# Fronts

*Release 0.9.3*

**Gabriel S. Gerlero**

**Feb 18, 2020**

# CONTENTS

Welcome to the API documentation for Fronts!

Don't forget to read the README and check out the examples on the project's GitHub page.

CHAPTER

ONE

# MAIN MODULE `FRONTS`

## 1.1 Solvers

| | |
|---|---|
| *solve*(D, Si, Sb[, dS_dob_bracket, radial, ...]) | Solve an instance of the general problem. |
| *solve_from_guess*(D, Si, Sb, o_guess, S_guess) | Solve an instance of the general problem starting from a guess of the solution. |
| *inverse*(o, S) | Solve the inverse problem. |

### 1.1.1 fronts.solve

`fronts.`**`solve`**(*D*, *Si*, *Sb*, *dS_dob_bracket=(-1.0, 1.0)*, *radial=False*, *ob=0.0*, *Si_tol=1e-06*, *maxiter=100*, *verbose=0*)

Solve an instance of the general problem.

Given a positive function *D*, scalars $S_i$, $S_b$ and $o_b$, and coordinate unit vector $\hat{\mathbf{r}}$, finds a function *S* of *r* and *t* such that:

$$\begin{cases} \dfrac{\partial S}{\partial t} = \nabla \cdot \left[ D\left(S\right) \dfrac{\partial S}{\partial r} \hat{\mathbf{r}} \right] & r > r_b(t), t > 0 \\ S(r, 0) = S_i & r > 0 \\ S(r_b(t), t) = S_b & t > 0 \\ r_b(t) = o_b \sqrt{t} \end{cases}$$

**Parameters**

- **`D`** (*`callable`*) – Twice-differentiable function that maps the range of *S* to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.
- **`Si`** (*`float`*) – $S_i$, the initial value of *S* in the domain.
- **`Sb`** (*`float`*) – $S_b$, the value of *S* imposed at the boundary.
- **`dS_dob_bracket`** (*`(float, float), optional`*) – An interval that contains the value of the derivative of *S* with respect to the Boltzmann variable *o* (i.e., $\frac{dS}{do}$) at the boundary in the solution. The interval can be made as wide as desired, at the cost of additional iterations required to obtain the solution. To refine a solution obtained previously with this same function, pass in that solution's final *dS_dob_bracket*. This parameter is always checked and a *ValueError* is raised if a *dS_dob_bracket* is found not valid for the problem.
- **`radial`** (*`{False, 'cylindrical', 'spherical'}, optional`*) – Choice of coordinate unit vector $\hat{\mathbf{r}}$. Must be one of the following:

- ***False* (default)** $\hat{\mathbf{r}}$ is any coordinate unit vector in rectangular (Cartesian) coordinates, or an axial unit vector in a cylindrical coordinate system
- **`'cylindrical'`** $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical coordinate system
- **`'spherical'`** $\hat{\mathbf{r}}$ is the radial unit vector in a spherical coordinate system

- **ob** (*float, optional*) – $o_b$, which determines the behavior of the boundary. The default is zero, which implies that the boundary always exists at $r = 0$. It must be strictly positive if *radial* is not *False*. Be aware that a non-zero value implies a moving boundary.
- **Si_tol** (*float, optional*) – Absolute tolerance for $S_i$.
- **maxiter** (*int, optional*) – Maximum number of iterations. A *RuntimeError* will be raised if the specified tolerance is not achieved within this number of iterations. Must be >= 2.
- **verbose** (*{0, 1, 2}, optional*) – Level of algorithm's verbosity. Must be one of the following:
  - 0 (default) : work silently.
  - 1 : display a termination report.
  - 2 : display progress during iterations.

