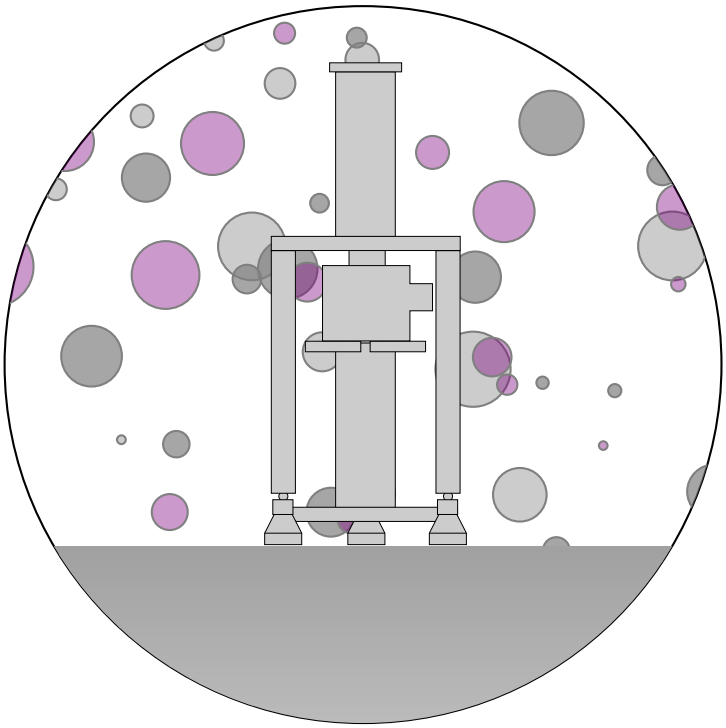


grat v. 1.0 Manual

Marcin Rajner
Politechnika Warszawska
Warsaw University of Technology

Contents

Chapter 1

Grat overview

1.1 Purpose

This program was created to make computation of atmospheric gravity correction more easy.

Version

v. 1.0

Date

2012-12-12

Author

Marcin Rajner
Politechnika Warszawska
(Warsaw University of Technology)

Warning

This program is written in Fortran90 standard but uses some featerus of 2003 specification (e.g., `'newunit='`). It was also written for `Intel Fortran Compiler` hence some commands can be unavailable for yours (e.g., <integer_parameter> for `IO` statements. This should be easily modifiable according to your output needs.> Also you need to have `iso_fortran_env` module available to guess the number of output_unit for your compiler. When you don't want a `log_file` and you don't switch `verbose` all unneceserry information whitch are normally collected goes to `/dev/null` file. This is ∗nix system default trash. For other system or file system organization, please change this value in `get_cmd_line` module.

1.2 section

Todo List

Chapter 2

ffff

Chapter 3

Todo List

Subprogram constants::ispline (u, x, y, b, c, d, n)

give source

Subprogram constants::jd (year, month, day, hh, mm, ss)

mjd!

Subprogram constants::spline (x, y, b, c, d, n)

give source

Subprogram get_cmd_line::is_numeric (string)

Add source name

Subprogram mod_green::convolve_moreverbose (latin, lonin, azimuth, azstep, distance, distancestep)

site height from model

Chapter 4

Data Type Index

4.1 Data Types List

Here are the data types with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Data Type Documentation

6.1 get_cmd_line::additional_info Type Reference

Public Attributes

- character(len=55), dimension(:),
  allocatable **names**

6.1.1 Detailed Description

Definition at line 58 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.2 aggf Module Reference

Public Member Functions

- subroutine compute_aggfdt (psi, aggfdt, delta_, aggf)

  Compute first derivative of AGGF with respect to temperature for specific angular distance (psi)
- subroutine read_tabulated_green (table, author)

  Wczytuje tablice danych AGGF.
- subroutine compute_aggf (psi, aggf_val, hmin, hmax, dh, if_normalization, t_zero, h, first_derivative_h, first_derivative_z, fels_type)

  This subroutine computes the value of atmospheric gravity green functions (AGGF) on the basis of spherical distance (psi)
- subroutine standard_density (height, rho, t_zero, fels_type)

  first derivative (respective to station height) micro Gal height / km
- subroutine standard_pressure (height, pressure, p_zero, t_zero, h_zero, if_simplificated, fels_type, inverted)

  Computes pressure [hPa] for specific height.
- subroutine transfer_pressure (height1, height2, pressure1, pressure2, temperature, polish_meteo)
- subroutine standard_gravity (height, g)

  Compute gravity acceleration of the Earth for the specific height using formula.
- real(sp) function geop2geom (geopotential_height)

  Compute geometric height from geopotential heights.

- subroutine surface_temperature (height, temperature1, temperature2, fels_type, tolerance)

  Iterative computation of surface temp. from given height using bisection method.
- subroutine standard_temperature (height, temperature, t_zero, fels_type)

  Compute standard temperature [K] for specific height [km].
- real function gn_thin_layer (psi)

  Compute AGGF GN for thin layer.
- integer function size_ntimes_denser (size_original, ndenser)

  returns numbers of arguments for n times denser size
- real(dp) function bouger (R_opt)

  Bouger plate computation.
- real(dp) function simple_def (R)

  Bouger plate computation see eq. page 288.

6.2.1 Detailed Description

Definition at line 9 of file aggf.f90.

6.2.2 Member Function/Subroutine Documentation

6.2.2.1 real(dp) function aggf::bouger (real(dp), optional *R_opt*)

Bouger plate computation.

Parameters

r_opt	height of point above the cylinder

Definition at line 479 of file aggf.f90.

6.2.2.2 subroutine aggf::compute_aggf (real(dp), intent(in) *psi,* real(dp), intent(out) *aggf_val,* real(dp), intent(in), optional *hmin,* real(dp), intent(in), optional *hmax,* real(dp), intent(in), optional *dh,* logical, intent(in), optional *if_normalization,* real(dp), intent(in), optional *t_zero,* real(dp), intent(in), optional *h,* logical, intent(in), optional *first_derivative_h,* logical, intent(in), optional *first_derivative_z,* character (len=*), intent(in), optional *fels_type*)

This subroutine computes the value of atmospheric gravity green functions (AGGF) on the basis of spherical distance (psi)

Parameters

in	*psi*	spherical distance from site [degree]
in	*h*	station height [km] (default=0)

Parameters

hmin	minimum height, starting point [km] (default=0)
hmax	maximum height. eding point [km] (default=60)
dh	integration step [km] (default=0.0001 -> 10 cm)
t_zero	temperature at the surface [K] (default=288.15=t0)

Definition at line 110 of file aggf.f90.

6.2.2.3 subroutine aggf::compute_aggfdt (real(dp), intent(in) *psi,* real(dp), intent(out) *aggfdt,* real(dp), intent(in), optional *delta_,* logical, intent(in), optional *aggf*)

Compute first derivative of AGGF with respect to temperature for specific angular distance (psi)

optional argument define (-dt;-dt) range See equation 19 in Huang et al. [2005] Same simple method is applied for aggf(gn) if `aggf` optional parameter is set to .true.

Warning

Please do not use `aggf`=.true. this option was added only for testing some numerical routines

Definition at line 27 of file aggf.f90.

6.2.2.4 real function aggf::gn_thin_layer (real(dp), intent(in) *psi*)

Compute AGGF GN for thin layer.

Simple function added to provide complete module but this should not be used for atmosphere layer See eq p. 491 in Merriam [1992]

Definition at line 455 of file aggf.f90.

6.2.2.5 subroutine aggf::read_tabulated_green (real(dp), dimension(:,:), intent(inout), allocatable *table,* character (len = *), intent(in), optional *author*)

Wczytuje tablice danych AGGF.

- merriam Merriam [1992]
- huang Huang et al. [2005]
- rajner Rajner [2013]

This is just quick solution for `example_aggf` program in `grat` see the more general routine `parse_green()`

Definition at line 66 of file aggf.f90.

6.2.2.6 real(dp) function aggf::simple_def (real(dp) *R*)

Bouger plate computation see eq. page 288.

Warburton and Goodkind [1977]

Definition at line 501 of file aggf.f90.

6.2.2.7 integer function aggf::size_ntimes_denser (integer, intent(in) *size_original,* integer, intent(in) *ndenser*)

returns numbers of arguments for n times denser size

i.e. * * * * –> * . . * . . * . . * (3 times denser)

Definition at line 470 of file aggf.f90.

6.2.2.8 subroutine aggf::standard_density (real(dp), intent(in) *height,* real(dp), intent(out) *rho,* real(dp), intent(in), optional *t_zero,* character(len = 22), optional *fels_type*)

first derivative (respective to station height) micro Gal height / km

direct derivative of equation 20 Huang et al. [2005] first derivative (respective to column height) according to equation 26 in Huang et al. [2005] micro Gal / hPa / km aggf GN micro Gal / hPa if you put the optional parameter `if_normalization=`.false. this block will be skipped by default the normalization is applied according to Merriam [1992] Compute air density for given altitude for standard atmosphere

using formulae 12 in Huang et al. [2005]

Parameters

in	*height*	height [km]
in	*t_zero*	if this parameter is given

Definition at line 194 of file aggf.f90.

6.2.2.9 subroutine aggf::standard_gravity (real(dp), intent(in) *height,* real(dp), intent(out) *g*)

Compute gravity acceleration of the Earth for the specific height using formula.

see Comitee on extension of the Standard Atmosphere [1976]

Definition at line 301 of file aggf.f90.

6.2.2.10 subroutine aggf::standard_pressure (real(dp), intent(in) *height,* real(dp), intent(out) *pressure,* real(dp), intent(in), optional *p_zero,* real(dp), intent(in), optional *t_zero,* real(dp), intent(in), optional *h_zero,* logical, intent(in), optional *if_simplificated,* character(len = 22), optional *fels_type,* logical, intent(in), optional *inverted*)

Computes pressure [hPa] for specific height.

See Comitee on extension of the Standard Atmosphere [1976] or Huang et al. [2005] for details. Uses formulae 5 from Huang et al. [2005]. Simplified method if optional argument if_simplificated = .true.

Definition at line 219 of file aggf.f90.

6.2.2.11 subroutine aggf::standard_temperature (real(dp), intent(in) *height,* real(dp), intent(out) *temperature,* real(dp), intent(in), optional *t_zero,* character (len=∗), intent(in), optional *fels_type*)

Compute standard temperature [K] for specific height [km].

if t_zero is specified use this as surface temperature otherwise use T0. A set of predifined temperature profiles ca be set using optional argument fels_type Fels [1986]

Parameters

in	*fels_type*	• US standard atmosphere (default) • tropical • subtropical_summer • subtropical_winter • subarctic_summer • subarctic_winter

Definition at line 369 of file aggf.f90.

6.2.2.12 subroutine aggf::transfer_pressure (real (dp), intent(in) *height1,* real (dp), intent(in) *height2,* real (dp), intent(in) *pressure1,* real(dp), intent(out) *pressure2,* real (dp), intent(in), optional *temperature,* logical, intent(in), optional *polish_meteo*)

Warning

OBSOLETE ROUTINE – use standard_pressure instead with optional args

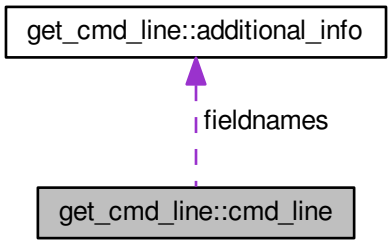
Definition at line 267 of file aggf.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/aggf.f90

6.3 get_cmd_line::cmd_line Type Reference

Collaboration diagram for get_cmd_line::cmd_line:



Public Attributes

- character(2) **switch**
- integer **fields**
- character(len=255), dimension(:), allocatable **field**
- type(additional_info), dimension(:), allocatable **fieldnames**

6.3.1 Detailed Description

Definition at line 61 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.4 constants Module Reference

Public Member Functions

- subroutine spline_interpolation (x, y, x_interpolated, y_interpolated)

  For given vectors x1, y1 and x2, y2 it gives x2interpolated for x1.
- subroutine spline (x, y, b, c, d, n)

  This subroutine was taken from.
- real function ispline (u, x, y, b, c, d, n)

  This subroutine was taken from.
- integer function ntokens (line)

  taken from ArkM http://www.tek-tips.com/viewthread.cfm?qid=1688013
- subroutine skip_header (unit, comment_char_optional)

  This routine skips the lines with comment chars (default '#') from opened files (unit) to read.
- real function jd (year, month, day, hh, mm, ss)

  downloaded from http://aa.usno.navy.mil/faq/docs/jd_formula.php
- real(dp) function **mjd** (date)
- subroutine **invmjd** (mjd, date)

Public Attributes

- integer, parameter dp = 8

  real (kind_real) => real (kind = 8)
- integer, parameter sp = 4

  real (kind_real) => real (kind = 4)
- real(dp), parameter t0 = 288.15

  surface temperature for standard atmosphere [K] (15 degC)
- real(dp), parameter g0 = 9.80665

  mean gravity on the Earth [m/s2]
- real(dp), parameter r0 = 6356.766

  Earth radius (US Std. atm. 1976) [km].
- real(dp), parameter p0 = 1013.25

  surface pressure for standard Earth [hPa]
- real(dp), parameter g = 6.672e-11

  Cavendish constant $[m^3/kg/s^2]$.
- real(dp), parameter r_air = 287.05

  dry air constant [J/kg/K]
- real(dp), parameter pi = 4∗atan(1.)

  pi = 3.141592... [ ]
- real(dp), parameter rho_crust = 2670

  mean density of crust [kg/m3]
- real(dp), parameter rho_earth = 5500

  mean density of Earth [kg/m3]

6.4.1 Detailed Description

Definition at line 5 of file constants.f90.

6.4.2 Member Function/Subroutine Documentation

6.4.2.1 real function constants::ispline (real(dp) *u,* real(dp), dimension(n) *x,* real(dp), dimension(n) *y,* real(dp), dimension(n) *b,* real(dp), dimension(n) *c,* real(dp), dimension(n) *d,* integer *n*)

This subroutine was taken from.

Todo give source

Definition at line 158 of file constants.f90.

6.4.2.2 real function constants::jd (integer, intent(in) *year,* integer, intent(in) *month,* integer, intent(in) *day,* integer, intent(in) *hh,* integer, intent(in) *mm,* integer, intent(in) *ss*)

downloaded from `http://aa.usno.navy.mil/faq/docs/jd_formula.php`

Todo mjd!

Definition at line 253 of file constants.f90.

6.4.2.3 subroutine constants::spline (real(dp), dimension(n) *x,* real(dp), dimension(n) *y,* real(dp), dimension(n) *b,* real(dp), dimension(n) *c,* real(dp), dimension(n) *d,* integer *n*)

This subroutine was taken from.

Todo give source

Definition at line 68 of file constants.f90.

6.4.2.4 subroutine constants::spline_interpolation (real(dp), dimension (:), intent(in), allocatable *x,* real(dp), dimension (:), intent(in), allocatable *y,* real(dp), dimension (:), intent(in), allocatable *x_interpolated,* real(dp), dimension (:), intent(out), allocatable *y_interpolated*)

For given vectors x1, y1 and x2, y2 it gives x2interpolated for x1.

uses `ispline` and `spline` subroutines

Definition at line 28 of file constants.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/constants.f90

6.5 get_cmd_line::dateandmjd Type Reference

Public Attributes

- real(dp) **mjd**
- integer, dimension(6) **date**

6.5.1 Detailed Description

Definition at line 46 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.6 get_cmd_line::file Type Reference

Public Attributes

- character(:), allocatable **name**
- character(len=50), dimension(5) **names** = ["z"
- integer **unit** = output_unit
- logical **if** = .false.
- logical **first_call** = .true.
- real(sp), dimension(4) **limits**
- real(sp), dimension(:), allocatable **lat**
- real(sp), dimension(:), allocatable **lon**
- real(sp), dimension(:), allocatable **time**
- real(sp), dimension(:), allocatable **level**
- integer, dimension(:,:),
  allocatable **date**
- real(sp), dimension(2) **latrange**
- real(sp), dimension(2) **lonrange**
- logical **if_constant_value**
- real(sp) **constant_value**
- real(sp), dimension(:,:,:),
  allocatable data

  4 dimension - lat , lon , level , mjd
- integer **ncid**
- integer **interpolation** = 1

6.6.1 Detailed Description

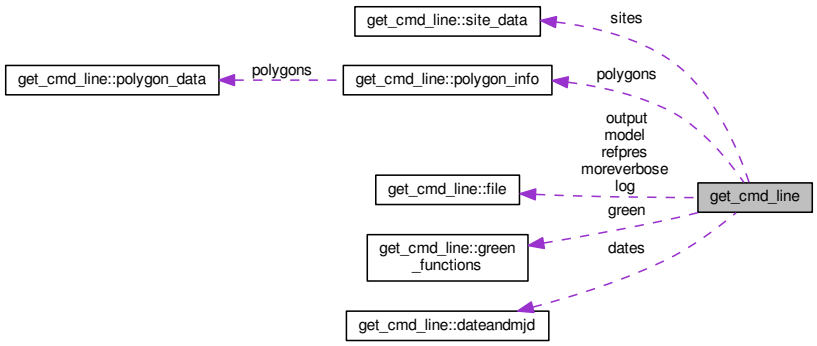
Definition at line 92 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.7 get_cmd_line Module Reference

Collaboration diagram for get_cmd_line:



Data Types

- type additional_info
- type cmd_line
- type dateandmjd
- type file
- type green_functions
- type polygon_data
- type polygon_info
- type site_data

Public Member Functions

- subroutine intro (program_calling)

  This subroutine counts the command line arguments.
- subroutine if_minimum_args (program_calling)

  Check if at least all obligatory command line arguments were given if not print warning.
- logical function if_switch_program (program_calling, switch)

  This function is true if switch is used by calling program or false if it is not.
- subroutine parse_option (cmd_line_entry, program_calling)

  This subroutine counts the command line arguments and parse appropriately.
- subroutine parse_green (cmd_line_entry)

  This subroutine parse -G option i.e. reads Greens function.
- integer function count_separator (dummy, separator)

  change the paths accordingly
- subroutine get_cmd_line_entry (dummy, cmd_line_entry, program_calling)

  This subroutine fills the fields of command line entry for every input arg.
- subroutine **get_model_info** (model, cmd_line_entry, field)
- subroutine parse_gmt_like_boundaries (cmd_line_entry)

  This subroutine checks if given limits for model are proper.
- subroutine read_site_file (file_name)

  Read site list from file.

