User's Guide for VADER v. 1.0

Mark Krumholz

June 20, 2014

Contents

1 License and Citations

This is a guide for users of the `VADER` software package. `VADER` is distributed under the terms of the GNU General Public License v. 3. A copy of the license notification is included in the main `VADER` directory. If you use `VADER` in any published work, please cite the paper Krumholz, M. R., & Forbes, J. C., 2014, submitted to *Astronomy & Computing.*

2 What Does `VADER` Do?

A full description of the model system that `VADER` simulates can be found in Krumholz, M. R., & Forbes, J. C., 2014, submitted to *Astronomy & Computing.* A short summary is that `VADER` solves the equations of mass, angular momentum, and energy conservation for a thin, axisymmetric, viscous disk. These equations are

$$\frac{\partial}{\partial t}\Sigma + \frac{1}{r}\frac{\partial}{\partial r}\left(r v_r \Sigma\right) = \dot{\Sigma}_{\mathrm{src}} \tag{1}$$

$$\frac{\partial}{\partial t}E + \frac{1}{r}\frac{\partial}{\partial r}\left[r v_r \left(E+P\right)\right] - \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{v_\phi \mathcal{T}}{2\pi r^2}\right) = \dot{E}_{\mathrm{src}}. \tag{2}$$

where v_ϕ is the orbital velocity in the disk, Σ, E, and P are the vertically-integrated mass per unit area, total energy per unit area (internal plus orbital plus potential), and pressure, $\mathcal{T}$ is the torque between adjacent rings of material, and $\dot{\Sigma}_{\mathrm{src}}$ and $\dot{E}_{\mathrm{src}}$ are the source terms for mass and total energy. The equations are written in a form where angular momentum conservation has already been incorporated, so a separate conservation equation is not needed for it. The radial velocity v_r appear in these equations is related to the torque by

$$v_r = \frac{1}{2\pi r \Sigma v_\phi (1+\beta)}\frac{\partial}{\partial r}\mathcal{T}, \tag{3}$$

where $\beta = \partial \ln v_\phi / \partial \ln r$ is the index of the rotation curve, and the torque is non-dimensionalized via the standard α parameterization as

$$\mathcal{T} = -2\pi r^2 \alpha (1-\beta) P. \tag{4}$$

Finally, the system obeys a general equation of state whereby the vertically integrated pressure is a function of the position, column density, and total internal energy,

$$P = P(r, \Sigma, E_{\mathrm{int}}) \tag{5}$$

where the internal energy is defined from the total energy and rotation curve by

$$E = \Sigma\left(\frac{v_\phi^2}{2} + \psi\right) + E_{\mathrm{int}}, \tag{6}$$

and ψ is the gravitational potential. The equation of state is characterize by two partial derivatives,

$$\gamma = 1 + \left.\frac{\partial P}{\partial E_{\mathrm{int}}}\right|_{r,\Sigma} \tag{7}$$

$$\delta = \frac{1}{\gamma - 1} \left.\frac{\partial \ln P}{\partial \ln \Sigma}\right|_{r,E_{\mathrm{int}}}. \tag{8}$$

`VADER` solves this system of equations on an arbitrarily-arranged grid of radial cells using a second-order accurate conservative finite volume method with implicit time discretization.

3 Installing and Configuring `VADER`

3.1 Getting `VADER`

The easiest way to get `VADER` is from the public bitbucket repository. On any system with git installed, `VADER` may be downloaded via the command

```
git clone https://krumholz@bitbucket.org/krumholz/vader.git
```

The entire repository may also be downloaded as a zip file from `https://bitbucket.org/krumholz/vader/downloads`.

3.2 Dependencies and Build Requirements

`VADER` has the following dependencies:

- The core c routines require the GNU scientific library (GSL) version $\geq$ 1.12.
- The Python wrapper routines require Python $\geq$ 2.6 (3.x also ok)
- The Python wrapper routines require numPy $\geq$ 1.6.1
- The Python wrapper routines require sciPy $\geq$ 0.7.0

`VADER`'s builtin `Makefile` expects that the GSL headers will be included in `C_INCLUDE_PATH`, and the object files included in `LD_LIBRARY_PATH`. If this is not the case, they can be added to the build path manually by editing the file `vader_csrc/Makefile`. The `VADER` build system also expects the environment variables `CC` and `MAKE` to be set, or to default to reasonable values.

4 Structure of a VADER Simulation

A VADER simulation is generally composed of several elements. Not all of these are strictly mandatory, but they will typically be found in most simulations.

1. **Initialize the Grid.** Almost all VADER functions require a `struct grid` object that defines the computational grid, the rotation curve that goes with it, and a number of ancillary quantities. In the c interface, grids can be allocated using the `gridAlloc` function, and can be both allocated and initialized using the `gridInit`, `gridInitTabulated`, `gridInitFlat`, and `gridInitKeplerian`.

2. **Allocate a Workspace.** Almost all VADER routines require a `struct wksp` object that provides a workspace for computations. This is allocated by calling the `wkspAlloc` function in c.

3. **Initialize the Data.** The user must declare and initialize arrays to contain the initial column density, vertically-integrated pressure, and, if using a complex equation of state, internal energy. These will be passed to the main simulation routine.

4. **Initialize the Parameters.** If the simulations require any additional parameters to be passed to the user-specific routines (see Section 5.1), these must be stored in a data structure that can be passed by reference.

5. **Allocate Arrays to Store Outputs.** A VADER simulation generally returns a series of outputs, representing snapshots of the state of the computation at a variety of times. The user must allocate memory to store these results.

6. **Run the Simulation.** The simulation is run via a call to the `driver` routine. This call runs the simulation and stores the output.

7. **Free Memory**. In a c simulation, the final step is to free the grid and workspace via calls to `gridFree` and `wkspFree`.

Example c programs can be found in the `vader_csrc/prob` directory; all files in that directory with names of the form `main_PROB.c` are examples. Several of these steps can be omitted for simulations using the Python interface (Section 7), which handles most of the allocation, de-allocation, and processing of data automatically.

5 Setting Up a New Problem

Most users of VADER will likely want to set up their own simulations, defined by their own choices of viscosity, equation of state, boundary conditions, and heating and cooling rates. Routines specific to particular problems are stored in the `vader_csrc/prob` subdirectory. To define a new problem, a user must create one mandatory file, and one optional file, in this directory.

5.1 The `userFunc` File

The mandatory file must have a name of the form `userFunc_MYPROB.c`, where `MYPROB` is the name the user wishes to assign to that problem. This file must define the following functions, all of which take the current state of the simulation as inputs:

- `userAlpha`: this defines the dimensionless viscosity α for the problem
- `userEOS`: this defines the equation of state parameters γ and δ
- `userMassSrc`: this defines the column density source function $\dot{\Sigma}_{\mathrm{src}}$
- `userIntEnSrc`: this defines the internal energy source function $\dot{E}_{\mathrm{int,src}}$
- `userIBC`: this defines the pressure and enthalpy values at the inner boundary condition
- `userOBC`: this defines the pressure and enthalpy values at the outer boundary condition

The full APIs for these functions are given in Section 9.2, and may also be found in the source file `vader_csrc/userFunc.h`. The easiest way to create a custom `userFunc_MYPROB.c` file is probably to copy the existing file `vader_csrc/prob/userFunc_none.c`, which defines all the required functions as no-ops that simply print a warning message and then return.

While all these functions must be defined in `userFunc_MYPROB.c`, their actual implementations can be no-ops, as is the case in `userFunc_none.c`. The functions are actually called only if the appropriate flags are set in calls to the `driver`, `advanceBE`, or `advanceCN` routines. For example, if `driver` is called but the input parameter `eos_func` is set to 0, then `userEOS` will never be called, and can safely be left as a no-op function.

All of the `user*` routines accept an argument `void *params`, as do the `driver`, `advanceBE`, and `advanceCN` routines from which they are called. The `params` variable can be used to pass arbitrary data into the `user*` routines, thereby allowing these functions to depend on additional parameters defined at runtime. The user must simply construct a structure containing whatever parameters he or she requires, obtain a pointer to that structure, cast it to void, pass it into `driver`, `advanceBE`, or `advanceCN`, and then cast the `void *params` value in the `user*` routine back to whatever data type it originally was.

5.2 The `main` File

The optional file has a name of the form `main_MYPROB.c`, where `MYPROB` is the same as the name used for `userFunc_MYPROB.c`. This file contains the `main` function for the program. It is required if the code is compiled in Executable Mode or Debug Mode, but is ignored if the code is compiled in Dynamically-Linked Library Mode.

The `main` routine generally follows the simulation structure described in Section 4. Example `main_MYPROB.c` files can be found in the `vader_csrc/prob` subdirectory.

6 Compiling VADER

VADER can be compiled either as a standalone executable, or as a dynamically-linked library. In the latter mode, it can be called from the python interface described in Section 7.

6.1 Executable Mode

To compile VADER in executable mode, from the main vader directory type

```
make exec PROB=MYPROB
```

where MYPROB is the name of the problem to be compiled. This command causes the files vader_csrc/userFunc_MYPROB.c and vader_csrc/main_MYPROB.c to be copied into the main vader_csrc directory and compiled along with the core VADER routines. Upon successful compilation, an executable vader.ex is placed in the bin subdirectory. The code is compiled without debugging symbols, and with a high level of optimization.

6.2 Debug Mode

To compile VADER in debug mode, from the main vader directory type

```
make debug PROB=MYPROB
```

where MYPROB is the name of the problem to be compiled. The results are identical to those for compilation in Executable Mode, except that debugging symbols are enabled and optimization is disabled.

6.3 Dynamically-Linked Library Mode

To compile VADER in dynamically-linked library mode, from the main vader directory type

```
make lib PROB=MYPROB
```

where MYPROB is the name of the problem to be compiled. This causes the file vader_csrc/userFunc_MYPROB.c, but *not* vader_csrc/main_MYPROB.c, to be copied into the main vader_csrc directory and compiled along with the core VADER routines. The code is compiled into a dynamically-linked library, libvader.x, where x is the extension appropriate for a dynamically-linked library on the system where the code is being compiled. The library is copied into the bin subdirectory. The code is compiled at a high level of optimization. If PROB is not specified, the default problem is none, which causes all the user* routines to be no-ops that just print a warning and then return. When compiled in this mode, the code can still be run on problems where no user-specific functions are used, and all quantities (viscosity, equation of state, inner and outer boundary conditions, source functions) are defined by constants.

6.4 Testing Mode

Testing mode is an additional option that can be combined with executable, debug, or dynamically-linked library mode. To compile in testing mode, do add the option `MODE=TEST` to any of the compilation commands above, for example

```
make lib PROB=MYPROB MODE=TEST.
```

This causes the `advanceBE`, `advanceCN`, and `driver` routines to return extra diagnostic information.

