Fronts

Release 0.9.8

Gabriel S. Gerlero

Jun 11, 2020

CONTENTS

Welcome to the reference documentation for Fronts. This documentation covers the usage of all available functions and classes.

For an introduction to the software, please refer to the README file, which is displayed on the project's GitHub and PyPI pages.

Users may also want to look at the example cases, available on the GitHub page under the examples directory.

CHAPTER

ONE

MAIN PACKAGE `fronts`

1.1 Solvers

solve(D, i, b[, radial, ob, itol, ...])	Solve a problem with a Dirichlet boundary condition.
solve_flowrate(D, i, Qb, radial[, ob, ...])	Solve a radial problem with a fixed-flowrate boundary condition.
solve_from_guess(D, i, b, o_guess, guess[, ...])	Alternative solver for problems with a Dirichlet boundary condition.
inverse(o, samples)	Extract D from samples of a solution.

1.1.1 fronts.solve

`fronts.`**`solve`**(*D*, *i*, *b*, *radial=False*, *ob=0.0*, *itol=0.001*, *d_dob_hint=None*, *d_dob_bracket=None*, *method='implicit'*, *maxiter=100*, *verbose=0*)

Solve a problem with a Dirichlet boundary condition.

Given a positive function D, scalars θ_i, θ_b and o_b, and coordinate unit vector $\hat{\mathbf{r}}$, finds a function θ of r and t such that:

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

Parameters

- **`D`** (callable or *sympy.Expression* or str or float) – Callable that evaluates D and its derivatives, obtained from the `fronts.D` module or defined in the same manner—i.e.:
  - `D(theta)` evaluates and returns D at `theta`
  - `D(theta, 1)` returns both the value of D and its first derivative at `theta`
  - `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

  When called by this function, `theta` is always a single float. However, calls as `D(theta)` should also accept a NumPy array argument.

  Alternatively, instead of a callable, the argument can be the expression of D in the form of a string or *sympy.Expression* with a single variable. In this case, the solver will differentiate and evaluate the expression as necessary.

- **i** (*float*) – Initial condition, θ_i.
- **b** (*float*) – Imposed boundary value, θ_b.
- **radial** (*{False, 'cylindrical', 'polar', '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'` or `'polar'`: $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical or polar coordinate system
  - `'spherical'`: $\hat{\mathbf{r}}$ is the radial unit vector in a spherical coordinate system
- **ob** (*float, optional*) – Parameter o_b, which determines the behavior of the boundary. The default is zero, which means that the boundary always exists at $r = 0$. It must be strictly positive if *radial* is not False. A non-zero value implies a moving boundary.

Returns

solution – See *Solution* 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.
- *niter (int)* – Number of iterations required to find the solution.
- *d_dob_bracket (sequence of two floats or None)* – If available, an interval that contains the value of $d\theta/do|_b$. May be used as the input *d_dob_bracket* in a subsequent call with a smaller *itol* for the same problem in order to avoid reduntant iterations. Whether this interval is available or not depends on the strategy used internally by the solver; in particular, this field is never *None* if a *d_dob_bracket* is passed when calling the function.

Return type *Solution*

Other Parameters

- **itol** (*float, optional*) – Absolute tolerance for the initial condition.
- **d_dob_hint** (*None or float, optional*) – Optional hint to the solver. If given, it should be a number close to the expected value of the derivative of θ with respect to the Boltzmann variable at the boundary (i.e., $d\theta/do|_b$) in the solution to be found. This parameter is typically not needed.
- **d_dob_bracket** (*None or sequence of two floats, optional*) – Optional search interval that brackets the value of $d\theta/do|_b$ in the solution. If given, the solver will use bisection to find a solution in which $d\theta/do|_b$ falls inside that interval (a *ValueError* will be raised for an incorrect interval). This parameter cannot be passed together with a *d_dob_hint*. It is also not needed in typical usage.
- **method** (*{'implicit', 'explicit'}, optional*) –

  Selects the integration method used by the solver:
  - `'implicit'` (default): uses a Radau IIA implicit method of order 5. A sensible default choice that will work for any problem
  - `'explicit'`: uses the DOP853 explicit method of order 8. As an explicit method, it trades off general solver robustness and accuracy for faster results in "well-behaved" cases. With this method, the second derivative of D is not needed. Requires SciPy 1.4.0 or later (Python 3 only)