- subroutine parse_dates (cmd_line_entry)

  Parse date given as 20110503020103 to yy mm dd hh mm ss and mjd.
- subroutine **string2date** (string, date)
- logical function is_numeric (string)

  Auxiliary function.
- logical function file_exists (string)

  Check if file exists , return logical.
- real(dp) function d2r (degree)

  degree -> radian
- real(dp) function r2d (radian)

  radian -> degree
- subroutine print_version (program_calling)

  Print version of program depending on program calling.
- subroutine print_settings (program_calling)

  Print settings.
- subroutine **print_help** (program_calling)
- subroutine **print_warning** (warn, unit)
- integer function nmodels (model)

  Counts number of properly specified models.

Public Attributes

- type(green_functions), dimension(:), allocatable **green**
- integer, dimension(2) **denser** = [1
- type(polygon_info), dimension(2) **polygons**
- real(kind=4) **cpu_start**
- real(kind=4) cpu_finish

  for time execution of program
- type(dateandmjd), dimension(:), allocatable **dates**
- type(site_data), dimension(:), allocatable **sites**
- integer fileunit_tmp

  unit of scratch file
- integer, dimension(8) execution_date

  To give time stamp of execution.
- character(len=2) method = "2D"

  computation method
- character(:), allocatable **filename_site**
- integer **fileunit_site**
- type(file) **log**
- type(file) **output**
- type(file) **moreverbose**
- type(file) **refpres**
- type(file), dimension(:), allocatable **model**
- character(len=40), dimension(5) **model_names** = ["pressure_surface"
- character(len=5), dimension(5) **green_names** = ["GN "
- logical if_verbose = .false.

  whether print all information
- logical **inverted_barometer** = .true.

- character(50), dimension(2) **interpolation_names** = ["nearest"
- character(len=255), parameter **form_header** = '(60("#"))'
- character(len=255), parameter **form_separator** = '(60("-"))'
- character(len=255), parameter **form_inheader** = '(("#"),1x,a56,1x,("#"))'
- character(len=255), parameter **form_60** = "(a,100(1x,g0))"
- character(len=255), parameter **form_61** = "(2x,a,100(1x,g0))"
- character(len=255), parameter **form_62** = "(4x,a,100(1x,g0))"
- character(len=255), parameter **form_63** = "(6x,100(x,g0))"
- character(len=255), parameter **form_64** = "(4x,4x,a,4x,a)"

6.7.1 Detailed Description

Definition at line 8 of file get_cmd_line.f90.

6.7.2 Member Function/Subroutine Documentation

6.7.2.1 integer function get_cmd_line::count_separator (character(∗), intent(in) *dummy,* character(1), intent(in), optional *separator*)

change the paths accordingly

Counts occurence of character (separator, default comma) in string

Definition at line 497 of file get_cmd_line.f90.

6.7.2.2 subroutine get_cmd_line::intro (character(len=∗) *program_calling*)

This subroutine counts the command line arguments.

Depending on command line options set all initial parameters and reports it

Definition at line 169 of file get_cmd_line.f90.

6.7.2.3 logical function get_cmd_line::is_numeric (character(len=∗), intent(in) *string*)

Auxiliary function.

check if argument given as string is valid number Taken from www

Todo Add source name

Definition at line 847 of file get_cmd_line.f90.

6.7.2.4 subroutine get_cmd_line::parse_dates (type(cmd_line) *cmd_line_entry*)

Parse date given as 20110503020103 to yy mm dd hh mm ss and mjd.

Warning

decimal seconds are not allowed

Definition at line 771 of file get_cmd_line.f90.

6.7.2.5 subroutine get_cmd_line::read_site_file (character(len=*), intent(in) *file_name*)

Read site list from file.

checks for arguments and put it into array `sites`

Definition at line 685 of file get_cmd_line.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.8 get_cmd_line::green_functions Type Reference

Public Attributes

- real(dp), dimension(:), allocatable **distance**
- real(dp), dimension(:), allocatable **data**
- logical **if**

6.8.1 Detailed Description

Definition at line 18 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.9 mod_data Module Reference

This modele gives routines to read, and write data.

Public Member Functions

- subroutine put_grd (model, time, level, filename_opt)

  Put netCDF COARDS compliant.
- subroutine read_netcdf (model)

  Read netCDF file into memory.
- subroutine get_variable (model, date)

  Get values from netCDF file for specified variables.
- subroutine nctime2date (model)

  Change time in netcdf to dates.
- subroutine get_dimension (model, i)

  Get dimension, allocate memory and fill with values.
- subroutine unpack_netcdf (model)

  Unpack variable.
- subroutine check (status)

  Check the return code from netCDF manipulation.
- subroutine get_value (model, lat, lon, val, level, method)

  Returns the value from model file.
- real function **bilinear** (x, y, aux)
- subroutine **invspt** (alp, del, b, rlong)

6.9.1 Detailed Description

This modele gives routines to read, and write data.

The netCDF format is widely used in geoscienses. Moreover it is self-describing and machine independent. It also allows for reading and writing small subset of data therefore very efficient for large datafiles (this case) net

Definition at line 10 of file mod_data.f90.

6.9.2 Member Function/Subroutine Documentation

6.9.2.1 subroutine mod_data::check (integer, intent(in) *status*)

Check the return code from netCDF manipulation.

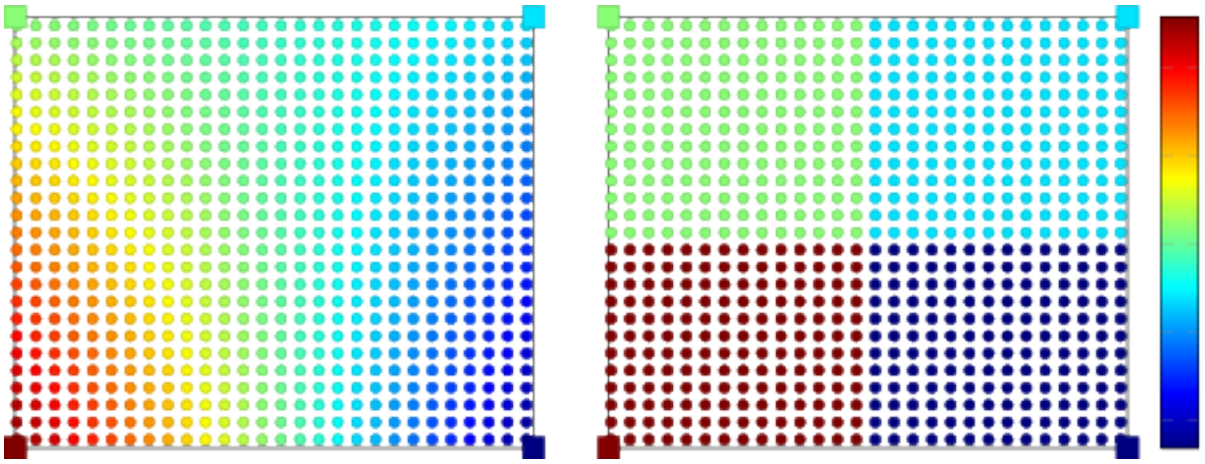
from net

Definition at line 216 of file mod_data.f90.

6.9.2.2 subroutine mod_data::get_value (type(file), intent(in) *model,* real(sp), intent(in) *lat,* real(sp), intent(in) *lon,* real(sp), intent(out) *val,* integer, intent(in), optional *level,* integer, intent(in), optional *method*)

Returns the value from model file.

if it is first call it loads the model into memory inspired by spotl Agnew [1997]

Get value can 

Definition at line 237 of file mod_data.f90.

6.9.2.3 subroutine mod_data::put_grd (type (file) *model,* integer *time,* integer *level,* character (∗), intent(in), optional *filename_opt*)

Put netCDF COARDS compliant.

for GMT drawing

Definition at line 25 of file mod_data.f90.

6.9.2.4 subroutine mod_data::unpack_netcdf (type(file) *model*)

Unpack variable.

from net

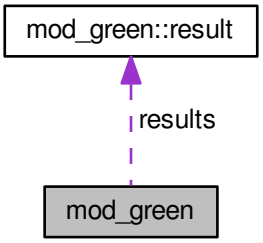
Definition at line 198 of file mod_data.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/mod_data.f90

6.10 mod_green Module Reference

Collaboration diagram for mod_green:



Data Types

- type result

Public Member Functions

- subroutine **green_unification** (green, green_common, denser)
- subroutine **spher_area** (distance, ddistance, azstp, area)
- subroutine **spher_trig** (latin, lonin, distance, azimuth, latout, lonout)
- subroutine **convolve** (site, green, results, denserdist, denseraz)
- subroutine convolve_moreverbose (latin, lonin, azimuth, azstep, distance, distancestep)

Public Attributes

- real(dp), dimension(:,:),
  allocatable **green_common**
- type(result), dimension(:),
  allocatable **results**

6.10.1 Detailed Description

Definition at line 1 of file mod_green.f90.

6.10.2 Member Function/Subroutine Documentation

6.10.2.1 subroutine mod_green::convolve_moreverbose (real(sp), intent(in) *latin,* real(sp), intent(in) *lonin,* real(sp), intent(in) *azimuth,* real(sp), intent(in) *azstep,* real(dp) *distance,* real(dp) *distancestep*)

Todo site height from model

Definition at line 179 of file mod_green.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/mod_green.f90

6.11 mod_polygon Module Reference

Public Member Functions

- subroutine read_polygon (polygon)

  Reads polygon data.
- subroutine chkgon (rlong, rlat, polygon, iok)

  check if point is in closed polygon
- integer function **if_inpoly** (x, y, coords)
- integer function ncross (x1, y1, x2, y2)

  finds whether the segment from point 1 to point 2 crosses the negative x-axis or goes through the origin (this is the signed crossing number)

6.11.1 Detailed Description

Definition at line 1 of file mod_polygon.f90.

6.11.2 Member Function/Subroutine Documentation

6.11.2.1 subroutine mod_polygon::chkgon (real(sp), intent(in) *rlong,* real(sp), intent(in) *rlat,* type(polygon_info), intent(in) *polygon,* integer, intent(out) *iok*)

check if point is in closed polygon

if it is first call it loads the model into memory inspired by spotl Agnew [1997] adopted to grat and Fortran90 syntax From original description

Definition at line 82 of file mod_polygon.f90.

6.11.2.2 integer function mod_polygon::ncross (real(sp), intent(in) *x1,* real(sp), intent(in) *y1,* real(sp), intent(in) *x2,* real(sp), intent(in) *y2*)

finds whether the segment from point 1 to point 2 crosses the negative x-axis or goes through the origin (this is the signed crossing number)

```
return value       nature of crossing
    4               segment goes through the origin
    2               segment crosses from below
    1               segment ends on -x axis from below
                     or starts on it and goes up
    0               no crossing
   -1               segment ends on -x axis from above
                     or starts on it and goes down
   -2               segment crosses from above
```

taken from spotl Agnew [1997] slightly modified

Definition at line 196 of file mod_polygon.f90.

6.11.2.3 subroutine mod_polygon::read_polygon (type(polygon_info) *polygon*)

Reads polygon data.

inspired by spotl Agnew [1997]

Definition at line 12 of file mod_polygon.f90.

The documentation for this module was generated from the following file:

- /home/mrajner/src/grat/src/mod_polygon.f90

6.12 get_cmd_line::polygon_data Type Reference

Public Attributes

- logical **use**
- real(sp), dimension(:,:), allocatable **coords**

6.12.1 Detailed Description

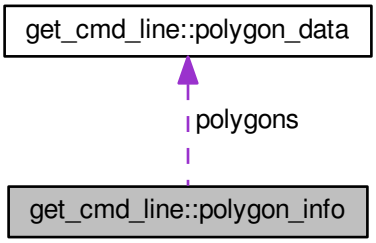
Definition at line 29 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.13 get_cmd_line::polygon_info Type Reference

Collaboration diagram for get_cmd_line::polygon_info:



Public Attributes

- integer **unit**
- character(:), allocatable **name**
- type(polygon_data), dimension(:), allocatable **polygons**
- logical **if**

6.13.1 Detailed Description

Definition at line 34 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

6.14 mod_green::result Type Reference

Public Attributes

- real(sp) **n** = 0.
- real(sp) **dt** = 0.
- real(sp) **e** = 0.
- real(sp) **dh** = 0.
- real(sp) **dz** = 0.

6.14.1 Detailed Description

Definition at line 9 of file mod_green.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/mod_green.f90

6.15 get_cmd_line::site_data Type Reference

Public Attributes

- character(:), allocatable **name**
- real(sp) **lat**
- real(sp) **lon**
- real(sp) **height**

6.15.1 Detailed Description

Definition at line 71 of file get_cmd_line.f90.

The documentation for this type was generated from the following file:

- /home/mrajner/src/grat/src/get_cmd_line.f90

Chapter 7

File Documentation

7.1 /home/mrajner/src/grat/src/aggf.f90 File Reference

This module contains utitlities for computing Atmospheric Gravity Green Functions.

Data Types

- module aggf

7.1.1 Detailed Description

This module contains utitlities for computing Atmospheric Gravity Green Functions. In this module there are several subroutines for computing AGGF and standard atmosphere parameters

Definition in file aggf.f90.