7 The Python Interface

When compiled in dynamically-linked library mode, `VADER` can be accessed through a python interface, which provides a convenient method of parsing parameter files, constructing grids, running simulations, and processing the results. The Python interface to `VADER` is implemented as a Python package. From the main `vader` directory, the package may be imported via the usual Python import procedure:

```
import vader
```

7.1 Structure of a Python-Based `VADER` Simulation

The Python interface to `VADER` provides several convenient routines to make it easier to set up and run simulations. The `vader.readParam` function can read and parse parameter files formatted as described in Section 7.2, and return a dict of parameters. This can then be passed to the `vader.grid` routine to construct a `class grid` object that contains al the grid information required to a run a simulation. Finally, once the grid has been constructed and initial conditions specified, a full simulation can be run through the `vader.driver` routine. This routine processes the inputs into a format suitable for passing to c, calls the c `driver` routine, processes the results into convenient form, and returns them. The usual structure of a Python-based `VADER` simulation is:

1. **Import the `VADER` package.**
2. **Read the parameter file.** This can be accomplished by

   ```
   paramDict = vader.readParam(fileName)
   ```

3. **Construct a grid.** This can be accomplished by

   ```
   grd = vader.grid(paramDict)
   ```

4. **Initialize the Data.** Construct `numpy` arrays for the initial column density, pressure, and, if using a complex equation of state, internal energy.

5. **Initialize Parameters for Passing to c.** If using custom `user*` routines that require additional parameters, construct a pointer to them suitable for passing to c. This can be done using the standard Python `ctypes` interface. In the most common case where the parameters consist of a series of real numbers, these can be stored in a `numpy` array, and then a pointer to them can be obtained via

   ```
   paramPtr = ctypes.byref(paramArr.ctypes.
                           data_as(ctypes.POINTER(ctypes.c_double)).
                           contents)
   ```

   where `paramArr` is the `numpy` array containing the parameter values.

6. **Run the simulation.** This can be accomplished by

   ```
   result = vader.driver(tState, tEnd, grd, col, pres, paramDict,
                         nOut=nOut, c_params=paramPtr)
   ```

   The `vader.driver` routine returns a `namedtuple` containing the results of the simulation.

Full APIs for all Python functions are given in Section 9.3.

Important warning: in python mode, the `vader` python package creates an interface to the `VADER` library compiled in dynamically-linked library mode using the functionality provided by `numpy.ctypeslib.load_library`. At present, `numpy.ctypes` does not provide any way to "unload" or "reload" a library. Once a library has been loaded into a particular instance of the python kernel, there is no way to modify it even if the underlying library is recompiled. This means that, if you alter and recompile `libvader`, you will have to start a new python kernel for those changes to take effect. There is at present no way to rebuild the library and then create a new interface to the modified library without restarting the python kernel.

7.2 Parameter Files

The `vader.readParam` function can parse files that describe the setup of `VADER` simulations. The format of the files is a series of key-value pairs formatted as

```
key = value
```

One such pair can appear per line. Blank lines, or lines for which the first non-whitespace character is `#`, are treated as comments and ignored. Known keys are listed in Table 1. Additional keys not in the table can be added freely, and will be ignored by `VADER`, allowing users to add their own parameters as they like.

Most keys map directly onto the parameters of the same name for the `driver` and/or `advanceBE` and `advanceCN` routines, and control the operations of those routines as described in the c APIs below. Some keys allow the special value `c_func`. Setting this value indicates that the quantity in question is not a constant, but instead should be determined by calling the appropriate `user*` routine that was compiled with the c code. This provides a mechanism for accessing problem-specific non-constant values of viscosity, equation of state, source functions, and boundary conditions. At an implementation level, if one of the keys that allow it is set to `c_func`, then the corresponding `*_func` parameter passed to the c function will be set to 1; otherwise it will be set to 0, and the corresponding `*_val` parameter will be set to the numerical value specified by the key.

8 The `VADER` Test Suite

`VADER` provides a number of tests of code accuracy and performance. These correspond to the tests published in the `VADER` method paper, Krumholz & Forbes, 2014, submitted to *Astronomy & Computing*. The tests are all implemented through the Python interface, and the scripts to run them are in the `test` subdirectory of the main `vader` directory. The test scripts provided are:

- `selfsim.py`: this tests the code against the analytic solution for evolution of a self-similar disk from Lynden-Bell & Pringle (1974, MNRAS, 168, 603)
- `selfsim_resolution.py`: this tests the code against the analytic solution for evolution of a self-similar disk from Lynden-Bell & Pringle (1974, MNRAS, 168, 603), and performs a convergence study by running the simulation at a variety of resolutions
- `ring.py`: this tests the code against the analytic solution for evolution of an initially-singular ring from Pringle (1981, ARA&A, 19, 137)
- `gidisk.py`: this tests the code against the analytic solution for a gravitational instability-dominated disk from Krumholz & Burkert (2010, ApJ, 724, 895), as extended by Forbes, Krumholz, & Burkert (2012, ApJ, 754, 48)
- `ringrad.py`: this tests the code by running the singular ring problem using a complex equation of state that includes both gas pressure- and radiation pressure-dominated regimes. No analytic solution is known for this case, but a benchmark is provided in the method paper against which future runs can be tested
- `performance.py`: this runs the code through a series of performance tests, using the self-similar disk and radiation pressure ring test problems; warning: this test takes a while to perform, and involves rebuilding `VADER` a number of times to test the performance of various code options that are set at compile- rather than run-time

All of these tests can be executed by doing

```
python TEST_NAME.py
```

Key	**Meaning**	**Allowed values**	**Default**
`nr`	Number of cells in the grid	Any `float`	None
`rmin`	Inner edge of grid	Any `float`	None
`rmax`	Outer edge of grid	Any `float`	None
`rot_curve_type`	Type of rotation curve	`flat`, `keplerian`, or `tabulated`	None
`rot_curve_velocity`	Rotation speed for `flat` rotation	Any `float`	None
`rot_curve_mass`	Mass of central object for `keplerian` rotation	Any `float`	None
`rot_curve_file`	Name of a text file containing a table of r v_ϕ values for `tabulated` rotation	Any `str`	None
`alpha`	Dimensionless viscosity α	Any `float` or `c_func`	None
`gamma`	Equation of state γ	Any `float` or `c_func`	None
`delta`	Equation of state δ	Any `float` or `c_func`	0
`ibc_pres_type`	Type of pressure boundary condition at grid inner edge	`fixed_mass_flux`, `fixed_torque_flux`, or `fixed_torque`	None
`ibc_pres_val`	Value for inner pressure boundary condition	Any `float` or `c_func`	None
`ibc_enth_type`	Type of enthalpy boundary condition at grid inner edge	`fixed_value`, `fixed_gradient`	None
`ibc_enth_val`	Value for inner enthalpy boundary condition	Any `float` or `c_func`	None
`obc_pres_type`	Type of pressure boundary condition at grid outer edge	`fixed_mass_flux`, `fixed_torque_flux`, or `fixed_torque`	None
`obc_pres_val`	Value for outer pressure boundary condition	Any `float` or `c_func`	None
`obc_enth_type`	Type of enthalpy boundary condition at grid outer edge	`fixed_value`, `fixed_gradient`	None
`obc_enth_val`	Value for outer enthalpy boundary condition	Any `float` or `c_func`	None
`mass_src`	Mass source function	Any `float` or `c_func`	0.0
`int_en_src`	Internal energy source function	Any `float` or `c_func`	0.0
`err_tol`	Implicit solver tolerance	Any `float`	10^{-6}
`dt_tol`	Maximum change per time step	Any `float`	0.1
`dt_start`	Starting time step	Any `float`	-1
`max_iter`	Maximum implicit iterations	Any positive `int`	40
`interp_order`	Enthalpy interpolation order	1, 2, or 3	2
`max_dt_increase`	Maximum time step increase	Any `float`	1.5
`dt_min`	Minimum time step	Any `float`	10^{-15}
`max_step`	Maximum time steps	Any `int`	-1
`method`	Advance method	`CN` or `BE`	`CN`
`verbosity`	Level of verbosity	0, 1, 2, or 3	0
`bspline_degree`	Degree of B-spline used for `tabulated` rotation curves	Any `int` > 1	6
`bspline_breakpoints`	Number of breakpoints in B-spline fit for `tabulated` rotation curves	Any `int` > 1	15

Table 1: Pre-defined keys in `VADER` parameter files. Default values indicate the values used in calls to the c routines if the corresponding key is absent.

from the main `vader` directory. Outputs from tests are placed in the `output` subdirectory of the main `vader` directory. Outputs consist of plots that can be compared to the published benchmarks in the methods paper.

9 Full Description of All Routines and Data Structures

A general note on naming conventions of arrays. All arrays with `_g` at the end of their names are arrays of `grd->nr+2` elements, of which the central `grd->nr` represent values at the centers of the computational grid, and elements `0` and `grd->nr+1` are ghost cells at the inner and outer boundary, respectively. Arrays with `_h` at the end of their names are arrays of `grd->nr+1` elements representing values at the edges of cells. Arrays with no subscript at the end of their names have `grd->nr` elements, and represent values at cell centers with no ghost zones.

9.1 Core c Routines

All c routines are in the `vader_csrc` directory.

9.1.1 c Data Structures

Data structures used throughout the code are defined in `vader.h`. These are:

```
/* Descriptor for a grid */
typedef struct {
  unsigned int nr;             /* Number of real cells */
  unsigned int linear;         /* Is this a linear or logarithmic grid? */
  double *r_g, *r_h;           /* Cell center, edge locations */
  double *dr_g;                /* Cell sizes / log sizes */
  double *area;                /* Area of a zone */
  double *vphi_g, *vphi_h;     /* Rotation curve */
  double *beta_g, *beta_h;     /* Logarithmic index of rotation curve */
  double *psiEff_g, *psiEff_h; /* Effective gravitational potential */
  double *g_h;                 /* Factor appearing in derivatives */
} grid;
```

and

```
/* Workspace for calculations */
typedef struct {
  double *pres_g, *presNew_g, *colNew, *colTmp;
  double *alpha_g, *hint_g, *hintL_g, *hintR_g;
  double *ppmwksp_g;
  double *fmLast_h, *fmNew_h;
  double *ftLast_h, *feLast_h;
  double *massSrcLast, *massSrcNew, *intEnSrc;
  double *eIntTmp, *eIntNew;
  double *gammaLast, *deltaLast, *gammaNew, *deltaNew;
  double *mSrc, *eSrc;
  gsl_vector *ud_g, *ld_g, *diag_g, *rhs_g, *presTmp_g;
#if AA_M > 0
  double *colHist, *presHist, *eIntHist;
  gsl_matrix *resid, *cov;
  gsl_vector *constraint, *aa_wgts;
#endif
} wksp;
```

9.1.2 advanceBE

Defined in advanceBE.h and advanceBE.c.

```
double
advanceBE(
          /* Time step and grid */
          const double t, const double dt, const grid *grd,
          /* Starting data */
          double *col, double *pres, double *eInt,
          /* Records of mass and energy transported off grid and added
             by sources */
          double *mBnd, double *eBnd, double *mSrc, double *eSrc,
          /* Equation of state parameters */
          const int eos_func, const double gamma_val,
          const double delta_val,
          /* Dimensionless viscosity parameters */
          const int alpha_func, const double alpha_val,
          /* Inner boundary condition parameters */
          const pres_bc_type pres_ibc, const enth_bc_type enth_ibc,
          const int ibc_func, const double ibc_pres_val,
          const double ibc_enth_val,
          /* Outer boundary condition parameters */
          const pres_bc_type pres_obc, const enth_bc_type enth_obc,
          const int obc_func, const double obc_pres_val,
          const double obc_enth_val,
          /* Source function parameters */
          const int massSrc_func, const double massSrc_val,
          const int intEnSrc_func, const double intEnSrc_val,
          /* Control and method parameters */
          const double errTol, const double dtTol,
          const unsigned int maxIter, const unsigned int interpOrder,
          const unsigned int noUpdate, const unsigned int verbose,
          const wksp *w,
          /* User-defined extra parameters */
          void *params,
          /* Diagnostic outputs */
          unsigned int *itCount
#ifdef TESTING_MODE
          , double *resid, unsigned int *rtype,
          double *advanceTime, double *nextIterTime,
          double *userTime
#endif
          );
```