- **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 nonnegative.
- **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: also display progress during iterations

See also:

solve_from_guess(), *solve_flowrate()*

Notes

This function works by transforming the partial differential equation with the Boltzmann transformation using *ode()* and then solving the resulting ODE repeatedly with the chosen integration method as implemented in the `scipy.integrate` module and a custom shooting algorithm. The boundary condition is satisfied exactly as the starting point, and the algorithm iterates with different values of $d\theta/do|_b$ until it finds the solution that also verifies the initial condition within the specified tolerance. Trial values of $d\theta/do|_b$ are selected automatically by default (using heuristics, which can also take into account an optional hint if passed by the user), or by bisecting an optional search interval. This scheme assumes that $d\theta/do|_b$ varies continuously with θ_i.

1.1.2 fronts.solve_flowrate

`fronts.`**`solve_flowrate`**(*D*, *i*, *Qb*, *radial*, *ob=1e-06*, *angle=6.283185307179586*, *height=None*, *itol=0.001*, *b_hint=None*, *b_bracket=None*, *method='implicit'*, *maxiter=100*, *verbose=0*)

Solve a radial problem with a fixed-flowrate boundary condition.

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

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

Parameters

- **D** (callable or *sympy.Expression* or str or float) – Callable that evaluates D and its derivatives, obtained from the `fronts.D` module or defined in the same manner—i.e.:

  – `D(theta)` evaluates and returns D at `theta`

  – `D(theta, 1)` returns both the value of D and its first derivative at `theta`

  – `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

  where `theta` is always a float in the latter two cases, but it may be either a single float or a NumPy array when *D* is called as `D(theta)`.

Alternatively, instead of a callable, the argument can be the expression of D in the form of a string or *sympy.Expression* with a single variable. In this case, the solver will differentiate and evaluate the expression as necessary.

- **i** (*float*) – Initial condition, θ_i.
- **Qb** (*float*) – Imposed flow rate of θ at the boundary, Q_b.

  The flow rate is considered in the direction of $\hat{\mathbf{r}}$: a positive value means that θ is flowing into the domain; negative values mean that θ flows out of the domain.
- **radial** (*{'cylindrical', 'polar'}*) – Choice of coordinate unit vector $\hat{\mathbf{r}}$. Must be one of the following:
  - `'cylindrical'` : $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical coordinate system
  - `'polar'` : $\hat{\mathbf{r}}$ is the radial unit vector in a polar coordinate system
- **ob** (*float, optional*) – Parameter o_b, which determines the behavior of the boundary. It must be positive. The boundary acts as a line source or sink in the limit where *ob* tends to zero.
- **angle** (*float, optional*) – Total angle covered by the domain. The default is 2π, which means that θ may flow through the boundary in all directions. Must be positive and no greater than 2π.
- **height** (*None or float, optional*) – Axial height of the domain if `radial=='cylindrical'`. Not allowed if `radial=='polar'`.

Returns

solution – See *Solution* 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.
- *niter (int)* – Number of iterations required to find the solution.
- *b_bracket (sequence of two floats or None)* – If available, an interval that contains the value of θ_b. May be used as the input *b_bracket* in a subsequent call with a smaller *itol* for the same problem in order to avoid reduntant iterations. Whether this interval is available or not depends on the strategy used internally by the solver; in particular, this field is never *None* if a *b_bracket* is passed when calling the function.

Return type *Solution*

Other Parameters

- **itol** (*float, optional*) – Absolute tolerance for the initial condition.
- **b_hint** (*None or float, optional*) – Optional hint to the solver. If given, it should be a number close to the expected value of θ at the boundary (i.e. θ_b) in the solution to be found.
- **b_bracket** (*None or sequence of two floats, optional*) – Optional search interval that brackets the value of θ_b in the solution. If given, the solver will use bisection to find a solution in which θ_b falls inside that interval (a *ValueError* will be raised for an incorrect interval). This parameter cannot be passed together with a *b_hint*.
- **method** (*{'implicit', 'explicit'}, optional*) –

  Selects the integration method used by the solver:
  - `'implicit'` (default): uses a Radau IIA implicit method of order 5. A sensible default choice that will work for any problem

