findhorizon Documentation

Release 0.1

Ian Hawke

May 02, 2014

Contents

Contents:

CHAPTER 1

findhorizon Module

1.1 Find Black Hole apparent horizons in an axisymmetric spacetime.

Black holes are usually described by their *horizon* enclosing the singularity. Locating any horizon in a general (typically numerically generated) spacetime can be very hard - see Thornburg's review [R1] for details. Here we restrict to the simpler problem of a specific type of axisymmetric spacetime, where the equation to solve reduces to a boundary value problem.

Strictly this module constructs *trapped surfaces*, which are surfaces where null geodesics (light rays) are ingoing. The apparent horizon is the outermost trapped surface.

1.1.1 Notes

The technical restrictions on the spacetime are

1. axisymmetric, so the singularities are on the z axis;
2. singularities have Brill-Lindquist type;
3. spacetime is conformally flat;
4. coordinates are chosen to obey maximal slicing with no shift;
5. data is time symmetric.

1.1.2 References

class findhorizon.**Spacetime**(*z_positions*, *masses*, *reflection_symmetric=False*)

Define an axisymmetric spacetime.

For an axisymmetric, vacuum spacetime with Brill-Lindquist singularities the only parameters that matter is the locations of the singularities (i.e. their z-location) and their bare masses.

Parameters **z_positions** : list of float

The location of the singularities on the z-axis.

masses : list of float

The bare masses of the singularities.

reflection_symmetry : bool, optional

Is the spacetime symmetric across the x-axis.

See also:

TrappedSurface class defining the trapped surfaces on a spacetime.

Examples

```
>>> schwarzschild = Spacetime([0.0], [1.0], True)
```

This defines standard Schwarzschild spacetime with unit mass.

```
>>> binary = Spacetime([-0.75, 0.75], [1.0, 1.1])
```

This defines two black holes, with the locations mirrored but different masses.

class findhorizon.**TrappedSurface**(*spacetime*, *z_centre=0.0*)

Store any trapped surface, centred on a particular point.

The trapped surface is defined in polar coordinates centred on a point on the z-axis; the z-axis is $\theta = 0$ or $\theta = \pi$.

Parameters **spacetime** : Spacetime

The spacetime on which the trapped surface lives.

z_centre : float

The z-coordinate about which the polar coordinate system describing the trapped surface is defined.

See also:

Spacetime class defining the spacetime.

Notes

With the restricted spacetime considered here, a trapped surface $h(\theta)$ satisfies a boundary value problem with the boundary conditions $h'(\theta = 0) = 0 = h'(\theta = \pi)$. If the spacetime is reflection symmetric about the x-axis then the boundary condition $h'(\theta = \pi/2) = 0$ can be used and the domain restricted to $0 \leq \theta \leq \pi/2$.

The shooting method is used here. In the reflection symmetric case the algorithm needs a guess for the initial horizon radius, $h(\theta = 0)$, and a single condition is enforced at $\pi/2$ to match to the boundary condition there.

In the general case we guess the horizon radius at two points, $h(\theta = 0)$ and $h(\theta = \pi)$ and continuity of both h *and* h' are enforced at the matching point $\pi/2$. The reason for this is a weak coordinate singularity on the axis at $\theta = 0, \pi$ which makes it difficult to integrate *to* these points, but possible to integrate *away* from them.

Examples

```
>>> schwarzschild = Spacetime([0.0], [1.0], True)
>>> ts1 = TrappedSurface(schwarzschild)
>>> ts1.find_r0([0.49, 0.51])
>>> ts1.solve_given_r0()
>>> print(round(ts1.r0[0], 9))
0.5
```

This example first constructs the Schwarzschild spacetime which, in this coordinate system, has the horizon with radius 0.5. The trapped surface is set up, the location of the trapped surface at $\theta = 0$ is found, which is (to the solver accuracy) at 0.5.

convert_to_cartesian()
When the solution is known in r, theta coordinates, compute the locations in cartesian coordinates (2 and 3d).

This function assumes that the trapped surface has been located and solved for.

See also:

solve_given_r0 find the trapped surface location in polar coordinates.

expansion(*theta*, *H*)
Compute the expansion for the given spacetime at a fixed point.

This function gives the differential equation defining the boundary value problem.

Parameters **theta** : float

The angular location at this point.

H : list of float

A vector of (h, h').

find_r0(*input_guess*, *full_horizon=False*)
Given some initial guess, find the correct starting location for the trapped surface using shooting.

This finds the horizon radius at $\theta = 0$ which, together with the differential equation, specifies the trapped surface location.

Parameters **input_guess** : list of float

Two positive reals defining the guess for the initial radius.

Note that the meaning is different depending on whether this is a "full" horizon or not. For a full horizon the numbers correspond to the guesses at $\theta = 0, \pi$ respectively. In the symmetric case where only one guess is needed the vector defines the interval within which a *unique* root must lie.

full_horizon : bool, optional

If the general algorithm is needed (ie, the domain should be $0 \leq \theta \leq \pi$ instead of $0 \leq \theta \leq \pi/2$).

This parameter is independent of the symmetry of the spacetime. If the spacetime is not symmetric this parameter will be ignored and the general algorithm always used. If the spacetime is symmetric it may still be necessary to use the general algorithm: for example, for two singularities it is possible to find a trapped surface surrounding just one singularity.

plot_2d(*ax*)
Given a matplotlib axis, plot the trapped surface.

Plots the surface in the x-z plane, together with the location of the singularities: marker style is used to indicate the mass of the singularity (will fail badly for masses significantly larger than 1).