Parameters:

- `t`: (INPUT) time at start of time step
- `dt`: (INPUT) size of time step
- `grd`: (INPUT) pointer to a `struct grid` that describes the simulation grid
- `col`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the column density at the start of the time step; at the end of the routine, it is updated to the new column density values
- `pres`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the vertically-integrated pressure at the start of the time step; at the end of the routine, it is updated to the new column pressure values
- `eInt`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the internal energy per unit area at the start of the time step; at the end of the routine, it is updated to the new internal energy values. If `eos_func` is set to 0, this will not be referenced, and can be set to `NULL`.
- `mBnd`: (OUTPUT) pointer to a two-element array; the mass advected across the inner and outer boundaries of the simulation domain during this time step will be added to these two elements, using a sign convention whereby transport in the $+r$ direction is positive
- `eBnd`: (OUTPUT) pointer to a two-element array; the total energy advected across the inner and outer boundaries of the simulation domain during this time step will be added to these two elements, using a sign convention whereby transport in the $+r$ direction is positive

- `mSrc`: (OUTPUT) pointer to an array of `grd->nr` elements; the total mass per unit area added to/subtracted form each cell by the source terms is added to it. If `massSrc_func` is 0, this array is not referenced, and may be set to `NULL`.
- `eSrc`: (OUTPUT) pointer to an array of `grd->nr` elements; the total energy per unit area added to/subtracted form each cell by the source terms is added to it. If `massSrc_func` and `intEnSrc_func` are both 0, this array is not referenced, and may be set to `NULL`.
- `eos_func`: (INPUT) a non-zero value indicates that `userEOS` should be called to set the equation of state parameters
- `gamma_val`: (INPUT) constant value of γ; ignored if `eos_func` is non-zero
- `delta_val`: (INPUT) constant value of δ; ignored if `eos_func` is non-zero
- `alpha_func`: (INPUT) a non-zero value indicates that `userAlpha` should be called to set the viscosity
- `alpha_val`: (INPUT) constant value of α; ignored if `alpha_func` is non-zero
- `pres_ibc`: (INPUT) type of inner pressure boundary condition; allowed values are `FIXED_MASS_FLUX`, `FIXED_TORQUE_FLUX`, and `FIXED_TORQUE`
- `enth_ibc`: (INPUT) type of inner enthalpy boundary condition; allowed values are `FIXED_ENTHALPY_VALUE` and `FIXED_ENTHALPY_GRADIENT`
- `ibc_func`: (INPUT) a non-zero value indicates that `userIBC` should be called to set the inner boundary condition
- `ibc_pres_val`: (INPUT) constant value for the mass flux, torque flux, or torque (depending on the value of `pres_ibc`) used with the inner pressure boundary condition; ignored if `ibc_func` is non-zero
- `ibc_enth_val`: (INPUT) constant value for the enthalpy value or enthalpy gradient (depending on the value of `enth_ibc`) use with the inner enthalpy boundary condition; ignored if `ibc_func` is non-zero
- `pres_obc`: (INPUT) identical to `pres_ibc`, but for the outer boundary
- `enth_obc`: (INPUT) identical to `enth_ibc`, but for the outer boundary
- `obc_func`: (INPUT) identical to `ibc_func`, but for the outer boundary
- `obc_pres_val`: (INPUT) identical to `ibc_pres_val`, but for the outer boundary
- `obc_enth_val`: (INPUT) identical to `ibc_enth_val`, but for the outer boundary

- `massSrc_func`: (INPUT) a non-zero value indicates that `userMassSrc` should be called to set the mass source function
- `massSrc_val`: (INPUT) constant value of the mass source function; ignored if `massSrc_func` is non-zero
- `intEnSrc_func`: (INPUT) a non-zero value indicates that `userIntEnSrc` should be called to set the mass source function
- `intEnSrc_val`: (INPUT) constant value of the mass source function; ignored if `intEnSrc_func` is non-zero
- `errTol`: (INPUT) maximum error in the iterative solver; used to test for convergence
- `dtTol`: (INPUT) maximum fractional change in column density / pressure / internal energy in fastest-varying cell; used to estimate the next time step
- `maxIter`: (INPUT) maximum number of iterations allowed in the implicit solver
- `interpOrder`: (INPUT) order of interpolation to use when reconstructing specific internal enthalpy at cell faces; 1 = piecewise constant, 2 = piecewise linear, 3 = piecewise parabolic
- `noUpdate`: (INPUT) if this is non-zero, then `col`, `pres`, `eInt`, `mBnd`, `eBnd`, `mSrc`, and `eSrc` are not altered, but the time step is still computed and returned
- `verbose`: (INPUT) if non-zero, the routine will print the residual and some related information after ever iteration
- `w`: (INPUT/OUTPUT) pointer to a `struct workspace` object
- `params`: (INPUT) pointer to memory containing user-defined parameters; this memory is not touched by this routine, but the pointer is passed to all `user*` routines
- `itCount`: (OUTPUT) number of iterations performed by the implicit solver
- `resid`: (OUTPUT, only in `TESTING_MODE`): pointer to an array of `maxIter` elements; the residual at the end of each iteration is returned in it
- `rtype`: (OUTPUT, only in `TESTING_MODE`): pointer to an array of `maxIter` elements; the type of residual (column density, pressure, or internal energy) at the end of each iteration is returned in it
- `advanceTime`: (OUTPUT, only in `TESTING_MODE`): pointer to a value measuring the time spent in the advance routine, in seconds; this is incremented by the time spent in the routine during this call

- `nextIterTime`: (OUTPUT, only in `TESTING_MODE`): pointer to a value measuring the time spent getting the next iterate (via Anderson acceleration, if `AA_M > 0`), in seconds; this is incremented by the time spent in the routine during this call
- `userTime`: (OUTPUT, only in `TESTING_MODE`): pointer to a value measuring the time spent in the `user*` routines, in seconds; this is incremented by the time spent in the routine during this call
- Return value: estimate for the size of the next time step; if convergence fails, the return value is -1.0

The `advanceBE` routine advances the simulation through a single time step using backwards Euler differencing. The arrays `col`, `pres`, and `eInt` specify the state at the start of the time step, and upon completion of a successful time step they are updated to the new values. The arrays `mBnd`, `eBnd`, `mSrc`, and `eSrc` are incremented by the values of mass and energy transported across the grid boundaries and added by the source terms during the time step. The return value is the estimated size of the next time step. If the iterative solver does not converge within `maxIter` iterations, or if the iteration procedure produces `Inf` or `NaN` values before the maximum number of iterations is reached, the code returns without updating any quantities except `itCount`, and the return value is set to -1.0.

9.1.3 `advanceCN`

Defined in `advanceCN.h` and `advanceCN.c`.

The API for this routine is identical to that for `advanceBE`. It differs only in that this routine uses Crank-Nicolson rather than backwards Euler time centering.

9.1.4 `applyBC`

Defined in `applyBC.h` and `applyBC.c`.

```
void
applyBC(
        /* Grid structure and viscosity values on grid */
        const grid *grd, const double *alpha_g,
        /* Inner boundary condition specifications */
        const pres_bc_type ibc_pres, const enth_bc_type ibc_enth,
        const double ibc_pres_val, const double ibc_enth_val,
        /* Outer boundary condition specifications */
        const pres_bc_type obc_pres, const enth_bc_type obc_enth,
        const double obc_pres_val, const double obc_enth_val,
        /* Outputs */
        double *pres_g, double *hint_g, double *pIn, double *qIn,
        double *pOut, double *qOut
        );
```

Parameters:

- `grd`: (INPUT) pointer to a `struct grid` that describes the simulation grid
- `alpha_g`: (INPUT) pointer to an array of `grd->nr+2` values of the viscosity α.
- `ibc_pres`: (INPUT) type of inner pressure boundary condition; allowed values are `FIXED_MASS_FLUX`, `FIXED_TORQUE_FLUX`, and `FIXED_TORQUE`
- `ibc_enth`: (INPUT) type of inner enthalpy boundary condition; allowed values are `FIXED_ENTHALPY_VALUE` and `FIXED_ENTHALPY_GRADIENT`
- `ibc_pres_val`: (INPUT) value for the mass flux, torque flux, or torque (depending on the value of `ibc_pres`) used with the inner pressure boundary condition
- `ibc_enth_val`: (INPUT) value for the enthalpy value or enthalpy gradient (depending on the value of `ibc_enth`) use with the inner enthalpy boundary condition
- `obc_pres`: (INPUT) identical to `ibc_pres`, but for the outer boundary
- `obc_enth`: (INPUT) identical to `ibc_enth`, but for the outer boundary
- `obc_pres_val`: (INPUT) identical to `ibc_pres_val`, but for the outer boundary
- `obc_enth_val`: (INPUT) identical to `ibc_enth_val`, but for the outer boundary
- `pres_g`: (INPUT/OUTPUT) pointer to an array of `grd->nr+2` values of the vertically-integrated pressure; on entry, values in elements $1 \cdots$ `grd->nr` are set to the values on the grid, and on return elements 0 and `grd->nr+1` are set to the values required to enforce the specified boundary condition
- `hint_g`: (INPUT/OUTPUT) pointer to an array of `grd->nr+2` values of the specific enthalpy; on entry, values in elements $1 \cdots$ `grd->nr` are set to the values on the grid, and on return elements 0 and `grd->nr+1` are set to the values required to enforce the specified boundary condition
- `pIn`: (OUTPUT) the inner ghost zone boundary value is set to `pres_g[0] = *qIn * pres_g[1] + *pIn`; this variable returns the value of `pIn`
- `qIn`: (OUTPUT) the inner ghost zone boundary value is set to `pres_g[0] = *qIn * pres_g[1] + *pIn`: this variable returns the value of `qIn`
- `pOut`: (OUTPUT) same as `pIn`, but for the outer boundary condition
- `qOut`: (OUTPUT) same as `qIn`, but for the outer boundary condition

This routine fills the ghost cells of the arrays `pres_g` and `hint_g` based on the input set of pressure and enthalpy boundary condition types and values.