  - `'explicit'`: uses the DOP853 explicit method of order 8. As an explicit method, it trades off general solver robustness and accuracy for faster results in "well-behaved" cases. With this method, the second derivative of D is not needed. Requires SciPy 1.4.0 or later (Python 3 only)

- **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 nonnegative.
- **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: also display progress during iterations

See also:

solve()

Notes

This function works by transforming the partial differential equation with the Boltzmann transformation using *ode()* and then solving the resulting ODE repeatedly with the chosen integration method as implemented in the `scipy.integrate` module and a custom shooting algorithm. The boundary condition is satisfied exactly as the starting point, and the algorithm iterates with different values of θ at the boundary until it finds the solution that also verifies the initial condition within the specified tolerance. Trial values of θ at the boundary are selected automatically by default (using heuristics, which can also take into account an optional hint if passed by the user), or by bisecting an optional search interval. This scheme assumes that θ at the boundary varies continuously with θ_i.

1.1.3 fronts.solve_from_guess

`fronts.`**`solve_from_guess`**(*D*, *i*, *b*, *o_guess*, *guess*, *radial=False*, *max_nodes=1000*, *verbose=0*)

Alternative solver for problems with a Dirichlet boundary condition.

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

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

Alternative to the main *solve()* function. This function requires a starting 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 starting 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 or *sympy.Expression* or str or float) – Callable that evaluates D and its derivatives, obtained from the `fronts.D` module or defined in the same manner—i.e.:
  - `D(theta)` evaluates and returns D at `theta`
  - `D(theta, 1)` returns both the value of D and its first derivative at `theta`
  - `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

  where `theta` may be a single float or a NumPy array.

  Alternatively, instead of a callable, the argument can be the expression of D in the form of a string or *sympy.Expression* with a single variable. In this case, the solver will differentiate and evaluate the expression as necessary.
- **i** (*float*) – Initial condition, θ_i.
- **b** (*float*) – Imposed boundary value, θ_b.
- **o_guess** (*numpy.array_like, shape (n_guess,)*) – Starting mesh in terms of the Boltzmann variable. Must be strictly increasing. `o_guess[0]` is taken as the value of the parameter 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. 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.
- **guess** (*float or numpy.array_like, shape (n_guess,)*) – Starting guess of the solution at the points in *o_guess*. If a single value, the guess is assumed uniform.
- **radial** (*{False, 'cylindrical', 'polar', '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'` or `'polar'`: $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical or polar coordinate system
  - `'spherical'`: $\hat{\mathbf{r}}$ is the radial unit vector in a spherical coordinate system

Returns

solution – See *Solution* 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.
- *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 *Solution*

Other Parameters

- **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: also display progress during iterations

See also:

solve()

Notes

This function works by transforming the partial differential equation with the Boltzmann transformation using *ode()* and then solving the resulting ODE with SciPy's collocation-based 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 $d\theta/do \to 0$ as $o \to \infty$).

1.1.4 fronts.inverse

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

Extract *D* from samples of a solution.

Given a function θ of *r* and *t*, and scalars θ_i, θ_b and o_b, finds a positive function *D* of the values of θ such that:

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

θ is taken as its values on a discrete 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 θ is known, expressed in terms of the Boltzmann variable. Must be strictly increasing.
- **`samples`** (*numpy.array_like, shape (n,)*) – Values of θ at o. Must be monotonic (either non-increasing or non-decreasing) and `samples[-1]` must be the initial value θ_i.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

D is guaranteed to be continuous; however, its derivatives are not.

Return type callable

See also:

o()

Notes

An o function of θ is constructed by interpolating the input data with a PCHIP monotonic cubic spline. The returned D uses the spline to evaluate the expressions that result from solving the Boltzmann-transformed equation for D.

1.2 Solutions

Solution	Solution to a problem.
BaseSolution	Base class for solutions using the Boltzmann transformation.

1.2.1 fronts.Solution

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

Solution to a problem.

Represents a continuously differentiable function θ of r and t such that:

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

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 an ODE obtained with *ode*. For any float or one-dimensional NumPy array `o` with values in the closed interval [*ob*, *oi*], `sol(o)[0]` are the values of θ at `o`, and `sol(o)[1]` are the values of the derivative $d\theta/do$ at *o`*. *sol* will only be evaluated in this interval.
- **ob** (*float*) – Parameter o_b, which determines the behavior of the boundary in the problem.
- **oi** (*float*) – Value of the Boltzmann variable at which the solution can be considered to be equal to the initial condition. Cannot be less than *ob*.
- **D** (*callable*) – Function to evaluate D at arbitrary values of the solution. Must be callable with a float or NumPy array as its argument.