7.2 aggf.f90

```
00001 !
      ================================================================================
00002 !> \file
00003 !! \brief This module contains utitlities for computing
00004 !! Atmospheric Gravity Green Functions
00005 !!
00006 !! In this module there are several subroutines for computing
00007 !! AGGF and standard atmosphere parameters
00008 !
      ================================================================================
00009 module aggf
00010
00011   use constants
00012   implicit none
00013
00014 contains
00015
00016 !
      ================================================================================
00017 !> \brief Compute first derivative of AGGF with respect to temperature
00018 !! for specific angular distance (psi)
00019 !!
00020 !! optional argument define (-dt;-dt) range
00021 !! See equation 19 in \cite Huang05
00022 !! Same simple method is applied for aggf(gn) if \c aggf optional parameter
00023 !! is set to \c .true.
00024 !! \warning Please do not use \c aggf=.true. this option was added only
00025 !! for testing some numerical routines
00026 !
      ================================================================================
00027 subroutine compute_aggfdt ( psi , aggfdt , delta_ , aggf )
00028   implicit none
```

```
00029   real(dp) , intent (in) :: psi
00030   real(dp) , intent (in) , optional :: delta_
00031   logical , intent (in) , optional :: aggf
00032   real(dp) , intent (out) :: aggfdt
00033   real(dp) :: deltat , aux , h_
00034
00035   deltat = 10. !< Default value
00036   if (present( delta_) )  deltat = delta_
00037   if (present( aggf ) .and. aggf ) then
00038     h_ = 0.001 ! default if we compute dggfdh using this routine
00039       if (present( delta_) )  h_ = deltat
00040     call compute_aggf( psi , aux , h = + h_ )
00041     aggfdt = aux
00042     call compute_aggf( psi , aux , h= -h_ )
00043     aggfdt = aggfdt - aux
00044     aggfdt = aggfdt / ( 2. * h_ )
00045   else
00046     call compute_aggf( psi , aux , t_zero = t0 + deltat )
00047     aggfdt = aux
00048     call compute_aggf( psi , aux , t_zero = t0 - deltat )
00049     aggfdt = aggfdt - aux
00050     aggfdt = aggfdt / ( 2. * deltat)
00051   endif
00052
00053
00054
00055 end subroutine
00056
00057 !
      ============================================================================
00058 !> Wczytuje tablice danych AGGF
00059 !! \li merriam \cite Merriam92
00060 !! \li huang   \cite Huang05
00061 !! \li rajner  \cite Rajnerdr
00062 !!
00063 !! This is just quick solution for \c example_aggf program
00064 !! in \c grat see the more general routine \c parse_green()
00065 !
      ============================================================================
00066 subroutine read_tabulated_green ( table , author )
00067   real(dp), intent (inout),dimension(:,:), allocatable :: table
00068   character ( len = * ) , intent (in) , optional       :: author
00069   integer                                              :: i , j
00070   integer                                              :: rows , columns ,
      file_unit
00071   character (len=255)                                  :: file_name
00072
00073   rows      = 85
00074   columns   = 6
00075   file_name = '../dat/merriam_green.dat'
00076
00077   if ( present(author) ) then
00078     if ( author .eq. "huang" ) then
00079       rows    = 80
00080       columns = 5
00081       file_name = '../dat/huang_green.dat'
00082     elseif( author .eq. "rajner" ) then
00083       rows    = 85
00084       columns = 5
00085       file_name = '../dat/rajner_green.dat'
00086     elseif( author .eq. "merriam" ) then
00087     else
00088       write ( * , * ) 'cannot find specified tables, using merriam instead'
00089     endif
00090   endif
00091
00092   if (allocated (table) ) deallocate (table)
00093   allocate ( table( rows , columns ) )
00094
00095   open (newunit = file_unit , file = file_name , action='read', status='old')
00096
00097   call skip_header(file_unit)
00098
00099   do i = 1 , rows
00100     read (file_unit,*) ( table( i , j ), j = 1 , columns )
00101   enddo
00102   close(file_unit)
00103 end subroutine
00104
00105
00106 !
      ============================================================================
00107 !> This subroutine computes the value of atmospheric gravity green functions
00108 !! (AGGF) on the basis of spherical distance (psi)
00109 !
      ============================================================================
00110 subroutine compute_aggf (psi , aggf_val , hmin , hmax , dh ,
```

```
      if_normalization, &
00111                       t_zero , h ,  first_derivative_h , first_derivative_z ,
     fels_type )
00112   implicit none
00113   real(dp), intent(in)          :: psi       !< spherical distance from site
      [degree]
00114   real(dp), intent(in),optional :: hmin ,  & !< minimum height, starting point
     [km]     (default=0)
00115                                 hmax ,  & !< maximum height. eding point    [km]
          (default=60)
00116                                 dh ,    & !< integration step              [km]
          (default=0.0001 -> 10 cm)
00117                                 t_zero, & !< temperature at the surface     [K]
          (default=288.15=t0)
00118                                 h         !< station height                [km]
          (default=0)
00119   logical, intent(in), optional :: if_normalization , first_derivative_h ,
     first_derivative_z
00120   character (len=*) , intent(in), optional  :: fels_type
00121   real(dp), intent(out)         :: aggf_val
00122   real(dp)                      :: r , z , psir , da , dz , rho , h_min , h_max
      , h_station , j_aux
00123
00124   h_min = 0.
00125   h_max = 60.
00126   dz    = 0.0001 !mrajner 2012-11-08 13:49
00127   h_station = 0.
00128
00129   if ( present(hmin) ) h_min    = hmin
00130   if ( present(hmax) ) h_max    = hmax
00131   if ( present(  dh) )    dz    = dh
00132   if ( present(   h) ) h_station = h
00133
00134
00135   psir = psi * pi / 180.
00136
00137   da = 2 * pi * r0**2 * ( 1 - cos(1. *pi/180.) )
00138
00139
00140   aggf_val=0.
00141   do z = h_min , h_max , dz
00142
00143     r = ( ( r0 + z )**2 + (r0 + h_station)**2 &
00144       - 2.*(r0 + h_station ) *(r0+z)*cos(psir) )**(0.5)
00145     call standard_density( z , rho , t_zero = t_zero ,
     fels_type = fels_type )
00146
00147     !> first derivative (respective to station height)
00148     !> micro Gal height / km
00149     if ( present( first_derivative_h) .and. first_derivative_h ) then
00150
00151       !! see equation 22, 23 in \cite Huang05
00152       !J_aux =  (( r0 + z )**2)*(1.-3.*((cos(psir))**2)) -2.*(r0 + h_station
      )**2  &
00153       !  + 4.*(r0+h_station)*(r0+z)*cos(psir)
00154       ! aggf_val =  aggf_val -  rho * (  J_aux  /  r**5  ) * dz
00155
00156       !> direct derivative of equation 20 \cite Huang05
00157       j_aux = (2.* (r0 ) - 2 * (r0 +z )*cos(psir)) / (2. * r)
00158       j_aux =  -r - 3 * j_aux * ((r0+z)*cos(psir) - r0)
00159       aggf_val =  aggf_val +  rho * (  j_aux  /  r**4  ) * dz
00160     else
00161       !> first derivative (respective to column height)
00162       !! according to equation 26 in \cite Huang05
00163       !! micro Gal / hPa / km
00164       if ( present( first_derivative_z) .and. first_derivative_z ) then
00165         if (z.eq.h_min) then
00166             aggf_val = aggf_val  &
00167               + rho*( ((r0 + z)*cos(psir) - ( r0 + h_station ) ) / ( r**3 ) )
00168         endif
00169       else
00170         !> aggf GN
00171         !! micro Gal / hPa
00172         aggf_val = aggf_val  &
00173          + rho * ( ( (r0 + z ) * cos( psir ) - ( r0 + h_station ) ) / ( r**3 )
     ) * dz
00174       endif
00175     endif
00176   enddo
00177
00178   aggf_val = -g * da * aggf_val * 1e8 * 1000
00179
00180   !> if you put the optional parameter \c if_normalization=.false.
00181   !! this block will be skipped
00182   !! by default the normalization is applied according to \cite Merriam92
00183   if ( (.not.present(if_normalization)) .or. (if_normalization)) then
00184     aggf_val= psir * aggf_val * 1e5  / p0
```

```
00185   endif
00186
00187 end subroutine
00188
00189 !
       ==========================================================================
00190 !> Compute air density for given altitude for standard atmosphere
00191 !!
00192 !! using formulae 12 in \cite Huang05
00193 !
       ==========================================================================
00194 subroutine standard_density ( height , rho , t_zero ,fels_type
       )
00195
00196   implicit none
00197   real(dp) , intent(in)  ::  height !< height [km]
00198   real(dp) , intent(in), optional  :: t_zero !< if this parameter is given
00199   character(len = 22) , optional :: fels_type
00200   !! surface temperature is set to this value,
00201   !! otherwise the T0 for standard atmosphere is used
00202   real(dp) , intent(out) :: rho
00203   real(dp)  :: p ,t
00204
00205   call standard_pressure(height , p , t_zero = t_zero,
      fels_type=fels_type)
00206   call standard_temperature(height , t , t_zero = t_zero,
      fels_type=fels_type)
00207
00208   ! pressure in hPa --> Pa
00209   rho= 100 * p / ( r_air * t )
00210 end subroutine
00211
00212 ! ==========================================================================
00213 !> \brief Computes pressure [hPa] for specific height
00214 !!
00215 !! See \cite US1976 or  \cite Huang05 for details.
00216 !! Uses formulae 5 from \cite Huang05.
00217 !! Simplified method if optional argument if_simplificated = .true.
00218 ! ==========================================================================
00219 subroutine standard_pressure (height, pressure , &
00220           p_zero , t_zero , h_zero,  if_simplificated ,fels_type , inverted)
00221   implicit none
00222   real(dp) , intent(in)             :: height
00223   real(dp) , intent(in) , optional :: t_zero , p_zero , h_zero
00224   character(len = 22) , optional :: fels_type
00225   logical          , intent(in) , optional :: if_simplificated
00226   logical          , intent(in) , optional :: inverted
00227   real(dp), intent(out) :: pressure
00228   real(dp) ::  lambda , sfc_height , sfc_temperature , sfc_gravity , alpha ,
      sfc_pressure
00229
00230   sfc_temperature = t0
00231   sfc_pressure = p0
00232   sfc_height = 0.
00233   sfc_gravity = g0
00234
00235   if (present(h_zero)) then
00236     sfc_height = h_zero
00237     call standard_temperature(sfc_height , sfc_temperature
      )
00238     call standard_temperature(sfc_height , sfc_temperature
      )
00239     call standard_gravity(sfc_height , sfc_gravity )
00240   endif
00241
00242   if (present(p_zero)) sfc_pressure = p_zero
00243   if (present(t_zero)) sfc_temperature = t_zero
00244
00245   lambda = r_air * sfc_temperature / sfc_gravity
00246
00247   if (present(if_simplificated) .and. if_simplificated ) then
00248     ! use simplified formulae
00249     alpha = -6.5
00250     pressure = sfc_pressure  &
00251       * ( 1 + alpha / sfc_temperature * (height-sfc_height)) &
00252       ** ( -sfc_gravity / (r_air * alpha / 1000. ) )
00253   else
00254     ! use precise formulae
00255     pressure = sfc_pressure * exp( -1000. * (height -sfc_height) / lambda )
00256   endif
00257   if (present(inverted).and.inverted) then
00258     pressure = sfc_pressure  / ( exp( -1000. * (height-sfc_height) / lambda ) )
00259   endif
00260 end subroutine
00261
00262 ! ==========================================================================
00263 ! > This will transfer pressure beetween different height using barometric
```

```
00264 ! formulae
00265 ! ============================================================================
00266 !> \warning OBSOLETE ROUTINE -- use standard_pressure instead with optional
       args
00267 subroutine transfer_pressure (height1 , height2 , pressure1 ,
       pressure2 , &
00268   temperature , polish_meteo )
00269   real (dp) , intent (in) :: height1 , height2 , pressure1
00270   real (dp) , intent (in), optional :: temperature
00271   real (dp) :: sfc_temp , sfc_pres
00272   logical , intent (in), optional :: polish_meteo
00273   real(dp) , intent(out) :: pressure2
00274
00275   sfc_temp = t0
00276
00277   ! formulae used to reduce press to sfc in polish meteo service
00278   if (present(polish_meteo) .and. polish_meteo) then
00279     sfc_pres = exp(log(pressure1) + 2.30259 * height1*1000. &
00280       /(18400.*(1+0.00366*((temperature-273.15) + 0.0025*height1*1000.)))  )
00281   else
00282   ! different approach
00283     if(present(temperature) ) then
00284       call surface_temperature( height1 , temperature ,
      sfc_temp )
00285     endif
00286     call standard_pressure(height1 , sfc_pres , t_zero=
      sfc_temp , &
00287       inverted=.true. , p_zero = pressure1 )
00288   endif
00289
00290   ! move from sfc to height2
00291   call standard_pressure(height2 , pressure2 , t_zero=sfc_temp
       , &
00292     p_zero = sfc_pres )
00293 end subroutine
00294
00295 ! ============================================================================
00296 !> \brief Compute gravity acceleration of the Earth
00297 !! for the specific height using formula
00298 !!
00299 !! see \cite US1976
00300 ! ============================================================================
00301 subroutine standard_gravity ( height , g )
00302   implicit none
00303   real(dp), intent(in)  :: height
00304   real(dp), intent(out) :: g
00305
00306   g= g0 * ( r0 / ( r0 + height ) )**2
00307 end subroutine
00308
00309
00310 ! ============================================================================
00311 !> \brief Compute geometric height from geopotential heights
00312 ! ============================================================================
00313 real(sp) function geop2geom (geopotential_height)
00314   real (sp) :: geopotential_height
00315
00316   geop2geom = geopotential_height * (r0 / ( r0 + geopotential_height )
       )
00317 end function
00318
00319
00320 ! ============================================================================
00321 !> Iterative computation of surface temp. from given height using bisection
00322 !! method
00323 ! ============================================================================
00324 subroutine surface_temperature (height , temperature1 , &
00325   temperature2, fels_type , tolerance)
00326   real(dp) , intent(in)  :: height , temperature1
00327   real(dp) , intent(out) :: temperature2
00328   real(dp) :: temp(3) , temp_ (3) , tolerance_ = 0.1
00329   character (len=*) , intent(in), optional  :: fels_type
00330   real(sp) , intent(in), optional  :: tolerance
00331   integer :: i
00332
00333   if (present(tolerance)) tolerance_ = tolerance
00334
00335   ! searching limits
00336   temp(1)=t0-150
00337   temp(3)=t0+ 50
00338
00339   do
00340     temp(2)= ( temp(1) + temp(3) ) /2.
00341
00342     do i = 1,3
00343       call standard_temperature(height , temp_(i) , t_zero=
      temp(i) , fels_type = fels_type )
```

```
00344     enddo
00345
00346     if (abs(temperature1 - temp_(2) ) .lt. tolerance_ ) then
00347       temperature2 = temp(2)
00348       return
00349     endif
00350
00351     if ( (temperature1 - temp_(1) ) * (temperature1 - temp_(2) ) .lt.0 ) then
00352       temp(3) = temp(2)
00353     elseif( (temperature1 - temp_(3) ) * (temperature1 - temp_(2) ) .lt.0 )
    then
00354       temp(1) = temp(2)
00355     else
00356       stop "surface_temp"
00357     endif
00358   enddo
00359 end subroutine
00360 ! ===================================================================================
00361 !> \brief Compute standard temperature [K] for specific height [km]
00362 !!
00363 !! if t_zero is specified use this as surface temperature
00364 !! otherwise use T0.
00365 !! A set of predifined temperature profiles ca be set using
00366 !! optional argument \argument fels_type
00367 !! \cite Fels86
00368 !
     ===================================================================================
00369 subroutine standard_temperature ( height , temperature ,
     t_zero , fels_type )
00370   real(dp) , intent(in)  :: height
00371   real(dp) , intent(out) :: temperature
00372   real(dp) , intent(in), optional  :: t_zero
00373   character (len=*) , intent(in), optional  :: fels_type
00374     !< \li US standard atmosphere (default)
00375     !! \li tropical
00376     !! \li subtropical_summer
00377     !! \li subtropical_winter
00378     !! \li subarctic_summer
00379     !! \li subarctic_winter
00380   real(dp) :: aux , cn , t
00381   integer :: i,indeks
00382   real , dimension (10) :: z,c,d
00383
00384   !< Read into memory the parameters of temparature height profiles
00385   !! for standard atmosphere
00386   !! From \cite Fels86
00387   z = (/11.0 , 20.1 , 32.1 , 47.4 , 51.4 , 71.7 , 85.7, 100.0, 200.0, 300.0/)
00388   c = (/-6.5,   0.0,   1.0,   2.75,  0.0,  -2.75, -1.97,  0.0,   0.0,   0.0/)
00389   d = (/ 0.3,   1.0,   1.0,   1.0,   1.0,   1.0,   1.0,   1.0,   1.0,   1.0/)
00390   t = t0
00391
00392   if ( present(fels_type)) then
00393     if (fels_type .eq. "US1976" ) then
00394     elseif(fels_type .eq. "tropical" ) then
00395       z=(/ 2.0  , 3.0, 16.5 , 21.5 , 45.0 , 51.0, 70.0 , 100.0 , 200.0 , 300.0
    /)
00396       c=(/-6.0 , -4.0, -6.7  , 4.0  , 2.2  , 1.0, -2.8  , -0.27  , 0.0   , 0.0
    /)
00397       d=(/ 0.5  , 0.5 , 0.3  , 0.5  , 1.0  , 1.0 , 1.0   , 1.0   , 1.0   , 1.0
    /)
00398       t=300.0
00399     elseif(fels_type .eq. "subtropical_summer" ) then
00400       z = (/ 1.5  , 6.5  , 13.0 , 18.0 , 26.0 , 36.0  , 48.0 ,  50.0 ,  70.0 ,
    100.0  /)
00401       c = (/-4.0 , -6.0  , -6.5  , 0.0  , 1.2  , 2.2  ,  2.5  ,  0.0   ,-3.0
    ,-0.025/)
00402       d = (/ 0.5  , 1.0   , 0.5  , 0.5  , 1.0  , 1.0   , 2.5   , 0.5   , 1.0
    , 1.0  /)
00403       t = 294.0
00404     elseif(fels_type .eq. "subtropical_winter" ) then
00405       z = (/ 3.0  ,10.0  , 19.0 , 25.0 , 32.0 , 44.5  , 50.0 ,  71.0 ,  98.0 ,
    200.0 /)
00406       c = (/-3.5 , -6.0  , -0.5  , 0.0  , 0.4  , 3.2  ,  1.6  , -1.8   , 0.7
    , 0.0 /)
00407       d = (/ 0.5  , 0.5   , 1.0  , 1.0  , 1.0  , 1.0   , 1.0   , 1.0   , 1.0
    , 1.0 /)
00408       t = 272.2
00409     elseif(fels_type .eq. "subarctic_summer" ) then
00410       z = (/  4.7 , 10.0 , 23.0 , 31.8 , 44.0 , 50.2  , 69.2 , 100.0 , 200.0 ,
    300.0 /)
00411       c = (/ -5.3 , -7.0 ,  0.0 ,  1.4 ,  3.0 , 0.7  , -3.3  , -0.2   , 0.0 ,
     0.0 /)
00412       d = (/  0.5 ,  0.3 ,  1.0 ,  1.0 ,  2.0 , 1.0   , 1.5   , 1.0   , 1.0 ,
     1.0 /)
00413       t = 287.0
00414     elseif(fels_type .eq. "subarctic_winter" ) then
00415       z = (/  1.0 ,  3.2 ,  8.5 , 15.5 , 25.0 , 30.0 , 35.0 , 50.0 , 70.0 , 100
```

```
     .0 /)
00416       c = (/  3.0 , -3.2 , -6.8 ,  0.0 , -0.6 ,  1.0 ,  1.2 ,  2.5 , -0.7 ,  -1
     .2 /)
00417       d = (/  0.4 ,  1.5 ,  0.3 ,  0.5 ,  1.0 ,  1.0 ,  1.0 ,  1.0 ,  1.0 ,   1
     .0 /)
00418       t = 257.1
00419     else
00420       print * ,
00421 "unknown fels_type argument: &        using US standard atmosphere 1976
      instead"
00422     endif
00423   endif
00424
00425   if (present(t_zero) ) then
00426     t=t_zero
00427   endif
00428
00429   do i=1,10
00430     if (height.le.z(i)) then
00431       indeks=i
00432       exit
00433     endif
00434   enddo
00435
00436   aux = 0.
00437   do i = 1 , indeks
00438     if (i.eq.indeks) then
00439       cn = 0.
00440     else
00441       cn = c(i+1)
00442     endif
00443       aux = aux + d(i) * ( cn - c(i) )  * log( cosh( (height - z(i)) / d(i) ) /
      cosh(z(i)/d(i)) )
00444   enddo
00445   temperature = t + c(1) * height/2. + aux/2.
00446 end subroutine
00447
00448 !
      ============================================================================
00449 !> \brief Compute AGGF GN for thin layer
00450 !!
00451 !! Simple function added to provide complete module
00452 !! but this should not be used for atmosphere layer
00453 !! See eq p. 491 in \cite Merriam92
00454 !
      ============================================================================
00455 real function gn_thin_layer (psi)
00456   implicit none
00457   real(dp) , intent(in) :: psi
00458   real(dp) :: psir
00459
00460   psir = psi * pi / 180.
00461   gn_thin_layer = 1.627 * psir / sin( psir / 2. )
00462 end function
00463
00464
00465 !
      ============================================================================
00466 !> \brief returns numbers of arguments for n times denser size
00467 !!
00468 !! i.e. * * * *  -->  * . . * . . * . . * (3 times denser)
00469 !
      ============================================================================
00470 integer function size_ntimes_denser (size_original, ndenser)
00471   integer, intent(in) :: size_original , ndenser
00472   size_ntimes_denser= (size_original - 1 ) * (ndenser +1 ) +
     1
00473 end function
00474
00475 !
      ============================================================================
00476 !> \brief Bouger plate computation
00477 !!
00478 !
      ============================================================================
00479 real(dp) function bouger ( R_opt )
00480   real(dp), optional :: r_opt !< height of point above the cylinder
00481   real(dp) :: aux
00482   real(dp) :: r
00483   real(dp) ::  h = 8.84 ! scale height of standard atmosphere
00484
00485   aux = 1
00486
00487   if (present( r_opt ) ) then
00488     r = r_opt
00489     aux = h  + r  - sqrt(  r**2  + (h/2. ) ** 2 )
00490     bouger = 2 * pi * g  * aux
```

```
00491   else
00492     aux = h
00493     bouger = 2 * pi * g * aux
00494     return
00495   endif
00496 end function
00497 !
       ==========================================================================
00498 !> \brief Bouger plate computation
00499 !! see eq. page 288 \cite Warburton77
00500 !
       ==========================================================================
00501 real(dp) function simple_def (R)
00502   real(dp) :: r ,delta
00503
00504   delta = 0.22e-11 * r
00505
00506   simple_def = g0 / r0 * delta * ( 2. - 3./2. * rho_crust / rho_earth
      &
00507     -3./4. * rho_crust / rho_earth * sqrt(2* (1. )) ) * 1000
00508 end function
00509
00510 !polish_meteo
00511
00512 end module
```

7.3 /home/mrajner/src/grat/src/constants.f90 File Reference

This module define some constant values used.