9.1.5 `driver`

Defined in `driver.h` and `driver.c`.

```
double
driver(
       /* Time parameters */
       const double tStart, const double tEnd,
       const double dtStart,
       /* Computational grid */
       const grid *grd,
       /* Input data */
       double *col, double *pres, double *eInt,
       /* Equation of state parameters */
       const int eos_func, const double gamma_val,
       const double delta_val,
       /* Dimensionless viscosity parameters */
       const int alpha_func, const double alpha_val,
       /* Inner boundary condition parameters */
       const pres_bc_type ibc_pres, const enth_bc_type ibc_enth,
       const int ibc_func, const double ibc_pres_val,
       const double ibc_enth_val,
       /* Outer boundary condition parameters */
       const pres_bc_type obc_pres, const enth_bc_type obc_enth,
       const int obc_func, const double obc_pres_val,
       const double obc_enth_val,
       /* Source function parameters */
       const int massSrc_func, const double massSrc_val,
       const int intEnSrc_func, const double intEnSrc_val,
       /* Control and method parameters */
       const double errTol, const double dtTol,
       const unsigned int maxIter, const unsigned int interpOrder,
       const double maxDtIncrease, const double dtMin,
       const int maxStep, const unsigned int useBE,
       const unsigned int verbosity, const wksp *w,
       /* User-defined extra parameters */
       void *params,
       /* Outputs */
       const unsigned int nOut, double *tOut, double *colOut,
       double *presOut, double *eIntOut,
       double *mBndOut, double *eBndOut,
       double *mSrcOut, double *eSrcOut,
       /* Diagnostic outputs */
       unsigned long *nStep, unsigned long *nIter,
       unsigned long *nFail
#ifdef TESTING_MODE
       , double *residSum, unsigned int *iterStep,
       double *driverTime, double *advanceTime,
       double *nextIterTime, double *userTime
#endif
       );
```

Parameters:

- `tStart`: (INPUT) start time of the simulation
- `tEnd`: (INPUT) end time of the simulation
- `dtStart`: (INPUT) value of first simulation time step; if set to a value ≤ 0, this is ignored and the size of the first time step is set automatically
- `grd`: (INPUT) pointer to a `struct grid` that describes the simulation grid
- `col`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the column density at the start of the simulation; at the end of the routine, it is updated to the final column density values
- `pres`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the vertically-integrated pressure at the start of the simulation; at the end of the routine, it is updated to the final column pressure values
- `eInt`: (INPUT/OUTPUT) pointer to an array of `grd->nr` elements containing the internal energy per unit area at the start of the simulation; at the end of the routine, it is updated to the final internal energy values. If `eos_func` is set to 0, this will not be referenced, and can be set to `NULL`.
- `eos_func`: (INPUT) a non-zero value indicates that `userEOS` should be called to set the equation of state parameters
- `gamma_val`: (INPUT) constant value of γ; ignored if `eos_func` is non-zero
- `delta_val`: (INPUT) constant value of δ; ignored if `eos_func` is non-zero
- `alpha_func`: (INPUT) a non-zero value indicates that `userAlpha` should be called to set the viscosity
- `alpha_val`: (INPUT) constant value of α; ignored if `alpha_func` is non-zero
- `pres_ibc`: (INPUT) type of inner pressure boundary condition; allowed values are `FIXED_MASS_FLUX`, `FIXED_TORQUE_FLUX`, and `FIXED_TORQUE`

- `enth_ibc`: (INPUT) type of inner enthalpy boundary condition; allowed values are `FIXED_ENTHALPY_VALUE` and `FIXED_ENTHALPY_GRADIENT`
- `ibc_func`: (INPUT) a non-zero value indicates that `userIBC` should be called to set the inner boundary condition
- `ibc_pres_val`: (INPUT) constant value for the mass flux, torque flux, or torque (depending on the value of `pres_ibc`) used with the inner pressure boundary condition; ignored if `ibc_func` is non-zero
- `ibc_enth_val`: (INPUT) constant value for the enthalpy value or enthalpy gradient (depending on the value of `enth_ibc`) use with the inner enthalpy boundary condition; ignored if `ibc_func` is non-zero
- `pres_obc`: (INPUT) identical to `pres_ibc`, but for the outer boundary
- `enth_obc`: (INPUT) identical to `enth_ibc`, but for the outer boundary
- `obc_func`: (INPUT) identical to `ibc_func`, but for the outer boundary
- `obc_pres_val`: (INPUT) identical to `ibc_pres_val`, but for the outer boundary
- `obc_enth_val`: (INPUT) identical to `ibc_enth_val`, but for the outer boundary
- `massSrc_func`: (INPUT) a non-zero value indicates that `userMassSrc` should be called to set the mass source function
- `massSrc_val`: (INPUT) constant value of the mass source function; ignored if `massSrc_func` is non-zero
- `intEnSrc_func`: (INPUT) a non-zero value indicates that `userIntEnSrc` should be called to set the mass source function
- `intEnSrc_val`: (INPUT) constant value of the mass source function; ignored if `intEnSrc_func` is non-zero
- `errTol`: (INPUT) maximum error in the iterative solver; used to test for convergence
- `dtTol`: (INPUT) maximum fractional change in column density / pressure / internal energy in fastest-varying cell; used to estimate the next time step
- `maxIter`: (INPUT) maximum number of iterations allowed in the implicit solver
- `interpOrder`: (INPUT) order of interpolation to use when reconstructing specific internal enthalpy at cell faces; 1 = piecewise constant, 2 = piecewise linear, 3 = piecewise parabolic

- `maxDtIncrease`: (INPUT) maximum factor by which the time step is allowed to increase per time step
- `dtMin`: (INPUT) minimum time step; the simulation terminates if the time step is ever smaller than `dtMin*(tEnd-tStart)`
- `maxStep`: (INPUT) maximum number of time steps allowed; if the simulation reaches this number of time steps, it terminates. Negative values indicate no limit.
- `useBE`: (INPUT) if non-zero, backwards Euler updating is used; otherwise Crank-Nicolson updating is used
- `verbosity`: (INPUT) level of verbosity in output printing; 0 = no output printed, 1 = print out time and time step every 100 time steps; 2 = print out time and time step every time step; 3 = print out time and time step every time step, and print out residual every implicit iteration
- `w`: (INPUT/OUTPUT) pointer to a `struct workspace` object
- `params`: (INPUT) pointer to memory containing user-defined parameters; this memory is not touched by this routine, but the pointer is passed to all `user*` routines
- `nOut`: (INPUT) number of times at which to store output
- `tOut`: (INPUT) pointer to array of `nOut` times at which to store output
- `colOut`: (OUTPUT) pointer to array of `nOut*grd->nr` elements used to store column densities at times specified by `tOut`. The first output time is stored in the elements `0`$\cdots$`grd->nr-1`, the next time in elements `grd->nr`$\cdots$`2*grd->nr-1`, etc.
- `presOut`: (OUTPUT) same as `colOut`, but used to store vertically-integrated pressures
- `eIntOut`: (OUTPUT) same as `colOut`, but used to store internal energy per unit area. This array is only used if `eos_func` is non-zero; otherwise it is not referenced, and can be set to `NULL`.
- `mBndOut`: (OUTPUT) pointer to array of `nOut*2` elements used to store the cumulative mass advected across the inner and outer boundaries of the simulation up to times `tOut`. The inner and outer boundary values at the first time are stored in elements 0 and 1, inner and outer boundary values at the second time are in elements 2 and 3, etc. The sign convention is such that values > 0 indicate transport in the $+r$ direction.
- `eBndOut`: (OUTPUT) pointer to array of `nOut*2` elements used to store the cumulative energy transported across the inner and outer boundaries of the simulation up to times `tOut`. The inner and outer boundary values at the first time are stored in elements 0 and 1, inner and outer boundary values at the second time are in elements 2 and 3, etc. The sign convention is such that values > 0 indicate transport in the $+r$ direction.

- mSrcOut: (OUTPUT) pointer to array of nOut*grd->nr elements used to store the cumulative column density added to every grid cell by the source terms up to times tOut. The first time values are stored in elements 0 · · · grd->nr-1, the next time in elements grd->nr · · · 2*grd->nr-1, etc. This array is only referenced if massSrc_func is non-zero; otherwise it may be set to NULL.
- mSrcOut: (OUTPUT) pointer to array of nOut*grd->nr elements used to store the cumulative energy per unit area added to every grid cell by the source terms up to times tOut. The first time values are stored in elements 0 · · · grd->nr-1, the next time in elements grd->nr · · · 2*grd->nr-1, etc. This array is only referenced if massSrc_func or intEnSrc_func is non-zero; otherwise it may be set to NULL.
- nStep: (OUTPUT) total number of time steps taken during the simulation
- nIter: (OUTPUT) total number of implicit iterations performed during the simulation, summed over all time steps
- nFail: (OUTPUT) total number of times the implicit solver failed to converge, necessitating a reduction in the time step
- residSum: (OUTPUT, only in TESTING_MODE): pointer to an array of maxIter elements; the value of the residual after N iterations, summed over all simulation time steps, is stored in element N
- iterStep: (OUTPUT, only in TESTING_MODE): pointer to an array of maxStep elements; returns the number of implicit iterations performed in each time step
- driverTime: (OUTPUT, only in TESTING_MODE): returns the time spent in the driver routine, in seconds
- advanceTime: (OUTPUT, only in TESTING_MODE): returns the time spent in the advance routine, in seconds
- nextIterTime: (OUTPUT, only in TESTING_MODE): returns the time spent the next iterate (via Anderson acceleration, if AA_M > 0) in the implicit solver, in seconds
- userTime: (OUTPUT, only in TESTING_MODE): returns the time spent in the user* routines, in seconds
- Return value: the final time in the simulation; equal to tEnd if the simulation successfully ran to completion

This is the main simulation driver routine. The user specifies the simulation start and end times, the initial state of the column density, pressure, and internal energy (if needed), as well as parameters describing the equation of state, viscosity, boundary conditions, and source terms. The routine then runs the simulation, storing the state of the simulation periodically at times specified by the user. On return, the column density, pressure, and internal energy arrays are updated to their values at the end of the simulation. If the simulation takes too many time steps, or the time step drops below the specified limit, the simulation terminates early; in this case the final state of the simulation is stored in the last slot of the output arrays, and the time at which the simulation terminated is returned.

9.1.6 `gridAlloc`

Defined in `init.h` and `init.c`.

```
grid *gridAlloc(const unsigned int nr);
```

Parameters:

- `nr`: (INPUT) number of grid cells
- Return value: a pointer to a `struct grid` object that is allocated but uninitialized

This routine allocates memory for a `struct grid` object. The object is uninitialized.

9.1.7 `gridFree`

Defined in `init.h` and `init.c`.

```
void gridFree(grid *grd);
```

Parameters:

- `grd`: the grid to be freed

This routine de-allocates memory for a `struct grid` object.

9.1.8 `gridInit`

Defined in `init.h` and `init.c`.

```
grid *gridInit(const unsigned int nr,
               const double *r_g, const double *r_h,
               const double *vphi_g, const double *vphi_h,
               const double *beta_g, const double *beta_h,
               const double *psiEff_g, const double *psiEff_h,
               const double *g_h,
               const unsigned int linear);
```

Parameters:

- `nr`: (INPUT) number of grid cells
- `r_g`: (INPUT) pointer to an array of `nr+2` values giving the cell-center positions of each grid cell, including two ghost zones; must be sorted in strictly increasing order
- `r_h`: (INPUT) pointer to an array of `nr+1` values giving the edge positions of each grid cell, excluding the ghost zones; must be sorted in strictly increasing order
- `vphi_g`: (INPUT) pointer to an array of `nr+2` values giving rotation curve velocities v_ϕ at cell centers, including two ghost cells
- `vphi_h`: (INPUT) pointer to an array of `nr+1` values giving rotation curve velocities v_ϕ at cell edges
- `beta_g`: (INPUT) pointer to an array of `nr+2` values giving values of $d \ln v_\phi / d \ln r$ at cell centers, including two ghost cells
- `beta_h`: (INPUT) pointer to an array of `nr+1` values giving values of $d \ln v_\phi / d \ln r$ at cell edges
- `psiEff_g`: (INPUT) pointer to an array of `nr+2` values giving values of the potential plus orbital energy per unit mass $\psi_{\text{eff}} = v_\phi^2/2 + \psi$ at cell centers, including two ghost cells
- `psiEff_h`: (INPUT) pointer to an array of `nr+1` values giving values of the potential plus orbital energy per unit mass $\psi_{\text{eff}} = v_\phi^2/2 + \psi$ at cell edges
- `g_h`: (INPUT) pointer to an array of `nr+1` values giving the derivative factor $2\pi/[v_\phi(1+\beta)]/(r\Delta \ln r)$ (for logarithmic grids) or $2\pi/[v_\phi(1+\beta)]/r$ (for linear grids) evaluated at cell edges
- `linear`: (INPUT) a non-zero value indicates that the grid should be treated as linear for the purposes of taking derivatives and making interpolations; a value of 0 indicates that the grid should be treated as logarithmic
- Return value: a pointer to a `struct grid` object that is allocated and initialized using the values in the input quantities

This routine constructs a `struct grid` object from the specified input values. This routine is the most general grid construction routine, as it allows the user to manually specify every quantity on the grid.