**Returns**

**solution** – See *SemiInfiniteSolution* for a description of the solution object. Additional fields specific to this solver are included in the object:

- ***o*** [numpy.ndarray, shape (n,)] Final solver mesh, in terms of the Boltzmann variable $o$.
- ***niter*** [int] Number of iterations required to find the solution.
- ***dS_dob_bracket*** [(float, float)] Subinterval of *dS_dob_bracket* that contains the value of dS/do at the boundary in the solution. May be used in a subsequent call with a smaller *Si_tol* to avoid reduntant iterations if wanting to refine a previously obtained solution.

**Return type** *SemiInfiniteSolution*

**See also:**

*solve_from_guess()*

### Notes

Given the expression of $r_b$ which specifies the location of the boundary, a fixed boundary can be had only if $o_b = 0$. Any other $o_b$ implies a moving boundary. This restriction affects radial problems in particular.

This function works by transforming the partial differential equation with the Boltzmann transformation using *ode* and then solving the resulting ODE repeateadly using the 'Radau' method as implemented in *scipy.integrate.solve_ivp*. The boundary condition is satisfied exactly as the starting point, and the algorithm iterates with different values of $\frac{dS}{do}$ at the boundary (chosen from within *dS_dob_bracket* using bisection) until it finds the solution that also satisfies the initial condition with the specified tolerance. This scheme assumes that $\frac{dS}{do}$ at the boundary varies continuously with $S_i$.

## 1.1.2 fronts.solve_from_guess

`fronts.`**`solve_from_guess`**(*D*, *Si*, *Sb*, *o_guess*, *S_guess*, *radial=False*, *max_nodes=1000*, *verbose=0*)

Solve an instance of the general problem starting from a guess of the solution.

Given a positive function *D*, scalars $S_i$, $S_b$ and $o_b$, and coordinate unit vector $\hat{\mathbf{r}}$, finds a function *S* of *r* and *t* such that:

$$\begin{cases} \dfrac{\partial S}{\partial t} = \nabla \cdot \left[ D\left(S\right) \dfrac{\partial S}{\partial r} \hat{\mathbf{r}} \right] & r > r_b(t), t > 0 \\ S(r, 0) = S_i & r > 0 \\ S(r_b(t), t) = S_b & t > 0 \\ r_b(t) = o_b \sqrt{t} \end{cases}$$

This function requires an initial mesh and guess of the solution. It is significantly less robust than *solve*, and will fail to converge in many cases that the latter can easily handle (whether it converges will usually depend heavily on the problem, the initial mesh and the guess of the solution; it will raise a *RuntimeError* on failure). However, when it converges it is usually faster than *solve*, which may be an advantage for some use cases. You should nonetheless prefer *solve* unless you have a particular use case for which you have found this function to be better.

Possible use cases include refining a solution (note that *solve* can do that too), optimization runs in which known solutions make good first approximations of solutions with similar parameters and every second of computing time counts, and in the implementation of other solving algorithms. In all these cases, *solve* should probably be used as a fallback for when this function fails.

**Parameters**

- **D** (*callable*) – Twice-differentiable function that maps the range of *S* to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.
- **Si** (*float*) – $S_i$, the initial value of *S* in the domain.
- **Sb** (*float*) – $S_b$, the value of *S* imposed at the boundary.
- **o_guess** (*numpy.array_like, shape (n_guess,)*) – Initial mesh in terms of the Boltzmann variable *o*. Must be strictly increasing. `o_guess[0]` is $o_b$, which determines the behavior of the boundary. If zero, it implies that the boundary always exists at $r = 0$. It must be strictly positive if *radial* is not *False*. Be aware that a non-zero value implies a moving boundary. On the other end, `o_guess[-1]` must be large enough to contain the solution to the semi-infinite problem.
- **S_guess** (*float or numpy.array_like, shape (n_guess,)*) – Initial guess of *S* at the points in *o_guess*. If a single value, the guess is interpreted as uniform.
- **radial** (*{False, 'cylindrical', 'spherical'}, optional*) – Choice of coordinate unit vector $\hat{\mathbf{r}}$. Must be one of the following:
  - ***False* (default)** $\hat{\mathbf{r}}$ is any coordinate unit vector in rectangular (Cartesian) coordinates, or an axial unit vector in a cylindrical coordinate system
  - **'cylindrical'** $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical coordinate system
  - **'spherical'** $\hat{\mathbf{r}}$ is the radial unit vector in a spherical coordinate system
- **max_nodes** (*int, optional*) – Maximum allowed number of mesh nodes.
- **verbose** (*{0, 1, 2}, optional*) – Level of algorithm's verbosity. Must be one of the following:
  - 0 (default) : work silently.
  - 1 : display a termination report.