Data Types

- module constants

7.3.1 Detailed Description

This module define some constant values used.

Definition in file constants.f90.

7.4 constants.f90

```
00001 !
       ==========================================================================
00002 !> \file
00003 !! This module define some constant values used
00004 !
       ==========================================================================
00005 module constants
00006
00007   implicit none
00008   integer , parameter :: dp = 8 !< real (kind_real) => real (kind = 8 )
00009   integer , parameter :: sp = 4 !< real (kind_real) => real (kind = 4 )
00010   real(dp) , parameter :: &
00011     T0          = 288.15,     &  !< surface temperature for standard atmosphere
      [K] (15 degC)
00012     g0          = 9.80665,    &  !< mean gravity on the Earth [m/s2]
00013     r0          = 6356.766,   &  !< Earth radius (US Std. atm. 1976)  [km]
00014     p0          = 1013.25,    &  !< surface pressure for standard Earth [hPa]
00015     G           = 6.672e-11,  &  !< Cavendish constant \f$[m^3/kg/s^2]\f$
00016     R_air       = 287.05,     &  !< dry air constant  [J/kg/K]
00017     pi          = 4*atan(1.), &  !< pi = 3.141592... [ ]
00018     rho_crust   = 2670  , &     !< mean density of crust [kg/m3]
00019     rho_earth   = 5500          !< mean density of Earth [kg/m3]
00020
00021 contains
00022
00023 !
       ==========================================================================
00024 !> For given vectors x1, y1 and x2, y2 it gives x2interpolated for x1
00025 !!
00026 !! uses \c ispline and \c spline subroutines
```

```
00027 !
      ================================================================================
00028 subroutine spline_interpolation(x,y, x_interpolated,
      y_interpolated)
00029   implicit none
00030   real(dp) , allocatable , dimension (:) ,intent(in) :: x, y, x_interpolated
00031   real(dp) , allocatable , dimension (:) , intent(out) :: y_interpolated
00032   real(dp) , dimension (:) , allocatable :: b, c, d
00033   integer :: i
00034
00035   allocate (b(size(x)))
00036   allocate (c(size(x)))
00037   allocate (d(size(x)))
00038   allocate (y_interpolated(size(x_interpolated)))
00039
00040   call spline( x , y, b , c, d, size(x))
00041
00042   do i=1, size(x_interpolated)
00043      y_interpolated(i) = ispline(x_interpolated(i) , x , y , b , c , d ,
      size (x) )
00044   enddo
00045
00046 end subroutine
00047
00048 !
      ================================================================================
00049 !> This subroutine was taken from
00050 !! \todo give source
00051 !
      ================================================================================
00052 !  Calculate the coefficients b(i), c(i), and d(i), i=1,2,...,n
00053 !  for cubic spline interpolation
00054 !  s(x) = y(i) + b(i)*(x-x(i)) + c(i)*(x-x(i))**2 + d(i)*(x-x(i))**3
00055 !  for  x(i) <= x <= x(i+1)
00056 !  Alex G: January 2010
00057 !-----------------------------------------------------------------------
00058 !  input..
00059 !  x = the arrays of data abscissas (in strictly increasing order)
00060 !  y = the arrays of data ordinates
00061 !  n = size of the arrays xi() and yi() (n>=2)
00062 !  output..
00063 !  b, c, d  = arrays of spline coefficients
00064 !  comments ...
00065 !  spline.f90 program is based on fortran version of program spline.f
00066 !  the accompanying function fspline can be used for interpolation
00067 !
      ================================================================================
00068 subroutine spline (x, y, b, c, d, n)
00069   implicit none
00070   integer n
00071   real(dp) :: x(n), y(n), b(n), c(n), d(n)
00072   integer i, j, gap
00073   real ::  h
00074
00075   gap = n-1
00076   ! check input
00077   if ( n < 2 ) return
00078   if ( n < 3 ) then
00079     b(1) = (y(2)-y(1))/(x(2)-x(1))   ! linear interpolation
00080     c(1) = 0.
00081     d(1) = 0.
00082     b(2) = b(1)
00083     c(2) = 0.
00084     d(2) = 0.
00085     return
00086   end if
00087   !
00088   ! step 1: preparation
00089   !
00090   d(1) = x(2) - x(1)
00091   c(2) = (y(2) - y(1))/d(1)
00092   do i = 2, gap
00093     d(i) = x(i+1) - x(i)
00094     b(i) = 2.0*(d(i-1) + d(i))
00095     c(i+1) = (y(i+1) - y(i))/d(i)
00096     c(i) = c(i+1) - c(i)
00097   end do
00098   !
00099   ! step 2: end conditions
00100   !
00101   b(1) = -d(1)
00102   b(n) = -d(n-1)
00103   c(1) = 0.0
00104   c(n) = 0.0
00105   if(n /= 3) then
00106     c(1) = c(3)/(x(4)-x(2)) - c(2)/(x(3)-x(1))
00107     c(n) = c(n-1)/(x(n)-x(n-2)) - c(n-2)/(x(n-1)-x(n-3))
```

```
00108     c(1) = c(1)*d(1)**2/(x(4)-x(1))
00109     c(n) = -c(n)*d(n-1)**2/(x(n)-x(n-3))
00110   end if
00111   !
00112   ! step 3: forward elimination
00113   !
00114   do i = 2, n
00115     h = d(i-1)/b(i-1)
00116     b(i) = b(i) - h*d(i-1)
00117     c(i) = c(i) - h*c(i-1)
00118   end do
00119   !
00120   ! step 4: back substitution
00121   !
00122   c(n) = c(n)/b(n)
00123   do j = 1, gap
00124     i = n-j
00125     c(i) = (c(i) - d(i)*c(i+1))/b(i)
00126   end do
00127   !
00128   ! step 5: compute spline coefficients
00129   !
00130   b(n) = (y(n) - y(gap))/d(gap) + d(gap)*(c(gap) + 2.0*c(n))
00131   do i = 1, gap
00132     b(i) = (y(i+1) - y(i))/d(i) - d(i)*(c(i+1) + 2.0*c(i))
00133     d(i) = (c(i+1) - c(i))/d(i)
00134     c(i) = 3.*c(i)
00135   end do
00136   c(n) = 3.0*c(n)
00137   d(n) = d(n-1)
00138 end subroutine spline
00139
00140
00141 !
      ! ============================================================================
00142 !> This subroutine was taken from
00143 !! \todo give source
00144 !
      ! ============================================================================
00145 !=======================================================================
00146 ! function ispline evaluates the cubic spline interpolation at point z
00147 ! ispline = y(i)+b(i)*(u-x(i))+c(i)*(u-x(i))**2+d(i)*(u-x(i))**3
00148 ! where  x(i) <= u <= x(i+1)
00149 !-----------------------------------------------------------------------
00150 ! input..
00151 ! u       = the abscissa at which the spline is to be evaluated
00152 ! x, y    = the arrays of given data points
00153 ! b, c, d = arrays of spline coefficients computed by spline
00154 ! n       = the number of data points
00155 ! output:
00156 ! ispline = interpolated value at point u
00157 !=======================================================================
00158 function ispline(u, x, y, b, c, d, n)
00159 implicit none
00160 real ispline
00161 integer n
00162 real(dp)::  u, x(n), y(n), b(n), c(n), d(n)
00163 integer ::  i, j, k
00164 real :: dx
00165
00166 ! if u is ouside the x() interval take a boundary value (left or right)
00167 if(u <= x(1)) then
00168   ispline = y(1)
00169   return
00170 end if
00171 if(u >= x(n)) then
00172   ispline = y(n)
00173   return
00174 end if
00175
00176 !*
00177 !  binary search for for i, such that x(i) <= u <= x(i+1)
00178 !*
00179 i = 1
00180 j = n+1
00181 do while (j > i+1)
00182   k = (i+j)/2
00183   if(u < x(k)) then
00184     j=k
00185     else
00186     i=k
00187    end if
00188 end do
00189 !*
00190 !  evaluate spline interpolation
00191 !*
00192 dx = u - x(i)
```

```
00193 ispline = y(i) + dx*(b(i) + dx*(c(i) + dx*d(i)))
00194 end function ispline
00195
00196 !
      ================================================================================
00197 !> taken from ArkM http://www.tek-tips.com/viewthread.cfm?qid=1688013
00198 !
      ================================================================================
00199 integer function ntokens(line)
00200 character,intent(in):: line*(*)
00201 integer i, n, toks
00202
00203 i = 1;
00204 n = len_trim(line)
00205 toks = 0
00206 ntokens = 0
00207 do while(i <= n)
00208    do while(line(i:i) == ' ')
00209      i = i + 1
00210      if (n < i) return
00211    enddo
00212    toks = toks + 1
00213    ntokens = toks
00214    do
00215      i = i + 1
00216      if (n < i) return
00217      if (line(i:i) == ' ') exit
00218    enddo
00219 enddo
00220 end function ntokens
00221
00222 !
      ================================================================================
00223 !> This routine skips the lines with comment chars (default '#')
00224 !! from opened files (unit) to read
00225 !
      ================================================================================
00226 subroutine skip_header ( unit , comment_char_optional )
00227   use iso_fortran_env
00228   implicit none
00229   integer , intent (in) :: unit
00230   character (len = 1) , optional :: comment_char_optional
00231   character (len = 60 ) :: dummy
00232   character (len = 1)  :: comment_char
00233   integer :: io_stat
00234
00235   if (present( comment_char_optional ) ) then
00236     comment_char = comment_char_optional
00237   else
00238     comment_char = '#'
00239   endif
00240
00241   read ( unit, * , iostat = io_stat) dummy
00242   if(io_stat == iostat_end) return
00243
00244   do while ( dummy(1:1) .eq. comment_char )
00245     read ( unit, * , iostat = io_stat ) dummy
00246     if(io_stat == iostat_end) return
00247   enddo
00248   backspace(unit)
00249 end subroutine
00250
00251 !> downloaded from  http://aa.usno.navy.mil/faq/docs/jd_formula.php
00252 !! \todo mjd!
00253 real function jd (year,month,day, hh,mm,ss)
00254   implicit none
00255   integer, intent(in) ::  year,month,day
00256   integer, intent(in) :: hh,mm, ss
00257   integer :: i , j , k
00258   i= year
00259   j= month
00260   k= day
00261   jd= k-32075+1461*(i+4800+(j-14)/12)/4+367*(j-2-(j-14)/12*12)/12-3*((i+4900+
      (j-14)/12)/100)/4  + (hh/24.) &
00262   + mm/(24.*60.) +ss/(24.*60.*60.) ! - 2400000.5
00263   return
00264 end function
00265
00266 !subroutine gdate (jd, year,month,day,hh,mm,ss)
00267 !  !! modyfikacja mrajner 20120922
00268 !  !! pobrane  http://aa.usno.navy.mil/faq/docs/jd_formula.php
00269 !  implicit none
00270 !  real, intent(in):: jd
00271 !  real :: aux
00272 !  integer,intent(out) :: year,month,day,hh,mm,ss
00273 !  integer :: i,j,k,l,n
00274
```

```
00275 !  l= int((jd+68569))
00276 !  n= 4*l/146097
00277 !  l= l-(146097*n+3)/4
00278 !  i= 4000*(l+1)/1461001
00279 !  l= l-1461*i/4+31
00280 !  j= 80*l/2447
00281 !  k= l-2447*j/80
00282 !  l= j/11
00283 !  j= j+2-12*l
00284 !  i= 100*(n-49)+i+l
00285
00286 !  year= i
00287 !  month= j
00288 !  day= k
00289
00290 !  aux= jd - int(jd) + 0.0001/86400 ! ostatni argument zapewnia poprawe
00291 !                                   ! jeżeli ss jest integer
00292 !  hh= aux*24
00293 !  mm= aux*24*60    - hh*60
00294 !  ss= aux*24*60*60 - hh*60*60 - mm*60
00295 !end subroutine
00296 real(dp) function mjd  (date)
00297   implicit none
00298   integer ,intent(in) :: date (6)
00299   integer :: aux (6)
00300   integer :: i , k
00301   real(dp) :: dayfrac
00302
00303   aux=date
00304   if ( aux(2) .le.  2) then
00305       aux(1)  = date(1) -  1
00306       aux(2) = date(2) +  12
00307   endif
00308   i = aux(1)/100
00309   k = 2 - i + int(i/4);
00310   mjd = int(365.25 * aux(1) ) - 679006
00311   dayfrac =  aux(4) / 24. + date(5)/(24. * 60. ) + date(6)/(24. * 3600. )
00312   mjd = mjd + int(30.6001*( aux(2) + 1)) + date(3) + k + dayfrac
00313 end function
00314
00315 subroutine invmjd (mjd , date)
00316   implicit none
00317   real(dp), intent (in) :: mjd
00318   integer , intent (out):: date (6)
00319   integer :: t1 ,t4 , h , t2 , t3 , ih1 , ih2
00320   real(dp) :: dayfrac
00321
00322   date =0
00323
00324   t1 = 1+ int(mjd) + 2400000
00325   t4 = mjd - int(mjd);
00326    h = int((t1 - 1867216.25)/36524.25);
00327   t2 = t1 + 1 + h - int(h/4)
00328   t3 = t2 - 1720995
00329   ih1 = int((t3 -122.1)/365.25)
00330   t1 = int(365.25 * ih1)
00331   ih2 = int((t3 - t1)/30.6001);
00332   date(3) = (t3 - t1 - int(30.6001 * ih2)) + t4;
00333   date(2) = ih2 - 1;
00334   if (ih2 .gt. 13) date(2) = ih2 - 13
00335    date(1) = ih1
00336   if (date(2).le. 2) date(1) = date(1) + 1
00337
00338   dayfrac = mjd - int(mjd) + 1./ (60*60*1000)
00339   date(4) = int(dayfrac * 24. )
00340   date(5) = ( dayfrac - date(4) / 24. ) * 60 * 24
00341   date(6) = ( dayfrac - date(4) / 24. - date(5)/(24.*60.)  ) * 60 * 24 *60
00342   if (date(6) .eq. 60 ) then
00343     date(6)=0
00344     date(5)=date(5) + 1
00345   endif
00346 end subroutine
00347
00348 end module constants
```

7.5 /home/mrajner/src/grat/src/example_aggf.f90 File Reference

This program shows some example of using AGGF module.

Functions/Subroutines

- program **example_aggf**
- subroutine simple_atmospheric_model ()

  Reproduces data to Fig.∼3 in.
- subroutine compare_tabulated_green_functions ()

  Compare tabulated green functions from different authors.
- subroutine compute_tabulated_green_functions ()

  Compute AGGF and derivatives.
- subroutine aggf_resp_fels_profiles ()

  Compare different vertical temperature profiles impact on AGGF.
- subroutine compare_fels_profiles ()

  Compare different vertical temperature profiles.
- subroutine aggf_resp_h ()

  Computes AGGF for different site height (h)
- subroutine aggf_resp_t ()

  This computes AGGF for different surface temperature.
- subroutine aggfdt_resp_dt ()

  This computes AGGFDT for different dT.
- subroutine aggf_resp_dz ()

  This computes AGGF for different height integration step.
- subroutine standard1976

  This computes standard atmosphere parameters.
- subroutine aggf_resp_hmax ()

  This computes relative values of AGGF for different atmosphere height integration.
- subroutine aux_heights (table)

  Relative value of aggf depending on integration height.
- subroutine **aggf_thin_layer** ()

7.5.1 Detailed Description

This program shows some example of using AGGF module.

Author

Marcin Rajner

Date

20121108

The examples are in contained subroutines

Definition in file example_aggf.f90.

7.5.2 Function/Subroutine Documentation

7.5.2.1 subroutine example_aggf::aux_heights (real(dp), dimension (:), intent(inout), allocatable *table*)

Relative value of aggf depending on integration height.

Auxiliary subroutine – height sampling for semilog plot

Definition at line 458 of file example_aggf.f90.

7.5.2.2 subroutine example_aggf::compare_fels_profiles ( )

Compare different vertical temperature profiles.

Using tables and formula from Fels [1986]

Definition at line 191 of file example_aggf.f90.

7.5.2.3 subroutine example_aggf::simple_atmospheric_model ( )

Reproduces data to Fig.∼3 in.

Warburton and Goodkind [1977]

Definition at line 39 of file example_aggf.f90.

7.5.2.4 subroutine example_aggf::standard1976 ( )

This computes standard atmosphere parameters.

It computes temperature, gravity, pressure, pressure (simplified formula) density for given height

Definition at line 386 of file example_aggf.f90.