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

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

Methods

__init__(sol, ob, oi, D)	Initialize self.
d_do([r, t, o])	Boltzmann-variable derivative of the solution.
d_dr(r, t)	Spatial derivative of the solution.
d_drb(t)	Spatial derivative of the solution at the boundary.
d_dt(r, t)	Time derivative of the solution.
d_dtb(t)	Time derivative of the solution at the boundary.
flux(r, t)	Diffusive flux.
fluxb(t)	Boundary flux.
rb(t)	Boundary location.

Attributes

b	Boundary value of the solution.
d_dob	Derivative of the solution with respect to the Boltzmann variable at the boundary.
i	Initial value of the solution.
ob	Parameter o_b.

__call__(*r=None*, *t=None*, *o=None*)
Evaluate the solution.

Evaluates and returns θ. May be called either with arguments *r* and *t*, or with just *o*.

Parameters

- **r** (*None or float or numpy.ndarray, shape (n,), optional*) – Location(s). If this parameter is used, *t* must also be given.
- **t** (*None or float or numpy.ndarray, optional*) – Time(s). Values must be positive.
- **o** (*None or float or numpy.ndarray, shape (n,) optional*) – Value(s) of the Boltzmann variable. If this parameter is used, neither *r* nor *t* can be given.

Returns

Return type float or numpy.ndarray, shape (n,)

property b
Boundary value of the solution.

Type float

d_do(*r=None*, *t=None*, *o=None*)
Boltzmann-variable derivative of the solution.

Evaluates and returns $d\theta/do$, the derivative of θ with respect to the Boltzmann variable. May be called either with arguments *r* and *t*, or with just *o*.

Parameters

- **r** (*None or float or numpy.ndarray, shape (n,), optional*) – Location(s). If this parameter is used, *t* must also be given.
- **t** (*None or float or numpy.ndarray, shape (n,), optional*) – Time(s). Values must be positive.

- **o** (*None or float or numpy.ndarray, shape (n,), optional*) – Value(s) of the Boltzmann variable. If this parameter is used, neither *r* nor *t* can be given.

Returns

Return type float or numpy.ndarray, shape (n,)

property d_dob

Derivative of the solution with respect to the Boltzmann variable at the boundary.

Type float

d_dr(*r*, *t*)

Spatial derivative of the solution.

Evaluates and returns $\partial\theta/\partial r$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s) along the coordinate.
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

d_drb(*t*)

Spatial derivative of the solution at the boundary.

Evaluates and returns $\partial\theta/\partial r|_b$. Equivalent to `self.d_dr(self.rb(t), t)`.

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

Returns

Return type float or numpy.ndarray, shape (n,)

d_dt(*r*, *t*)

Time derivative of the solution.

Evaluates and returns $\partial\theta/\partial t$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s).
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

d_dtb(*t*)

Time derivative of the solution at the boundary.

Evaluates and returns $\partial\theta/\partial t|_b$. Equivalent to `self.d_dt(self.rb(t), t)`.

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

Returns

Return type float or numpy.ndarray, shape (n,)

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

Returns the diffusive flux of θ in the direction $\hat{\mathbf{r}}$, equal to $-D(\theta)\partial\theta/\partial r$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s).
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

fluxb(*t*)
Boundary flux.

Returns the diffusive flux of θ at the boundary, in the direction $\hat{\mathbf{r}}$. Equivalent to `self.flux(self.rb(t), t)`.

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

Returns

Return type float or numpy.ndarray, shape (n,)

property i
Initial value of the solution.

Type float

property ob
Parameter o_b.

Type float

rb(*t*)
Boundary location.

Returns r_b, the location of the boundary.

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

Parameters **t** (*float or numpy.ndarray*) – Time(s). Values must not be negative.

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

Return type float or numpy.ndarray

1.2.2 fronts.BaseSolution

class fronts.**BaseSolution**(*sol*, *D*)
Base class for solutions using the Boltzmann transformation.

