{i,j+1} & \partial_{xy} f_{i,j} & \partial_{xy} f_{i,j+1} \\ \partial_x f_{i+1,j} & \partial_x f_{i+1,j+1} & \partial_{xy} f_{i+1,j} & \partial_{xy} f_{i+1,j+1} \end{bmatrix}.$$

Constructing this matrix requires 16 values: the function at the four corners of the subinterval, the first derivatives with respect to x and with respect to y at the corners, and the cross derivatives $\partial_{xy} f$ at the corners.

20.3 Multivariate Interpolation

Let's now generalize the compact expressions above for piecewise-bilinear and bicubic interpolation, to piecewise interpolation in N dimensions. For linear interpolation, we have

$$\tilde{f}(x_1, x_2, \ldots, x_N) = \sum_{p_1,p_2,\ldots,p_N=1}^{2} \mathcal{L}^{p_1,p_2,\ldots,p_N} \prod_{k=1}^{N} \ell_{p_k}(u_k)$$

Likewise, for cubic interpolation, we have

$$\tilde{f}(x_1, x_2, \ldots, x_N) = \sum_{p_1,p_2,\ldots,p_N=1}^{4} \mathcal{C}^{p_1,p_2,\ldots,p_N} \prod_{k=1}^{N} c_{p_k}(u_k).$$

The coefficients $\mathcal{L}^{p_1,p_2,\ldots,p_N}$ and $\mathcal{C}^{p_1,p_2,\ldots,p_N}$ cannot easily be expressed in closed form, but they are relatively straightforward to construct algorithmically.

The summations in expressions above can be regarded as the contraction (over all indices) of a pair of rank-N tensors. In the cubic case, the components of the first tensor correspond to the coefficients $\mathcal{C}^{p_1,p_2,\ldots,p_N}$, while the second tensor is formed by taking N outer products between the vectors

$$\mathbf{c}(u_k) = \begin{bmatrix} c_1(u_k) \\ c_2(u_k) \\ c_3(u_k) \\ c_4(u_k) \end{bmatrix} \quad (k = 1, \ldots, N)$$

Hence, this kind of multivariate interpolation is known as tensor product interpolation.

PYTHON MODULE INDEX

FORTRAN MODULE INDEX

INDEX

Symbols

T

U

V