7.6 example_aggf.f90

```
00001 ! ============================================================================
00002 !> \file
00003 !! \brief This program shows some example of using AGGF module
00004 !! \author Marcin Rajner
00005 !! \date 20121108
00006 !!
00007 !! The examples are in contained subroutines
00008 ! ============================================================================
00009 program example_aggf
00010
00011   !> module with subroutines for calculating Atmospheric Gravity Green
      Fucntions
00012   use aggf
00013   use constants
00014   implicit none
00015
00016
00017
00018
00019 ! call standard1976 ()
00020 ! call aggf_resp_hmax ()
00021 ! call aggf_resp_dz ()
00022 ! call aggf_resp_t ()
00023 ! call aggf_resp_h ()
00024 ! call aggfdt_resp_dt ()
00025 ! call compare_fels_profiles ()
00026 ! call compute_tabulated_green_functions ()
00027 ! call aggf_thin_layer ()
00028 ! call aggf_resp_fels_profiles ()
00029 ! call compare_tabulated_green_functions ()
00030  call simple_atmospheric_model()
00031
00032
00033
00034 contains
00035
00036 ! ============================================================================
00037 !> \brief Reproduces data to Fig.~3 in \cite Warburton77
00038 ! ============================================================================
00039 subroutine simple_atmospheric_model ()
00040   real(dp) :: r ! km
00041   integer :: iunit
00042
00043   open (newunit=iunit,file="/home/mrajner/dr/rysunki/simple_approach.dat" ,&
00044     action = "write")
00045     do r = 0. , 25*8
00046     write ( iunit ,  * ) , r , bouger( r_opt= r) * 1e8, & !conversion to
      microGal
```

```
00047       simple_def(r) * 1e8
00048   enddo
00049
00050 end subroutine
00051 ! ========================================================================
00052 !> \brief Compare tabulated green functions from different authors
00053 ! ========================================================================
00054 subroutine compare_tabulated_green_functions
      ()
00055   integer :: i , j , file_unit , ii , iii
00056   real(dp), dimension(:,:), allocatable :: table , results
00057   real(dp), dimension(:,:), allocatable :: parameters
00058   real(dp), dimension(:), allocatable :: x1, y1 ,x2 , y2 , x, y ,
      x_interpolated, y_interpolated
00059   integer :: how_many_denser
00060   character(len=255), dimension(3) :: authors
00061   integer , dimension(3) :: columns
00062
00063   authors=["rajner", "merriam" , "huang"]
00064   ! selected columns for comparison in appropriate tables
00065   columns=[2 , 2, 2]
00066
00067   how_many_denser=0
00068
00069   ! reference author
00070   call read_tabulated_green(table , author = authors(1) )
00071   allocate (results(size_ntimes_denser(size(table(:,1)),
      how_many_denser) , 0 : size(authors) ))
00072
00073   ! fill abscissa in column 0
00074   ii = 1
00075   do i = 1 ,  size (table(:,1) ) - 1
00076     do j = 0 , how_many_denser
00077         results(ii,0) = table(i,1 ) + j * (table(i+1, 1) -table(i,1) ) / (
      how_many_denser + 1 )
00078         ii=ii+1
00079     enddo
00080   enddo
00081   ! and the last element
00082   results( size (results(:,0) )  , 0) =  table( size(table(:,1)) ,1 )
00083
00084   ! take it as main for all series
00085   allocate(x_interpolated( size ( results(:,0))))
00086   x_interpolated = results(:,0)
00087
00088   open (newunit = file_unit , file = "../examples/compare_aggf.dat", action=
      "write")
00089
00090   ! for every author
00091   do i= 1, size(authors)
00092     print * , trim( authors( i ) )
00093     call read_tabulated_green(table , author = authors(i) )
00094     allocate(x( size (table(:,1))))
00095     allocate(y( size (table(:,2))))
00096     x = table(:,1)
00097     y = table(:, columns(i))
00098     call spline_interpolation( x , y , x_interpolated,
      y_interpolated )
00099     if (i.gt.1) then
00100       y_interpolated = ( y_interpolated - results(:,1) ) / results(:,1)  * 100.
00101     endif
00102
00103     results(:, i ) = y_interpolated
00104     deallocate(x,y)
00105   enddo
00106
00107   write (file_unit , '(<size(results(1,:))>f20.5)' ) ( results(i , :) , i = 1 ,
       size(results( :,1)) )
00108   close(file_unit)
00109 end subroutine
00110
00111 ! ========================================================================
00112 !> \brief Compute AGGF and derivatives
00113 ! ========================================================================
00114 subroutine compute_tabulated_green_functions
      ()
00115   integer :: i , file_unit
00116   real(dp) :: val_aggf , val_aggfdt ,val_aggfdh, val_aggfdz
00117   real(dp), dimension(:,:), allocatable :: table , results
00118
00119   ! Get the spherical distances from Merriam92
00120   call read_tabulated_green( table , author = "merriam")
00121
00122   open  ( newunit = file_unit, &
00123           file    = '../dat/rajner_green.dat', &
00124           action  = 'write' &
00125         )
```

```
00126
00127   ! print header
00128   write ( file_unit,*) '# This is set of AGGF computed using module ', &
00129   'aggf from grat software'
00130   write ( file_unit,*) '# Normalization according to Merriam92'
00131   write ( file_unit,*) '# Marcin Rajner'
00132   write ( file_unit,*) '# For detail see www.geo.republika.pl'
00133   write ( file_unit,'(10(a23))')  '#psi[deg]', &
00134     'GN[microGal/hPa]'       , 'GN/dT[microGal/hPa/K]' , &
00135     'GN/dh[microGal/hPa/km]' , 'GN/dz[microGal/hPa/km]'
00136
00137   do i= 1, size(table(:,1))
00138     call compute_aggf( table(i,1) , val_aggf   )
00139     call compute_aggfdt( table(i,1) , val_aggfdt )
00140     call compute_aggf( table(i,1) , val_aggfdh , first_derivative_h
      =.true. )
00141     call compute_aggf( table(i,1) , val_aggfdz , first_derivative_z
      =.true. )
00142     write ( file_unit, '(10(e23.5))' ) &
00143       table(i,1) , val_aggf , val_aggfdt , val_aggfdh, val_aggfdz
00144   enddo
00145   close(file_unit)
00146 end subroutine
00147
00148 ! ========================================================================
00149 !> \brief Compare different vertical temperature profiles impact on AGGF
00150 ! ========================================================================
00151 subroutine aggf_resp_fels_profiles ()
00152   character (len=255) ,dimension (6) :: fels_types
00153   real (dp) :: val_aggf
00154   integer :: i , j, file_unit
00155   real(dp), dimension(:,:), allocatable :: table
00156
00157   ! All possible optional arguments for standard_temperature
00158   fels_types = (/ "US1976"             , "tropical",   &
00159                   "subtropical_summer" , "subtropical_winter" , &
00160                   "subarctic_summer"   , "subarctic_winter"    /)
00161
00162   open  ( newunit = file_unit, &
00163           file    = '../examples/aggf_resp_fels_profiles.dat' , &
00164           action  = 'write' &
00165         )
00166
00167   call read_tabulated_green(table)
00168
00169   ! print header
00170   write ( file_unit , '(100(a20))' ) &
00171     'psi', ( trim( fels_types(i) ) , i = 1 , size (fels_types) )
00172
00173   ! print results
00174   do i = 1 , size (table(:,1))
00175     write (file_unit, '(f20.6$)') table(i,1)
00176     do j = 1 , size(fels_types)
00177       call compute_aggf(table(i,1), val_aggf ,fels_type=fels_types(
      j))
00178       write (file_unit, '(f20.6$)') val_aggf
00179     enddo
00180     write(file_unit, *)
00181   enddo
00182   close(file_unit)
00183 end subroutine
00184
00185
00186 ! ========================================================================
00187 !> \brief Compare different vertical temperature profiles
00188 !!
00189 !! Using tables and formula from \cite Fels86
00190 ! ========================================================================
00191 subroutine compare_fels_profiles ()
00192   character (len=255) ,dimension (6) :: fels_types
00193   real (dp) :: height , temperature
00194   integer :: i , file_unit
00195
00196   ! All possible optional arguments for standard_temperature
00197   fels_types = (/ "US1976"             , "tropical",   &
00198                   "subtropical_summer" , "subtropical_winter" , &
00199                   "subarctic_summer"   , "subarctic_winter"    /)
00200
00201   open  ( newunit = file_unit, &
00202           file    = '../examples/compare_fels_profiles.dat' , &
00203           action  = 'write' &
00204         )
00205
00206   ! Print header
00207   write ( file_unit , '(100(a20))' ) &
00208     'height', ( trim( fels_types(i) ) , i = 1 , size (fels_types) )
00209
```

```
00210   ! Print results
00211   do height = 0. , 70. , 1.
00212     write ( file_unit , '(f20.3$)' ) , height
00213     do i = 1 , size (fels_types)
00214       call standard_temperature &
00215         ( height , temperature , fels_type = fels_types(i) )
00216       write ( file_unit , '(f20.3$)' ),  temperature
00217     enddo
00218     write ( file_unit , * )
00219   enddo
00220   close(file_unit)
00221 end subroutine
00222
00223 ! ========================================================================
00224 !> \brief Computes AGGF for different site height (h)
00225 ! ========================================================================
00226 subroutine aggf_resp_h ()
00227   real(dp), dimension(:,:), allocatable :: table , results
00228   integer :: i, j, file_unit , ii
00229   real(dp) :: val_aggf
00230
00231   ! Get the spherical distances from Merriam92
00232   call read_tabulated_green( table , author = "merriam")
00233
00234   ! Specify the output table and put station height in first row
00235   allocate ( results( 0 : size (table(:,1)) , 7 ) )
00236   results(0,1) = 1./0     ! Infinity in first header
00237   results(0,3) = 0.0      !   0 m
00238   results(0,3) = 0.001    !   1 m
00239   results(0,4) = 0.01     !  10 m
00240   results(0,5) = 0.1      ! 100 m
00241   results(0,6) = 1.       !   1 km
00242   results(0,7) = 10.      !  10 km
00243
00244   ! write results to file
00245   open ( &
00246     newunit = file_unit, &
00247     file    = '../examples/aggf_resp_h.dat', &
00248     action  = 'write' &
00249     )
00250
00251   write (file_unit, '(8(F20.8))' ) results(0, :)
00252   do i =1 , size (table(:,1))
00253     ! denser sampling
00254     do ii = 0,8
00255       results( i , 1 )  = table(i,1) + ii * (table(i+1,1) - table(i,1)) / 9.
00256       ! only compute for small spherical distances
00257       if (results(i, 1) .gt. 0.2 ) exit
00258       write (file_unit, '(F20.7,$)') , results(i,1)
00259       do j =  2 , size(results(1,: ) )
00260         call compute_aggf(results(i,1) , val_aggf, dh=0.0001, h =
      results(0,j))
00261         results(i,j) = val_aggf
00262         write (file_unit,'(f20.7,1x,$)') results(i,j)
00263       enddo
00264       write (file_unit,*)
00265     enddo
00266   enddo
00267   close (file_unit)
00268 end subroutine
00269
00270 ! ========================================================================
00271 !> \brief This computes AGGF for different surface temperature
00272 ! ========================================================================
00273 subroutine aggf_resp_t ()
00274   real(dp), dimension(:,:), allocatable :: table , results
00275   integer :: i, j , file_unit
00276   real(dp) :: val_aggf
00277
00278   ! read spherical distances from Merriam
00279   call read_tabulated_green( table )
00280
00281   ! Header in first row with surface temperature [K]
00282   allocate ( results(0 : size (table(:,1)) , 4 ) )
00283   results(0,1) = 1./0
00284   results(0,2) = t0 +   0.
00285   results(0,3) = t0 +  15.0
00286   results(0,4) = t0 + -45.0
00287   do i =1 , size (table(:,1))
00288     results( i , 1 )  = table(i,1)
00289     do j =  2 , 4
00290     call compute_aggf( results(i , 1 ) , val_aggf, dh = 0.00001,
      t_zero = results(0, j) )
00291     results(i,j) = val_aggf
00292     enddo
00293   enddo
00294
```

```
00295   ! Print results to file
00296   open ( newunit = file_unit , &
00297          file    = '../examples/aggf_resp_t.dat' , &
00298          action  = 'write')
00299   write (file_unit , '(4F20.5)' ) &
00300     ( (results(i,j) , j=1,4) , i = 0, size ( table(:,1) ) )
00301   close (file_unit)
00302 end subroutine
00303
00304 ! ============================================================================
00305 !> \brief This computes AGGFDT for different dT
00306 ! ============================================================================
00307 subroutine aggfdt_resp_dt ()
00308   real(dp), dimension(:,:), allocatable :: table , results
00309   integer :: i, j , file_unit
00310   real(dp) :: val_aggf
00311
00312   ! read spherical distances from Merriam
00313   call read_tabulated_green( table )
00314
00315   ! Header in first row with surface temperature [K]
00316   allocate ( results(0 : size (table(:,1)) , 6 ) )
00317   results(0,1) = 1./0
00318   results(0,2) = 1.
00319   results(0,3) = 5.
00320   results(0,4) = 10.
00321   results(0,5) = 20.
00322   results(0,6) = 50.
00323   do i =1 , size (table(:,1))
00324     results( i , 1 )  = table(i,1)
00325     do j =  2 , 6
00326     call compute_aggfdt( results(i , 1 ) , val_aggf, results(0, j
      ) )
00327     results(i,j) = val_aggf
00328     enddo
00329   enddo
00330
00331   ! Print results to file
00332   open ( newunit = file_unit , &
00333          file    = '../examples/aggfdt_resp_dt.dat' , &
00334          action  = 'write')
00335   write (file_unit , '(6F20.5)' ) &
00336     ( (results(i,j) , j=1,6) , i = 0, size ( table(:,1) ) )
00337   close (file_unit)
00338 end subroutine
00339
00340 ! ============================================================================
00341 !> \brief This computes AGGF for different height integration step
00342 ! ============================================================================
00343 subroutine aggf_resp_dz ()
00344   real(dp), dimension(:,:), allocatable :: table , results
00345   integer :: file_unit , i , j
00346   real(dp) :: val_aggf
00347
00348   open ( newunit = file_unit, &
00349          file    = '../examples/aggf_resp_dz.dat', &
00350          action='write')
00351
00352   ! read spherical distances from Merriam
00353   call read_tabulated_green( table )
00354
00355   ! Differences in AGGF(dz) only for small spherical distances
00356   allocate ( results( 0 : 29 , 0: 5 ) )
00357   results = 0.
00358
00359   ! Header in first row [ infty and selected dz follow on ]
00360   results(0,0) = 1./0
00361   results(0,1:5)=(/ 0.0001, 0.001, 0.01, 0.1, 1./)
00362
00363   do i = 1 , size ( results(:,1) ) - 1
00364     results(i,0) = table(i , 1 )
00365     do j = 1 , size (results(1,:) ) - 1
00366     call compute_aggf( results(i,0) , val_aggf , dh = results(0,j)
    )
00367     results(i, j) =  val_aggf
00368     enddo
00369
00370     ! compute relative errors from column 2 for all dz with respect to column 1
00371     results(i,2:) = abs((results(i,2:) - results(i,1)) / results(i,1) * 100 )
00372   enddo
00373
00374   ! write result to file
00375   write ( file_unit , '(<size(results(1,:))>f14.6)' ) &
00376     ((results(i,j), j=0,size(results(1,:)) - 1), i=0,size(results(:,1)) - 1)
00377   close(file_unit)
00378 end subroutine
00379
```

```
00380 ! ============================================================================
00381 !> \brief This computes standard atmosphere parameters
00382 !!
00383 !! It computes temperature, gravity, pressure, pressure (simplified formula)
00384 !! density for given height
00385 ! ============================================================================
00386 subroutine standard1976  !()
00387   real(dp) :: height , temperature , gravity , pressure , pressure2 , density
00388   integer :: file_unit
00389
00390   open ( newunit = file_unit , &
00391          file    = '../examples/standard1976.dat', &
00392          action  = 'write' )
00393   ! print header
00394   write ( file_unit , '(6(a12))' ) &
00395     'height[km]', 'T[K]' , 'g[m/s2]' , 'p[hPa]', 'p_simp[hPa]' , 'rho[kg/m3]'
00396   do height=0.,98.
00397     call standard_temperature( height , temperature )
00398     call standard_gravity( height , gravity )
00399     call standard_pressure( height , pressure )
00400     call standard_pressure( height , pressure2 ,
      if_simplificated = .true. )
00401     call standard_density( height , density )
00402     ! print results to file
00403     write( file_unit,'(5f12.5, e12.3)'), &
00404     height,temperature , gravity , pressure , pressure2 , density
00405   enddo
00406   close( file_unit )
00407 end subroutine
00408
00409 ! ============================================================================
00410 !> \brief This computes relative values of AGGF for different atmosphere
00411 !! height integration
00412 ! ============================================================================
00413 subroutine aggf_resp_hmax ()
00414   real (dp) , dimension (10) :: psi
00415   real (dp) , dimension (:)   , allocatable :: heights
00416   real (dp) , dimension (:,:) , allocatable :: results
00417   integer :: file_unit , i , j
00418   real(dp) :: val_aggf
00419
00420   ! selected spherical distances
00421   psi=(/0.000001, 0.000005,0.00001, 1,  2, 3 , 5, 10 , 90 ,  180 /)
00422
00423   ! get heights (for nice graph) - call auxiliary subroutine
00424   call aux_heights( heights )
00425
00426   open ( newunit = file_unit , &
00427          file    = '../examples/aggf_resp_hmax.dat', &
00428          action  = 'write')
00429
00430   allocate ( results( 0:size(heights)-1 , 1+size(psi) ) )
00431
00432   do j=0 , size (results(:,1))
00433       results( j , 1 ) = heights(j)
00434
00435     do i = 1 , size(psi)
00436       call compute_aggf( psi(i) , val_aggf , hmax = heights(j) , dh
       = 0.00001 )
00437       results(j,i+1) = val_aggf
00438
00439       !> Relative value of aggf depending on integration height
00440       if (j.gt.0) then
00441         results(j,i+1) = results(j,i+1) / results(0,i+1) * 100
00442       endif
00443     enddo
00444   enddo
00445
00446   ! print header
00447   write(file_unit , '(a14,SP,100f14.5)' ),"#wys\psi", (psi(j) , j= 1,size(psi))
00448   ! print results
00449   do i=1, size (results(:,1))-1
00450     write(file_unit, '(100f14.3)' ) (results(i,j), j = 1, size(psi)+1 )
00451   enddo
00452   close(file_unit)
00453 end subroutine
00454
00455 ! ============================================================================
00456 !> \brief Auxiliary subroutine -- height sampling for semilog plot
00457 ! ============================================================================
00458 subroutine aux_heights ( table )
00459   real(dp) , dimension (:), allocatable, intent(inout) :: table
00460   real(dp) , dimension (0:1000) :: heights
00461   real(dp) :: height
00462   integer :: i , count_heights
00463
00464   heights(0) =60
```

```
00465   i=0
00466   height=-0.001
00467   do while (height.lt.60)
00468     i=i+1
00469     if (height.lt.0.10) then
00470       height=height+2./1000
00471     elseif(height.lt.1) then
00472       height=height+50./1000
00473     else
00474       height=height+1
00475     endif
00476     heights(i)= height
00477     count_heights=i
00478   enddo
00479   allocate ( table( 0 : count_heights ) )
00480   table(0 : count_heights ) = heights( 0 : count_heights )
00481 end subroutine
00482
00483 subroutine aggf_thin_layer ()
00484   integer :: file_unit , i
00485   real(dp) , dimension (:,:), allocatable :: table
00486
00487   ! read spherical distances from Merriam
00488   call read_tabulated_green(table)
00489   do i = 1 , size (table(:,1))
00490     write(*,*) table(i,1:2) , gn_thin_layer(table(i,1))
00491   enddo
00492
00493 end subroutine
00494 end program
```

7.7 /home/mrajner/src/grat/src/get_cmd_line.f90 File Reference

This module sets the initial values for parameters reads from command line and gives help it allows to specify commands with or without spaces therefore it is convienient to use with auto completion of names.