- 2 : display progress during iterations.

**Returns**

**solution** – See *SemiInfiniteSolution* for a description of the solution object. Additional fields specific to this solver are included in the object:

- **o** [numpy.ndarray, shape (n,)] Final solver mesh, in terms of the Boltzmann variable o.
- **niter** [int] Number of iterations required to find the solution.
- **rms_residuals** [numpy.ndarray, shape (n-1,)] RMS values of the relative residuals over each mesh interval.

**Return type** *SemiInfiniteSolution*

**See also:**

`solve()`

### Notes

Given that the location of the boundary is expressed in terms of the Boltzmann variable, a fixed boundary can be had only if `o_guess[0]` is 0. Any other `o_guess[0]` implies a moving boundary. This restriction affects radial problems in particular.

This function works by transforming the partial differential equation with the Boltzmann transformation using *ode* and then solving the resulting ODE with SciPy's boundary value problem solver *scipy.integrate.solve_bvp* and a two-point Dirichlet condition that matches the boundary and initial conditions of the problem. Upon that solver's convergence, it runs a final check on whether the candidate solution also satisfies the semi-infinite condition (which implies $\frac{dS}{do} \to 0$ as $o \to \infty$).

## 1.1.3 fronts.inverse

`fronts.`**`inverse`**(*o*, *S*)

Solve the inverse problem.

Given a function *S* of *r* and *t*, and scalars $S_i$, $S_b$ and $o_b$, finds a positive function *D* of the values of *S* such that:

$$\begin{cases} \dfrac{\partial S}{\partial t} = \dfrac{\partial}{\partial r}\left(D\left(S\right)\dfrac{\partial S}{\partial r}\right) & r > r_b(t), t > 0 \\ S(r,0) = S_i & r > 0 \\ S(r_b(t),t) = S_b & t > 0 \\ r_b(t) = o_b\sqrt{t} & \end{cases}$$

*S* is input via its values on a finite set of points expressed in terms of the Boltzmann variable. Problems in radial coordinates are not supported.

**Parameters**

- **o** (*numpy.array_like, shape (n,)*) – Points where *S* is known, expressed in terms of the Boltzmann variable. Must be strictly increasing.
- **S** (*numpy.array_like, shape (n,)*) – Values of the solution at *o*. Must be monotonic (either non-increasing or non-decreasing) and `S[-1]` must be $S_i$.

**Returns** **D** – Twice-differentiable function that maps the range of *S* to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order)

as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Return type** callable

**See also:**

*o()*

### Notes

An *o(S)* function is constructed by interpolating the input data with a PCHIP monotonic cubic spline. The function *D* is then constructed by applying the expressions that result from solving the Boltzmann-transformed equation for *D*.

## 1.2 Solutions

| | |
|---|---|
| *SemiInfiniteSolution* | Continuous solution to a semi-infinite problem. |
| *Solution* | Base class for solutions using the Boltzmann transformation. |

### 1.2.1 fronts.SemiInfiniteSolution

**class** fronts.**SemiInfiniteSolution**(*sol*, *ob*, *oi*, *D*)

Continuous solution to a semi-infinite problem.