Parameters **ax** : axis object

Matplotlib axis on which to do the plot.

shooting_function(*r0*)
The function used in the shooting algorithm.

This is the symmetric algorithm from integrating over $0 \leq \theta \leq \pi/2$. The difference between the derivative at the end point and the boundary condition is the error to be minimized.

Parameters **r0** : float

Initial guess for the horizon radius, as outlined above.

Returns **float** :

The error at the end point.

shooting_function_full (*r0*)

The function used in the shooting algorithm.

This is the full algorithm from integrating over $0 \leq \theta \leq \pi$. The difference between the solution and its derivative at the matching point is the error to be minimized.

Parameters **r0** : list of float

Initial guess for the horizon radius, as outlined above.

Returns **list of float** :

The error at the matching point.

solve_given_r0 (*full_horizon=False*)

Given the correct value for the initial radius, find the horizon.

This function does not find the correct radius for the trapped surface, but solves (in polar coordinates) for the complete surface location given the correct initial guess.

Parameters **full_horizon** : bool, optional

If the general algorithm is needed (ie, the domain should be $0 \leq \theta \leq \pi$ instead of $0 \leq \theta \leq \pi/2$).

This parameter is independent of the symmetry of the spacetime. If the spacetime is not symmetric this parameter will be ignored and the general algorithm always used. If the spacetime is symmetric it may still be necessary to use the general algorithm: for example, for two singularities it is possible to find a trapped surface surrounding just one singularity.

See also:

find_r0 finds the correct initial radius.

findhorizon.**find_horizon_binary** (*z=0.5*, *mass1=1.0*, *mass2=1.0*)

Utility function to find horizons for the general case.

This returns the horizon for a spacetime with precisely two singularities of mass [mass1, mass2] located at $\pm z$. That is, we work in the frame where the location of the horizons is symmetric.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Returns **ts** : TrappedSurface

Only returns the single surface found, expected to be the common horizon.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For larger separations we should not expect a common horizon.

findhorizon.**find_horizon_binary_symmetric**(*z=0.5*, *mass=1.0*)

Utility function to find horizons for reflection symmetric case.

This returns the horizon for a spacetime with precisely two singularities of identical mass located at $\pm z$.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Returns **ts** : TrappedSurface

Only returns the single surface found, expected to be the common horizon.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For larger separations we should not expect a common horizon.

findhorizon.**find_individual_horizon_binary_symmetric**(*z=0.5*, *mass=1.0*)

Utility function to find horizons for reflection symmetric case.

This returns two trapped surface for a spacetime with precisely two singularities of identical mass located at $\pm z$. These should be trapped surfaces about only one singularity.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Returns **ts1, ts2** : TrappedSurface

Returns the trapped surfaces found.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0.45 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For smaller separations we should not expect individual horizons.

findhorizon.**find_inner_outer_horizon_binary_symmetric**(*z=0.5*, *mass=1.0*)

Utility function to find horizons for reflection symmetric case.

This returns two trapped surface for a spacetime with precisely two singularities of identical mass located at $\pm z$. The outer surface is the apparent horizon; the inner surface is just a trapped surface.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Returns **ts1, ts2** : TrappedSurface

Returns the trapped surfaces found.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For larger separations we should not expect a common horizon. The inner horizon is based on a similar fit but in the narrower range $0.6 \leq z \leq 0.7$ and so it is very likely that this function will fail for $z < 0.42$.

findhorizon.**plot_horizon_3d**(*tss*)

Plot a list of horizons.

Parameters **tss** : list of TrappedSurface

All the trapped surfaces to visualize.

findhorizon.**solve_plot_binary**(*z=0.5*, *mass1=1.0*, *mass2=1.0*)

Utility function to find horizons for general case.

This returns the horizon for a spacetime with precisely two singularities of different mass located at $\pm z$.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass1, mass2 : float, optional

The mass of the singularities.

Notes

The initial guess is not easily determined, so performance is poor and the algorithm not robust

findhorizon.**solve_plot_binary_3d**(*z=0.5*, *mass1=1.0*, *mass2=1.0*)

Utility function to plot horizons in 3d for general case.

This returns the horizon for a spacetime with precisely two singularities of different mass located at $\pm z$.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass1, mass2 : float, optional

The mass of the singularities.

Notes

The initial guess is not easily determined, so performance is poor and the algorithm not robust

findhorizon.**solve_plot_symmetric**(*z=0.5*, *mass=1.0*)

Utility function to find horizons for reflection symmetric case.

This returns the horizon for a spacetime with precisely two singularities of identical mass located at $\pm z$.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For larger separations we should not expect a common horizon.

findhorizon.**solve_plot_symmetric_3d**(*z=0.5*, *mass=1.0*)

Utility function to plot horizon in 3d for reflection symmetric case.

This returns the horizon for a spacetime with precisely two singularities of identical mass located at $\pm z$.

Parameters **z** : float, optional

The distance from the origin of the singularities (ie the two singularities are located at [-z, +z]).

mass : float, optional

The mass of the singularities.

Notes

The initial guess for the horizon location is based on fitting a cubic to the results constructed for $0 \leq z \leq 0.75$ for the unit mass case. The radius should scale with the mass. For larger separations we should not expect a common horizon.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[R1] J. Thornburg, “Event and Apparent Horizon Finders for 3+1 Numerical Relativity”, Living Reviews in Relativity 10 (3) 2007. http://dx.doi.org/10.12942/lrr-2007-3.

f

f