9.1.9 `gridInitFlat`

Defined in `init.h` and `init.c`.

```
grid *gridInitFlat(const unsigned int nr, const double rmin,
                   const double rmax, const double vphi,
                   const unsigned int linear);
```

Parameters:

- `nr`: (INPUT) number of grid cells
- `rmin`: (INPUT) radius of the inner edge of the innermost grid cell
- `rmax`: (INPUT) radius of the outer edge of the outermost grid cell
- `vphi`: (INPUT) constant rotation curve speed
- `linear`: (INPUT) a non-zero value indicates that the grid should be treated as linear for the purposes of taking derivatives and making interpolations; a value of 0 indicates that the grid should be treated as logarithmic
- Return value: a pointer to a `struct grid` object that is allocated and initialized using the values in the input quantities

This routine constructs a grid of `nr` cells from `rmin` to `rmax`, spaced uniformly linearly or logarithmically in radius, depending on the value of `linear`. The rotation curve on the grid is flat, with a constant value `vphi`.

9.1.10 `gridInitKeplerian`

Defined in `init.h` and `init.c`.

```
grid *gridInitKeplerian(const unsigned int nr, const double rmin,
                        const double rmax, const double m,
                        const unsigned int linear);
```

Parameters:

- `nr`: (INPUT) number of grid cells
- `rmin`: (INPUT) radius of the inner edge of the innermost grid cell
- `rmax`: (INPUT) radius of the outer edge of the outermost grid cell
- `m`: (INPUT) mass of central object, in CGS units

- `linear`: (INPUT) a non-zero value indicates that the grid should be treated as linear for the purposes of taking derivatives and making interpolations; a value of 0 indicates that the grid should be treated as logarithmic
- Return value: a pointer to a `struct grid` object that is allocated and initialized using the values in the input quantities

This routine constructs a grid of `nr` cells from `rmin` to `rmax`, spaced uniformly linearly or logarithmically in radius, depending on the value of `linear`. The rotation curve is Keplerian, with a central object of mass `m`.

9.1.11 `gridInitTabulated`

Defined in `init.h` and `init.c`.

```
grid *gridInitTabulated(const unsigned int nr, const double *rTab,
                        const double *vTab, const int nTab,
                        const double rmin, const double rmax,
                        const unsigned int bspline_degree,
                        const unsigned int bspline_breakpoints,
                        const unsigned int linear);
```

Parameters:

- `nr`: (INPUT) number of grid cells
- `rTab`: (INPUT) pointer to an array of tabulated radii
- `vTab`: (INPUT) pointer to an array of tabulated v_ϕ values
- `nTab`: (INPUT) number of elements in the `rTab` and `vTab` arrays
- `rmin`: (INPUT) radius of the inner edge of the innermost grid cell
- `rmax`: (INPUT) radius of the outer edge of the outermost grid cell
- `bspline_degree`: (INPUT) degree of the basis spline fit used to interpolate the tabulated rotation curve
- `bspline_breakpoints`: (INPUT) number of breakpoints in the basis spline fit used to interpolate the tabulated rotation curve
- `linear`: (INPUT) a non-zero value indicates that the grid should be treated as linear for the purposes of taking derivatives and making interpolations; a value of 0 indicates that the grid should be treated as logarithmic

- Return value: a pointer to a `struct grid` object that is allocated and initialized using the values in the input quantities

This routine constructs a grid of `nr` cells from `rmin` to `rmax`, spaced uniformly linearly or logarithmically in radius, depending on the value of `linear`. The rotation curve on the grid, and its derivative and integral (the potential), are determined by constructing a basis spline interpolation, using the specified number of breakpoints and degree, to the input table of (r, v_ϕ) values. The input values must satisfy `nTab >= bspline_breakpoints + bspline_degree - 2`, and the smallest and largest values in `rTab` must be such that the centers of all grid cells (including ghost cells) are between them.

9.1.12 `ppmExtrap`

Defined in `ppmExtrap.h` and `ppmExtrap.c`.

```
void ppmExtrap(const int nx, const double *dx, double *v, double *vL,
               double *vR, double *workspace);
```

Parameters:

- `nx`: (INPUT) number of array elements
- `dx`: (INPUT) pointer to an array of `nx` elements giving the sizes of array cells
- `v`: (INPUT) pointer to an array of `nx` elements giving average values in cells
- `vL`: (OUTPUT) pointer to an array of `nx` elements giving values at cell left edges
- `vR`: (OUTPUT) pointer to an array of `nx` elements giving values at cell right edges
- `workspace`: (INPUT/OUTPUT) pointer to an array of `nx` elements that can be used as workspace

This routine implements the Colella & Woodward (1984, J. Comp. Phys., 54, 174-201) piecewise parabolic method for estimating quantities on the left and right sides of cell interfaces from the cell-averaged values on a non-uniform grid. The inputs are the number of cells, their sizes, and their average values, while the outputs are the values at the cell left and right edges.

9.1.13 `rotCurveSpline`

Defined in `rotCurveSpline.h` and `rotCurveSpline.c`.

```
void
rotCurveSpline(const double *rTab, const double *vTab,
               const unsigned int nTab,
               const unsigned int bspline_degree,
               const unsigned int bspline_breakpoints,
               const double *r, const unsigned int nr,
               double *vphi, double *psi,
               double *beta);
```

Parameters:

- `rTab`: (INPUT) pointer to an array of `nTab` elements giving radial positions in the data table; must be ordered from lowest to highest radius
- `vTab`: (INPUT) pointer to an array of `nTab` elements giving rotation curve velocities v_ϕ in the data table
- `nTab`: (INPUT) number of entries in the input data table
- `bspline_degree`: (INPUT) degree of the basis spline fit used to interpolate the tabulated rotation curve
- `bspline_breakpoints`: (INPUT) number of breakpoints in the basis spline fit used to interpolate the tabulated rotation curve
- `r`: (INPUT) pointer to an array of `nr` radii at which interpolated values are to be computed; must be ordered from lowest to highest radius
- `nr`: (INPUT) number of radii at which interpolated values are to be computed
- `vphi`: (OUTPUT) pointer to an array of `nr` at which interpolated values of v_ϕ are to be stored
- `psi`: (OUTPUT) pointer to an array of `nr` at which interpolated values of the gravitational potential ψ are to be stored
- `beta`: (OUTPUT) pointer to an array of `nr` at which interpolated values of the rotation curve index $d\ln v_\phi/d\ln r$ are to be stored

This function takes a table of (r, v_ϕ) values as inputs and uses basis spline interpolation to construct a fitting function to them. This fitting function is then used to generate a series of interpolated v_ϕ values, and a corresponding set of gravitational potential ψ and rotation curve index $d\ln v_\phi/d\ln r$ values at a specified set of fitting points. The gravitational potential is computed in a gauge where $\psi = 0$ at the outermost fitting point. The input data table and fitting points must be `rTab[0] <= r[0]=` and `rTab[nTab-1] >= r[nr-1]`. The quantities `bspline_breakpoints` and `bspline_degree` specify how many breakpoints there should be in the b-spline fit, and what degree the fit should be, and must satisfy `nTab >= bspline_breakpoints + bspline_degree - 2`. Optimal choices of these parameters depend on the nature of the input data: sparsely-sampled, noisy data should choose smaller values, while well-sampled, low-noise data should use larger values.

9.1.14 `testingMode`

Defined in `testingMode.c`.

```
int testingMode();
```

Parameters:

- Return value: this function returns 1 if the code was compiled with `MODE=TEST`, 0 otherwise

This function provides a way of determining whether the code was compiled in testing mode, so that the appropriate interface to `advanceBE`, `advanceCN`, and `driver` can be selected.

9.1.15 `wkspAlloc`

Defined in `init.h` and `init.c`.

```
wksp *wkspAlloc(const unsigned int nr);
```

Parameters:

- `nr`: (INPUT) number of grid cells in the workspace
- Return value: a `struct wksp` object for which memory has been allocated.

This routine allocated memory for a `struct wksp` object.

9.1.16 `wkspFree`

Defined in `init.h` and `init.c`.

```
void wkspFree(wksp *w);
```

Parameters:

- `w`: pointer to the `struct wksp` object to be de-allocated.

This routine de-allocates a `struct wksp` object.

9.2 User-Implemented c Routines

All the routines in this section are problem-dependent, and it is up to the user to implement code for these routines that matches the specified API and defines the physical model to be used in his or her simulation. All of these routines take an argument `params`, of type `void *`. The `user*` routines are all called from `advanceBE`, `advanceCN`, or `driver`; these routines also accept the `params` argument, and they simply pass it on to the `user*` routines. This argument is provided to make it possible to pass arbitrary information into the `user*` routines. It is up to the user to cast this pointer to the appropriate data structure and to parse the data to which it points. If no additional parameters are needed in the `user*` routines, the user may simply choose not to reference the `params` argument, and may then pass `NULL` to the corresponding argument slot in `advanceBE`, `advanceCN`, or `driver`.