Represents a continuously differentiable function θ of *r* and *t* such that:

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

Parameters

- **sol** (*callable*) – Solution to an ODE obtained with *ode*. For any float or *numpy.ndarray* `o`, `sol(o)[0]` are the values of θ at `o`, and `sol(o)[1]` are the values of the derivative $d\theta/do$ at `o`.
- **D** (*callable*) – Function to evaluate D at arbitrary values of the solution. Must be callable with a float or NumPy array as its argument.

See also:

ode

__init__ (*sol*, *D*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(sol, D)	Initialize self.
d_do([r, t, o])	Boltzmann-variable derivative of the solution.
d_dr(r, t)	Spatial derivative of the solution.
d_dt(r, t)	Time derivative of the solution.
flux(r, t)	Diffusive flux.

__call__ (*r=None*, *t=None*, *o=None*)
Evaluate the solution.

Evaluates and returns θ. May be called either with arguments *r* and *t*, or with just *o*.

Parameters

- **r** (*None or float or numpy.ndarray, shape (n,), optional*) – Location(s). If this parameter is used, *t* must also be given.
- **t** (*None or float or numpy.ndarray, optional*) – Time(s). Values must be positive.
- **o** (*None or float or numpy.ndarray, shape (n,) optional*) – Value(s) of the Boltzmann variable. If this parameter is used, neither *r* nor *t* can be given.

Returns

Return type float or numpy.ndarray, shape (n,)

d_do (*r=None*, *t=None*, *o=None*)
Boltzmann-variable derivative of the solution.

Evaluates and returns $d\theta/do$, the derivative of θ with respect to the Boltzmann variable. May be called either with arguments *r* and *t*, or with just *o*.

Parameters

- **r** (*None or float or numpy.ndarray, shape (n,), optional*) – Location(s). If this parameter is used, *t* must also be given.
- **t** (*None or float or numpy.ndarray, shape (n,), optional*) – Time(s). Values must be positive.
- **o** (*None or float or numpy.ndarray, shape (n,), optional*) – Value(s) of the Boltzmann variable. If this parameter is used, neither *r* nor *t* can be given.

Returns

Return type float or numpy.ndarray, shape (n,)

d_dr (*r*, *t*)
Spatial derivative of the solution.

Evaluates and returns $\partial\theta/\partial r$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s) along the coordinate.
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

d_dt (*r*, *t*)
Time derivative of the solution.

Evaluates and returns $\partial\theta/\partial t$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s).
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

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

Returns the diffusive flux of θ in the direction $\hat{\mathbf{r}}$, equal to $-D(\theta)\partial\theta/\partial r$.

Parameters

- **r** (*float or numpy.ndarray, shape (n,)*) – Location(s).
- **t** (*float or numpy.ndarray, shape (n,)*) – Time(s). Values must be positive.

Returns

Return type float or numpy.ndarray, shape (n,)

1.3 Boltzmann transformation

ode(D[, radial, catch_errors])	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*, *catch_errors=False*)

Transform the PDE into an ODE.

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

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

into an ordinary differential equation (ODE) where θ 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 θ itself and `y[1]` to its first derivative $d\theta/do$.

Parameters

- **`D`** (*`callable`*) – Callable as `D(theta, 1)`, which must return both D and its first derivative evaluated at `theta`. If the returned *jac* will be used, it must also be callable as `D(theta, 2)` to obtain D, its first derivative, and its second derivative evaluated at `theta`. To allow vectorized usage of *fun* and *jac*, *D* must be able to accept a NumPy array as `theta`.
- **`radial`** (*`{False, 'cylindrical', 'polar', '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'` or `'polar'`: $\hat{\mathbf{r}}$ is the radial unit vector in a cylindrical or polar 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)`. In non-vectorized usage, `o` is a float and `y` is any sequence of two floats, and *fun* returns a NumPy array with shape (2,). For vectorized usage, `o` and `y` must be NumPy arrays with shapes (n,) and (2,n) respectively, and the return is a NumPy array of shape (2,n).
- **jac** (*callable*) – Function that returns the Jacobian matrix of the right-hand side of the system. The calling signature is `jac(o, y)`. In non-vectorized usage, `o` is a scalar and `y` is an array-like with shape (2,n), and the return is a NumPy array with shape (2,2). In vectorized usage, `o` and `y` must be NumPy arrays with shapes (n,) and (2,n) respectively, and the return is a NumPy array of shape (2,2,n).