Its methods describe a continuous solution to a problem of finding a function *S* of *r* and *t* such that:

$$\frac{\partial S}{\partial t} = \nabla \cdot \left[ D\left(S\right) \frac{\partial S}{\partial r} \hat{\mathbf{r}} \right]$$

with *r* bounded at $r_b(t) = o_b\sqrt{t}$ on the left and unbounded to the right. For $r < r_b(t)$, the methods will evaluate to NaNs.

**Parameters**

- **sol** (*callable*) – Solution to the corresponding ODE obtained with *ode*. For any *o* in the closed interval [*ob*, *oi*], `sol(o)[0]` is the value of *S* at *o*, and `sol(o)[1]` is the value of the derivative $\frac{dS}{do}$ at *o*. *sol* will only be evaluated inside this interval.
- **ob** (*float*) – $o_b$, which determines the behavior of the boundary.
- **oi** (*float*) – Value of the Boltzmann variable at which the solution can be considered to be equal to the initial condition. Must be $\geq o_b$.
- **D** (*callable*) – *D* used to obtain *sol*. Must be the same function that was passed to *ode*.

**See also:**

*solve*, *solve_from_guess*, *ode*

**__init__**(*sol*, *ob*, *oi*, *D*)

Initialize self. See help(type(self)) for accurate signature.

### Methods

| | |
|---|---|
| S([r, t, o]) | $S$, the unknown function. |
| __init__(sol, ob, oi, D) | Initialize self. |
| dS_do([r, t, o]) | $\frac{dS}{do}$, derivative of $S$ with respect to the Boltzmann variable. |
| dS_dr(r, t) | $\frac{\partial S}{\partial r}$, spatial derivative of $S$. |
| dS_dt(r, t) | $\frac{\partial S}{\partial t}$, time derivative of $S$. |
| flux(r, t) | Diffusive flux of $S$. |
| rb(t) | $r_b$, the location of the boundary. |

**S** (*r=None*, *t=None*, *o=None*)
$S$, the unknown function.

May be called either with parameters *r* and *t*, or with just *o*.

**Parameters**

- **r** (*float or numpy.ndarray, optional*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*. If this parameter is used, you must also pass *t* and cannot pass *o*.
- **t** (*float or numpy.ndarray, optional*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable ith *r*. Values must be positive. If this parameter is used, you must also pass *r* and cannot pass *o*.
- **o** (*float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, you cannot pass *r* or *t*.

**Returns** **S** – If *o* is passed, the return is of the same type and shape as *o*. Otherwise, return is a float if both *r* and *t* are floats, or a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**dS_do** (*r=None*, *t=None*, *o=None*)
$\frac{dS}{do}$, derivative of $S$ with respect to the Boltzmann variable.

May be called either with parameters *r* and *t*, or with just *o*.

**Parameters**

- **r** (*float or numpy.ndarray, optional*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*. If this parameter is used, you must also pass *t* and cannot pass *o*.
- **t** (*float or numpy.ndarray, optional*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive. If this parameter is used, you must also pass *r* and cannot pass *o*.
- **o** (*float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, you cannot pass *r* or *t*.

**Returns** **dS_do** – If *o* is passed, the return is of the same type and shape as *o*. Otherwise, the return is a float if both *r* and *t* are floats, or a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**dS_dr** (*r*, *t*)
$\frac{\partial S}{\partial r}$, spatial derivative of $S$.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s) along the coordinate. If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **dS_dr** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**dS_dt**(*r*, *t*)

$\frac{\partial S}{\partial t}$, time derivative of *S*.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **dS_dt** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**flux**(*r*, *t*)

Diffusive flux of *S*.

Returns the diffusive flux of *S* in the direction $\hat{\mathbf{r}}$, equal to $-D(S)\frac{\partial S}{\partial r}$.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **flux** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**rb**(*t*)

$r_b$, the location of the boundary.

This is the point where the boundary condition of the problem is imposed.