Data Types

- module get_cmd_line
- type get_cmd_line::green_functions
- type get_cmd_line::polygon_data
- type get_cmd_line::polygon_info
- type get_cmd_line::dateandmjd
- type get_cmd_line::additional_info
- type get_cmd_line::cmd_line
- type get_cmd_line::site_data
- type get_cmd_line::file

7.7.1 Detailed Description

This module sets the initial values for parameters reads from command line and gives help it allows to specify commands with or without spaces therefore it is convienient to use with auto completion of names.

Definition in file get_cmd_line.f90.

7.8 get_cmd_line.f90

```
00001 ! ================================================================================
00002 !> \file
00003 !! \brief This module sets the initial values for parameters
00004 !! reads from command line and gives help
00005 !! it allows to specify commands with or without spaces therefore it is
00006 !! convienient to use with auto completion of names
00007 ! ================================================================================
00008 module get_cmd_line
00009   use iso_fortran_env
```

```
00010   use constants
00011
00012   implicit none
00013
00014   !-----------------------------------------------
00015   ! Greens function
00016   !-----------------------------------------------
00017
00018   type green_functions
00019     real(dp),allocatable,dimension(:) :: distance
00020     real(dp),allocatable,dimension(:) :: data
00021     logical  :: if
00022   end type
00023   type(green_functions), allocatable , dimension(:) :: green
00024   integer :: denser(2) = [1,1]
00025
00026   !-----------------------------------------------
00027   ! polygons
00028   !-----------------------------------------------
00029   type polygon_data
00030     logical :: use
00031     real(sp), allocatable , dimension (:,:) :: coords
00032   end type
00033
00034   type polygon_info
00035     integer :: unit
00036     character(:), allocatable  :: name
00037     type(polygon_data) , dimension (:) , allocatable :: polygons
00038     logical :: if
00039   end type
00040
00041   type(polygon_info) , dimension (2) :: polygons
00042
00043   !-----------------------------------------------
00044   ! dates
00045   !-----------------------------------------------
00046   type dateandmjd
00047     real(dp) :: mjd
00048     integer,dimension (6) :: date
00049   end type
00050
00051   real(kind=4) :: cpu_start , cpu_finish  !< for time execution of program
00052   type(dateandmjd) , allocatable,dimension (:) :: dates
00053
00054
00055   !-----------------------------------------------
00056   ! command line entry
00057   !-----------------------------------------------
00058   type additional_info
00059     character (len=55) ,allocatable ,dimension(:) :: names
00060   end type
00061   type cmd_line
00062     character(2) :: switch
00063     integer ::      fields
00064     character (len=255) ,allocatable ,dimension(:) :: field
00065     type (additional_info), allocatable , dimension(:) ::
      fieldnames
00066   end type
00067
00068   !-----------------------------------------------
00069   ! site information
00070   !-----------------------------------------------
00071   type site_data
00072    character(:), allocatable :: name
00073    real(sp)                  :: lat,lon,height
00074   end type
00075
00076   type(site_data) , allocatable , dimension(:) :: sites
00077
00078   ! various
00079   !-----------------------------------------------
00080   integer :: fileunit_tmp              !< unit of scratch file
00081   integer,dimension(8):: execution_date !< To give time stamp of execution
00082   character (len = 2) :: method = "2D"  !< computation method
00083
00084   !-----------------------------------------------
00085   ! Site names file
00086   !-----------------------------------------------
00087   character(:), allocatable &
00088           :: filename_site
00089   integer :: fileunit_site
00090
00091
00092   type file
00093     character(:), allocatable &
00094           :: name
00095     ! varname , lonname,latname,levelname , timename
```

```
00096     character(len=50) :: names(5) = [ "z", "lon", "lat","level","time"]
00097
00098     integer :: unit = output_unit
00099
00100     ! if file was determined
00101     logical :: if =.false.
00102
00103     ! to read into only once
00104     logical :: first_call =.true.
00105
00106     ! boundary of model e , w ,s ,n
00107     real(sp):: limits(4)
00108
00109 !     resolution of model in lon lat
00110 !    real(sp):: resolution(2)
00111
00112     real(sp) , allocatable ,dimension(:) :: lat , lon , time ,level
00113     integer , allocatable , dimension(:,: ) :: date
00114
00115     real (sp), dimension(2) :: latrange , lonrange
00116
00117     ! todo
00118     logical :: if_constant_value
00119     real(sp):: constant_value
00120
00121     ! data
00122     !> 4 dimension - lat , lon , level , mjd
00123     ! todo
00124     real(sp) , allocatable , dimension (:,:,:) :: data
00125
00126     ! netcdf identifiers
00127     integer :: ncid
00128     integer :: interpolation = 1
00129   end type
00130
00131   ! External files
00132   type(file) ::  log  , output , moreverbose , refpres
00133   type(file) , allocatable, dimension (:) :: model
00134
00135   character (len =40) :: model_names (5) = ["pressure_surface" , &
00136     "temperature_surface" , "topography" , "landsea" , "pressure levels" ]
00137
00138
00139   character(len=5) :: green_names(5) = [ "GN   ", "GN/dt", "GN/dh","GN/dz","GE
      "]
00140
00141
00142   ! Verbose information and the output for \c log_file
00143   logical :: if_verbose  = .false.  !< whether print all information
00144   logical :: inverted_barometer  = .true.
00145
00146   character (50) :: interpolation_names (2) &
00147     = [ "nearest" , "bilinear" ]
00148
00149   !-----------------------------------------------------
00150   ! For preety printing
00151   !-----------------------------------------------------
00152   character(len=255), parameter ::  &
00153     form_header    = '(60("#"))' , &
00154     form_separator = '(60("-"))' , &
00155     form_inheader  = '(("#"),1x,a56,1x,("#"))' , &
00156     form_60        = "(a,100(1x,g0))",          &
00157     form_61        = "(2x,a,100(1x,g0))",    &
00158     form_62        = "(4x,a,100(1x,g0))",       &
00159     form_63        = "(6x,100(x,g0))",       &
00160     form_64        = "(4x,4x,a,4x,a)"
00161
00162
00163 contains
00164 ! ================================================================================
00165 !> This subroutine counts the command line arguments
00166 !!
00167 !! Depending on command line options set all initial parameters and reports it
00168 ! ================================================================================
00169 subroutine intro (program_calling)
00170   implicit none
00171   integer :: i, j
00172   character(len=255) :: dummy, dummy2,arg
00173   character(len=*) :: program_calling
00174   type(cmd_line) :: cmd_line_entry
00175
00176   if(iargc().eq.0) then
00177     write(output_unit , '(a)' ) , 'Short description: ./'//program_calling//'
      -h'
00178     call exit
00179   else
00180     open(newunit=fileunit_tmp,status='scratch')
```

```
00181     write (fileunit_tmp,form_61) "command invoked"
00182     call get_command(dummy)
00183     write (fileunit_tmp,form_62) trim(dummy)
00184     do i = 1 , iargc()
00185       call get_command_argument(i,dummy)
00186       ! allow specification like '-F file' and '-Ffile'
00187       call get_command_argument(i+1,dummy2)
00188       if (dummy(1:1).eq."-") then
00189         arg = trim(dummy)
00190       else
00191         arg=trim(arg)//trim(dummy)
00192       endif
00193       if(dummy2(1:1).eq."-".or.i.eq.iargc()) then
00194          call get_cmd_line_entry(arg, cmd_line_entry ,
      program_calling = program_calling)
00195       endif
00196     enddo
00197
00198     call if_minimum_args( program_calling = program_calling )
00199
00200     ! Where and if to log the additional information
00201     if (log%if) then
00202       ! if file name was given then automaticall switch verbose mode
00203       if_verbose = .true.
00204       open (newunit = log%unit, file = log%name , action = "write" )
00205     else
00206       ! if you don't specify log file, or not switch on verbose mode
00207       ! all additional information will go to trash
00208       ! Change  /dev/null accordingly if your file system does not
00209       ! support this name
00210       if (.not.if_verbose) then
00211         open (newunit=log%unit, file = "/dev/null", action = "write" )
00212       endif
00213     endif
00214   endif
00215 end subroutine
00216
00217 ! ================================================================================
00218 !> Check if at least all obligatory command line arguments were given
00219 !! if not print warning
00220 ! ================================================================================
00221 subroutine if_minimum_args ( program_calling )
00222   implicit none
00223   character (*) , intent(in) :: program_calling
00224
00225   if (program_calling.eq."grat" ) then
00226
00227     if (size(sites) .eq. 0) then
00228       write(error_unit, * ) "ERROR:", program_calling
00229       write(error_unit, * ) "ERROR:", "no sites!"
00230       call exit
00231     endif
00232   elseif(program_calling.eq."polygon_check" ) then
00233   endif
00234 end subroutine
00235
00236 ! ================================================================================
00237 !> This function is true if switch is used by calling program or false if it
00238 !! is not
00239 ! ================================================================================
00240 logical function if_switch_program (program_calling , switch )
00241   implicit none
00242   character(len=*), intent (in) :: program_calling
00243   character(len=*),  intent (in) :: switch
00244   character, dimension(:) , allocatable :: accepted_switch
00245   integer :: i
00246
00247   ! default
00248   if_switch_program=.false.
00249
00250   ! depending on program calling decide if switch is permitted
00251   if (program_calling.eq."grat") then
00252     allocate( accepted_switch(15) )
00253     accepted_switch = [ "V" , "f" , "S", "B" , "L" , "G" , "P" , "p", &
00254       "o" , "F" , "I" , "D" , "d" , "v" , "h"   ]
00255   elseif(program_calling.eq."polygon_check") then
00256     allocate( accepted_switch(12) )
00257     accepted_switch = [ "V" , "f" , "A", "B" , "L" , "P" , "o", "S" , &
00258       "h" , "v" , "I" , "i"]
00259   elseif(program_calling.eq."value_check") then
00260     allocate( accepted_switch(9) )
00261     accepted_switch = [ "V" , "F" , "o", "S" , "h" , "v" , "I" , "D" , "L"]
00262   else
00263     if_switch_program=.true.
00264     return
00265   endif
00266
```

```
00267   ! loop trough accepted switches
00268   do i =1, size (accepted_switch)
00269     if (switch(2:2).eq.accepted_switch(i)) if_switch_program=.
      true.
00270   enddo
00271 end function
00272
00273 ! ============================================================================
00274 !> This subroutine counts the command line arguments and parse appropriately
00275 ! ============================================================================
00276 subroutine parse_option (cmd_line_entry , program_calling)
00277   type(cmd_line),intent(in):: cmd_line_entry
00278   character(len=*), optional :: program_calling
00279   integer :: i
00280
00281   ! all the command line option are stored in tmp file and later its decide
00282   ! if it is written to STDOUT , log_file or nowwhere
00283     select case (cmd_line_entry%switch)
00284     case ('-h')
00285       call print_help(program_calling)
00286       call exit
00287     case ('-v')
00288       call print_version(program_calling)
00289       call exit()
00290     case ('-V')
00291       if_verbose = .true.
00292       write(fileunit_tmp, form_62) 'verbose mode' ,trim(log%name)
00293       if (len(trim(cmd_line_entry%field(1))).gt.0) then
00294         log%if = .true.
00295         log%name = trim(cmd_line_entry%field(1))
00296         write(fileunit_tmp, form_62) 'the log file was set:' ,log%name
00297       endif
00298     case ('-S')
00299       ! check if format is proper for site
00300       ! i,e. -Sname,B,L[,H]
00301       if (.not. allocated(sites)) then
00302         if (  is_numeric(cmd_line_entry%field(2)) &
00303         .and.is_numeric(cmd_line_entry%field(3)) &
00304         .and.index(cmd_line_entry%field(1), "/" ).eq.0 &
00305         .and.(.not.cmd_line_entry%field(1).eq. "Rg" ) &
00306         ) then
00307             allocate (sites(1))
00308             sites(1)%name = trim(cmd_line_entry%field(1))
00309             read ( cmd_line_entry%field(2) , * ) sites(1)%lat
00310             if (abs(sites(1)%lat).gt.90.) &
00311               sites(1)%lat = sign(90.,sites(1)%lat)
00312             read ( cmd_line_entry%field(3) , * ) sites(1)%lon
00313             if (sites(1)%lon.ge.360.) sites(1)%lon = mod(sites(1)%lon,360.)
00314             if (is_numeric(cmd_line_entry%field(4) ) ) then
00315               read ( cmd_line_entry%field(4) , * ) sites(1)%height
00316             endif
00317             write(fileunit_tmp, form_62) 'the site was set (BLH):' , &
00318               sites(1)%name, sites(1)%lat , sites(1)%lon , sites(1)%height
00319         else
00320           ! or read sites from file
00321           if (file_exists(cmd_line_entry%field(1) ) ) then
00322             write(fileunit_tmp, form_62) 'the site file was set:' , &
00323               cmd_line_entry%field(1)
00324             call read_site_file(cmd_line_entry%field(1))
00325           elseif(index(cmd_line_entry%field(1), "/" ).ne.0 &
00326               .or.cmd_line_entry%field(1).eq."Rg")  then
00327             call parse_gmt_like_boundaries(
      cmd_line_entry )
00328           else
00329             call print_warning( "site" , fileunit_tmp)
00330           endif
00331         endif
00332       else
00333         call print_warning( "repeated" , fileunit_tmp)
00334       endif
00335     case ("-I")
00336       write( fileunit_tmp , form_62 , advance="no" ) "interpolation method was
       set:"
00337       do i = 1 , cmd_line_entry%fields
00338         if (is_numeric(cmd_line_entry%field(i))) then
00339           read ( cmd_line_entry%field(i) , * ) model(i)%interpolation
00340           write(fileunit_tmp , '(a10,x,$)' ) interpolation_names(model(i)
      %interpolation)
00341           if (model(i)%interpolation.gt.size(interpolation_names)) then
00342             model(i)%interpolation=1
00343           endif
00344         endif
00345       enddo
00346       write(fileunit_tmp , *)
00347     case ("-L")
00348       moreverbose%if=.true.
00349       moreverbose%name=cmd_line_entry%field(1)
```

```
00350       moreverbose%names(1) = cmd_line_entry%fieldnames(1)%names(1)
00351       write (fileunit_tmp , form_62) "printing additional information"
00352       if (len(moreverbose%name).gt.0 .and. moreverbose%name.ne."") then
00353         open (newunit = moreverbose%unit , file = moreverbose%name , action
     = "write" )
00354       endif
00355     case ("-B")
00356       if (cmd_line_entry%field(1).eq."N" ) inverted_barometer=.false.
00357     case ('-D')
00358       call parse_dates( cmd_line_entry )
00359     case ('-F')
00360       allocate(model(cmd_line_entry%fields))
00361       do i = 1, cmd_line_entry%fields
00362         call get_model_info(model(i) , cmd_line_entry , i )
00363       enddo
00364     case ("-G")
00365       call parse_green(cmd_line_entry)
00366     case ('-M')
00367       method = cmd_line_entry%field(1)
00368       write(fileunit_tmp, form_62), 'method was set: ' , method
00369     case ('-o')
00370       output%if=.true.
00371       output%name=cmd_line_entry%field(1)
00372       write(fileunit_tmp, form_62), 'output file was set: ' , output%name
00373       if (len(output%name).gt.0.and. output%name.ne."") then
00374         open (newunit = output%unit , file = output%name , action = "write"
     )
00375       endif
00376     case ('-P')
00377       do i = 1 , 2 !size(cmd_line_entry%field)
00378         polygons(i)%name=cmd_line_entry%field(i)
00379         if (file_exists((polygons(i)%name))) then
00380           write(fileunit_tmp, form_62), 'polygon file was set: ' , polygons(i)
    %name
00381           polygons(i)%if=.true.
00382           ! todo
00383 !          call read_polygon (polygons(i))
00384         else
00385           write(fileunit_tmp, form_62), 'file do not exist. Polygon file was
     IGNORED'
00386         endif
00387       enddo
00388     case default
00389       write(fileunit_tmp,form_62), "unknown argument: IGNORING"
00390     end select
00391     return
00392 end subroutine
00393
00394 ! ========================================================================
00395 !> This subroutine parse -G option i.e. reads Greens function
00396 ! ========================================================================
00397 subroutine parse_green ( cmd_line_entry)
00398   type (cmd_line)  :: cmd_line_entry
00399   character (60) :: filename
00400   integer :: i , iunit , io_status , lines ,  ii
00401   integer :: fields(2)= [1,2]
00402   real (sp) , allocatable , dimension(:) :: tmp
00403
00404     write(fileunit_tmp , form_62) "Green function file was set:"
00405     allocate (green(cmd_line_entry%fields))
00406
00407     do i = 1 , cmd_line_entry%fields
00408
00409       if (i.eq.6) then
00410         if (is_numeric(cmd_line_entry%field(i))) then
00411           read( cmd_line_entry%field(i), *) denser(1)
00412           if (is_numeric(cmd_line_entry%fieldnames(i)%names(1))) then
00413             read( cmd_line_entry%fieldnames(i)%names(1), *) denser(2)
00414           endif
00415           return
00416          endif
00417       endif
00418
00419       if (.not.file_exists(cmd_line_entry%field(i)) &
00420       .and. (.not. cmd_line_entry%field(i).eq."merriam" &
00421       .and. .not. cmd_line_entry%field(i).eq."huang" &
00422       .and. .not. cmd_line_entry%field(i).eq."rajner" )) then
00423         cmd_line_entry%field(i)="merriam"
00424       endif
00425
00426
00427       !> change the paths accordingly
00428       if (cmd_line_entry%field(i).eq."merriam") then
00429         filename="/home/mrajner/src/grat/dat/merriam_green.dat"
00430         if (i.eq.1) fields = [1,2]
00431         if (i.eq.2) fields = [1,3]
00432         if (i.eq.3) fields = [1,4]
```

```
00433         if (i.eq.4) fields = [1,4]
00434         if (i.eq.5) fields = [1,6]
00435       elseif(cmd_line_entry%field(i).eq."huang") then
00436         filename="/home/mrajner/src/grat/dat/huang_green.dat"
00437         if (i.eq.1) fields = [1,2]
00438         if (i.eq.2) fields = [1,3]
00439         if (i.eq.3) fields = [1,4]
00440         if (i.eq.4) fields = [1,5]
00441         if (i.eq.5) fields = [1,6]
00442       elseif(cmd_line_entry%field(i).eq."rajner") then
00443         filename="/home/mrajner/src/grat/dat/rajner_green.dat"
00444         if (i.eq.1) fields = [1,2]
00445         if (i.eq.2) fields = [1,3]
00446         if (i.eq.3) fields = [1,4]
00447         if (i.eq.4) fields = [1,5]
00448         if (i.eq.5) fields = [1,6]
00449       elseif(file_exists(cmd_line_entry%field(i))) then
00450         filename = cmd_line_entry%field(i)
00451         if (size(cmd_line_entry%fieldnames).ne.0 .and. allocated(cmd_line_entry
      %fieldnames(i)%names)) then
00452           do ii=1, 2
00453             if(is_numeric(cmd_line_entry%fieldnames(i)%names(ii) ) )
      then
00454               read( cmd_line_entry%fieldnames(i)%names(ii), *) fields(ii)
00455             endif
00456           enddo
00457         endif
00458       endif
00459
00460       allocate(tmp(max(fields(1),fields(2))))
00461       lines = 0
00462       open ( newunit =iunit,file=filename,action="read")
00463       do
00464         call skip_header(iunit)
00465         read (iunit , * , iostat = io_status)
00466         if (io_status == iostat_end) exit
00467         lines = lines + 1
00468       enddo
00469       allocate (green(i)%distance(lines))
00470       allocate (green(i)%data(lines))
00471       rewind(iunit)
00472       lines = 0
00473       do
00474         call skip_header(iunit)
00475         lines = lines + 1
00476         read (iunit , * , iostat = io_status) tmp
00477         if (io_status == iostat_end) exit
00478         green(i)%distance(lines) = tmp(fields(1))
00479         green(i)%data(lines)     = tmp(fields(2))
00480       enddo
00481       deallocate(tmp)
00482       close(iunit)
00483       if (cmd_line_entry%field(i).eq."merriam" .and. i.eq.4) then
00484         green(i)%data = green(i)%data * (-1.)
00485       endif
00486       if (cmd_line_entry%field(i).eq."huang" .and. (i.eq.3.or.i.eq.4)) then
00487         green(i)%data = green(i)%data * 1000.
00488       endif
00489       write(fileunit_tmp , form_63) trim(green_names(i)), &
00490         trim(cmd_line_entry%field(i)),":", fields
00491     enddo
00492 end subroutine
00493
00494 ! ================================================================================
00495 !> Counts occurence of character (separator, default comma) in string
00496 ! ================================================================================
00497 integer function count_separator (dummy , separator)
00498   character(*) , intent(in) ::dummy
00499   character(1), intent(in), optional  :: separator
00500   character(1)  :: sep
00501   character(:), allocatable :: dummy2
00502   integer :: i
00503
00504   dummy2=dummy
00505   sep = ","
00506   if (present(separator)) sep = separator
00507   count_separator=0
00508   do
00509     i = index(dummy2, sep)
00510     if (i.eq.0) exit
00511     dummy2 = dummy2(i+1:)
00512     count_separator=count_separator+1
00513   enddo
00514 end function
00515
00516
00517 ! ================================================================================
```

```
00518 !> This subroutine fills the fields of command line entry for every input arg
00519 ! ================================================================================
00520 subroutine get_cmd_line_entry (dummy , cmd_line_entry ,
       program_calling )