Other Parameters **catch_errors** (*bool, optional*) – Whether to catch exceptions that may be attributed to a domain error of *D* and convert them to NaN (or +/-Inf) values in the returns of *fun* and *jac*. If True, the following exceptions will be caught as domain errors:

- *ValueError* and *ArithmeticError* (the latter includes *ZeroDivisionError*) raised by a call to *D*
- *ZeroDivisionError* when attempting to divide by a zero value returned by *D*
- *TypeError* when assigning to the return array (usually because Python arithmetic inside *D* caused that function to return a complex value)

Returning NaN or infinite values signals the domain error to a caller that does not expect *fun* and *jac* to raise exceptions to indicate this condition (such as SciPy).

This option is useful in non-vectorized usage, and particularly where the invocation of *D* might use native Python mathematical functions and types. It is less relevant in vectorized usage and other cases that involve only NumPy types and functions, as those will not cause these exceptions by default.

If False (default), the exceptions will be allowed to propagate to the callers of *fun* and *jac*.

See also:

BaseSolution(), *o()*

Notes

If *radial* is other than *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:

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

Parameters

- **r** (*float or numpy.ndarray*) – Location(s). If an array, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If an array, 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 an array 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 $\partial o/\partial r$ evaluated at (*r*, *t*).

Parameters

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

- **t** (*float or numpy.ndarray*) – Time(s). If an array, 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 an array 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 $\partial o/\partial t$ evaluated at (*r*, *t*).

Parameters

- **r** (*float or numpy.ndarray*) – Location(s). If an array, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If an array, 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 an array 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 an array, it must have a shape broadcastable with *t*.
- **t** (*float or numpy.ndarray*) – Time(s). If an array, it must have a shape broadcastable with *o*. Values must not be negative.

Returns **r** – The return is a float if both *o* and *t* are floats. Otherwise it is an array 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 array, it must have a shape broadcastable with *r*. Values must not be zero.
- **r** (*float or numpy.ndarray*) – Location(s). If a NumPy array, 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 an array 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** (*None or float or numpy.ndarray, optional*) – Location(s). If this parameter is used, *t* must also be given. If an array, it must have a shape broadcastable with *t*.
- **t** (*None or float or numpy.ndarray, optional*) – Time(s). If an array, it must have a shape broadcastable with *r*. Values must be positive.
- **o** (*None or float or numpy.ndarray, optional*) – Value(s) of the Boltzmann variable. If this parameter is used, neither *r* nor *t* can be given.

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

Return type float or numpy.ndarray

See also:

o()

CHAPTER

TWO

MODULE FRONTS.D: DIFFUSIVITY FUNCTIONS

D.constant(D0)	Return a constant *D* function.
D.power_law(k[, a, epsilon])	Return a power-law *D* function.
D.brooks_and_corey(n[, l, alpha, Ks, k, nu, ...])	Return a Brooks and Corey moisture diffusivity function.
D.van_genuchten([n, m, l, alpha, Ks, k, nu, ...])	Return a Van Genuchten moisture diffusivity function.
D.from_expr(expr[, vectorized, max_derivatives])	Create a *D* function from a SymPy-compatible expression.
D.richards(C, kr[, Ks, k, nu, g])	Return a moisture diffusivity function for a Richards equation problem.

2.1 fronts.D.constant

fronts.D.**constant**(*D0*)

Return a constant *D* function.

Given D_0, returns the function D:

$$D(\theta) = D_0$$

Parameters **D0** (*float*) – D_0, a positive constant

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

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=1.0*, *epsilon=0.0*)

Return a power-law *D* function.

Given the scalars a, k and ε, returns a function *D* defined as:

$$D(\theta) = a\theta^k + \varepsilon$$

Parameters

- **k** (*float*) – Exponent
- **a** (*float, optional*) – Constant factor. The default is 1.
- **epsilon** (*float, optional*) – ε, the deviation term. The default is 0.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

Return type callable

Notes

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

2.3 fronts.D.brooks_and_corey

`fronts.D.`**`brooks_and_corey`**(*n*, *l=1.0*, *alpha=1.0*, *Ks=None*, *k=None*, *nu=1e-06*, *g=9.81*, *theta_range=0.0, 1.0*)

Return a Brooks and Corey moisture diffusivity function.