**Parameters** **t** (*float or numpy.ndarray*) – Time(s). Values must be positive.

**Returns** **rb** – The return is of the same type and shape as *t*.

**Return type** float or numpy.ndarray

### Notes

Depending on $o_b$, the boundary may be fixed at $r = 0$ or it may move with time.

### 1.2.2 fronts.Solution

**class** fronts.**Solution**(*sol*, *D*)

Base class for solutions using the Boltzmann transformation.

Its methods describe a continuous solution to any problem of finding a function $S$ of $r$ and $t$ such that:

$$\frac{\partial S}{\partial t} = \nabla \cdot \left[ D\left(S\right) \frac{\partial S}{\partial r} \hat{\mathbf{r}} \right]$$

**Parameters**

- **sol** (*callable*) – Solution to an ODE obtained with *ode*. For any float or *numpy.ndarray* *o*, `sol(o)[0]` are the values of $S$ at *o*, and `sol(o)[1]` the values of the derivative *dS/do* at *o*.
- **D** (*callable*) – *D* used to obtain *sol*. Must be the same function that was passed to *ode*.

**See also:**

*ode*

**__init__**(*sol*, *D*)

Initialize self. See help(type(self)) for accurate signature.

#### Methods

| | |
|---|---|
| *S*([r, t, o]) | *S*, the unknown function. |
| *__init__*(sol, D) | Initialize self. |
| *dS_do*([r, t, o]) | $\frac{dS}{do}$, derivative of $S$ with respect to the Boltzmann variable. |
| *dS_dr*(r, t) | $\frac{\partial S}{\partial r}$, spatial derivative of $S$. |
| *dS_dt*(r, t) | $\frac{\partial S}{\partial t}$, time derivative of $S$. |
| *flux*(r, t) | Diffusive flux of $S$. |

**S**(*r=None*, *t=None*, *o=None*)

*S*, the unknown function.

May be called either with parameters $r$ and $t$, or with just $o$.

**Parameters**

- **r** (*float or numpy.ndarray, optional*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with $t$. If this parameter is used, you must also pass $t$ and cannot pass $o$.
- **t** (*float or numpy.ndarray, optional*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable ith $r$. Values must be positive. If this parameter is used, you must also pass $r$ and cannot pass $o$.
- **o** (*float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, you cannot pass $r$ or $t$.

**Returns** **S** – If $o$ is passed, the return is of the same type and shape as $o$. Otherwise, return is a float if both $r$ and $t$ are floats, or a *numpy.ndarray* of the shape that results from broadcasting $r$ and $t$.

**Return type** float or numpy.ndarray

**dS_do**(*r=None*, *t=None*, *o=None*)
$\frac{dS}{do}$, derivative of *S* with respect to the Boltzmann variable.

May be called either with parameters *r* and *t*, or with just *o*.

**Parameters**

- **r** (*float or numpy.ndarray, optional*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*. If this parameter is used, you must also pass *t* and cannot pass *o*.
- **t** (*float or numpy.ndarray, optional*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive. If this parameter is used, you must also pass *r* and cannot pass *o*.
- **o** (*float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, you cannot pass *r* or *t*.

**Returns** **dS_do** – If *o* is passed, the return is of the same type and shape as *o*. Otherwise, the return is a float if both *r* and *t* are floats, or a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**dS_dr**(*r*, *t*)
$\frac{\partial S}{\partial r}$, spatial derivative of *S*.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s) along the coordinate. If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **dS_dr** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**dS_dt**(*r*, *t*)
$\frac{\partial S}{\partial t}$, time derivative of *S*.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **dS_dt** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**flux**(*r*, *t*)
Diffusive flux of *S*.

Returns the diffusive flux of *S* in the direction $\hat{\mathbf{r}}$, equal to $-D(S)\frac{\partial S}{\partial r}$.

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.

- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **flux** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

## 1.3 Boltzmann transformation

| | |
|---|---|
| *ode*(D[, radial]) | Transform the PDE into an ODE. |
| *o*(r, t) | Transform to the Boltzmann variable. |
| *do_dr*(r, t) | Spatial derivative of the Boltzmann transformation. |
| *do_dt*(r, t) | Time derivative of the Boltzmann transformation. |
| *r*(o, t) | Transform back from the Boltzmann variable into *r*. |
| *t*(o, r) | Transform back from the Boltzmann variable into *t*. |
| *as_o*([r, t, o]) | Transform to the Boltzmann variable if called with *r* and *t*. |

### 1.3.1 fronts.ode

fronts.**ode**(*D*, *radial=False*)

Transform the PDE into an ODE.

Given a positive function *D* and coordinate unit vector $\hat{\mathbf{r}}$, transform the partial differential equation (PDE) in which *S* is the unknown function of *r* and *t*:

$$\frac{\partial S}{\partial t} = \nabla \cdot \left[ D\left(S\right) \frac{\partial S}{\partial r} \hat{\mathbf{r}} \right]$$

into an ordinary differential equation (ODE) where *S* is an unknown function of the Boltzmann variable *o*.

This function returns the *fun* and *jac* callables that may be used to solve the ODE with the solvers included with SciPy (*scipy.integrate* module). The second-order ODE is expressed as a system of first-order ODEs with independent variable *o* where `y[0]` in *fun* and *jac* correspond to the value of the function *S* itself and `y[1]` to its first derivative $\frac{dS}{do}$.

*fun* and *jac* support both non-vectorized usage (where their first argument is a float) as well as vectorized usage (when *numpy.ndarray* objects are passed as both arguments).

**Parameters**

- **D** (*callable*) – Twice-differentiable function that maps the range of *S* to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.
- **radial** (*{False, 'cylindrical', 'spherical'}, optional*) – Choice of coordinate unit vector $\hat{\mathbf{r}}$. Must be one of the following:
  - ***False*** **(default)** $\hat{\mathbf{r}}$ is any coordinate unit vector in rectangular (Cartesian) coordinates, or an axial unit vector in a cylindrical coordinate system
  - **'cylindrical'** $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical coordinate system
  - **'spherical'** $\hat{\mathbf{r}}$ is the radial unit vector in a spherical coordinate system

**Returns**

- **fun** (*callable*) – Function that returns the right-hand side of the system. The calling signature is `fun(o, y)`.
- **jac** (*callable*) – Function that returns the Jacobian matrix of the right-hand side of the system. The calling signature is `jac(o, y)`.

**See also:**

*Solution()*, *o()*

### Notes

If *radial* is not *False*, the PDE is undefined at $r = 0$, and therefore the returned ODE is also undefined for $o = 0$.

## 1.3.2 fronts.o

`fronts.o(r, t)`

Transform to the Boltzmann variable.

Returns the Boltzmann variable at the given *r* and *t*, which is the result of applying the Boltzmann transformation:

$$o(r, t) = \frac{r}{\sqrt{t}}$$

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **o** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**See also:**

*do_dr()*, *do_dt()*, *r()*, *t()*, *as_o()*

## 1.3.3 fronts.do_dr

`fronts.do_dr(r, t)`

Spatial derivative of the Boltzmann transformation.

Returns the partial derivative $\frac{\partial o}{\partial r}$ evaluated at (*r*,'t').

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **do_dr** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**See also:**

*o()*, *do_dt()*

### 1.3.4 fronts.do_dt

fronts.**do_dt**(*r*, *t*)

Time derivative of the Boltzmann transformation.

Returns the partial derivative $\frac{\partial o}{\partial t}$ evaluated at (*r*,'t').

**Parameters**

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive.

**Returns** **do_dt** – The return is a float if both *r* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *r* and *t*.

**Return type** float or numpy.ndarray

**See also:**

*o()*, *do_dr()*

### 1.3.5 fronts.r

fronts.**r**(*o*, *t*)

Transform back from the Boltzmann variable into *r*.