00521   character(*) :: dummy
00522   character(:), allocatable :: dummy2
00523   type (cmd_line),intent(out) :: cmd_line_entry
00524   character(1) :: separator=","
00525   character(len=*) , intent(in) , optional :: program_calling
00526   integer :: i , j , ii , jj
00527
00528   cmd_line_entry%switch = dummy(1:2)
00529   write(fileunit_tmp, form_61) , dummy
00530   if (.not.if_switch_program(program_calling, cmd_line_entry
      %switch)) then
00531     write ( fileunit_tmp , form_62 ) "this switch is IGNORED by program "//
      program_calling
00532     return
00533   endif
00534
00535
00536   dummy=dummy(3:)
00537
00538   cmd_line_entry%fields = count_separator(dummy) + 1
00539   allocate(cmd_line_entry%field (cmd_line_entry%fields) )
00540
00541   ! if ":" separator is present in command line allocate
00542   ! additional array for fieldnames
00543   if (count_separator(dummy, ":" ).ge.1) then
00544     allocate(cmd_line_entry%fieldnames (cmd_line_entry%fields) )
00545   endif
00546   do i = 1 , cmd_line_entry%fields
00547     j = index(dummy, separator)
00548     cmd_line_entry%field(i) = dummy(1:j-1)
00549     if (i.eq.cmd_line_entry%fields) cmd_line_entry%field(i)=dummy
00550     dummy=dummy(j+1:)
00551
00552     ! separate field and fieldnames
00553     if ( index(cmd_line_entry%field(i),":").ne.0 ) then
00554       dummy2 = trim(cmd_line_entry%field(i))//":"
00555       allocate ( cmd_line_entry%fieldnames(i)%names(count_separator
      (dummy2,":") - 1 ))
00556       do ii = 1, size(cmd_line_entry%fieldnames(i)%names)+1
00557         jj = index(dummy2, ":")
00558         if (ii.eq.1) then
00559           cmd_line_entry%field(i) = dummy2(1:jj-1)
00560         else
00561           cmd_line_entry%fieldnames(i)%names(ii-1) = dummy2(1:jj-1)
00562         endif
00563         dummy2 = dummy2(jj+1:)
00564       enddo
00565     endif
00566   enddo
00567   call parse_option(cmd_line_entry , program_calling =
      program_calling)
00568 end subroutine
00569
00570 subroutine get_model_info ( model , cmd_line_entry , field)
00571   type(cmd_line),intent(in):: cmd_line_entry
00572   type(file),intent(inout):: model
00573   integer :: field , i
00574
00575   model%name = trim(cmd_line_entry%field(field))
00576   if (model%name.eq."") return
00577   if ( file_exists(model%name) ) then
00578     write (fileunit_tmp , form_62) , trim(model_names(field) )
00579     write(fileunit_tmp, form_63), trim(model%name)
00580
00581     do i =1 , size (model%names)
00582       if (size(cmd_line_entry%fieldnames).gt.0) then
00583         if (i.le.size (cmd_line_entry%fieldnames(field)%names) &
00584           .and. cmd_line_entry%fieldnames(field)%names(i).ne."" &
00585           ) then
00586           model%names(i) = cmd_line_entry%fieldnames(field)%names(i)
00587         endif
00588       endif
00589       write(fileunit_tmp, form_63, advance="no") , trim( model%names(i))
00590     enddo
00591     model%if=.true.
00592     write(fileunit_tmp, form_63)
00593   elseif(is_numeric(model%name)) then
00594     model%if_constant_value=.true.
00595     read (model%name , * ) model%constant_value
00596     write (fileunit_tmp , form_62) , trim(model_names(field) )
00597     write(fileunit_tmp, form_63), 'constant value was set: ' , model
      %constant_value
00598     model%if_constant_value=.true.
```

```
00599   else
00600     write (fileunit_tmp , form_63 ) "no (correct) model in field: ", field
00601   endif
00602 end subroutine
00603
00604
00605 ! ============================================================================
00606 !> This subroutine checks if given limits for model are proper
00607 ! ============================================================================
00608 subroutine parse_gmt_like_boundaries ( cmd_line_entry
      )
00609   implicit none
00610   real(sp) :: limits (4) , resolution (2) =[1,1]
00611   real(sp) :: range_lon , range_lat , lat , lon
00612   character(10) :: dummy
00613   integer :: i , ii
00614   type (cmd_line) , intent (in) :: cmd_line_entry
00615   character(:) ,allocatable :: text
00616   integer :: n_lon , n_lat
00617
00618   text = cmd_line_entry%field(1)
00619
00620   do i=1,3
00621     if ( is_numeric(text(1:index(text, "/"))) ) then
00622       read ( text(1:index(text, "/")) , * )  limits(i)
00623     else
00624       if (text.eq."Rg" ) then
00625         limits=[0. , 360. , -90 , 90. ]
00626       endif
00627     endif
00628     text=text(index(text,"/")+1:)
00629   enddo
00630
00631   if ( is_numeric(text(1:)) ) then
00632     read ( text(1:) , * )  limits(4)
00633   else
00634     call print_warning("boundaries")
00635   endif
00636
00637   do i = 1 ,2
00638     if (limits(i).lt. -180. .or. limits(i).gt.360. ) then
00639       call print_warning("boundaries")
00640     else
00641       if (limits(i).lt.0.) limits(i)=limits(i)+360.
00642     endif
00643   enddo
00644   do i =3,4
00645     if (limits(i).lt. -90. .or. limits(i).gt.90. ) then
00646       call print_warning("boundaries")
00647     endif
00648   enddo
00649   if (limits(3).gt.limits(4)) then
00650     call print_warning("boundaries")
00651   endif
00652
00653   if (is_numeric(cmd_line_entry%field(2) ) ) then
00654     read (cmd_line_entry%field(2) , * ) resolution(1)
00655     resolution(2) = resolution(1)
00656   endif
00657   if (is_numeric(cmd_line_entry%field(3) ) ) then
00658     read (cmd_line_entry%field(3) , * ) resolution(2)
00659   endif
00660
00661   range_lon=limits(2) - limits(1)
00662   if (range_lon.lt.0) range_lon = range_lon + 360.
00663   range_lat=limits(4) - limits(3)
00664   n_lon = floor( range_lon / resolution(1)) + 1
00665   n_lat = floor( range_lat / resolution(2)) + 1
00666   allocate (sites( n_lon * n_lat ) )
00667
00668   do i = 1 , n_lon
00669     lon = limits(1) + (i-1) * resolution(1)
00670     if (lon.ge.360.) lon = lon - 360.
00671     do ii = 1 , n_lat
00672       lat = limits(3) + (ii-1) * resolution(2)
00673       sites( (i-1) * n_lat + ii  )%lon = lon
00674       sites( (i-1) * n_lat + ii  )%lat = lat
00675     enddo
00676   enddo
00677
00678 end subroutine
00679
00680 ! ============================================================================
00681 !> Read site list from file
00682 !!
00683 !! checks for arguments and put it into array \c sites
00684 ! ============================================================================
```

```
00685 subroutine read_site_file ( file_name )
00686   character(len=*) , intent(in) ::  file_name
00687   integer :: io_status , i , good_lines = 0 , number_of_lines = 0 , nloop
00688   character(len=255) ,dimension(4)  :: dummy
00689   character(len=255) :: line_of_file
00690   type(site_data) :: aux
00691
00692
00693
00694     open ( newunit = fileunit_site , file = file_name, &
00695             iostat = io_status ,status = "old" , action="read" )
00696
00697     ! two loops, first count good lines and print rejected
00698     ! second allocate array of sites and read coordinates into it
00699       nloops: do nloop = 1, 2
00700       if (nloop.eq.2) allocate(sites(good_lines))
00701       if (number_of_lines.ne.good_lines) then
00702         call print_warning("site_file_format")
00703       endif
00704       good_lines=0
00705       line_loop:do
00706         read ( fileunit_site , '(a)' , iostat = io_status ) line_of_file
00707         if ( io_status == iostat_end)  exit  line_loop
00708         number_of_lines = number_of_lines + 1
00709         ! we need at least 3 parameter for site (name , B , L )
00710         if (ntokens(line_of_file).ge.3) then
00711           ! but no more than 4 parameters (name , B , L, H)
00712           if (ntokens(line_of_file).gt.4) then
00713             read ( line_of_file , * ) dummy(1:4)
00714           else
00715             read ( line_of_file , * ) dummy(1:3)
00716             ! if site height was not given we set it to zero
00717             dummy(4)="0."
00718           endif
00719         endif
00720         ! check the values given
00721         if(    is_numeric(trim(dummy(2)))   &
00722           .and.is_numeric(trim(dummy(3)))   &
00723           .and.is_numeric(trim(dummy(4)))   &
00724           .and.ntokens(line_of_file).ge.3 ) then
00725
00726           aux%name= trim(dummy(1))
00727           read( dummy(2),*) aux%lat
00728           read(dummy(3),*) aux%lon
00729           read(dummy(4),*) aux%height
00730
00731 !          ! todo
00732           if (aux%lat.ge.-90 .and. aux%lat.le.90) then
00733             if (aux%lon.ge.-180 .and. aux%lon.le.360) then
00734               good_lines=good_lines+1
00735               if (nloop.eq.2) then
00736                 sites(good_lines)%name= trim(dummy(1))
00737                 read(dummy(2),*) sites(good_lines)%lat
00738                 read(dummy(3),*) sites(good_lines)%lon
00739                 read(dummy(4),*) sites(good_lines)%height
00740               endif
00741             else
00742               if (nloop.eq.2) write ( fileunit_tmp, form_63) "rejecting (lon
      limits):" , line_of_file
00743             endif
00744           else
00745             if (nloop.eq.2) write ( fileunit_tmp, form_63) "rejecting (lat
      limits):" , line_of_file
00746           endif
00747
00748         else
00749           ! print it only once
00750           if (nloop.eq.2) then
00751               write ( fileunit_tmp, form_63) "rejecting (args):      " ,
     line_of_file
00752           endif
00753         endif
00754       enddo line_loop
00755       if (nloop.eq.1) rewind(fileunit_site)
00756     enddo nloops
00757
00758   ! if longitude <-180, 180> change to <0,360) domain
00759   do i =1 , size (sites)
00760     if (sites(i)%lon.lt.0) sites(i)%lon= sites(i)%lon + 360.
00761     if (sites(i)%lon.eq.360) sites(i)%lon= 0.
00762   enddo
00763 end subroutine
00764
00765
00766 ! ========================================================================
00767 !> Parse date given as 20110503020103  to yy mm dd hh mm ss and mjd
00768 !!
```

```
00769 !! \warning decimal seconds are not allowed
00770 ! ============================================================================
00771 subroutine parse_dates (cmd_line_entry )
00772   type(cmd_line) cmd_line_entry
00773   integer , dimension(6) :: start , stop
00774   real (sp) :: step =6. ! step in hours
00775   integer :: i
00776
00777   call string2date(cmd_line_entry%field(1), start)
00778   write (fileunit_tmp , form_62) "start date:" , start
00779   if (cmd_line_entry%field(2).eq."".or.cmd_line_entry%fields.le.1) then
00780     stop = start
00781   else
00782     call string2date(cmd_line_entry%field(2), stop )
00783     write (fileunit_tmp , form_62) "stop date: " , stop
00784   endif
00785   if (is_numeric(cmd_line_entry%field(3)).and.cmd_line_entry%fields
      .ge.3) then
00786     read(cmd_line_entry%field(3),*) step
00787     write (fileunit_tmp , form_62) "interval [h]:" , step
00788   endif
00789
00790   allocate (dates( int( ( mjd(stop) - mjd(start) ) / step * 24. + 1 ) ))
00791   do i = 1 , size(dates)
00792     dates(i)%mjd = mjd(start) + ( i -1 ) * step / 24.
00793     call invmjd( dates(i)%mjd , dates(i)%date)
00794   enddo
00795 end subroutine
00796
00797 subroutine string2date ( string , date )
00798   integer , dimension(6) ,intent(out):: date
00799   character (*) , intent(in) :: string
00800   integer :: start_char , end_char , j
00801
00802   ! this allow to specify !st Jan of year simple as -Dyyyy
00803   date = [2000 , 1 , 1 , 0 ,0 ,0]
00804
00805   start_char = 1
00806   do j = 1 , 6
00807     if (j.eq.1) then
00808       end_char=start_char+3
00809     else
00810       end_char=start_char+1
00811     endif
00812       if (is_numeric(string(start_char : end_char) )) then
00813       read(string(start_char : end_char),*) date(j)
00814     endif
00815     start_char=end_char+1
00816   enddo
00817
00818 end subroutine
00819
00820
00821 !subroutine sprawdzdate(mjd)
00822 !  real:: mjd
00823 !    if
      (mjd.gt.jd(data_uruchomienia(1),data_uruchomienia(2),data_uruchomienia(3),data_uruchomienia(4),data_uruchomienia(5),dat
00824 !      write (*,'(4x,a)') "Data późniejsza niż dzisiaj. KOŃCZĘ!"
00825 !      call exit
00826 !    elseif (mjd.lt.jd(1980,1,1,0,0,0)) then
00827 !      write (*,'(4x,a)') "Data wcześniejsza niż 1980-01-01. KOŃCZĘ!"
00828 !      call exit
00829 !    endif
00830 !    if (.not.log_E) then
00831 !      data_koniec=data_poczatek
00832 !      mjd_koniec=mjd_poczatek
00833 !    endif
00834 !    if (mjd_koniec.lt.mjd_poczatek) then
00835 !      write (*,*) "Data końcowa większa od początkowej. KOŃCZĘ!"
00836 !      write (*,form_64) "Data końcowa większa od początkowej. KOŃCZĘ!"
00837 !    endif
00838 !end subroutine
00839
00840 ! ============================================================================
00841 !> Auxiliary function
00842 !!
00843 !! check if argument given as string is valid number
00844 !! Taken from www
00845 !! \todo Add source name
00846 ! ============================================================================
00847 function is_numeric(string)
00848   implicit none
00849   character(len=*), intent(in) :: string
00850   logical :: is_numeric
00851   real :: x
00852   integer :: e
00853   read(string,*,iostat=e) x
```

```
00854   is_numeric = e == 0
00855 end function
00856
00857
00858 ! ================================================================================
00859 !> Check if file exists , return logical
00860 ! ================================================================================
00861 logical function file_exists(string)
00862   implicit none
00863   character(len=*), intent(in) :: string
00864   logical :: exists
00865   real :: x
00866   integer :: e
00867   if (string =="") then
00868     file_exists=.false.
00869     return
00870   endif
00871   inquire(file=string, exist=exists)
00872   file_exists=exists
00873 end function
00874
00875
00876 ! ================================================================================
00877 !> degree -> radian
00878 ! ================================================================================
00879 real(dp) function d2r (degree)
00880   real(dp) , intent (in) :: degree
00881   d2r= pi / 180.0 * degree
00882 end function
00883
00884 ! ================================================================================
00885 !> radian -> degree
00886 ! ================================================================================
00887 real(dp) function r2d ( radian )
00888   real(dp), intent (in) :: radian
00889   r2d= 180. / pi * radian
00890 end function
00891
00892 ! ================================================================================
00893 !> Print version of program depending on program calling
00894 ! ================================================================================
00895 subroutine print_version (program_calling)
00896   implicit none
00897   character(*) :: program_calling
00898   write(log%unit , form_header )
00899   if (program_calling.eq."grat" ) then
00900     write(log%unit,form_inheader ) , 'grat v. 1.0'
00901     write(log%unit,form_inheader ) , 'Last modification: 20120910'
00902   elseif(program_calling.eq."polygon_check") then
00903     write(log%unit,form_inheader ) , 'polygon_check v. 1.0'
00904     write(log%unit,form_inheader ) , 'Last modification: 20120910'
00905     write(log%unit,form_inheader ) , ''
00906     write(log%unit,form_inheader ) , 'Check if given point (given with -S)'
00907     write(log%unit,form_inheader ) ,
00908 'is included or excluded usig &      specific polygon file'
00909   elseif(program_calling.eq."value_check") then
00910     write(log%unit,form_inheader ) , 'value_check v. 1.0'
00911     write(log%unit,form_inheader ) , 'Last modification: 20120910'
00912     write(log%unit,form_inheader ) , ''
00913     write(log%unit,form_inheader ) , 'Check data value for given point (given
      with -S)'
00914   endif
00915   write(log%unit,form_inheader ) , ''
00916   write(log%unit,form_inheader ) , 'Marcin Rajner'
00917   write(log%unit,form_inheader ) , 'Politechnika Warszawska'
00918   write(log%unit,form_inheader ) , '(Warsaw University of Technology)'
00919   write(log%unit , form_header )
00920 end subroutine
00921
00922 ! ================================================================================
00923 !> Print settings
00924 ! ================================================================================
00925 subroutine print_settings ( program_calling )
00926   implicit none
00927   logical :: exists
00928   character (len=255):: dummy
00929   integer :: io_status , j
00930   character(*) :: program_calling
00931
00932   call print_version( program_calling = program_calling)
00933   call date_and_time( values = execution_date )
00934   write(log%unit,
00935 '("Program started:",1x,i4,2("-",i2.2), &
      1x,i2.2,2(":",i2.2),1x,"(",SP,i3.2,"h UTC)")'),          &
00936     execution_date(1:3),execution_date(5:7),execution_date(4)/60
00937   write(log%unit, form_separator)
00938
```

```
00939   inquire(fileunit_tmp, exist=exists)
00940   if (exists) then
00941     write (log%unit, form_60 ) 'Summary of command line arguments'
00942
00943     !-------------------------------------------------
00944     ! Cmd line summary (from scratch file)
00945     !-------------------------------------------------
00946     rewind(fileunit_tmp)
00947     do
00948       read(fileunit_tmp,'(a80)', iostat = io_status ) dummy
00949       if ( io_status == iostat_end)  exit
00950       write (log%unit, '(a80)') dummy
00951     enddo
00952
00953     !-------------------------------------------------
00954     ! Site summary
00955     !-------------------------------------------------
00956     write(log%unit, form_separator)
00957     write(log%unit, form_60 ) "Processing:", size(sites), "sites"
00958     write(log%unit, '(2x,a,t16,3a15)') "Name" , "lat [deg]" , "lon [deg]" ,"H
       [m]"
00959     do j = 1,size(sites)
00960       write(log%unit, '(2x,a,t16,3f15.4)') &
00961         sites(j)%name, sites(j)%lat, sites(j)%lon , sites(j)%height
00962       if (j.eq.10) exit
00963     enddo
00964     if (size(sites).gt.10) write(log%unit , form_62 ) &
00965       "and", size(sites)-10, "more"
00966
00967     !-------------------------------------------------
00968     ! Computation method summary
00969     !-------------------------------------------------
00970     if (program_calling.eq."grat" ) then
00971       write(log%unit, form_separator)
00972       write(log%unit, form_60 ) "Method used:", method
00973     endif
00974
00975     write(log%unit, form_separator)
00976     write(log%unit, form_60 ) "Interpolation data:", &
00977     interpolation_names(model%interpolation)(1:7)
00978
00979
00980
00981   endif
00982 end subroutine
00983
00984 subroutine print_help (program_calling)
00985   implicit none
00986   character(*) :: program_calling
00987   type help_fields
00988     character(2) :: switch
00989     character(255), allocatable,dimension(:) :: description
00990     character(255):: example=""
00991   end type
00992   ! todo change array size
00993   type(help_fields) help(9)
00994   integer :: i , j
00995
00996   help(1)%switch =      "-h"
00997   allocate(help(1)%description(1))
00998   help(1)%description(1) = "print help"
00999
01000   help(2)%switch =      "-v"
01001   allocate(help(2)%description(1))
01002   help(2)%description(1) = "print version and author"
01003
01004   help(3)%switch = "-S"
01005   allocate(help(3)%description(1))
01006   help(3)%description(1) = "set site(s) coordinates"
01007   help(3)%example = "-R0/20/30/40 or -Rg (=R0/360/-90/90) same as GMT"
01008
01009   help(4)%switch = "-L"
01010 !  allocate(help(4)%description(4))
01011 !  help(4)%description(1) = "prints additional information"
01012 !  help(4)%description(2) = "syntax: -L[filename]"
01013 !    help(4)%example = "-L[filename]"
01014 !  help(4)%example = "todo"//''//''//"fdf"
01015
01016   write(log%unit , form_60) , 'Summary of available options for program '//
       program_calling
01017   do i = 1 , size (help)
01018   if(if_switch_program(program_calling , help(i)%switch ))
       then
01019     write(log%unit , form_61) ,trim(help(i)%switch)
01020     if(allocated(help(i)%description)) then
01021       do j = 1 , size(help(i)%description)
01022         write (log%unit , form_62 ) trim(help(i)%description(j))
```

```
01023         if (.not.help(i)%example(1:1).eq."") then
01024 !          write(log%unit , form_63) , trim(help(i)%description(j)example)
01025         endif
01026       enddo
01027     endif
01028   endif
01029   enddo
01030
01031 end subroutine
01032
01033 subroutine print_warning (  warn , unit)
01034  implicit none
01035  character (len=*)  :: warn
01036  integer , optional :: unit
01037  integer :: def_unit
01038
01039  def_unit=fileunit_tmp
01040  if (present(unit) ) def_unit=unit
01041
01042  if (warn .eq. "site_file_format") then
01043   write(def_unit, form_63) "Some records were rejected"
01044   write(def_unit, form_63) "you should specify for each line at least 3[4]
      parameters in free format:"
01045   write(def_unit, form_63) "name lat lon [H=0] (skipped)"
01046 elseif(warn .eq. "boundaries") then
01047   write(def_unit, form_62) "something wrong with boundaries. IGNORED"
01048 elseif(warn .eq. "site") then
01049   write(def_unit, form_62) "something wrong with -S specification. IGNORED"
01050 elseif(warn .eq. "repeated") then
01051   write(def_unit, form_62) "reapeted specification. IGNORED"
01052 elseif(warn .eq. "dates") then
01053   write(def_unit, form_62) "something wrong with date format -D. IGNORED"
01054 endif
01055 end subroutine
01056
01057
01058 ! ================================================================================
01059 !> Counts number of properly specified models
01060 ! ================================================================================
01061 integer function nmodels (model)
01062   type(file) , allocatable, dimension (:) :: model
01063   integer :: i
01064
01065   nmodels = 0
01066
01067   do i = 1 , size (model)
01068     if (model(i)%if) nmodels =nmodels + 1
01069     if (model(i)%if_constant_value) nmodels =nmodels + 1
01070   enddo
01071 end function
01072
01073 end module get_cmd_line
```