9.2.1 `userAlpha`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userAlpha(
          /* Inputs */
          const double t, const grid *grd,
          const double *col, const double *pres,
          const double *eInt, const double *gamma,
          const double *delta, void *params,
          /* Outputs */
          double *alpha
          );
```

Parameters:

- `t`: (INPUT) current simulation time
- `grd`: (INPUT) pointer to the `struct grid` object for the simulation
- `col`: (INPUT) pointer to an array of `grd->nr` elements giving the current column densities on the grid
- `pres`: (INPUT) pointer to an array of `grd->nr` elements giving the current vertically-integrated pressures on the grid
- `eInt`: (INPUT) pointer to an array of `grd->nr` elements giving the current internal energy per unit area on the grid; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero

- `gamma`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state γ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero

- `delta`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state δ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero

- `params`: (INPUT) a pointer to a block of memory containing parameters

- `alpha`: (OUTPUT) pointer to an array of `grd->nr` elements containing the dimensionless viscosity α at every grid cell center

The routine receives as inputs the current state of the simulation, including the time, grid, and values of all quantities being evolved (column density, vertically-integrated pressure, internal energy per unit area) and all quantities derives from them (equation of state parameters γ, δ). It also receives whatever data is passed through `params`. The user must implement code that, given these inputs, computes the dimensionless viscosity α at the center of every grid cell and stores the result in the array `alpha`. This routine is called from hyperref[sssec:advanceBE]`advanceBE`, `advanceCN`, and `driver`, but only if `alpha_func` is non-zero.

9.2.2 `userEOS`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userEOS(
        /* Inputs */
        const double t, const grid *grd,
        const double *col, const double *pres, const double *eInt,
        void *params,
        /* Outputs */
        double *gamma, double *delta
        );
```

Parameters:

- `t`: (INPUT) current simulation time

- `grd`: (INPUT) pointer to the `struct grid` object for the simulation

- `col`: (INPUT) pointer to an array of `grd->nr` elements giving the current column densities on the grid

- `pres`: (INPUT) pointer to an array of `grd->nr` elements giving the current vertically-integrated pressures on the grid
- `eInt`: (INPUT) pointer to an array of `grd->nr` elements giving the current internal energy per unit area on the grid
- `params`: (INPUT) a pointer to a block of memory containing parameters
- `gamma`: (OUTPUT) pointer to an array of `grd->nr` elements giving the current equation of state γ value in every grid cell
- `delta`: (OUTPUT) pointer to an array of `grd->nr` elements giving the current equation of state δ value in every grid cell

The routine receives as inputs the current state of the simulation, including the time, grid, and values of all quantities being evolved (column density, vertically-integrated pressure, internal energy per unit area), as well as whatever data is passed through `params`. The user must implement code to compute the equation of state parameters γ and δ in every grid cell and store the result in the arrays `gamma` and `delta`. This routine is called from `advanceBE`, `advanceCN`, and `driver`, but only if `eos_func` is non-zero.

9.2.3 `userIBC`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userIBC(
        /* Inputs */
        const double t, const grid *grd,
        const double *col, const double *pres, const double *eInt,
        const double *gamma, const double *delta,
        const pres_bc_type ibc_pres, const enth_bc_type ibc_enth,
        void *params,
        /* Outputs */
        double *ibc_pres_val, double *ibc_enth_val
        );
```

Parameters:

- `t`: (INPUT) current simulation time
- `grd`: (INPUT) pointer to the `struct grid` object for the simulation
- `col`: (INPUT) pointer to an array of `grd->nr` elements giving the current column densities on the grid

- `pres`: (INPUT) pointer to an array of `grd->nr` elements giving the current vertically-integrated pressures on the grid
- `eInt`: (INPUT) pointer to an array of `grd->nr` elements giving the current internal energy per unit area on the grid; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `gamma`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state γ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `delta`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state δ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `params`: (INPUT) a pointer to a block of memory containing parameters
- `ibc_pres`: (INPUT) the type of pressure boundary condition being used; will be one of `FIXED_MASS_FLUX`, `FIXED_TORQUE_FLUX`, or `FIXED_TORQUE`
- `ibc_enth`: (INPUT) the type of enthalpy boundary condition being used; will be one of `FIXED_ENTHALPY_VALUE` or `FIXED_ENTHALPY_GRADIENT`
- `ibc_pres_val`: (OUTPUT) value of the inner pressure boundary condition; depend on the value of `ibc_pres` this will be interpreted as a value for the mass flux across the boundary, the torque flux across the boundary, or the torque in the ghost zone adjacent to the boundary
- `ibc_enth_val`: (OUTPUT) value of the inner enthalpy boundary condition; depend on the value of `ibc_enth` this will be interpreted as a value for the either the internal enthalpy in the first ghost zone, or for the gradient of internal enthalpy at the inner cell edge

The routine receives as inputs the current state of the simulation, including the time, grid, and values of all quantities being evolved (column density, vertically-integrated pressure, internal energy per unit area) and all quantities derives from them (equation of state parameters γ, δ). It also receives whatever data is passed through `params`. The user must implement code that computes the value of the inner pressure and specific enthalpy boundary conditions, and stores those values in `ibc_pres_val` and `ibc_enth_val`. The way that these values will be interpreted depends on the values of `ibc_pres` and `ibc_enth`, which specify whether the value expected is, for pressure, a mass flux, a torque flux, or an absolute torque, and, for enthalpy, whether it is an value of the specific enthalpy or a value for the gradient of specific enthalpy. For all fluxes, the sign convention is that positive values indicate transport in the $+r$ direction. This routine is called from `advanceBE`, `advanceCN`, and `driver`, but only if `ibc_func` is non-zero.

9.2.4 `userIntEnSrc`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userIntEnSrc(
             /* Inputs */
             const double t, const grid *grd,
             const double *col, const double *pres,
             const double *eInt, const double *gamma,
             const double *delta, void *params,
             /* Outputs */
             double *intEnSrc
             );
```

Parameters:

- `t`: (INPUT) current simulation time
- `grd`: (INPUT) pointer to the `struct grid` object for the simulation
- `col`: (INPUT) pointer to an array of `grd->nr` elements giving the current column densities on the grid
- `pres`: (INPUT) pointer to an array of `grd->nr` elements giving the current vertically-integrated pressures on the grid
- `eInt`: (INPUT) pointer to an array of `grd->nr` elements giving the current internal energy per unit area on the grid; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `gamma`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state γ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `delta`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state δ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero

- `params`: (INPUT) a pointer to a block of memory containing parameters
- `intEnSrc`: (OUTPUT) pointer to an array of `grd->nr` elements giving the value of the source term for internal energy per unit area in each cell

The routine receives as inputs the current state of the simulation, including the time, grid, and values of all quantities being evolved (column density, vertically-integrated pressure, internal energy per unit area) and all quantities derives from them (equation of state parameters γ, δ). It also receives whatever data is passed through `params`. The user must implement code that computes the source term for internal energy per unit area $\dot{E}_{\mathrm{int,src}}$ in each cell and stores those values in the array `intEnSrc`. This routine is called from `advanceBE`, `advanceCN`, and `driver`, but only if `intEnSrc_func` is non-zero.

9.2.5 `userMassSrc`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userMassSrc(
            /* Inputs */
            const double t, const grid *grd,
            const double *col, const double *pres,
            const double *eInt, const double *gamma,
            const double *delta, void *params,
            /* Outputs */
            double *massSrc
            );
```

Parameters:

- `t`: (INPUT) current simulation time
- `grd`: (INPUT) pointer to the `struct grid` object for the simulation
- `col`: (INPUT) pointer to an array of `grd->nr` elements giving the current column densities on the grid
- `pres`: (INPUT) pointer to an array of `grd->nr` elements giving the current vertically-integrated pressures on the grid
- `eInt`: (INPUT) pointer to an array of `grd->nr` elements giving the current internal energy per unit area on the grid; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero

- `gamma`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state γ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `delta`: (INPUT) pointer to an array of `grd->nr` elements giving the current equation of state δ value in every grid cell; note that this array is updated and contains meaningful data only if the value of `eos_func` passed to `advanceBE`, `advanceCN`, or `driver` is not zero
- `params`: (INPUT) a pointer to a block of memory containing parameters
- `massSrc`: (OUTPUT) pointer to an array of `grd->nr` elements giving the value of the source term for mass per unit area in each cell

The routine receives as inputs the current state of the simulation, including the time, grid, and values of all quantities being evolved (column density, vertically-integrated pressure, internal energy per unit area) and all quantities derives from them (equation of state parameters γ, δ). It also receives whatever data is passed through `params`. The user must implement code that computes the source term for mass per unit area $\dot{\Sigma}_{\mathrm{src}}$ in each cell and stores those values in the array `massSrc`. This routine is called from `advanceBE`, `advanceCN`, and `driver`, but only if `massSrc_func` is non-zero.

9.2.6 `userOBC`

Defined in `userFunc.h` and `userFunc_PROB.c`.

```
void
userOBC(
        /* Inputs */
        const double t, const grid *grd,
        const double *col, const double *pres, const double *eInt,
        const double *gamma, const double *delta,
        const pres_bc_type obc_pres, const enth_bc_type obc_enth,
        void *params,
        /* Outputs */
        double *obc_pres_val, double *obc_enth_val
        );
```

The API and parameter definitions are identical to the corresponding ones for `userIBC`, except that this routine is applied to the outer rather than the inner boundary condition. It is called from `advanceBE`, `advanceCN`, and `driver`, but only if `obc_func` is non-zero.

9.3 Python Routines

All Python routines are defined by files in the `vader` or `vader/test` subdirectories of the main `vader` directory.

9.3.1 `vader.advance`

Defined in `interface.py`.

```
def advance(t, dt, grd, col, pres, paramDict, eInt=None,
            no_update=False, c_params=None, **kwargs):
```

Parameters:

- `t`: (`float`, INPUT) time at start of time step
- `dt`: (`float`, INPUT) size of time step
- `grd` : (`class grid`, INPUT) grid object describing the simulation grid
- `col` : (`numpy.array`, INPUT/OUTPUT) array of starting column densities; on return, contains column densities at end of time step
- `pres` : (`numpy.array`, INPUT/OUTPUT) array of starting vertically-integrated pressures; on return, contains column densities at end of time step
- `paramDict`: (`dict`, INPUT) dict of simulation parameters; can be set manually, but usually returned by `vader.readParam`
- `eInt`: (`numpy.array`, INPUT/OUTPUT) array of starting internal energies; must be set if `gamma` or `delta` in `paramDict` are specified as `c_func` instead of constants, ignored otherwise
- `no_update`: (`bool`, INPUT) if `True`, the state quantities will not be altered, and the original values will be returned
- `c_params`: (`c_void_p`, INPUT) pointer to parameters to be passed into the c functions
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below.

This function calls the c `advanceBE` or `advanceCN` routine, depending on the value of `method` in the `paramDict`. Other values in `paramDict` are used to set the corresponding arguments in the c functions. The return value of the function is a `namedtuple` whose elements are:

- `success`: (`bool`) `True` if the iteration converged, `False` if not
- `dtNew`: (`float`) estimated value of next time step
- `nIter`: (`long`) number of implicit iterations performed
- `mBnd`: (`numpy.array, shape (2)`) mass transported across inner and outer boundaries during this time step; positive values indicate transport in $+r$ direction
- `eBnd`: (`numpy.array, shape (2)`) energy transported across inner and outer boundaries during this time step; positive values indicate transport in $+r$ direction
- `mSrc`: (`numpy array, shape (grd.nr)`) mass column density added to each cell by the source terms during this time step; returned only if `mass_src` is set to `c_func` in `paramDict`; otherwise this is `None`
- `eSrc`: (`numpy array, shape (grd.nr)`) energy per unit area added to each cell by the source terms during this time step; returned only if `mass_src` or `int_en_src` is set to `c_func` in `paramDict`; otherwise this is `None`
- `resid`: (`numpy.array, shape (paramDict['max_iter'])`) residual after each iteration; returned only if the code is compiled in Testing Mode
- `rtype`: (`numpy.array of uintc, shape (paramDict['max_iter'])`) indicates if the maximum residual after each iteration was in column density, pressure, or internal energy, with 1 = column density, 2 = pressure, 3 = internal energy; returned only if the code is compiled in Testing Mode

9.3.2 `vader.driver`

Defined in `interface.py`.