**Parameters**

- **o** (*float or numpy.ndarray*) – Value(s) of the Boltzmann variable. If a *numpy.ndarray*, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *o*. Values must be positive.

**Returns** **r** – The return is a float if both *o* and *t* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *o* and *t*.

**Return type** float or numpy.ndarray

**See also:**

*o()*, *t()*

### 1.3.6 fronts.t

fronts.**t**(*o*, *r*)

Transform back from the Boltzmann variable into *t*.

**Parameters**

- **o** (*float or numpy.ndarray*) – Value(s) of the Boltzmann variable. If a *numpy.ndarray*, it must have a shape broadcastable with *r*.

- **r** (*float or numpy.ndarray*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *o*.

**Returns** **t** – The return is a float if both *o* and *r* are floats. Otherwise it is a *numpy.ndarray* of the shape that results from broadcasting *o* and *r*.

**Return type** float or numpy.ndarray

**See also:**

*o()*, *r()*

### 1.3.7 fronts.as_o

fronts.**as_o**(*r=None*, *t=None*, *o=None*)

Transform to the Boltzmann variable if called with *r* and *t*. Passes the values through if called with *o* only. On other combinations of arguments, it raises a *TypeError* with a message explaining valid usage.

This function is a helper to define other functions that may be called either with *r* and *t*, or with just *o*.

**Parameters**

- **r** (*float or numpy.ndarray, optional*) – Location(s). If a *numpy.ndarray*, it must have a shape broadcastable with *t*. If this parameter is used, you must also pass *t* and cannot pass *o*.
- **t** (*float or numpy.ndarray, optional*) – Time(s). If a *numpy.ndarray*, it must have a shape broadcastable with *r*. Values must be positive. If this parameter is used, you must also pass *r* and cannot pass *o*.
- **o** (*float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, you cannot pass *r* or *t*.

**Returns** **o** – Passes *o* through if it is given. Otherwise, it returns `o(r,t)`.

**Return type** float or numpy.ndarray

**See also:**

*o()*

CHAPTER

TWO

# SUBMODULE FRONTS.D: INCLUDED D FUNCTIONS

| | |
|---|---|
| D.constant([D]) | Return a constant *D* function. |
| D.power_law(k[, a, epsilon]) | Return a power-law *D* function. |
| D.van_genuchten([n, m, l, alpha, Ks, S_range]) | Return a Van Genuchten moisture diffusivity function. |
| D.richards(K, C) | Return a moisture diffusivity function for a Richards equation problem. |

## 2.1 fronts.D.constant

fronts.D.**constant**(*D=1.0*)

Return a constant *D* function.

Given a positive constant *D*, returns the function *D*:

$$D(S) = D$$

**Parameters** **D** (*float*) – A positive constant.

**Returns** **D** – Function that maps any value of *S* to the given constant. It can be called as `D(S)` to obtain the value. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function, which are always zero, are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Return type** callable

### Notes

This function is not particularly useful: a constant *D* will turn a diffusion problem into a linear one, which has an exact solution and no numerical solvers are necessary. However, it is provided here given that it is the simplest supported function.

## 2.2 fronts.D.power_law

fronts.D.**power_law**(*k*, *a=1*, *epsilon=0*)

Return a power-law *D* function.

Given the scalars $a$, $k$ and $\varepsilon$, returns a function *D* defined as:

$$D(S) = aS^k + \varepsilon$$

**Parameters**

- **k** (*float*) – Exponent
- **a** (*float*) – Constant factor
- **epsilon** (*float*) – $\varepsilon$, the deviation term

**Returns** **D** – Twice-differentiable function that maps *S* to values according to the expression. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Return type** callable

### Notes

Keep in mind that, depending on the parameters, the returned *D* does not necessarily map every value of *S* to a positive value.