7.9 /home/mrajner/src/grat/src/grat.f90 File Reference

Functions/Subroutines

- program **grat**

7.9.1 Detailed Description

Definition in file grat.f90.

7.10 grat.f90

```
00001 !
      ================================================================================
00002 !> \file
00003 !! \mainpage Grat overview
00004 !! \section Purpose
00005 !! This program was created to make computation of atmospheric gravity
00006 !! correction more easy.
00007 !!
00008 !! \version v. 1.0
```

```
00009 !! \date 2012-12-12
00010 !! \author Marcin Rajner\n
00011 !! Politechnika Warszawska\n
00012 !! (Warsaw University of Technology)
00013 !! \line program
00014 !!
00015 !! \warning This program is written in Fortran90 standard but uses some
       featerus
00016 !! of 2003 specification (e.g., \c 'newunit='). It was also written
00017 !! for <tt>Intel Fortran Compiler</tt> hence some commands can be unavailable
00018 !! for yours (e.g., \c <integer_parameter> for \c IO statements. This should be
00019 !! easily modifiable according to your output needs.>
00020 !! Also you need to have \c iso_fortran_env module available to guess the
       number
00021 !! of output_unit for your compiler.
00022 !! When you don't want a \c log_file and you don't switch \c verbose all
00023 !! unneceserry information whitch are normally collected goes to \c /dev/null
00024 !! file. This is *nix system default trash. For other system or file system
00025 !! organization, please change this value in \c get_cmd_line module.
00026 !!
00027 !! \section section
00028 !! \subpage todo
00029 !! \page ffff
00030 !
       ================================================================================
00031 program grat
00032   use iso_fortran_env
00033   use get_cmd_line
00034   use mod_polygon
00035   use mod_data
00036   use mod_green
00037
00038
00039   implicit none
00040   real(sp) :: x , y , z , lat ,lon ,val(0:100) !tmp variables
00041   integer :: i , j , ii, iii
00042
00043   !> program starts here with time stamp
00044   call cpu_time(cpu_start)
00045
00046   ! gather cmd line option decide where to put output
00047   call intro( program_calling = "grat" )
00048
00049   ! print header to log: version, date and summary of command line options
00050   call print_settings(program_calling = "grat")
00051
00052   ! read polygons
00053   do i =1 , 2
00054    call read_polygon(polygons(i))
00055   enddo
00056
00057   ! read models into memory
00058   do i =1 , size(model)
00059     if (model(i)%if) call read_netcdf( model(i) )
00060   enddo
00061
00062   ! todo refpres in get_cmd-line
00063 !  if (refpres%if) then
00064     refpres%name="/home/mrajner/src/grat/data/refpres/vienna_p0.grd"
00065     call read_netcdf(refpres)
00066 !  endif
00067
00068   allocate (results(size(sites)*max(size(dates),1)))
00069   iii=0
00070   do j = 1 , max(size (dates),1)
00071     if(size(dates).gt.0)  write(output%unit, '(i4,5(i2.2))', advance ="no")
      dates(j)%date
00072
00073     do ii = 1 , min(2,size(model))
00074       if (model(ii)%if) call get_variable( model(ii) , date = dates(j)%date)
00075     enddo
00076
00077
00078
00079 !todo
00080     do i = 1 , size(sites)
00081       write(output%unit, '(2f15.5f)', advance ="no") sites(i)%lat ,sites(i)%lon
00082       iii=iii+1
00083       call convolve(sites(i) , green , results(iii), denserdist = denser(1) ,
      denseraz = denser(2))
00084       write (output%unit,'(15f13.5)') , results(iii)%e ,results(iii)%n  ,
      results(iii)%dt , results(iii)%dh, results(iii)%dz
00085     enddo
00086   enddo
00087
00088 ! print '(15f13.5)',  results(maxloc (results%e))%e - results(minloc
       (results%e))%e       ,&
```

```
00089 !           results(maxloc (results%n))%n - results(minloc (results%n))%n
       ,&
00090 !           results(maxloc (results%dh))%dh - results(minloc (results%dh))%dh
       ,&
00091 !           results(maxloc (results%dz))%dz - results(minloc (results%dz))%dz
       ,&
00092 !           results(maxloc (results%dt))%dt - results(minloc (results%dt))%dt
00093
00094
00095   call cpu_time(cpu_finish)
00096   write(log%unit, '(/,"Execution time:",1x,f16.9," seconds")') cpu_finish -
      cpu_start
00097   write(log%unit, form_separator)
00098 ! hellow ro
00099   print * , model(6)%level
00100   print *
00101   lat =00
00102   lon = 00
00103   call get_value(model(7),lat,lon, val(0))
00104   do i =1, size(model(6)%level)
00105   call get_value(model(6),lat,lon, val(i), level = i, method=2)
00106   enddo
00107   print  '(30f10.2)', lat , lon , (val(i), i=0,size(model(6)%level))
00108   print  '(30f10.2)' , lat , lon , (geop2geom(val(i)/1000)*1000., i=0,
      size(model(6)%level))
00109
00110 end program
```

Appendix A

Polygon

This examples show how the exclusion of selected polygons works

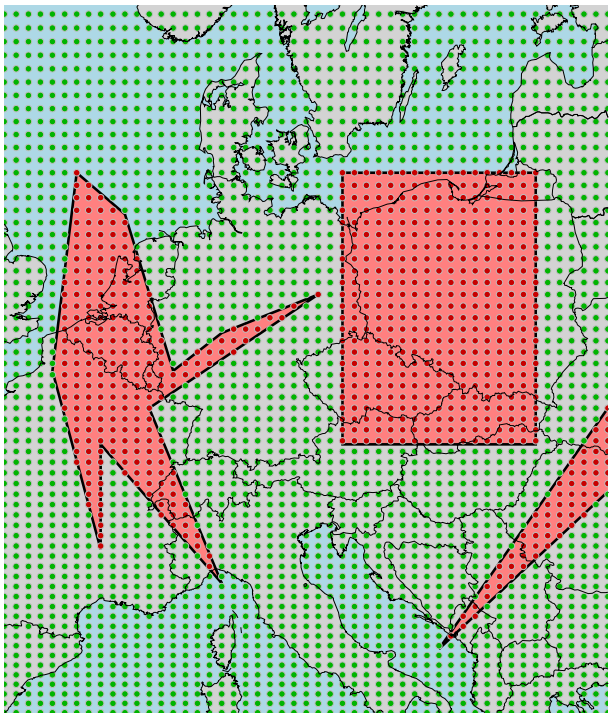


Figure A.1: If only excluded polygons (red area) are given all points falling in it will be excluded (red points) all other will be included

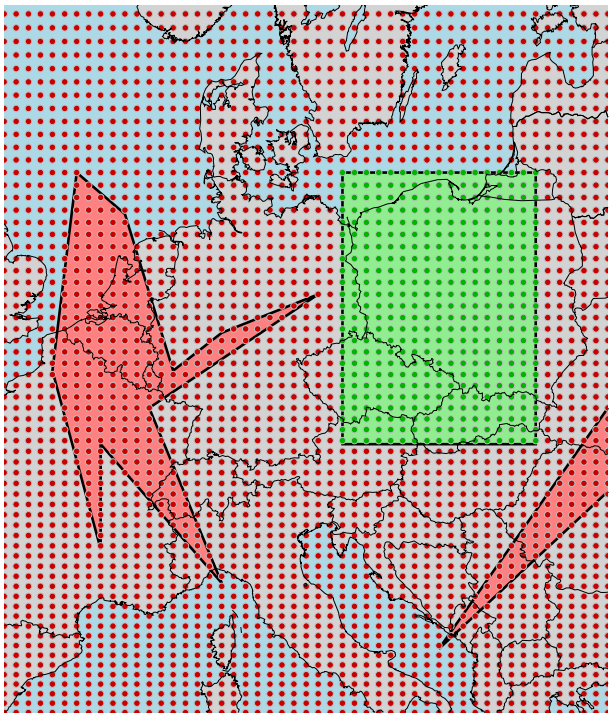


Figure A.2: If at least one included are are given (green area) than all points which not fall into included area will be excluded

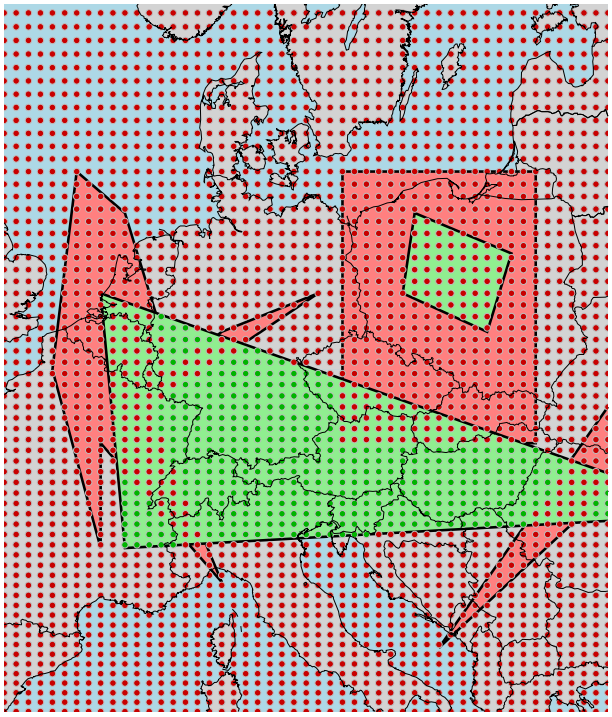


Figure A.3: If there is overlap of polygons the exclusion has higher priority

Appendix B

Interpolation

Figure B.1: Interpoloation

Bibliography

netcdf. URL https://www.unidata.ucar.edu/software/netcdf/.

D. C. Agnew. NLOADF: a program for computing ocean-tide loading. *J. Geophys. Res.*, 102:5109–5110, 1997.

COESA Comitee on extension of the Standard Atmosphere. U.S. Standard Atmosphere, 1976. Technical report, 1976.

S. B. Fels. Analytic Representations of Standard Atmosphere Temperature Profiles. *Journal of Atmospheric Sciences*, 43:219–222, January 1986. doi: 10.1175/1520-0469(1986)043<0219:AROSAT>2.0.CO;2.

Y. Huang, J. Guo, C. Huang, and X. Hu. Theoretical computation of atmospheric gravity green's functions. *Chinese Journal of Geophysics*, 48(6):1373–1380, 2005.

J. B. Merriam. Atmospheric pressure and gravity. *Geophysical Journal International*, 109(3):488–500, 1992. ISSN 1365-246X. doi: 10.1111/j.1365-246X.1992.tb00112.x. URL http://dx.doi.org/10.1111/j.1365-246X.1992.tb00112.x.

M. Rajner. *Wyznaczanie atmosferycznych poprawek grawimetrycznych na podstawie numerycznych modeli pogody*. PhD thesis, Politechnika Warszawska, Wydział Geodezji i Kartografii, 2013. URL http://www.geo.republika.pl/pub.

R. J. Warburton and J. M. Goodkind. The influence of barometeic-pressure variations on gravity. *Geophys. J. R. Astron. Society*, 48:281–292, 1977.

Index