```
def driver(tStart, tEnd, grd, col, pres, paramDict, nOut=50,
           outTimes=None, eInt=None, c_params=None, **kwargs):
```

Parameters:

- `tStart`: (`float`, INPUT) time at start of simulation
- `tEnd`: (`float`, INPUT) simulation end time
- `grd` : (`class grid`, INPUT) grid object describing the simulation grid
- `col` : (`numpy.array`, INPUT/OUTPUT) array of starting column densities; on return, contains column densities at end of time step
- `pres` : (`numpy.array`, INPUT/OUTPUT) array of starting vertically-integrated pressures; on return, contains column densities at end of time step

- `paramDict`: (`dict`, INPUT) dict of simulation parameters; can be set manually, but usually returned by `vader.readParam`
- `nOut`: (`int`, INPUT) number of times at which to store output; outputs are only stored if `nOut >= 1`
- `outTimes`: (`numpy.array`, INPUT) exact times at which to store output; if set, this overrides `nOut`
- `eInt`: (`numpy.array`, INPUT/OUTPUT) array of starting internal energies; must be set if `gamma` or `delta` in `paramDict` are specified as `c_func` instead of constants, ignored otherwise
- `c_params`: (`c_void_p`, INPUT) pointer to parameters to be passed into the c functions
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below.

This function calls the c `driver` routine, using the values in `paramDict` to set the call parameters. If `nOut` is ≥ 1 and `outTimes` is not set, then the array of output times passed to the c routine is automatically set to an array of values uniformly spaced in time from `tStart` to `tEnd`. The return value of the function is a `namedtuple` whose elements are:

- `tFin`: (`float`) time at which simulation ended
- `tOut`: (`numpy.array, shape (nOut)`) times at which output is stored
- `colOut`: (`numpy.array, shape (nOut, grd.nr)`) 2D array of column densities stored at specified times
- `presOut`: (`numpy.array, shape (nOut, grd.nr)`) 2D array of vertically-integrated pressures stored at specified times
- `eIntOut`: (`numpy.array, shape (nOut, grd.nr)`) 2D array of internal energies per unit area stored at specified times; returned only if `gamma` or `delta` in `paramDict` are set to `c_func`; otherwise this is `None`
- `mBndOut`: (`numpy.array, shape (nOut, 2)`) cumulative mass transported across the inner and outer boundaries up to the specified time; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction
- `eBndOut`: (`numpy.array, shape (nOut, 2)`) cumulative energy transported across the inner and outer boundaries up to the specified time; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction

- `mSrcOut`: (`numpy.array, shape (nOut, grd.nr)`) cumulative mass column density added to each cell by the source terms up to the specified time; returned only if `mass_src` in `paramDict` is set to `c_func`; otherwise this is `None`

- `eSrcOut`: (`numpy.array, shape (nOut, grd.nr)`) cumulative energy per unit area added to each cell by the source terms up to the specified time; returned only if `mass_src` or `int_en_src` in `paramDict` is set to `c_func`; otherwise this is `None`

- `nStep`: (`long`) total number of simulation time steps

- `nIter`: (`long`) total number of implicit iterations, summed over all time steps

- `nFail`: (`long`) total number of times the implicit solver failed to converge

- `residSum`: (`numpy.array, shape (paramDict['max_iter'])`) sum of residuals after a given iteration number, summed over all time steps; returned only if the code is compiled in Testing Mode

- `iterStep`: (`numpy.array of uintc, shape (paramDict['max_step'])`): total number of iterations performed in each time step; returned only if the code is compiled in Testing Mode

9.3.3 `vader.grid`

Defined in `grid.py`.

This is a class with the following attributes:

- `nr`: (`int`) number of cells

- `r`: (`numpy.array, shape(grd.nr)`) cell center positions

- `r_h`: (`numpy.array, shape(grd.nr+1)`) cell edge positions

- `r_g`: (`numpy.array, shape(grd.nr+2)`) cell center positions, including ghost zones at the inner and outer edges

- `linear`: (`bool`) `True` for linearly-spaced grids, `False` for logarithmically spaced

- `dr`: (`numpy.array, shape (grd.nr)`) cell sizes, `dr = r_h[1:] - r_h[:-1]`

- `dr_h`: (`numpy.array, shape (grd.nr+1)`) cell center to cell center distances, `dr_h = r_g[1:] - r_g[:-1]`

- `dlnr`: (`numpy.array, shape (grd.nr)`) logarithmic cell sizes, `dlnr = ln(r_h[1:]/r_h[:-1])`

- `dlnr_h`: (`numpy.array, shape (grd.nr+1)`) logarithmic cell center to cell center distances, `dlnr_h = ln(r_g[1:]/r_g[:-1])`

- `area`: (`numpy.array, shape (grd.nr)`) cell areas
- `vphi`: (`numpy.array, shape (grd.nr)`) rotation curve velocities at cell centers
- `vphi_h`: (`numpy.array, shape (grd.nr+1)`) rotation curve velocities at cell edges
- `vphi_g`: (`numpy.array, shape (grd.nr+2)`) rotation curve velocities at cell centers, including ghost zones
- `beta`: (`numpy.array, shape (grd.nr)`) rotation curve index $\beta = d \ln v_\phi / d \ln r$ at cell centers
- `beta_h`: (`numpy.array, shape (grd.nr+1)`) rotation curve index $\beta = d \ln v_\phi / d \ln r$ at cell edges
- `beta_g`: (`numpy.array, shape (grd.nr+2)`) rotation curve index $\beta = d \ln v_\phi / d \ln r$ at cell centers, including ghost zones
- `psi`: (`numpy.array, shape (grd.nr)`) gravitational potential at cell centers
- `psi_h`: (`numpy.array, shape (grd.nr+1)`) gravitational potential at cell edges
- `psi_g`: (`numpy.array, shape (grd.nr+2)`) gravitational potential at cell centers, including ghost zones
- `psiEff`: (`numpy.array, shape (grd.nr)`) gravitational plus orbital energy per unit mass at cell centers
- `psiEff_h`: (`numpy.array, shape (grd.nr+1)`) gravitational plus orbital energy per unit mass at cell edges
- `psiEff_g`: (`numpy.array, shape (grd.nr+2)`) gravitational plus orbital energy per unit mass at cell centers, including ghost zones
- `g_h`: (`numpy.array, shape (grd.nr+1)`) the quantity $g = 2\pi/[v_\phi(1+\beta)\Delta r]$ (for linear grids), or $g = 2\pi/[v_\phi(1+\beta)r\Delta \ln r]$ (for logarithmic grids), computed at cell edges
- `c_grd`: (`c_void_p`) a pointer to a c struct grid object associated with this grid
- `c_wksp`: (`c_void_p`) a pointer to a c struct wksps object associated with this grid

The class has one externally-visible function, which uses an input `paramDict` to construct an instance of the class.

```
def __init__(self, paramDict, rotCurveFun=None, rotCurveTab=None,
             r_h=None, r_g=None):
```

Parameters:

- `paramDict`: (`dict`, INPUT) dict of simulation parameters; can be set manually, but usually returned by `vader.readParam`
- `rotCurveFun`: (`callable`, INPUT) a function to compute the rotation curve; it must take two arguments: a `numpy.array` of positions, and the `paramDict`, and return 3 `numpy.array`s giving the rotation curve velocity, the potential, and the index β at the input positions. If this parameter is not `None`, then all information related to the rotation curve in `paramDict` is ignored.
- `rotCurveTab`: (`numpy.array, shape (2, N)`, INPUT) an array containing a tabulated rotation curve; elements `[0,:]` give the positions, and elements `[1,:]` give the velocities at those positions. If this parameter is not `None`, then all information related to the rotation curve in `paramDict` is ignored.
- `r_h`: (`numpy.array, shape (N)`) an array of cell edge positions. If this parameter is not `None`, then all information related to the grid in `paramDict` is ignored.
- `r_g`: (`numpy.array, shape (N+1)`) an array of cell center positions, including two ghost cells. If this parameter is not `None`, then all information related to the grid in `paramDict` is ignored. This must be set if `r_h` is set, and vice-versa.

This function constructs a `class grid` object using the information found in the `paramDict`. By default the grid is either uniform in linear or logarithmic radius, with edges specified by `rmin` and `rmax` in `paramDict`, but this can be overridden by setting the keywords `r_g` and `r_h`, allowing arbitrary grids. Similarly, the rotation curve is by default set to `flat`, `keplerian`, or `tabulated`, depending on the value specified in the `paramDict`, but this can be overridden by the user setting the keyword `rotCurveFun`. This option allows the user to supply an arbitrary Python callable that returns the rotation curve velocity, potential, and index β versus position, and builds the rotation curve from that. The user can construct `tabulated` rotation curves in two ways. First, the user may supply a `numpy.array` of shape `(2,N)` through the keyword `rotCurveTab`. Second, it the `paramDict` specifies that the rotation curve is `tabulated`, and `rotCurveTab` is `None`, the routine expects the `paramDict` to include the keyword `rot_curve_file` specifying the name of an ASCII file containing a series of `r vphi` values from which the rotation curve table can be read. For either of the tabulated options, the rotation curve, potential, and index on the computational grid are constructed by calling `rotCurveSpline` in `libvader`. The degree and number of breakpoints in the B-spline fit are controlled by `bspline_degree` and `bspline_breakpoints` in `paramDict`.

9.3.4 `vader.readParam`

Defined in `readParam.py`.

```
def readParam(paramFile, noCheck=False):
```

Parameters:

- `paramFile`: (`str`, INPUT) name of a parameter file to be read
- `noCheck`: (`bool`, INPUT) if `TRUE`, the parameter file is not checked to see if it contains all the (key, value) pairs required to run a `VADER` simulation
- Return value: this function returns a `dict` containing the (key, value) pairs read from the specified file

This routine parses a parameter file. The file is expected to be formatted as described in Section 7.2. Unless `noCheck` is set to `True`, it also checks that the parameter file contains all the (key, value) pairs that are mandatory for running a `VADER` simulation; this are `alpha`, `gamma`, `ibc_pres_type`, `ibc_enth_type`, `ibc_pres_val`, `obc_pres_type`, `obc_enth_type`, and `obc_pres_val`, and raises a `ValueError` if a mandatory parameter is found to be missing. The function returns the parsed file as a `dict` suitable for passing to other python `VADER` routines.

9.3.5 `vader.test.gidisk`

Defined in `vader/test/gidisk.py`.