Given the saturated hydraulic conductivity K_S and parameters α, n, l, θ_r and θ_s, the Brooks and Corey moisture diffusivity function *D* is defined as:

$$D(\theta) = \frac{K_S S_e^{1/n+l+1}}{\alpha n(\theta_s - \theta_r)}$$

where:

$$S_e = \frac{\theta - \theta_r}{\theta_s - \theta_r}$$

and θ is water content.

Parameters

- **n** (*float*) – *n* parameter.
- **l** (*float, optional*) – *l* parameter. The default is 1.
- **alpha** (*float, optional*) – α parameter. The default is 1. Must be positive.
- **Ks** (*None or float, optional*) – K_S, the saturated hydraulic conductivity. Must be positive. If neither *Ks* nor *k* are given, the saturated hydraulic conductivity is assumed to be 1.
- **k** (*None or float, optional*) – Intrinsic permeability of the porous medium. Can be given in place of *Ks*, which results in the saturated hydraulic conductivity being computed using $K_S = kg/\nu$. Must be positive.
- **nu** (*float, optional*) – ν, the kinematic viscosity of the wetting fluid. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 1e-6, approximately the kinematic viscosity of water at 20°C in SI units.
- **g** (*float, optional*) – Magnitude of the gravitational acceleration. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 9.81, the gravity of Earth in SI units.
- **theta_range** (*sequence of two floats, optional*) – (θ_r, θ_s), where θ_r is the minimum (also known as residual) water content and θ_s is the maximum water content. The default is (0, 1). θ_s must be greater than θ_r.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

Return type callable

References

[1] BROOKS, R.; COREY, T. Hydraulic properties of porous media. Hydrology Papers, Colorado State University, 1964, vol. 24, p. 37.

2.4 fronts.D.van_genuchten

fronts.D.**van_genuchten**(*n=None*, *m=None*, *l=0.5*, *alpha=1.0*, *Ks=None*, *k=None*, *nu=1e-06*, *g=9.81*, *theta_range=0.0, 1.0*)

Return a Van Genuchten moisture diffusivity function.

Given the saturated hydraulic conductivity K_S and parameters α, *m*, *l*, θ_r and θ_s, the Van Genuchten moisture diffusivity function *D* is defined as:

$$D(\theta) = \frac{(1-m)K_S}{\alpha m(\theta_s - \theta_r)} S_e^{l-\frac{1}{m}} \left((1 - S_e^{\frac{1}{m}})^{-m} + (1 - S_e^{\frac{1}{m}})^{m} - 2\right)$$

where:

$$S_e = \frac{\theta - \theta_r}{\theta_s - \theta_r}$$

and θ is water content.

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

Parameters

- **n** (*float, optional*) – *n* parameter in the Van Genuchten model. Must be >1. Either *n* or *m* must be given (but not both).
- **m** (*float, optional*) – *m* parameter in the Van Genuchten model. Must be strictly between 0 and 1. Either *n* or *m* must be given (but not both).
- **l** (*float, optional*) – Pore connectivity parameter. The default is 0.5.
- **alpha** (*float, optional*) – α parameter of the Van Genuchten model. The default is 1. Must be positive.
- **Ks** (*None or float, optional*) – K_S, the saturated hydraulic conductivity. Must be positive. If neither *Ks* nor *k* are given, the saturated hydraulic conductivity is assumed to be 1.
- **k** (*None or float, optional*) – Intrinsic permeability of the porous medium. Can be given in place of *Ks*, which results in the saturated hydraulic conductivity being computed using $K_S = kg/\nu$. Must be positive.
- **nu** (*float, optional*) – ν, the kinematic viscosity of the wetting fluid. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 1e-6, approximately the kinematic viscosity of water at 20°C in SI units.
- **g** (*float, optional*) – Magnitude of the gravitational acceleration. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 9.81, the gravity of Earth in SI units.
- **theta_range** (*sequence of two floats, optional*) – (θ_r, θ_s), where θ_r is the minimum (also known as residual) water content and θ_s is the maximum water content. The default is (0, 1). θ_s must be greater than θ_r.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`

- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

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.

2.5 fronts.D.from_expr

`fronts.D.`**`from_expr`**(*expr*, *vectorized=True*, *max_derivatives=2*)

Create a D function from a SymPy-compatible expression.