## 2.3 fronts.D.van_genuchten

`fronts.D.`**`van_genuchten`**(*n=None*, *m=None*, *l=0.5*, *alpha=1.0*, *Ks=1.0*, *S_range=(0.0, 1.0)*)

Return a Van Genuchten moisture diffusivity function.

Given the parameters $K_s$, $\alpha$, *m*, *l*, $S_r$ and $S_s$, the Van Genuchten moisture diffusivity function *D* is defined as:

$$D(S) = \frac{(1-m)K_s}{\alpha m(S_s - S_r)} S_e^{(l-\frac{1}{m})} \left((1 - S_e^{\frac{1}{m}})^{-m} + (1 - S_e^{\frac{1}{m}})^{m} - 2\right)$$

where:

$$S_e = \frac{S - S_r}{S_s - S_r}$$

and *S* is the saturation (or moisture content).

In common usage, the *m* parameter is replaced with an *n* parameter so that $m = 1 - \frac{1}{n}$. This function supports either parameter.

**Parameters**

- **n** (*float, optional*) – *n* parameter in the Van Genuchten model. Must be >1. You must pass either *n* or *m* (but not both).
- **m** (*float, optional*) – *m* parameter in the Van Genucthen model. Must be strictly between 0 and 1. You must pass either *n* or *m* (but not both).
- **l** (*float, optional*) – Pore connectivity parameter. The default is 0.5. Must be strictly between 0 and 1.
- **alpha** (*float, optional*) – $\alpha$ parameter of the Van Genucthen model. The default is 1. Must be positive.
- **Ks** (*float, optional*) – $K_s$, the hydraulic conductivity when saturated. The default is 1. Must be positive.
- **S_range** ((*float, float*), *optional*) – the tuple ($S_r$, $S_s$), where $S_r$ is the minimum (or residual) saturation and $S_s$ the maximum saturation. The default is (0, 1). $S_s$ must be greater than $S_r$.

**Returns** **D** – Twice-differentiable function that maps values of *S* in the open interval ($S_r$, $S_s$) to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Return type** callable

### Notes

The expression used is the one found in Van Genuchten's original paper [1], but with the addition of the optional *l* parameter.

### References

[1] VAN GENUCHTEN, M. Th. A closed-form equation for predicting the hydraulic conductivity of unsaturated soils. Soil science society of America journal, 1980, vol. 44, no 5, p. 892-898.

## 2.4 fronts.D.richards

`fronts.D.`**`richards`**(*K*, *C*)

Return a moisture diffusivity function for a Richards equation problem.

Given the functions *K* and *C*, returns the function:

$$D(S) = \frac{K(S)}{C(S)}$$

This function helps the conversion of horizontal Richards equation problems (for which those two functions are parameters) into moisture diffusivity problems that can be solved using this library.

**Parameters**

- **`K`** (*`callable`*) – Hydraulic conductivity function, defined in terms of saturation. A twice-differentiable function that maps values of *S* to positive values. It can be called as `K(S)` to evaluate it at *S*. It can also be called as `K(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *K* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.
- **`C`** (*`callable`*) – Capillary capacity function, defined in terms of saturation. A twice-differentiable function that maps values of *S* to positive values. It can be called as `C(S)` to evaluate it at *S*. It can also be called as `C(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *C* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Returns** **D** – Twice-differentiable function that maps values of *S* in the domains of both *K* and *C* to positive values. It can be called as `D(S)` to evaluate it at *S*. It can also be called as `D(S, n)` with *n* equal to 1 or 2, in which case the first *n* derivatives of the function evaluated at the same *S* are included (in order) as additional return values. While mathematically a scalar function, *D* operates in a vectorized fashion with the same semantics when *S* is a *numpy.ndarray*.

**Return type** callable

- genindex

- search

# INDEX

## Symbols



## A



## C



## D



## F



## I



## O



## P



## R



## S



## T



## V