```
def gidisk(paramFile, nOut=16, outTimes=None, verbosity=0, **kwargs):
```

Parameters:

- `paramFile`: (`str`, INPUT) name of the parameter file to use
- `outTimes`: (`numpy.array`, INPUT) array specifying times at which to save output
- `nOut`: (`int`, INPUT) number of output times to save, uniformly spaced; ignored if `outTimes` is not `None`
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` constructed from `paramFile` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below

This function sets up and runs a test comparing `VADER`'s result to the Krumholz & Burkert (2010, ApJ, 724, 895) analytic solution for a steady-state gravitational instability-dominated disk. The file `paramFile` contains the parameter to use for the run; in addition to the usual ones defined in Table 1, the parameter file must contain the following keys: `eta` gives the dimensionless turbulence dissipation rate, `n_orbit` specifies the number of outer orbits to run, `t_Q` specifies the timescale on which $Q \to 1$ in units of the local orbital time, `init_col` and `init_vdisp` give the initial column density and velocity dispersion relative to the steady-state value, `ibc_vdisp` and `obc_vdisp` give the velocity dispersion and the inner and outer boundaries relative to the steady state value, and `dt_init` gives the initial time step in units of the outer orbital time. An example parameter file can be found in `vader/test/gidisk.param`. After the simulation completes, the function returns a `namedtuple` containing the following entries:

- `x`: (`numpy.array, shape (grd.nr)`) dimensionless cell center position, in units where the outer edge of the disk is at $x = 1$
- `T`: (`numpy.array, shape (nOut)`) output times, in units where the orbital period at the disk outer edge is 1
- `colOut`: (`numpy.array, shape (nOut, grd.nr)`) column density at times `T` and positions `x`, in units where the steady state solution is at the disk edge is 1
- `presOut`: (`numpy.array, shape (nOut, grd.nr)`) vertically-integrated pressure at times `T` and positions `x`, in units where the steady state solution for the column density times velocity dispersion squared at the disk edge is 1
- `Q`: (`numpy.array, shape (nOut, grd.nr)`) Toomre Q at times `T` and positions `x`
- `colSteady`: (`numpy.array, shape (grd.nr)`) steady-state solution for column density versus position `x`, in the same dimensionless units as `colOut`
- `presSteady`: (`numpy.array, shape (grd.nr)`) steady-state solution for vertically-integrated pressure versus position `x`, in the same dimensionless units as `presOut`
- `nStep`: (`long`) total number of simulation time steps
- `nIter`: (`long`) total number of implicit iterations, summed over all time steps
- `nFail`: (`long`) total number of times the implicit solver failed to converge

9.3.6 `vader.test.ring`

Defined in `vader/test/ring.py`.

```
def ring(paramFile, nOut=65, outTimes=None, **kwargs):
```

Parameters:

- `paramFile`: (`str`, INPUT) name of the parameter file to use
- `outTimes`: (`numpy.array`, INPUT) array specifying times at which to save output
- `nOut`: (`int`, INPUT) number of output times to save, uniformly spaced; ignored if `outTimes` is not `None`
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` constructed from `paramFile` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below

This function tests VADER against the Pringle (1981, ARA&A, 19, 137) solution for the evolution of an initially-singular ring of material. The file paramFile contains the parameter to use for the run; in addition to the usual ones defined in Table 1, the parameter file must contain the following keys: ring_mass gives the mass of the ring, init_temp gives the initial temperature of the gas, col_ratio gives the ratio of column densities between the cell containing the ring and all other cells, ring_loc gives the location of the ring, kinematic_visc gives the kinematic viscosity, and end_time gives the simulation end time in units of t_s, the characteristic viscous evolution time for the problem. An example parameter file can be found in vader/test/ring.param.The function runs the simulation and then returns a namedtuple containing the following elements:

- x: (numpy.array, shape (grd.nr)) dimensionless cell center position, in units where the initial ring location is $x = 1$
- tau: (numpy.array, shape (nOut)) output times, in units where the characteristic viscous evolution time t_s is 1
- colOut: (numpy.array, shape (nOut, grd.nr)) column densities at times tau and positions x, normalized to col0 = ring_mass/(np.pi*ring_loc**2)
- colExact: (numpy.array, shape (nOut, grd.nr)) exact analytic solution for the column density at times tau and positions x, normalized to col0
- err: (numpy.array, shape (nOut, grd.nr)) fractional error between analytic and exact solutions for column density at times tau and positions x
- presOut: (numpy.array, shape (nOut, grd.nr)) vertically-integrated pressure at times tau and positions x, normalized to pres0 = col0*kB*init_temp/(mu*mH), where kB is Boltzmann's constant, mH is the mass of a hydrogen atom, and mu=2.33
- l1err: (numpy.array, shape (nOut)) L^1 norm error on the numerical solution as compared to the analytic one, at times tau
- mBndOut: (numpy.array, shape (nOut, 2)) cumulative mass transported across the inner and outer boundaries up to the specified time, normalized to col0*ring_loc**2; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction
- mDiskOut: (numpy.array, shape (nOut)) total mass of all material on the computational grid at times tau, normalized to col0*ring_loc**2
- eBndOut: (numpy.array, shape (nOut, 2)) cumulative energy transported across the inner and outer boundaries up to the specified time, normalized to pres0*ring_loc**2*(gamma-1); positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction

- `eDiskOut`: (`numpy.array, shape (nOut)`) total energy of all material on the computational grid at times `tau`, normalized to `pres0*ring_loc**2*(gamma-1)`
- `nStep`: (`long`) total number of simulation time steps
- `nIter`: (`long`) total number of implicit iterations, summed over all time steps
- `nFail`: (`long`) total number of times the implicit solver failed to converge

9.3.7 `vader.test.ringrad`

Defined in `vader/test/ringrad.py`.

```
def ringrad(paramFile, nOut=65, outTimes=None, **kwargs):
```

Parameters:

- `paramFile`: (`str`, INPUT) name of the parameter file to use
- `outTimes`: (`numpy.array`, INPUT) array specifying times at which to save output
- `nOut`: (`int`, INPUT) number of output times to save, uniformly spaced; ignored if `outTimes` is not `None`
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` constructed from `paramFile` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below

This function tests `VADER` by running the same setup as `vader.test.ring`, but with an equation of state that includes contributions from both radiation and gas pressure. The keys expected in the `paramFile` are identical to those in the `vader.test.ring` test (except that the key `init_temp` is renamed `init_teff`, to better reflect its physical meaning in this test), plus the additional key `f_z0`, which defines the relative importance of radiation and gas pressure in the initial conditions; see the Krumholz & Forbes method paper for the full definition. An example parameter file can be found in `vader/test/ringrad.param`. The function runs the simulation and then returns a `namedtuple` containing the following elements:

- `x`: (`numpy.array, shape (grd.nr)`) dimensionless cell center position, in units where the initial ring location is $x = 1$
- `tau`: (`numpy.array, shape (nOut)`) output times, in units where the characteristic viscous evolution time t_s is 1
- `colOut`: (`numpy.array, shape (nOut, grd.nr)`) column densities at times `tau` and positions `x`

- `colExact`: (`numpy.array, shape (nOut, grd.nr)`) exact analytic solution for the column density at times `tau` and positions `x`
- `err`: (`numpy.array, shape (nOut, grd.nr)`) fractional error between analytic and exact solutions for column density at times `tau` and positions `x`
- `presOut`: (`numpy.array, shape (nOut, grd.nr)`) vertically-integrated pressure at times `tau` and positions `x`
- `pGasOut`: (`numpy.array, shape (nOut, grd.nr)`) vertically-integrated gas pressure at times `tau` and positions `x`
- `pRadOut`: (`numpy.array, shape (nOut, grd.nr)`) vertically-integrated radiation pressure at times `tau` and positions `x`
- `eIntOut`: (`numpy.array, shape (nOut, grd.nr)`) internal energy per unit area at times `tau` and positions `x`
- `eGravOut`: (`numpy.array, shape (nOut, grd.nr)`) gravitational potential energy per unit area at times `tau` and positions `x`
- `eOrbOut`: (`numpy.array, shape (nOut, grd.nr)`) orbital kinetic energy per unit area at times `tau` and positions `x`
- `eOut`: (`numpy.array, shape (nOut, grd.nr)`) total energy per unit area at times `tau` and positions `x`
- `tempOut`: (`numpy.array, shape (nOut, grd.nr)`) effective temperature T_{eff} at times `tau` and positions `x`
- `eBndOut`: (`numpy.array, shape (nOut, 2)`) cumulative energy transported across the inner and outer boundaries up to the specified time; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction
- `eDiskOut`: (`numpy.array, shape (nOut)`) total energy of all material on the computational grid at times `tau`
- `nStep`: (`long`) total number of simulation time steps
- `nIter`: (`long`) total number of implicit iterations, summed over all time steps
- `nFail`: (`long`) total number of times the implicit solver failed to converge

Note that the outputs of this routine differ from those of `vader.test.ring` in that column densities and pressure for this routine are in physical rather than normalized units.

9.3.8 `vader.test.selfsim`

Defined in `vader/test/selfsim.py`

```
def selfsim(paramFile, nOut=65, outTimes=None, **kwargs):
```

Parameters:

- `paramFile`: (`str`, INPUT) name of the parameter file to use
- `outTimes`: (`numpy.array`, INPUT) array specifying times at which to save output
- `nOut`: (`int`, INPUT) number of output times to save, uniformly spaced; ignored if `outTimes` is not `None`
- `kwargs`: (`dict`, INPUT) any additional keywords specified here are appended to the `paramDict` constructed from `paramFile` before the simulation is run
- Return value: the function returns a `namedtuple` whose elements are described below

This function tests `VADER` by running a simulation that is compared to the analytic solution of Lynden-Bell & Pringle (1974, MNRAS, 168, 603) for the evolution of a self-similar disk. The keys expected in the `paramFile`, beyond the standard ones defined in Table 1, are: `R0`, which gives the initial scale radius, `nu0`, which gives the kinematic viscosity at the initial scale radius, `Mdot0`, which gives the accretion rate onto the central point mass at 1 viscous evolution time, `init_temp`, which gives the initial gas temperature, and `end_time`, which gives the end time of the simulation in viscous evolution times. An example parameter file can be found in `vader/test/selfsim.param`. The routine runs the simulation and then returns the results in a `namedtuple` containing the following elements:

- `x`: (`numpy.array, shape (grd.nr)`) dimensionless cell center position, in units where the initial scale radius is $x = 1$
- `T`: (`numpy.array, shape (nOut)`) output times, in units where the characteristic viscous evolution time t_s is 1
- `colOut`: (`numpy.array, shape (nOut, grd.nr)`) column densities at times `T` and positions `x`, normalized to `col1 = Mdot0/(3*np.pi*nu0)`
- `colExact`: (`numpy.array, shape (nOut, grd.nr)`) exact analytic solution for the column density at times `T` and positions `x`, normalized to `col1`
- `err`: (`numpy.array, shape (nOut, grd.nr)`) fractional error between analytic and exact solutions for column density at times `T` and positions `x`
- `presOut`: (`numpy.array, shape (nOut, grd.nr)`) vertically-integrated pressure at times `T` and positions `x`, normalized to `pres1 = col1*kB*init_temp/(mu*mH)`, where `kB` is Boltzmann's constant, `mH` is the mass of a hydrogen atom, and `mu=2.33`

- `l1err`: (`numpy.array, shape (nOut)`) L^1 norm error on the numerical solution as compared to the analytic one, at times `T`
- `mBndOut`: (`numpy.array, shape (nOut, 2)`) cumulative mass transported across the inner and outer boundaries up to the specified time, normalized to `col1*R0**2`; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction
- `mDiskOut`: (`numpy.array, shape (nOut)`) total mass of all material on the computational grid at times `T`, normalized to `col1*R0**2`
- `eBndOut`: (`numpy.array, shape (nOut, 2)`) cumulative energy transported across the inner and outer boundaries up to the specified time, normalized to `pres1*R0**2*(gamma-1)`; positive values indicate transport in the $+r$ direction, negative values indicate transport in the $-r$ direction
- `eDiskOut`: (`numpy.array, shape (nOut)`) total energy of all material on the computational grid at times `tau`, normalized to `pres1*R0**2*(gamma-1)`
- `nStep`: (`long`) total number of simulation time steps
- `nIter`: (`long`) total number of implicit iterations, summed over all time steps
- `nFail`: (`long`) total number of times the implicit solver failed to converge

10 Revision History

1. Version 1.0, 6/2014 – initial release