Parameters

- **`expr`** (*sympy.Expression* or str or float) – SymPy-compatible expression containing up to one free symbol.
- **`vectorized`** (*`bool, optional`*) – Whether the returned D must be compatible with a solver that uses vectorized calls.

  If True (default), the first argument passed to D may always be either a float or a NumPy array. However, if False, calls as `D(theta, 1)` or `D(theta, 2)` will assume that `theta` is a single float, which may allow for optimizations that speed up the evaluations required by a solver that does not use vectorized calls.

  Note that, regardless of this setting, calls to D that do not ask for any derivatives (i.e., calls as `D(theta)`) will always take floats and arrays interchangeably. This behavior ensures that D is always compatible with the solution classes.
- **`max_derivatives`** (*`int, optional`*) – Highest-order derivative of D that may be required. Can be 0, 1 or 2. The default is 2.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- If *max_derivatives* >= 1, `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- If *max_derivatives* is 2, `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

If *vectorized* is True, the argument `theta` may be a single float or a NumPy array in all cases. If *vectorized* is False, `theta` may be either a float or an array when *D* is called as `D(theta)`, but it must be a float otherwise.

Return type callable

Notes

Users will rarely need to call this function, as all built-in solver functions already do so themselves when they receive an expression as *D*.

2.6 fronts.D.richards

`fronts.D.`**`richards`**(*C*, *kr*, *Ks=None*, *k=None*, *nu=1e-06*, *g=9.81*)

Return a moisture diffusivity function for a Richards equation problem.

Given *K_S* and the functions *C* and k_r (whose argument is either water content or saturation), returns the function:

$$D(\theta) = \frac{K_S k_r(\theta)}{C(\theta)}$$

This function helps transform problems of the horizontal Richards equation (for which K_S, k_r, and *C* are known parameters) into problems of the moisture diffusivity equation that can be solved with this library.

Parameters

- **`C`** (*`callable`*) – Capillary capacity function (also known as hydraulic capacity function). A twice-differentiable function that maps values of θ to positive values. It can be called as `C(theta)` to evaluate it at `theta`. It can also be called as `C(theta, n)` with `n` equal to 1 or 2, in which case the first `n` derivatives of the function evaluated at the same `theta` are included (in order) as additional return values. While mathematically a scalar function, *C* operates in a vectorized fashion with the same semantics when `theta` is a *numpy.ndarray*.
- **`kr`** (*`callable`*) – k_r, the relative permeability function (also known as relative conductivity function). A twice-differentiable function that maps values of θ to positive values (usually between 0 and 1). It can be called as `kr(theta)` to evaluate it at `theta`. It can also be called as `kr(theta, n)` with `n` equal to 1 or 2, in which case the first `n` derivatives of the function evaluated at the same `theta` are included (in order) as additional return values. While mathematically a scalar function, *kr* operates in a vectorized fashion with the same semantics when `theta` is a *numpy.ndarray*.
- **`Ks`** (*`None or float, optional`*) – K_S, the saturated hydraulic conductivity. Must be positive. If neither *Ks* nor *k* are given, the saturated hydraulic conductivity is assumed to be 1.
- **`k`** (*`None or float, optional`*) – Intrinsic permeability of the porous medium. Can be given in place of *Ks*, which results in the saturated hydraulic conductivity being computed using $K_S = kg/\nu$. Must be positive.
- **`nu`** (*`float, optional`*) – ν, the kinematic viscosity of the wetting fluid. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 1e-6, approximately the kinematic viscosity of water at 20°C in SI units.
- **`g`** (*`float, optional`*) – Magnitude of the gravitational acceleration. Only used if *k* is passed instead of *Ks*. Must be positive. Defaults to 9.81, the gravity of Earth in SI units.

Returns

D –

Function to evaluate D and its derivatives:

- `D(theta)` evaluates and returns D at `theta`
- `D(theta, 1)` returns both the value of D and its first derivative at `theta`
- `D(theta, 2)` returns the value of D, its first derivative, and its second derivative at `theta`

In all cases, the argument `theta` may be a single float or a NumPy array.

Return type callable

- genindex
- search

INDEX

Symbols

A

B

C

D

F

I

O

P

R

S

T

V